TOHOKU UNIVERSITY

Graduate School of Information Sciences

A Hardware-Software Co-designed Cache Memory System for Energy-efficient Microprocessors

(高エネルギ効率マイクロプロセッサのための

ハードウェア・ソフトウェア協調型キャッシュメモリシステムに関する研究)

A dissertation submitted for the degree of

Doctor of Philosophy (Information Sciences)

Department of Computer and Mathematical Sciences

by

Masayuki SATO

January 16, 2012

A Hardware-Software Co-designed Cache Memory System for Energy-efficient Microprocessors

Masayuki Sato

Abstract

Performance improvements of microprocessors have relied on both the CMOS process technology and the computer architecture design. The advance in CMOS process technology has shrunk the size of the transistors. This results in increases in the number of transistors on a chip, and clock frequency, and a reduction in switching power of transistors. The computer architecture design contributes to the utilization of these transistors. A lot of transistors are used for microarchitecture design for effectively using the transistors as pipelining, out-of-order execution, speculative execution, and cache memories. As a result, microprocessors have achieved improvements of performance per power consumption.

However, in recent years, the advance in the CMOS process technology hardly contributes to the performance improvement without a significant increase of the power consumption. The power density increases and the heating problem becomes more serious as the integrated density of the transistors increases. Therefore, while the number of transistors and their power consumption increase, effective usage of the transistors as on-chip hardware resource becomes the main challenge in the microprocessor design. If the hardware resource is not used for performance improvement, it only consumes energy. Therefore, in the future, the computer architecture design becomes more important for microprocessors to improve performance per power consumption.

Under this situation, this dissertation focuses on cache memories, which play important roles in modern microprocessors. The performance of microprocessors has significantly increased while the access speed of DRAM memory has slowly improved. As a result, the performance gap between microprocessors and main memories has gradually widened. If microprocessors access data in main memories, they have to wait for the arrival of the data coming from main memories. The waiting time degrades resource utilization of microprocessors, and causes significant performance degradation. Therefore, modern microprocessors have cache memories on a chip. Cache memories store recently used data of executing applications, and supply the data when microprocessors require them again, instead of main memories. In general, a lot of data are reused from cache memories because almost all the applications have data access locality by some form. As a result, microprocessors can improve the performances since they can avoid the necessity of waiting for data coming from main memories for a long time.

However, cache memories have two problems. One is growth in power consumption. Recent advance in the CMOS process technology increases the ratio of static power consumption to the

total power consumption. Static power consumption is proportional to the area. Since cache memories occupy a large portion of the area of microprocessors, their static power is not ignorable. The other is performance degradation by inter-thread cache conflicts. In recent years, chip multiprocessors (CMPs) have become a major form of microprocessors. A CMP includes multiple cores, and each of them can execute one thread. Therefore, CMPs can increase their performances as the number of cores increases. In general, these cores share a cache memory on a chip. This sharing causes inter-thread cache conflicts. Inter-thread cache conflicts occur when the thread competes for cache capacity resource against the other simultaneously executed threads. Such a competition disturbs the effective usage of cache resource. As a result, the performance of CMPs would not improve as expected. Inter-thread cache conflicts are caused in situations where inter-thread kickouts and capacity shortage occur. Data fetched by one thread are evicted by data of another thread in inter-thread kickouts. When the former thread uses the data again, cache miss occurs. Capacity shortage is the situation where the cache capacity is too small to store the data of all threads. Hence, data required to improve the performance of each thread cannot be stored. Consequently, inter-thread cache conflicts cause performance degradation on CMPs.

Past researches have demonstrated that dynamic cache resizing mechanisms, especially the power-aware dynamic cache partitioning mechanism in this dissertation, can solve these problems around cache memories. The mechanism divides the cache into some parts and independently manages them. These parts are exclusively allocated to the executed threads. Since the allocated parts for a thread cannot store data of other threads, the mechanism can avoid inter-thread kickouts. Moreover, the mechanism can disable power supply for the parts that are not allocated, and hence the power consumption of the cache memory reduces. However, the mechanism cannot solve the capacity shortage problem, which is the other cause of inter-thread cache conflicts. Therefore, it is strongly required that the mechanism and cache memories improve the utilization efficiency of cache resource, e.g., to improve the performance without capacity growth, or to keep the performance with a smaller capacity. Hence, the objective of the dissertation is to achieve the effective usage of cache resource. To this end, this dissertation proposes three approaches to the effective usage of cache capacity for the power-aware dynamic cache partitioning mechanisms, which are included in a hardware-software co-designed cache memory system for microprocessors to achieve a high performance and a low energy consumption.

As the first approach, this dissertation discusses a voting-based working set assessment scheme for dynamic cache resizing mechanisms. Dynamic cache resizing mechanisms, e.g. the power-aware dynamic cache partitioning mechanism, enables only required cache capacities for threads based on working set assessment, and can reduce energy without significant performance degradation. In the mechanism, the accuracy of working set assessment is important to both keep the high performance and further enhance the energy saving. However, cache access

characteristics of threads affect the results of working set assessment. The mechanism monitors cache accesses and collects them as statistical data for assessment. Such data sometimes include exceptional cache access behaviors, which are irregular and ignorable for keeping the high performance. Once such behaviors are included as the statistical data, they cause an enlargement of the estimated working set size, resulting in an increase in the number of allocated parts. While these increases by exceptional cache access behaviors reduce some cache misses, its performance contributions are very small. As a result, energy consumption increases without significant performance improvement.

To avoid the harmful effects of exceptional cache access behaviors, this dissertation proposes a voting-based working set assessment scheme. The proposed scheme ignores the effect of exceptional cache access behaviors using fine-grain working set assessment and majority voting. The proposed scheme assesses the working set sizes of threads by using shorter sampling periods than that of the original scheme. These periods are called voting periods. In these voting periods, there are some periods affected by the exceptional cache access behaviors. However, the number of affected periods is smaller than that of periods that do not include exceptional cache access behaviors. Hence, by using the majority voting, the proposed scheme filters the results of affected periods and employs the results of normal periods. Consequently, the proposed scheme can distinguish and ignore the voting period including such behaviors. From the simulation results, it is observed that the energy consumption of the dynamic cache resizing mechanism reduces by up to 24%, and 10% on an average, without significant performance degradation.

As the second approach, this dissertation discusses a capacity-efficient insertion policy for on-cache data management. In general, the capacities of cache memories are smaller than those of main memories. Hence, cache memories have to manage stored data, e.g. deciding which data block should be evicted and written back to main memories when storing a new data block. The LRU replacement policy, which is the commonly used data management policy, is based on the temporal locality, in which the most-recently used data block is high potential to be reused. However, the existence of dead-on-fill blocks is the problem. A recent growth in application complexity and an increase in variations of characteristics have enlarged the number of data blocks that are not reused inherently, or reused after a very long time. If such data blocks are stored in cache memories, they become dead-on-fill blocks. Dead-on-fill blocks are the blocks that are not reused after being stored in cache memories. These blocks occupy cache memories but do not contribute to performance improvement. Therefore, these blocks only waste the cache capacity without performance contribution.

To evict these blocks as quickly as possible, this dissertation proposes a capacity-efficient insertion policy. The proposed policy flexibly adjusts resident priority for newly coming blocks, while always storing blocks with the highest resident priority in the LRU replacement policy. There is a trade-off to achieve the proposed policy. If a newly coming block is a dead-on-fill block, it should be stored with low resident priority for early eviction. On the other hand, if a newly

coming block are not dead-on-fill block, i.e., reusable blocks, storing it with low resident priority causes the early eviction of it before being reused, resulting in an additional cache miss. To consider this trade-off, the proposed policy decides resident priority for newly coming blocks by monitoring first reuses to reusable blocks in lowest resident priority. Hence, cache memories can keep only reusable blocks and early evict dead-on-fill blocks. As a result, the policy reduces the number of dead-on-fill blocks and helps the power-aware dynamic cache partitioning mechanism reduce the allocated capacity to threads without significant performance degradation. Simulation results show that the proposed policy can reduce the energy consumption by up to 30% and 6% on an average. Moreover, the proposed policy is also effective for the power-aware dynamic cache partitioning mechanism in the CMP.

As the third approach, this dissertation discusses a thread scheduling method based on working set assessment for CMPs. In many CMPs, multiple threads share a single cache. Since the working set sizes of threads highly depends on the cache access characteristics of threads, the sum of working set sizes of the thread combinations that share a cache may differ from one thread combination to another. If the threads with large working set sizes share a cache, capacity shortage occurs and degrades the performance. On the other hand, if the threads with small working set sizes share a cache, capacity shortage does not occur. The results of the preliminary evaluation show that performance degradation by capacity shortage becomes larger as the sum of working set sizes increases. Especially, performance degradation becomes significant when the sum of working set sizes of threads exceeds cache capacity.

Based on the preliminary evaluation results, this dissertation proposes a thread scheduling method based on working set assessment. Recently, a CMP includes multiple cache memories, each of which is shared by multiple cores. In this case, the thread combinations sharing a cache memory can be flexibly changed. Hence, the proposed thread scheduling method changes the thread combinations so that the capacity shortage problem is alleviated. The proposed scheduling method consists of two stages, assessment of working set sizes of threads and decision of thread assignments to cores by a scheduling algorithm. The algorithm decides the assignments so that threads with large working sets do not share a same cache, and a thread with the largest working set is coupled with a thread with the smallest working set. It helps the power-aware dynamic cache partitioning mechanism allocate a sufficient number of cache parts to threads. Simulation results show that the proposed method achieves a 1.9% higher performance than the average over all the combinations of thread assignments, and a 8.1% higher performance than the worst-case scheduling.

In conclusion, these three approaches can improve the utilization efficiency of cache resource. Therefore, the approaches in this dissertation will contribute to innovation of computer architecture design, and realization of microprocessor with high performance and low energy consumption.

Acknowledgements

This dissertation would not have been carried out without a lot of support of many people. The author would like to acknowledge all of them gratefully.

First of all, I would like to express grateful gratitude to Professor Hiroaki Kobayashi, my supervisor, who was abundantly helpful and offered invaluable assistance. I benefited immensely from not only his support, counsel, and encouragement, but also introduction to research fields during the past eight years. I would like to thank Professor Michitaka Kameyama and Professor Takafumi Aoki for their thoughtful review of this dissertation and their helpful comments. I wish to express my gratitude to Associate Professor Hiroyuki Takizawa and Assistant Professor Ryusuke Egawa for their technical and personal support and their hot encouragement.

I would like to thank Processor Emeritus Michael Flynn of Stanford University, Associate Professor Hideaki Goto, Associate Professor Kentaro Sano, and Associate Professor Kenichi Suzuki of Tohoku Institute of Technology for their valuable and helpful comments. Thanks go out to Ms. Maki Takahashi, Ms. Rikako Hasegawa, and all the members of Cyberscience Center of Tohoku University for their support to my research activities and my comfort laboratory life.

I would also like to express my appreciation to all the members of Kobayashi,

Goto, and Takizawa Laboratory, Graduate School of Information Sciences, Tohoku University. Gratefully thanks go to Dr. Isao Kotera for establishing the power-aware dynamic cache partitioning mechanism that is the basis of this dissertation. Special thanks go to Mr. Yusuke Tobo for partial support of the experimental results in this dissertation. Sincerely thanks go to the member of the processor project team, Mr. Yusuke Funaya, Mr. Gao Ye, Mr. Norihiro Tsuge, and Mr. Takumi Takai.

I express the deep appreciation to my family. I want to thank my parents for their affectionate encouragement and much support. I also want to thank my grand mother and my sister. Finally, I want to thank my partner for emotional support.

January 16, 2012

Masayuki Sato

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

In the last four decades, performance improvement of microprocessors has continued based on both advance in CMOS process technology and innovations of computer architecture design. The advance in CMOS process technology has been shrinking the size of transistors. This plays the following three important roles for performance improvement [1]. First, it increases the number of transistors on a chip, and enables microprocessors to mount a large amount of hardware resource for calculation. Second, it increases the switching speed of transistors to bring an increase in the clock frequency of microprocessors, by which the number of operations per unit time can be increased. Third, it decreases the power consumption per transistor, to realize that a larger number of transistors can be switched at less power consumption. As a result, microprocessors have achieved improvements of performance per power consumption.

The innovations of computer architecture design have been required to effectively use a large amount of hardware resource on a chip. A huge amount of hardware resource enables to implement a large number of execution units.

New microarchitecture designs such as pipelining [2], out-of-order execution [3], and speculative execution [4] have been invented to improve the performance of microprocessors. As a result, these innovations can process a large number of instructions per cycle.

As the throughput of the processor increases, its memory system has to achieve a higher bandwidth and a short access latency balanced with the processor throughput. However, the data supply rate from main memories cannot satisfy the data request rate from the microprocessors. To satisfy it, multi-level cache hierarchy [5] is employed using a large amount of hardware resource to hide the latency between microprocessors and main memories. By shortening the latency, cache memories can increase the data supply rate. These innovations described above have contributed to the performance improvement of microprocessors.

However, in recent years, the further advance in CMOS process technology hardly contributes to the performance improvement without significant energy increase. As the integrated density of transistors increases, the power density of microprocessors also increases [6]. Hence, to realize higher performance microprocessors, a serious problem is a growth in power consumption per chip. In addition, an increase in power density causes more heat [7], and cooling has been becoming more difficult than before. Therefore, the innovations of computer architecture design will play more important roles for performance improvement and energy reduction in the future [8, 9].

Moreover, while the amount of hardware resource on a chip has been grown, improvement of resource utilization has become one of the main problems in microprocessor design. In general, a microprocessor mounts hardware resource that can be integrated by the current technology, considering trade-offs between

performance of all executed applications and hardware cost. However, when executing an application, all the resource cannot be always used. In such a case, excessively provided hardware resource on the microprocessor does not contribute to the performance of executing the application, while only consuming energy. Consequently, by improving utilization efficiency of hardware resource, it is expected that microprocessors achieve their performance improvement without a significant increase of energy consumption.

1.2 Objective of the Dissertation

To realize high performance and low energy microprocessors, this dissertation focuses on cache memories, which are one of the most important components of modern microprocessors for high performance.

Recently, it has become very difficult to reduce the memory access latency. On the other hand, the performance of microprocessors is increasing [10]. As a result, a microprocessor has to wait for a very long time to access the main memory, and the long memory latency leads to severe performance degradation [11]. To overcome this problem, the so-called memory wall problem [12], microprocessors have adopted cache memories, which are small-capacity but high-speed memories. Accordingly, cache memories are requisite for modern microprocessors to avoid the long memory access latency and thereby achieve a high performance.

Although cache memories are essential for microprocessors as mentioned above, they also have a large impact on energy consumption of microprocessors. Recently, the capacity of a cache memory has been growing and occupies a large area of a chip. In deep-submicron CMOS technology, leakage current, which is proportional to the area, is a large portion of the overall energy consumption [13]. As a result, a large part of energy is consumed by cache memories. For instance, 16% of the total power is consumed by cache memories in Alpha 21264 [14], 21% in Pentium Pro [15], and 40% in StrongARM 110 [16]. In the case of multi-level cache, a large portion of overall energy is consumed by the last-level cache memory, which is placed at the nearest position to main memories, e.g. 16.5% in second-level cache of Niagara 2 and 18% in third-level cache of 3.4GHz Xeon Tulsa [17]. From the above, it is obvious that a cache memory is one of the most important building blocks to design high performance and low

energy microprocessors.

The contributions of cache memories to the performance clearly depend on memory access characteristics of executed threads. Hence, there is room to improve the cache utilization ratio by adapting data management and hardware resource management to individual threads.

It has also been becoming difficult for a single thread to use more hardware resource in parallel, because instruction-level parallelism is limited [18]. Therefore, CMPs [19, 20] have become commonplace. By simultaneously executing multiple threads by multiple cores, CMPs can effectively use on-chip hardware resource based on thread-level parallelism [21], and provide a good trade-off between performance and energy consumption. However, in almost all CMPs, multiple threads share one cache [22], and the cache sharing causes inter-thread cache conflicts. It causes a harmful effect on efficiency of cache usage, resulting in performance degradation of CMPs. Therefore, it is strongly required to solve such a cache conflict problem.

It should be noted that there are two causes of inter-thread cache conflicts. One is *inter-thread kickouts (ITKO)* [23], in which data fetched by one thread are evicted by data of another thread. If the former thread again accesses the data that have already been replaced by the latter thread, the cache access results in a cache miss that does not occur in single thread execution. For this reason, replacement by one thread may increase the execution time of another thread. The other cause is *capacity shortage*. If cache capacities requested by threads sharing a cache are too large, the sum of requested capacities exceeds the capacity of the cache. In this case, their working sets cannot be stored in the cache because the number of ways allocated to each thread is limited. As a result, severe performance degradations may occur on simultaneously executed

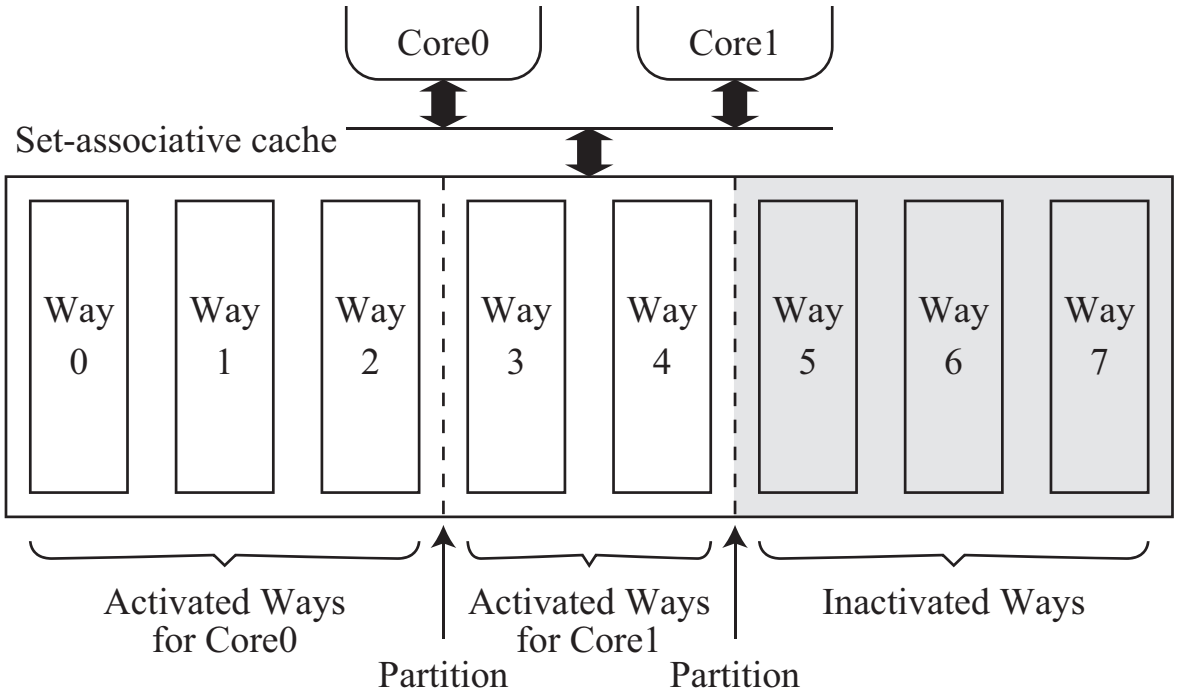


Figure 1.1: Basic concept of the power-aware dynamic cache partitioning mechanism.

threads.

Under this situation, this dissertation focuses on a dynamic cache resizing mechanism, especially *the power-aware dynamic cache partitioning mechanism* [24, 25, 26]. There are two objectives in the power-aware dynamic cache partitioning. One is the reduction of inter-thread kickouts, and the other is the reduction of energy consumption.

The basic concept of the mechanism is shown in Figure 1.1. The mechanism is applied to a set-associative cache shared by two cores. The cache is divided into three parts in a unit of way. In Figure 1.1, the first part that consists of ways 0, 1, and 2 is allocated to Core0. The second part that consists of ways 3 and 4 is allocated to Core1. The third part that consists of ways 5, 6, and 7 is not allocated. The first and second parts are exclusively allocated to Core0 and

Core1, respectively. Hence, one core cannot touch the part allocated to the other core, and hence the mechanism can avoid inter-thread cache conflicts. Moreover, the third part is disabled, and the power supply to this part is cut off by using power gating [27]. This power control mechanism contributes to the reduction of the power consumption of the cache. These parts are resized flexibly and dynamically according to the cache capacity required by each thread. As a result, the mechanism can reduce energy consumption without significant performance degradation.

Although the mechanism is effective to avoid inter-thread kickouts and reduce energy consumption, there is still room to further improve the performance. Among the two causes inter-thread cache conflicts, capacity shortage cannot be avoided because allocatable capacity to threads is limited by the number of maximum ways in the cache. Therefore, it is strongly required that the mechanism and cache memories improve the utilization efficiency of cache resource, e.g., to improve performance without capacity growth, or to keep the performance with a smaller capacity. Hence, the objective of this dissertation is to achieve the improvement of utilization efficiency of cache resource. If the objective is achieved, the power-aware dynamic cache partitioning mechanisms can avoid inter-thread cache conflicts by capacity shortage, and reduce the energy consumption without significant performance degradation.

To this end, this dissertation proposes three approaches as follows. The first approach is a monitoring scheme to avoid excessive allocation of the ways for the power-aware dynamic cache partitioning mechanism. The second approach is a cache management policy to early evict unused cache blocks to increase the capacity for reusable blocks. The third approach is a thread scheduling method to adjust the combination of threads sharing one cache.

In the first approach, this dissertation discusses a monitoring scheme of the power-aware dynamic cache partitioning mechanism. If the ways are allocated excessively by exceptional cache access behaviors, the mechanism increases the number of allocated ways to threads. As a result, the energy consumption grows without performance improvement. Hence, improvement of estimation accuracy is very important for the efficiency of the power-aware dynamic cache partitioning.

In the second approach, this dissertation discusses a cache management policy. In general, the capacity of cache memories is smaller than that of main memories. Hence, some data cannot be kept in the cache until they are used. Furthermore, some data might inherently be not reused because the application does not access them again. The blocks of these data are not reused after they are inserted in the cache. Such blocks are called *dead-on-fill blocks* [28]. If these blocks occupy a cache, the number of allocated ways increases to store both reusable blocks and dead-on-fill ones. In this way, dead-on-fill blocks disturb the energy reduction by the power-aware dynamic cache partitioning mechanism. Therefore, the reduction of dead-on-fill blocks is ignorable for the efficiency of the power-aware dynamic cache partitioning.

The third approach is a thread scheduling method that avoids capacity shortage. If threads with large working set sizes share a cache, inter-thread cache conflicts by capacity shortage is not avoidable. However, it is still possible to prevent the threads with large working sets from sharing a cache by thread scheduling. Recently, CMPs consist of many execution cores and shared cache memories. This enables an operating system or its thread scheduler to flexibly change the combination of threads share a cache. As a result, capacity shortage caused by the threads with large working set sharing the cache can be avoided.

Accordingly, these three approaches are crucial for cache memories with the power-aware dynamic cache partitioning mechanism. In this dissertation, all these three approaches are totally proposed as a hardware-software co-designed cache memory system, which can realize microprocessors with high performance and low energy consumption. The main contribution of this dissertation is to show that these three approaches in the proposed cache system can improve the utilization of cache memories.

1.3 Organization of the Dissertation

This dissertation is organized as follows. Chapter 1 describes the background and objectives of this dissertation. Chapter 1 indicates that cache memories play important roles for performance improvement and energy saving of microprocessors.

Chapter 2 proposes a voting-based working set assessment scheme for dynamic cache resizing mechanisms. This scheme is effective to improve the estimation accuracy of the working set size of a thread. By estimating the working set size more accurately, dynamic cache resizing mechanisms become more effective to improve performance and energy efficiency.

Chapter 3 proposes a capacity-efficient insertion policy for data management. The policy is designed to early evict dead-on-fill blocks that are not reused in the future. As a result, it can reduce the cache capacity used by each thread, and more cache capacity can be deactivated.

Chapter 4 proposes a thread scheduling method based on working set assessment for CMPs. In the scheduling method, threads are scheduled so that they do not share a cache memory if the total of their working set sizes exceeds the cache memory size. As a result, a larger cache capacity can be allocated to a thread with a large working set size, and it is expected that the overall system performance improves.

Finally, Chapter 5 describes concluding remarks of this dissertation.

Chapter 2

A Voting-based Working Set Assessment Scheme for Dynamic Cache Resizing Mechanisms

2.1 Introduction

To achieve high performance and energy reduction of CMPs, one promising approach is dynamic cache resizing, in which each partitioned cache region is independently managed to find an appropriate allocation of cache resources to threads. A lot of previous researches have focused on the dynamic cache resizing mechanisms.

For energy saving, Albonesi has discussed the effects of changing the number of enabled ways of a set-associative cache on performance and energy consumption [29]. Yang et al. have considered exploiting the effects of dynamic cache resizing mechanisms by using hybrid organization of both way-based and

set-based resizing mechanisms [30]. Powell et al. have proposed the DRI i-cache [31], which can reduce static energy consumption on instruction caches. The way-adaptable cache mechanism [24] demonstrates the effect of the dynamic control mechanism allowing data caches to find a good trade-off between performance and energy consumption. To adapt to fine-grain phase changes of applications, Pokam et al. have considered the cooperation between cache resizing and compiler optimization [32]. The mechanisms to save energy for the non-uniform cache architecture (NUCA) [33], which is an enhanced cache architecture, was also proposed [34].

For cache partitioning on CMPs, dynamic cache partitioning mechanisms [35, 36, 37, 38] can exclusively allocate ways to each thread and avoid inter-thread kickouts. Srikantaiah et al. have proposed the adaptive set pinning. This proposal is also a cache partitioning mechanism, which exclusively allocates some of cache sets to each thread [39]. Lee et al. have proposed the novel address mapping scheme to achieve dynamic set-based cache partitioning [40]. Chang et al. have introduced the multiple time-sharing partitions, in which cache resources are allocated from the viewpoint of both capacity and occupied time [41]. Some cache partitioning strategies for NUCA have also been proposed by Dybdahl et al. [42] and Huh et al. [43]. Furthermore, to achieve both energy saving and cache partitioning on CMPs, the power-aware dynamic cache partitioning mechanism [26] was proposed.

In this chapter, estimation accuracy of cache capacity necessary for threads is discussed. Accurate estimation is very important for the dynamic cache resizing mechanisms to improve energy efficiency. If the estimated capacity of a thread is too small against the working set size, a core executing the thread cannot extract its potential performance. On the other hand, if the estimated capacity

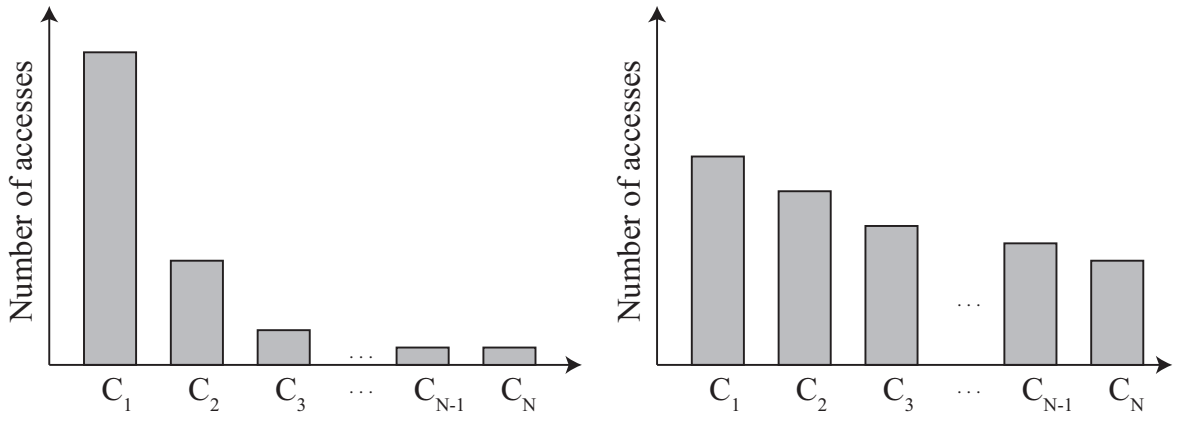
is too large, the mechanisms cannot allocate ways to other threads that require more ways, or energy consumption may be increased without any performance gain due to underutilized ways.

This chapter assumes the power-aware dynamic cache partitioning mechanism that estimates working set sizes of threads by sampling their cache accesses in a certain period. This mechanism conducts locality assessment to estimate the working set sizes using the sampled accesses. However, observation of cache accesses behaviors indicates that there are some exceptional behaviors not following the trend of the overall accesses. Such behaviors might influence the locality assessment and result in an increase in the number of allocated ways, even though this usually degrades energy efficiency. Some other approaches to estimate working set sizes of threads do not consider the exceptional behaviors [44, 45, 46].

To avoid the increase of energy consumption by exceptional behaviors, this chapter proposes a new scheme for working set assessment called a voting-based working set assessment scheme. The conventional cache partitioning mechanisms conduct locality assessment based on the sampling data of one long sampling period, which is also used as the time length of holding the number of ways. However, in the proposed scheme, a locality assessment result of one sampling period is decided based on majority voting among locality assessment results of several short sampling periods, called *voting periods*. Dividing one sampling period to several voting periods enables the mechanism to identify the periods including exceptional behaviors, and to ignore the locality assessment results of such periods by adopting the results of majority voting. Applying the proposed scheme to the power-aware dynamic cache partitioning mechanism, the number of ways allocated to each thread and the energy consumption of the

cache memory are reduced without significant performance degradation. Consequently, this will improve energy efficiency of CMPs with large shared cache memories.

The rest of this chapter is organized as follows. In Section 2.2, the power-aware dynamic cache partitioning mechanism considered in this dissertation is described, and the impact of exceptional behaviors is discussed. Section 2.3 proposes a new scheme for the power-aware dynamic cache partitioning mechanism based on the discussions in Section 2.2. In Section 2.4, the proposed scheme is evaluated in terms of performance and energy consumption. Finally, Section 2.5 concludes this chapter.



(a) High-locality Model (b) Low-locality Model

Figure 2.1: Stack distance profiling.

2.2 The Power-Aware Dynamic Cache Partitioning Mechanism

This section firstly introduces a locality assessment metric, which is important for judging whether the number of allocated ways is appropriate or not. Second, the function of deciding the number of ways based on the locality assessment metric is described. Finally, the effect of exceptional cache access behavior is pointed out.

2.2.1 Metric for Locality Assessment

To estimate the number of ways required by each thread, a metric to judge whether the thread requires more ways or not is essential. For this purpose, the power-aware cache partitioning mechanism carries out locality assessment using *stack distance profiling* [47, 48, 49].

Figure 2.1 shows two examples of the results of stack distance profiling. Let $C_1, C_2, ..., C_N$ be N counters for an N-way set-associative cache with the LRU replacement policy. C_i counts the number of accesses to the i-th line in the LRU

stack. Therefore, counters C_1 and C_N are used to count the numbers of accesses to MRU lines and that to LRU lines, respectively. If a thread has a high locality, its cache accesses are concentrated on the MRU and its nearby lines as shown in Figure 2.1(a). However, if a thread has a low locality, its accesses are widely distributed from MRU to LRU as shown in Figure 2.1(b). From the above, it is that the ratio, which is defined as D in the following equation, represents the cache access locality of a thread.

$$D = \frac{C_1}{C_N}. \tag{2.1}$$

If a thread has a high locality, D of the thread becomes small. On the other hand, if it has a low locality, D becomes large.

2.2.2 The Control Scheme of the Power-Aware Dynamic Cache Partitioning Mechanism

To grasp the number of ways requested by each thread, the mechanism samples cache accesses in a certain period, called a *sampling period*. After the sampling period, the mechanism has an opportunity to decide to change the number of allocated ways to each thread. This opportunity is called an *adaptation opportunity*. To adjust the number of ways to dynamic phase changes due to cache access characteristics of each thread, the sampling period and the adaptation opportunity are alternately repeated.

At an adaptation opportunity, the mechanism decides whether the number of allocated ways to each thread should be changed. For this purpose, the mechanism uses three functions: *the local observation function*, *the global observation function*, and *the way-allocation function*.

The Local Observation Function

This function judges whether each thread requires more ways or not by locality assessment at adaptation opportunities. The function uses D and thresholds t_1 and t_2 ($t_1 < t_2$). If $D < t_1$, the function sends a signal *dec* as a locality assessment result to the global observation function to suggest deallocating one way from the thread and inactivating it. On the other hand, if $t_2 < D$, the function makes a signal *inc* to suggest allocating one way to the thread and activating it. If $t_1 < D < t_2$, the function generates a signal *keep* to suggest keeping the current number of ways.

The Global Observation Function

This function actually decides whether to resize the cache based on both the current signal and the signals at the past adaptation opportunities from the local observation function. In the case that the signal changes at every adaptation opportunity, performance degradation occurs due to a lot of writeback. To avoid this, the global observation function avoids frequent transitions between activation and inactivation of the ways at every adaptation opportunity.

To avoid frequent inactivation, the function employs an asymmetric state machine as shown in Figure 2.2. The state machine changes its state when *inc* or *dec* is input from the local observation function. When *inc* is given to the state machine, it outputs a cache up-sizing control signal *INC* and then always transits to State “000” from any state. However, in the case of *dec* given, the machine works conservatively to generate a down-sizing signal *DEC* and transit to State “111”.

As a result, if the function receives *inc*, it decides to allocate an additional

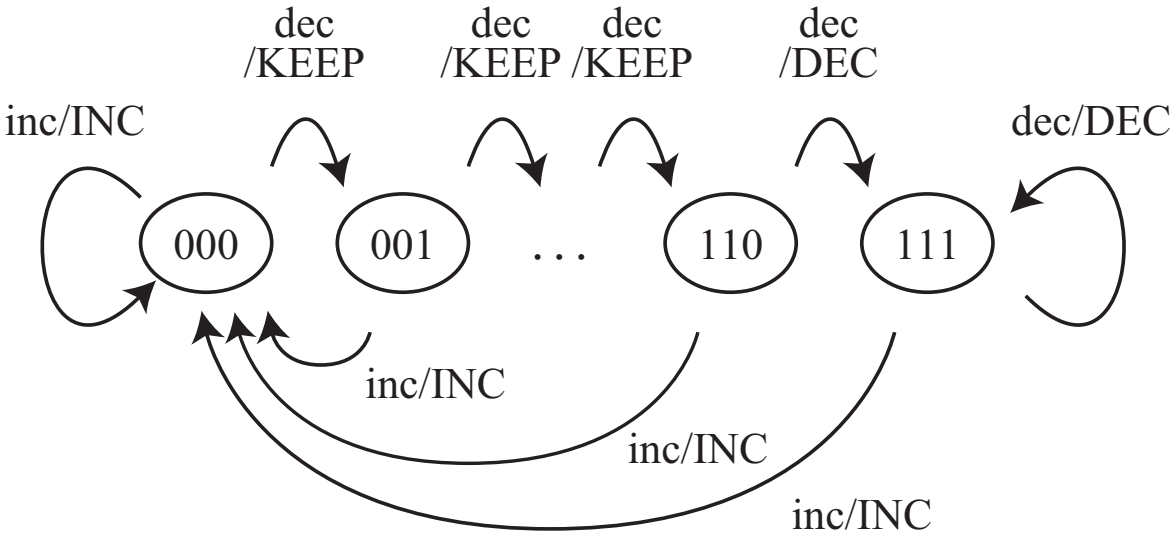


Figure 2.2: 3-bit state machine to decide the way-adaptation.

way to the thread immediately to avoid performance degradation due to insensitivity to *inc*. However, to inactivate one way, the function requires continual inputs of *dec* since the past adaptation opportunities to avoid performance degradation due to a lot of write back.

The Way-Allocation Function

This function considers changing allocation of ways to threads when the mechanism activates all ways and the threads require more ways. For example, if two threads share a cache and one thread has a larger working set than the other, the former thread should take more cache resources.

The function assumes that the number of ways required by each thread is proportional to the degree of their locality. After calculating D_i from the cache access samples of the i-th thread, the following inequality is used to determine

whether the number of ways allocated to threads should be increased or decreased.

$$\begin{cases} Alloc_j \mathrel{+}= 1 \\ Alloc_k \mathrel{-}= 1, \end{cases} \tag{2.2}$$

where

$$\begin{aligned} Alloc_i &= (\text{the number of allocated way to the } i\text{-th} \\ &\quad \text{threads}), \\ j &= \arg\max_{0 \le i < n} D_i, \\ k &= \arg\min_{0 \le i < n} D_i, \\ n &= (\text{the number of threads}), \end{aligned}$$

and satisfies the following conditions.

$$Alloc_{all} = \sum_{i=0}^{n-1} Alloc_i, \tag{2.3}$$

where $Alloc_{all}$ denotes the cache associativity. If $D_j = D_k$, allocation of ways is not changed.

2.2.3 Exceptional Cache Access Behaviors and their Effect on the Mechanism

Observing cache access behaviors of threads, a certain number of LRU accesses sometimes happen in a very short period. In many cases, these accesses do not follow the overall trend of cache access behaviors. Hence, these accesses are considered exceptional behaviors of cache accesses. If sampled accesses including

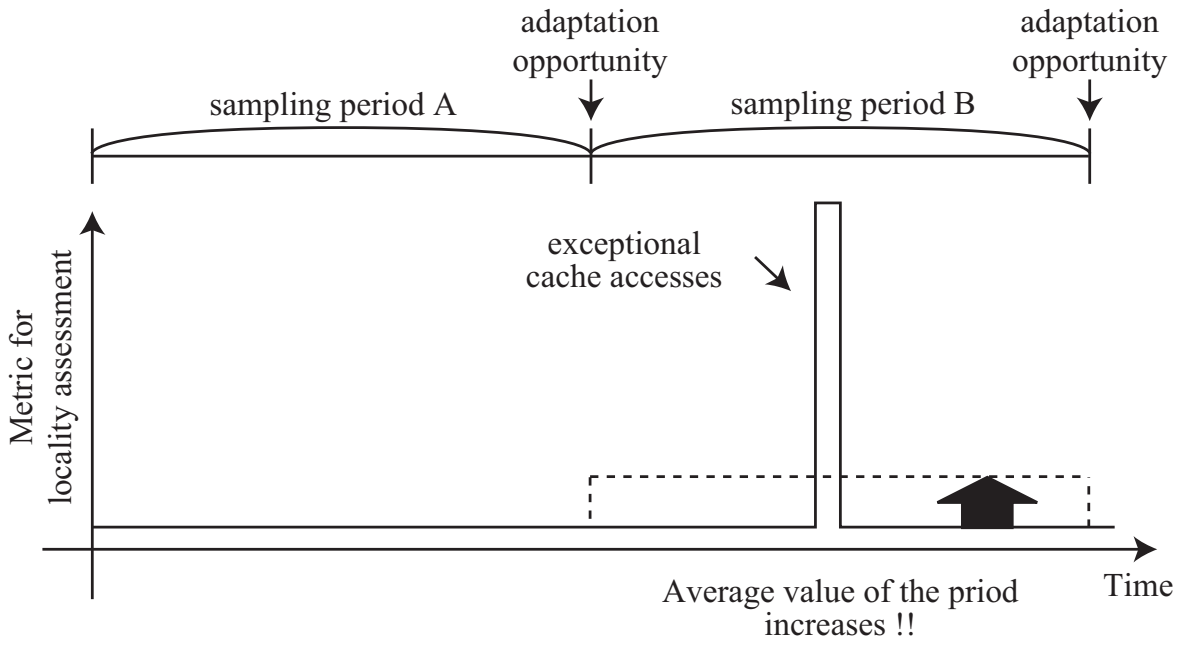


Figure 2.3: Concept of exceptional behaviors of cache accesses and its effect on locality assessment.

exceptional behaviors are used for locality assessment, the mechanism causes the excessive activation of the ways. As a result, the mechanism loses a chance to reduce energy consumption.

Figure 2.3 shows examples of two sampling periods to describe the effect of exceptional behaviors. In the figure, sampling period *A* does not include exceptional behaviors. On the other hand, sampling period *B* includes such behaviors. In this situation, comparing the average locality assessment metric D of the period A with that of the period B, the former is obviously larger than the latter. This difference between the periods A and B causes different locality assessment results while the overall trends of two periods are almost the same. Thus, such an exceptional behavior in period B may cause a misjudgment and increase the number of activated ways, resulting in degradation of energy efficiency.

The reason of energy efficiency degradation is that the newly activated way

by the exceptional behaviors may not be effectively used. At an adaptation opportunity, if the behaviors are observed temporarily in the previous sampling period and do not appear in the next period, the activated way does not contribute to performance improvement.

Moreover, even if exceptional behaviors iteratively occur, performance does not always improve by adapting to the behaviors. When the mechanism adapts the cache to such behaviors and activates a new way, some cache accesses may hit. However, the number of accesses saved by this adaptation is smaller than that by exceptional behaviors because, in general, the number of accesses to LRU lines gradually decreases as the number of ways increases. Hence performance impact of the saved accesses is small.

In these cases, exceptional behaviors cause the increase in the number of activated ways, which results in the growth in energy consumption. However, the performance improvement is not significant. Hence, energy efficiency of the cache mechanism degrades. To avoid degradation of energy efficiency in such cases, the mechanism should be insensitive to exceptional behaviors.

One possible solution is to increase the length of the sampling periods. If the interval becomes long, average D of the sampling period B in Figure 2.3 decreases. However, there are two drawbacks of using a long sampling period. First, if the sampling period becomes long, the probability of including multiple exceptional behaviors in one sampling period increases. In this case, the usage of a long period cannot reduce average D. Second, once the number of ways is increased by exceptional behaviors, it also takes a long time to reduce the number of activated ways because of the long sampling period.

In addition, it becomes further difficult for the mechanism to immediately inactivate ways because of the effect of the global observation function. The mechanism cannot help using the global observation function to avoid frequent transitions between activation and inactivation of the ways. However, because several sampling periods are needed to inactivate the ways after activating them, the global observation function also increases energy consumption if excessively adapting the cache to exceptional behaviors. Moreover, it is not preferable that the global observation function avoids an immediate increase in the number of ways. If the global observation function always be less sensitive to *inc* signal, e.g., continual inputs of *inc* in several periods are needed to actually increase the number of ways as well as *dec*, performance degradation becomes large by ignoring the *inc* signal that are not affected by exceptional behaviors. Consequently, a new technique is required to negate only the effect of exceptional behaviors, without changing the length of the sampling periods and the global observation function, resulting in a reduction in the number of activated ways.

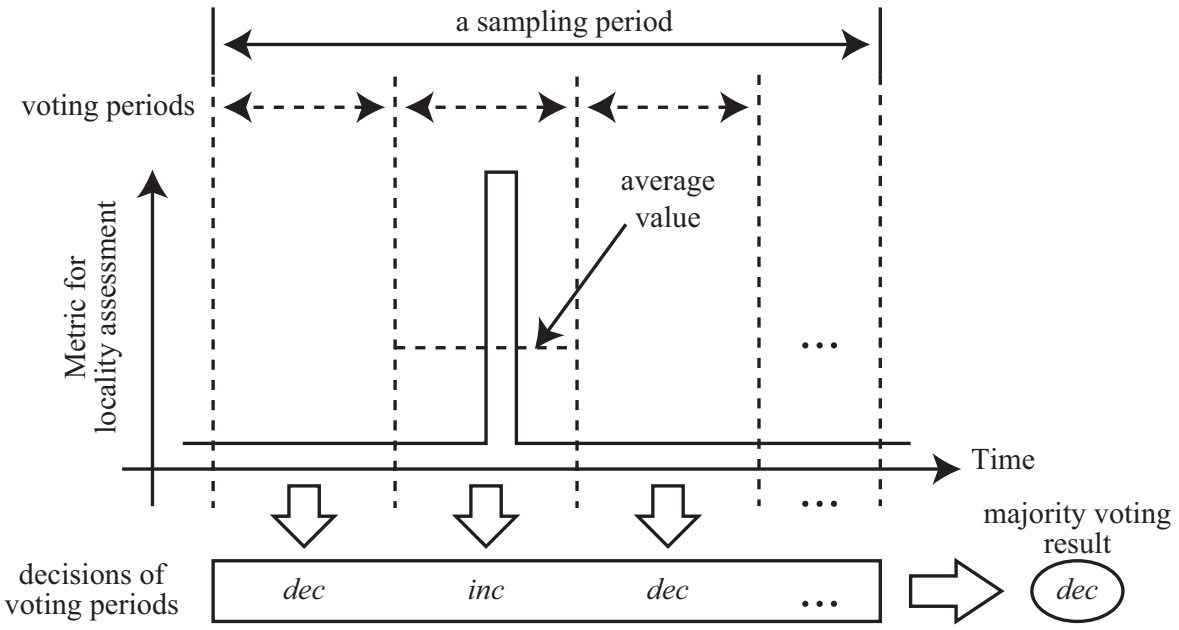


Figure 2.4: Way to reduce the effects of exceptional behaviors by the proposed scheme.

2.3 A Voting-Based Working Set Assessment Scheme

To reduce the effect of exceptional behaviors, a voting-based working set assessment scheme is proposed in this section. The scheme consists of two main functions. One is a voting-based local observation function. The other is a voting-based way-allocation function.

2.3.1 A Voting-Based Local Observation Function

To properly assess cache access locality, the voting-based local observation function identifies exceptional behaviors and avoids excessive adaptation to them.

First, to identify exceptional behaviors, the proposed scheme divides a sampling period into several short sub-periods, which are called *voting periods*. Figure 2.4 illustrates the relationship between a sampling period including an exceptional behavior and voting periods. In the figure, the second voting period

includes an exceptional behavior, and thereby the locality assessment of the voting period results in *inc*. However, the other voting periods in the figure do not include exceptional behaviors. Among these voting periods, only the second period makes the different decision. Hence, the scheme can detect that the second voting period contains an exceptional behavior.

Second, the scheme makes a final decision based on a series of locality assessment results that are generated by voting periods. To reduce the impact of exceptional behaviors on the final judgment, the scheme should consider only locality assessment results of voting periods that do not include any exceptional behavior. To realize such a control, the proposed scheme uses majority voting among the locality assessment results, which are made by the voting periods between two consecutive adaptation opportunities. If some voting periods generate different locality assessment results against the overall trend, the scheme can figure out that these voting periods include exceptional behaviors and therefore ignore their results by majority voting. Moreover, if almost all voting periods have the same locality assessment results, the scheme can decide that cache access behavior is stable, and hence the scheme decides to adjust the number of ways based on the results. In the case of Figure 2.4, the proposed scheme can reject the locality assessment result of the second voting period by majority voting because only this period includes the exceptional behavior and outputs the different results.

The local observation function of the proposed scheme works as follows. At the end of every voting period, the result of locality assessment of a thread in the period, i.e. *inc*, *dec*, or *keep* is decided by Eq. (2.1) and two thresholds (t_1, t_2). If an adaptation opportunity comes during a voting period, the voting period is immediately terminated and the locality assessment of the period is performed

using the statistics information collected until then. At the adaptation opportunity, if the number of votes to *dec* is equal to that to *inc*, or if that to *keep* is the largest, the function outputs *keep*. Otherwise, the function outputs the signal associated with the most votes.

In the proposed scheme, the length of the voting periods is an important parameter to appropriately assess the cache access locality. The length should be as short as possible to isolate only exceptional behaviors from the others and to obtain more votes at the adaptation opportunity. However, too short voting periods cannot include a sufficient number of cache accesses to statistically capture the locality. The proposed scheme uses an *access-based interval* [26] to decide the length of the voting periods, in which the end of the periods comes after a certain number of cache accesses. By using this interval, the proposed scheme can ensure that a certain number of sampled accesses are guaranteed in one voting period. In addition, an adaptation opportunity comes at a fixed *time-based interval* [37] to make the adaptation frequency moderate.

2.3.2 A Voting-Based Way-Allocation Function

Suppose that all ways are activated and already allocated to the threads. Then, if a thread with a smaller working set size has more ways than other threads, some of ways should be reallocated to the latter threads. For reallocation, the proposed scheme again uses the majority voting results for the way-allocation function.

The number of votes to *inc* means how strongly a thread demands a large cache capacity. If the number of votes to *inc* of one thread is larger than that of other thread, a larger cache capacity should be allocated to the former thread. However, simple comparison of the numbers of *inc* votes among threads is unfair

for threads with fewer ways, because the number of votes in one sampling period is proportional to the number of accesses, which increases with the number of allocated ways. Hence, the function must consider the number of allocated ways. Due to this reason, *weighted votes* (wv) is defined, which is the number of votes to *inc* divided by the number of allocated ways, and the scheme decides the allocation of ways to the i-th threads as follows.

$$\begin{cases} Alloc_j \mathrel{+}= 1 \\ Alloc_k \mathrel{-}= 1, \end{cases} \tag{2.4}$$

where

$$\begin{aligned} Alloc_i &= \text{(the number of allocated ways to the } i\text{-th thread)}, \\ j &= \arg\max_{0 \le i < n} wv_i, \\ k &= \arg\min_{0 \le i < n} wv_i, \\ wv_i &= inc_i \ / \ Alloc_i, \\ inc_i &= \text{(the number of votes to } \textit{inc} \text{ of the } i\text{-th thread)}. \end{aligned}$$

Using the above equations, the function can consider not only the number of votes but also the number of allocated ways to achieve fair comparison between two different threads.

Table 2.1: Parameters of the simulated architecture for the voting-based scheme.

Core
8-issue out-of-order, 2GHz, 32nm technology
Memory
L1 I-Cache: 32kB, 4-way, 64B-line, 1 cycle latency
L1 D-Cache: 32kB, 4-way, 64B-line, 1 cycle latency
L2 Cache: 2MB, 32-way, 64B-line, 14 cycle latency
Main Memory: 200 cycle latency

2.4 Evaluations

2.4.1 Experimental Setup

A simulator including the proposed scheme has been developed based on the M5 Simulator System [50] and CACTI 6.5 [51, 52, 53]. Table 2.1 shows the simulation parameters of a modeled processor and memory hierarchy. The proposed scheme with the power-aware dynamic cache partitioning mechanism is applied to the L2 cache, which is a 2MB, 32-way set-associative cache in our evaluation.

In the power-aware dynamic cache partitioning mechanism, thresholds (t_1, t_2) are required as mentioned in Section 2.2.2. According to the previous work [26], $(t_1, t_2) = (0.001, 0.005)$ is a fine-tuned parameter set to maintain a certain performance, and these values are used. Benchmarks examined on the simulator are selected from the SPEC CPU2006 benchmark suite [54]. Each simulation is done by executing first one billion cycles of the simulated processor.

2.4.2 Evaluation Results of a Single-Core Processor

This section shows evaluation results in the cases of executing a single thread on a single-core processor, to clarify the effect of the proposed scheme. Before

Table 2.2: Classification of the benchmarks by DVR.

Class	DVR	Benchmarks
I	≥ 0.25	soplex, bwaves, tonto, h264ref, omnetpp, gobmk, astar
II	< 0.25	libquantum, GemsFDTD, milc, gamess, lbm, calculix, wrf
III	$= 0$	zeusmp, bzip2

showing the results, the benchmarks are classified into three classes based on their characteristics observed from the experimental results.

For classification of the benchmarks, *Dissenting Vote Rate (DVR)* is defined as a metric. DVR is a ratio of the number of dissenting votes to the number of all votes. For example, when the voting-based local observation function judges that the number of ways should be decreased, the votes to *inc* and *keep* in this sampling period are dissenting votes. Since it is considered that the voting periods against the final judge of the sampling period include exceptional behaviors, an increase in DVR indicates an increase in the occurrence frequency of exceptional behaviors.

Table 2.2 shows the classes of the benchmarks. A benchmark whose DVR is larger than 0.25 is classified into Class I. The benchmarks in Class I are supposed that they include more exceptional behaviors than the other benchmarks because DVR of the benchmarks in this class is larger than those of the other benchmarks. A benchmark whose DVR is smaller than 0.25 is classified into Class II. The benchmarks in this class include some exceptional behaviors, but its frequency is lower than that of benchmarks in Class I. Finally, a benchmark whose DVR is zero is classified into Class III. One reason that their DVRs become zero is that they have a few accesses to the L2 cache and hence exceptional behaviors do not occur. Another reason is that their voting periods

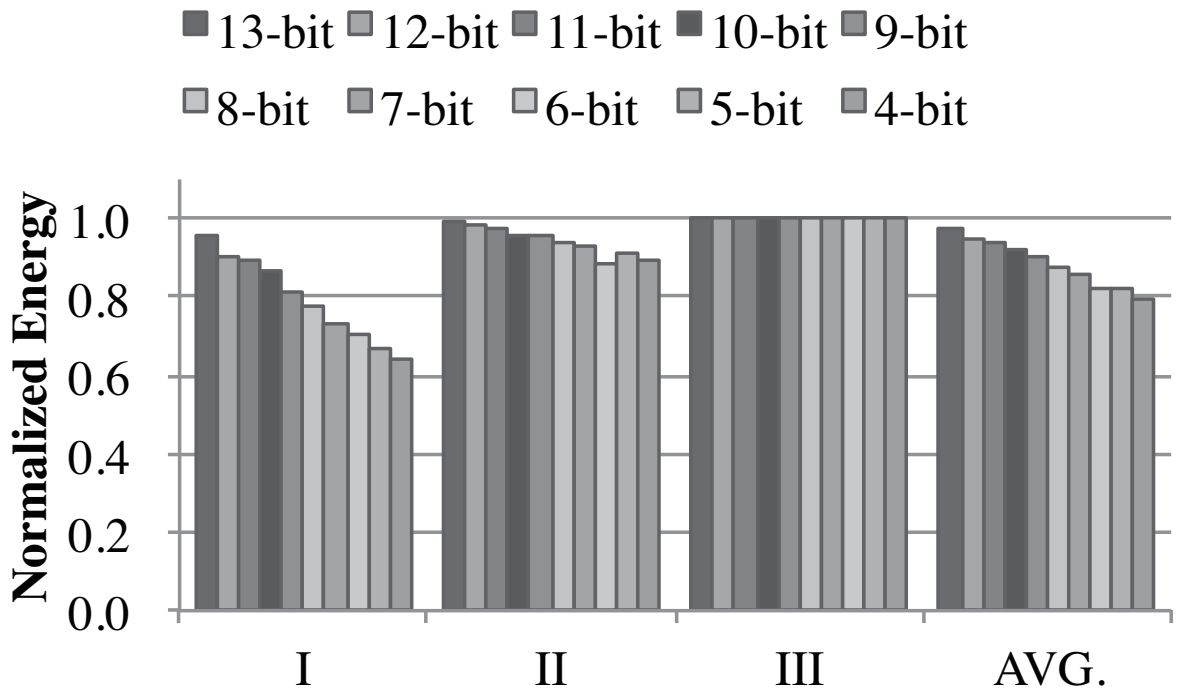


Figure 2.5: Energy consumptions in the case of a single-core processor.

become longer than sampling periods because of very few accesses. Under this situation, dissenting votes cannot occur because voting is performed only once in a sampling period.

Figure 2.5 shows the energy consumptions of the L2 cache, when its power-aware dynamic cache partitioning mechanism is controlled by the proposed voting-based working set assessment scheme. In the figure, "n-bit" indicates the voting-based control scheme with an n-bit saturating counter that decides the length of a voting period. When the counter of each thread is saturated, the scheme performs locality assessment for the thread and votes the result of the voting period. The horizontal axis indicates the class of benchmarks, and the vertical axis means the energy consumption normalized by that in the case of "without the voting scheme." From this figure, it is demonstrated that the voting-based partitioned cache can reduce the energy consumption for the benchmarks in

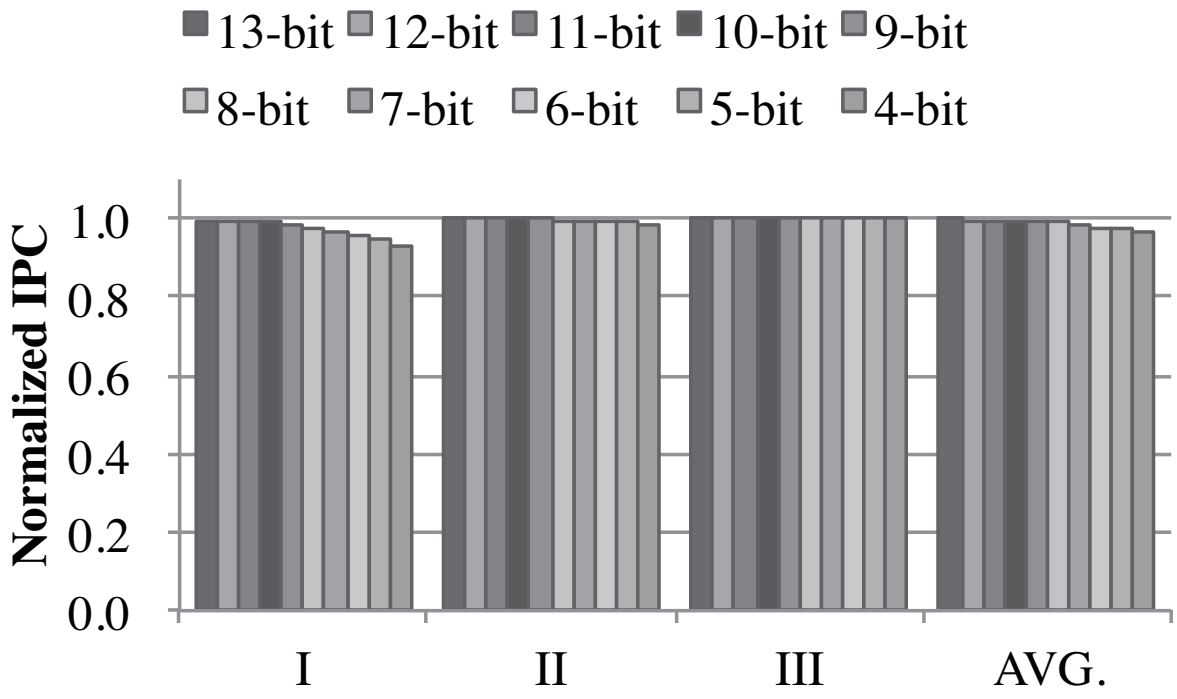


Figure 2.6: Performances in the case of a single-core processor.

Classes I and II. This indicates that the superiority of the voting-based scheme over the without-voting one becomes significant if cache accesses of the benchmarks in Classes I and II often include exceptional behaviors. In this figure, the energy consumption monotonically decreases with the number of counter bits. This is because that the short voting periods help the voting-based scheme to finely distinguish the periods including exceptional behaviors from the other periods. As a result, the voting-based partitioned cache can find a lot of chances to reduce the number of allocated ways.

Figure 2.6 shows performances by the voting-based partitioned cache. In this figure, the horizontal axis indicates the class of benchmark. The vertical axis indicates the relative IPC, which is the IPC normalized by that of the without-voting partitioned cache. This figure shows that performance degradations are less than 1% on an average. Therefore, the voting-based scheme does not give a

harmful effect on performances.

Figure 2.6 also shows that the performances of the benchmarks in Class I slightly degrade. However, the performance degradation can be reduced if the number of counter bits is large, hence the negative performance impact of the proposed scheme is small enough. In addition, for the benchmarks in Class II, the voting-based partitioned cache can attain the same performance as the without-voting one. This is because their DVRs are low and hence they do not often cause exceptional behaviors. Therefore, these two caches similarly adjust the allocation, resulting in the lesser performance degradation of Class II than that of Class I. Finally, for every benchmark in Class III, the voting-based partitioned cache can achieve the same performance as the without-voting one. As their DVRs are zero and exceptional behaviors do not occur, there is no difference between their cache control results.

Moreover, Figures 2.5 and 2.6 show that the voting-based scheme can improve energy efficiency. Even though energy consumptions are significantly reduced, performance degradations are moderate. For example, the voting-based partitioned cache with a 13-bit saturating counter can achieve a 2.8% reduction in energy consumption while a 0.2% performance degradation on an average. The number of counter bits can be also adjusted so as to make the cache more low-energy-oriented. The cache with an 8-bit saturating counter can achieve a 15% reduction in energy consumption while a 1% performance degradation on an average. Even in the worst performance case of *soplex*, the scheme can reduce energy consumption by 35%, while a 7% performance degradation. These results indicate that the proposed scheme can improve energy efficiency of the cache with the power-aware dynamic cache partitioning by ignoring exceptional

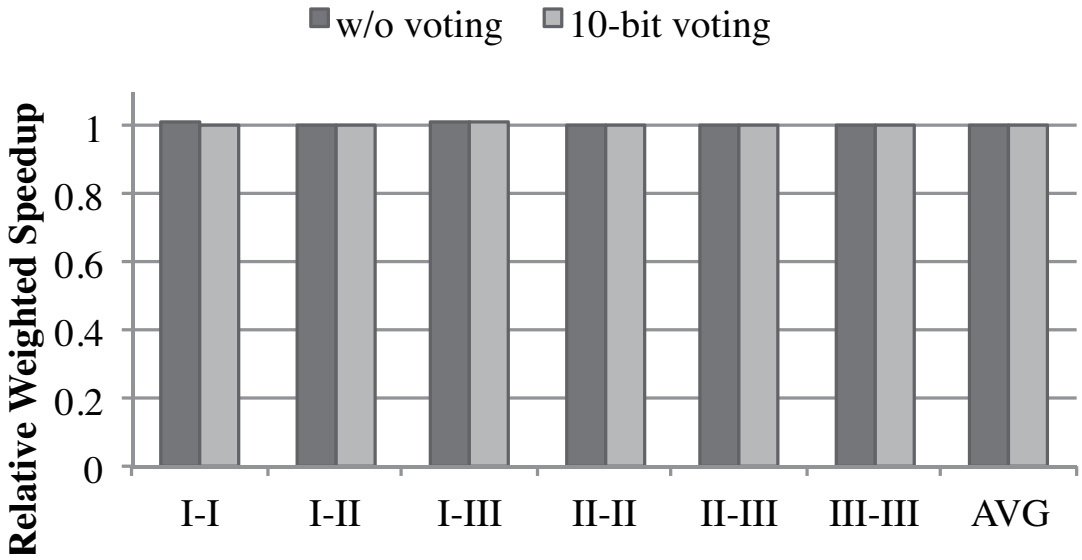


Figure 2.7: Performances in the case of multi-thread execution on the 2-core CMP.

behaviors. In the case of lower bit saturating counters, from 7-bit to 4-bit in Figures 2.5 and 2.6, the performance and the energy monotonically decrease with the number of counter bits. These results suggest that the number of counter bits is not very small to avoid significant performance degradation by the proposed scheme.

2.4.3 Evaluation Results of a 2-Core CMP

This section shows the evaluation results of the voting-based partitioned cache in the case of multi-thread execution on a 2-core CMP, by the performance and the energy consumption. Considering the number of benchmarks in Table 2.2, 120 combinations of two threads are generated. In this way, these combinations are grouped into five combination classes: I-I, I-II, I-III, II-II, II-III, and III-III, each of which denotes classes of two benchmarks. Hereafter, the results are shown by an average value of each combination class.

A performance metric used to evaluate overall performance of CMPs is Weighted Speedup [55], which is defined as follows.

$$(WeightedSpeedup) = \sum_{i=0}^{N-1} \frac{MultiIPC_i}{SingleIPC_i}. \tag{2.5}$$

Here, N is the number of threads, $SingleIPC_i$ is the original performance of the i-th thread in the case of single-thread execution. $MultiIPC_i$ is the performance of the i-th thread in the case of multi-thread execution. The performances of both $SingleIPC_i$ and $MultiIPC_i$ are measured in IPC. Since the performance is normalized by IPC of single-thread execution, the metric becomes fair for low IPC threads, unlike throughput that is simply the sum of IPC of all threads. The performances are normalized by those without the power-aware dynamic cache partitioning mechanisms.

In the following evaluation, 10-bit saturating counters for the proposed scheme are used. This is because the voting-based partitioned cache can achieve 99% performance on an average compared with the conventional cache, in which a thread can use the entire capacity, and performance degradation is moderate. In addition, the results are normalized by those of the conventional cache, to compare the cache with the voting-based partitioned cache with not only the without-voting partitioned one but also the conventional one.

Figure 2.7 shows the performance of each combination class. The horizontal axis shows the combination class name. The vertical axis shows relative weighted speedup, which is normalized by that of the conventional cache without partitioning. In the figure, “w/o voting”, and “10-bit voting” mean the without-voting partitioned cache, and the voting-based one with the 10-bit counters, respectively. Figure 2.7 shows that the performances of the voting-based partitioned cache are lower than those of the without-voting one. However, their

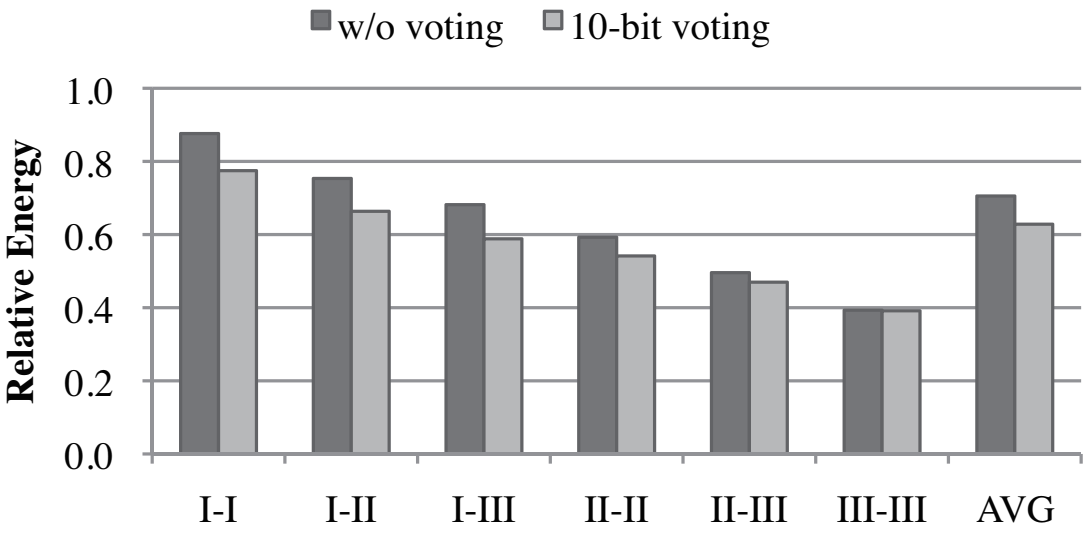
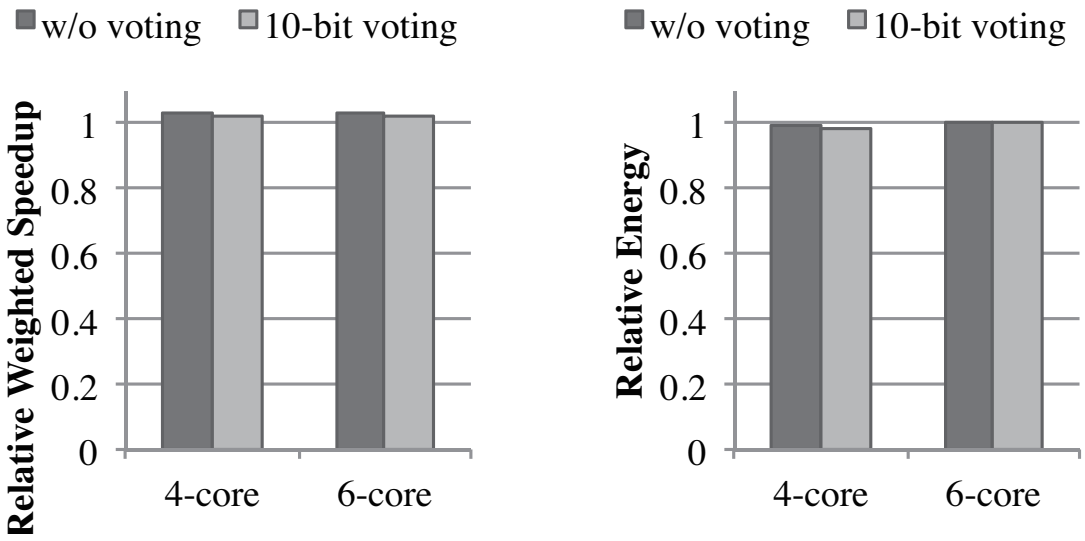


Figure 2.8: Energy consumptions in the case of multi-thread execution on the 2-core CMP.

performance difference is about 1% and quite small as discussed in the previous section. Moreover the voting-based partitioned cache can improve the performance compared with the conventional cache. Hence, the voting-based partitioned cache still maintains the performance improvements by the effect of cache partitioning, avoiding inter-thread kickouts.

In terms of energy efficiency, the voting-based scheme is superior to the without-voting scheme. Figure 2.8 shows the average energy consumption for executing thread combinations in each combination class. Figures 2.7 and 2.8 suggest that the voting-based partitioned cache can reduce the energy consumption by about 10% on an average without severe performance degradations; the performance degradation is 0.4% on an average in this evaluation. In all the experimented benchmark combinations, the maximum energy reduction is 24%



(a) Performances

(b) Energy consumptions

Figure 2.9: Performances and energy consumptions in the case of the 4-core and 6-core CMPs.

while the maximum performance degradation is 2%. Accordingly, the voting-based scheme can improve the energy efficiency of the cache with the power-aware dynamic cache partitioning mechanism.

2.4.4 Evaluation Results of 4-Core and 6-Core CMPs

To confirm the potential of the proposed scheme in many-core environments, performance evaluations using 4-core and 6-core CMPs are carried out. In the experiments, the benchmarks in Class I is only used since they have large impacts on the voting-based partitioned cache as observed in the previous sections. Figure 2.9 shows the average weighted speedups and the energy consumptions by 4-core and 6-core CMPs in each scheme. The result of the 4-core CMP shows the average of the results of 35 combinations, each of which consists of four benchmarks. Likewise, the evaluation of the 6-core CMP shows the average of the results of seven combinations, each of which consists of six benchmarks.

Although the energy consumptions of the voting-based partitioned cache are reduced in the 2-core CMP, the performances and the energy consumptions of the voting-based partitioned cache are comparable with that of the without-voting partitioned one in both the 4-core CMP and the 6-core CMP. Observing partitioning behaviors, inactivation of the ways hardly occurs in both the partitioned caches because a 2MB cache is too small to be shared by the 4 cores or 6 cores. This means that the threads require more ways while there is no inactivated way, and hence the energy consumptions are not reduced in the 4-core and the 6-core CMPs. Therefore, to achieve energy reduction in many-core environments by the proposed scheme, it is necessary that cache capacity and associativity are more larger than those of the experimented environments, respectively.

2.5 Conclusions

The power-aware dynamic cache partitioning mechanisms are promising to realize energy-efficient computing on multi-core processors. In these mechanisms, it is important to accurately estimate the number of ways required by a thread, because inaccurate resizing of the cache degrades energy efficiency. This chapter has discussed exceptional behaviors of cache accesses and their effects on the locality assessment results to be used to estimate the number of required ways. To reduce the negative effects, this chapter proposed a voting-based working set assessment scheme that decides the number of activated ways based on majority voting of the results in several short periods.

By using the proposed scheme for the power-aware dynamic cache partitioning mechanism, the power consumption is decreased by up to 24%, and 10% on an average without significant performance degradation in the case of the 2-core CMP. These results suggest that the power-aware dynamic cache partitioning mechanism with the proposed scheme can make the shared cache more energy-efficient than that with the conventional scheme. In addition, the proposed scheme can still improve energy efficiency in the cases of CMPs with four and six cores. Moreover, the results also show that the proposed scheme can find a better trade-off between performance and energy by adjusting the length of a voting period. In the case of an appropriate voting period length, the performance degradation induced by the proposed scheme becomes quite small.

Chapter 3

A Capacity-efficient Insertion Policy for On-Cache Data Management

3.1 Introduction

One important technical issue of the power-aware dynamic cache partitioning mechanism is the existence of *dead-on-fill blocks* [28], which are blocks occupied by the data that are not reused after being stored in the cache. In recent years, various applications have become more complex and its data set becomes larger. As a result, some applications use most data only once, others take a long time to reuse data. If such data are once inserted in the cache, they become dead-on-fill blocks and are evicted from the cache without being reused. In the case of a lot of dead-on-fill blocks, the power-aware dynamic cache partitioning mechanism would allocate an enough number of ways to the thread to store both reusable blocks and dead-on-fill ones, even though dead-on-fill blocks do not contribute

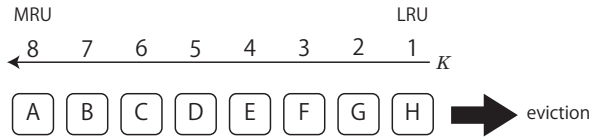


Figure 3.1: Concept of the LRU chain (focusing on a single set of an 8-way set-associative cache).

to performance improvement but consume energy. Accordingly, there is room to reduce the number of allocated ways to the threads by reducing the number of dead-on-fill blocks in the cache.

In addition, dead-on-fill blocks need a long time to be evicted if the Least Recently Used (LRU) replacement policy is used for cache data management. The LRU replacement policy is commonly used as a data management policy of cache memories because data management based on the temporal locality of reference have a positive impact on performance for almost all applications. Based on the temporal locality, the policy assumes MRU (Most Recently Used) data have the highest probability to be reused. To show the management of the LRU replacement policy in this chapter, the resident priority of each block is indicated using the LRU chain [56], which is a structure to manage the resident priority of each block. Figure 3.1 depicts the LRU chain of a set in an 8-way set-associative cache. The left-most side of the LRU chain (K=8) is the highest resident priority position, i.e., the MRU position. The right-most side of the LRU chain (K=1) is the lowest resident priority position, the LRU position. In the LRU replacement policy, if a newly coming block is inserted into the LRU chain, the block is placed at the MRU position. At the same time, the block at the LRU position is evicted and is written back to the lower-level in the memory hierarchy. The resident priorities of the other blocks decrease; the block that was placed at n-th LRU position ($K = n$) moves to the $(n-1)$-th position ($K = n-1$),

where $2 \leq n \leq 8$. However, such a priority management method does not care about dead-on-fill blocks. Every new block is placed at the MRU position. This means that the highest resident priority is also given to dead-on-fill blocks as well as reusable ones. Once dead-on-fill blocks are placed at the MRU position, they require a long time to be placed at the LRU position, and to be evicted.

Especially, the power-aware dynamic cache partitioning mechanism is likely to suffer from dead-on-fill blocks. In general, the mechanism is applied to the last-level cache memory (LLC), which is the nearest cache to the main memory. This is because LLC has the largest impact on energy and inter-thread cache conflicts. In recent years, capacity and associativity are growing in LLC, and the growth of associativity increases the length of the LRU chain. If the length becomes longer, the eviction of dead-on-fill blocks requires a longer time than the case of the lower associativity cache. As a result, the number of dead-on-fill blocks in the cache increases. If dead-on-fill blocks are evicted as early as possible, the power-aware dynamic cache partitioning mechanism can become more efficient.

This chapter proposes a new cache insertion policy named dynamic LRU-K insertion policy to evict dead-on-fill blocks earlier. While the conventional LRU policy places a newly coming block at the MRU position that is the highest resident priority position, the proposed policy inserts a new block into the K-th LRU position that is the K-th lowest resident priority position in an n-way set associative cache ($1 \leq K \leq n$). Furthermore, the policy evaluates the reusability of cache blocks and dynamically determines K during execution. As a result, the proposed policy can evict dead-on-fill blocks earlier than the LRU policy, while keeping reusable blocks. If the number of dead-on-fill blocks in the cache is reduced, the power-aware dynamic cache partitioning mechanism can

deactivate the ways storing dead-on-fill blocks without significant performance degradation. Therefore, the proposed policy allows the mechanism to carry out capacity-efficient resizing.

The rest of this chapter is organized as follows. Section 3.2 describes the motivation and related work. Section 3.3 proposes the dynamic LRU-K insertion policy for dynamic cache resizing mechanisms. In Section 3.4, the proposal is evaluated in terms of energy consumption and performance. Section 3.5 concludes this chapter.

3.2 Motivation

3.2.1 Related Work

Data management policies are important for improving the cache efficiency, and a lot of researches have been performed in this field. In this chapter, the main idea to reduce dead-on-fill blocks is to change the position in the LRU chain to which a new block is inserted, called an *insertion position*. The Segmented LRU (SLRU) replacement policy [57] was originally proposed for cache management of a disk system. Its basic concept is the same as the proposal in this chapter. However, the insertion position, called the SLRU parameter, is not discussed well in [57]. They only indicate that the length between the MRU position and the insertion position should be 60-80% of the overall length of the LRU chain. Qureshi et al. proposed the adaptive insertion policy [58] and Jaleel et al. proposed its enhanced version [59], in which the insertion position is switched either the MRU position or the LRU position. However, their policies do not insert blocks into the other positions. Khan et al. [60] presented an insertion position selection mechanism. Using decision tree analysis, the mechanism selects the insertion position for each application from the candidate positions. The number of candidates is still limited because of the verification overhead of decision tree analysis. In addition, the limit of the insertion position deprives chances to further improve the reusability or to keep the high performance. Compared with the above studies, this chapter aims at the flexible configuration of the insertion position without a significant increase in its hardware and control overheads. The proposed policy can insert any priority position in the LRU chain.

There are some insertion policies based on other replacement policies. The dynamic re-referential interval prediction (DRRIP) [56] was based on the Not

Recently Used (NRU) replacement policy, which is employed in the industrial microprocessors [61, 62]. However, the insertion position of their proposal is limited to three positions. While the performance of the original NRU policy does not exceed that of the LRU policy, the performance of DRRIP is higher than that of the LRU policy. This fact indicates that the flexibility of insertion position is also important to improve the performance for various replacement policies. The pseudo insertion/promotion policy (PIPP) [63] changes the insertion position flexibly to any priority positions. However, their policy is based on the frequency-based promotion used in the Least Frequently Used policy [64]. In the frequency-based promotion, a block is promoted to the one higher priority position when the block is accessed. This is the disadvantage of PIPP because the frequency-based promotion significantly reduces the reusability of the data preferred by the recency-based promotion. In the recency-based promotion, the accessed blocks are promoted to the highest resident priority position as with the LRU replacement policy. Chaudhuri [28] have indicated that the performance of PIPP does not exceed that of other intelligent insertion policies. Chaudhuri has also proposed the Pseudo-LIFO policy (pLIFO) in his paper. However, the policy does not have any promotion activity, and the performance of pLIFO cannot exceed that of DRRIP in [56]. These results also let us confirm that the recency-based promotion used in the major cache management policies is basically important to improve the performance of cache memories. Based on these observations, this chapter focuses on the insertion position for the LRU replacement policy, the representative policy with the recency-based promotion. Flexibility of the insertion position for the other policies with the recency-based promotion, e.g., the NRU replacement policy, should be discussed in future work.

Another advanced replacement policy, the Shepherd Cache [65] was proposed

for emulating the Belady's OPT [66] that is an optimal replacement algorithm using offline profiling. Recently, the performance gap between the LRU replacement policy and the Belady's OPT is growing as the associativity of set-associative caches is increasing. This gap was investigated in [67]. However, to emulate optimal replacement online, a large hardware overhead is required. This fact is not preferable for dynamic cache resizing mechanisms such as the power-aware dynamic cache partitioning mechanism because they aim at energy reduction. Some researches combined different two policies, especially the LRU policy and the LFU policy. ARC [68] divides a cache into two parts. One part is managed by the LRU policy and the other part is by the LFU policy. These parts are dynamically resized by cache access characteristics. Adaptive Cache [69] tracks the replacement activity of two policies, and a victim block is selected by imitating the policy that is achieving the smaller miss rate. These policies require additional tag arrays to track the activity of the two policies, and their hardware costs are still large. Loh [70] has proposed the adaptive multi-queue policy specialized for 3-dimensional DRAM caches.

Dead-block prediction methods have also been proposed to reduce the dead-on-fill blocks. The focus of these methods is on a dead block, which is reusable but no longer reused. While the concept of dead-on-fill blocks is included in the concept of dead blocks in a larger sense, this dissertation focuses on only dead-on-fill blocks. This is because the ratio of dead-on-fill blocks is larger than that of dead blocks, which will be shown in Figure 3.2 in Section 3.2.2. In future, both dead-on-fill blocks and dead blocks should be considered because early eviction of dead blocks also has the potential to reduce the number of ways even in the benchmark with a lot of reusable lines. Burger et al. have investigated the ratio of dead blocks in the cache [71]. Their results also indicate that the ratio of

dead blocks is large, in which dead blocks include dead-on-fill blocks. Chaudhuri [28] have investigated the number of accesses to each block, in which the block becomes dead-on-fill one.

Dead-block prediction is used for two objectives. One objective is energy reduction. Cache Decay [72], IATAC [73], and Adaptive Mode Control [74] have been proposed to turn off cache lines that store dead blocks. HDOS [75] can immediately turn off these lines by detecting memory instructions that leave dead blocks in the cache. Drowsy Cache [76] changes the mode of cache lines including dead blocks into the drowsy mode, in which static energy consumption is reduced by a lower supply voltage. The other objective is performance improvement. IRGD [77] predicts dead blocks using inter-reference gap, which is the interval between two accesses to the same block. The counter-based cache replacement [78] predicts dead blocks by counting the number of accesses and its histories. Cache Bursts [79] makes tolerant predictions against irregular cache accesses while the prior prediction methods detect dead blocks by histories of cache accesses.

Cache bypassing methods can also reduce dead-on-fill blocks, in which the data causing dead-on-fill blocks are bypassed and not stored in the cache. Tyson et al. [80] have proposed the first approach of cache bypassing using a table structure associated with load and store instructions, as with the 2-bit branch prediction scheme. Johnson et al. [81] have proposed cache bypassing based on address location and access frequency of the data. In addition, Johnson et al. have also proposed a bypass buffer that stores data that are reused in a very short term. Such data are valuable to be stored in the cache, but are likely to become dead blocks immediately. The bypass buffer stores such data instead

of cache memories. The counter-based cache replacement and bypassing algorithm [78] can realize bypassing of dead-on-fill blocks using same hardware with dead-block prediction. Sandberg et al. [82] have proposed a software-based approach using a prefetch instruction for non-temporal data in a real microprocessor [83]. They use StatStack [84] that is an approximate profiling method of stack distance profiling [49]. However, in general, prediction misses on bypassing may cause a large performance degradation because the blocks are not stored at all. The trade-off between performance and energy reduction by cache bypassing for the power-aware dynamic cache partitioning mechanism is beyond the scope of this dissertation and should be discussed in future work.

Furthermore, this chapter focuses on the effect of dead-on-fill blocks to the power-aware dynamic cache partitioning mechanism. Dead-on-fill blocks occupy the cache without performance contribution. Hence, the existence of dead-on-fill blocks waste the allocated ways to the threads, i.e., the ways are overwhelmed with dead-on-fill blocks. If this problem is solved, it is possible to improve the energy efficiency of the power-aware dynamic cache partitioning mechanism. Previous researches about the mechanism do not discuss this point. The contributions of this chapter are to propose a new insertion policy for early eviction of dead-on-fill block and to clarify its effectiveness on the power-aware dynamic cache partitioning mechanism.

3.2.2 Cache Block Reusability

To understand how many data are reused, reusability that is the ratio of reusable blocks and dead-on-fill blocks is preliminary investigated. The simulation for investigating the cache block reusability is carried out as follows. A simulator

Table 3.1: Parameters of the simulated architecture to investigate reusability.

Core
8-issue out-of-order, 2GHz, 32nm CMOS technology
Memory
L1 I-Cache: 32kB, 2-way, 64B-line, 1 cycle latency L1 D-Cache: 32kB, 2-way, 64B-line, 1 cycle latency L2 Cache: 512kB, 8-way, 64B-line, 10-cycle latency L3 Cache: 2MB, 32-way, 64B-line, 20-cycle latency Main Memory: 200-cycle latency

based on the M5 simulator system [50] and CACTI 6.5 [51, 52, 53] has been developed. Parameters used in the simulation are listed in Table 3.1. The power-aware dynamic cache partitioning mechanism is used in the last-level L3 cache. The mechanism can adjust a trade-off between performance improvement and energy reduction using some parameters. For simplicity of the discussion in this chapter, a performance-oriented parameter configuration is used, which reduces the energy consumption as long as the performance loss is not significant. Cache resizing is performed every 5 million cycles. Benchmark programs examined on the simulator are selected from the SPEC CPU2006 Benchmark suite [54]. Each simulation is done by executing 2 billion instructions after 1 billion instructions.

Figure 3.2 shows the ratio of reusable blocks and dead-on-fill ones in the L3 cache in every benchmark. A lot of dead-on-fill blocks exist in most of the benchmarks. This figure brings out that 63% of all blocks are dead-on-fill blocks on an average. In addition, the ratio of dead-on-fill blocks to the overall blocks significantly depends on the benchmark programs. For example, 30% of blocks are dead-on-fill ones in `h264ref`, and 93% in `namd`.

From these observations, it is clear that a large part of the cache is filled

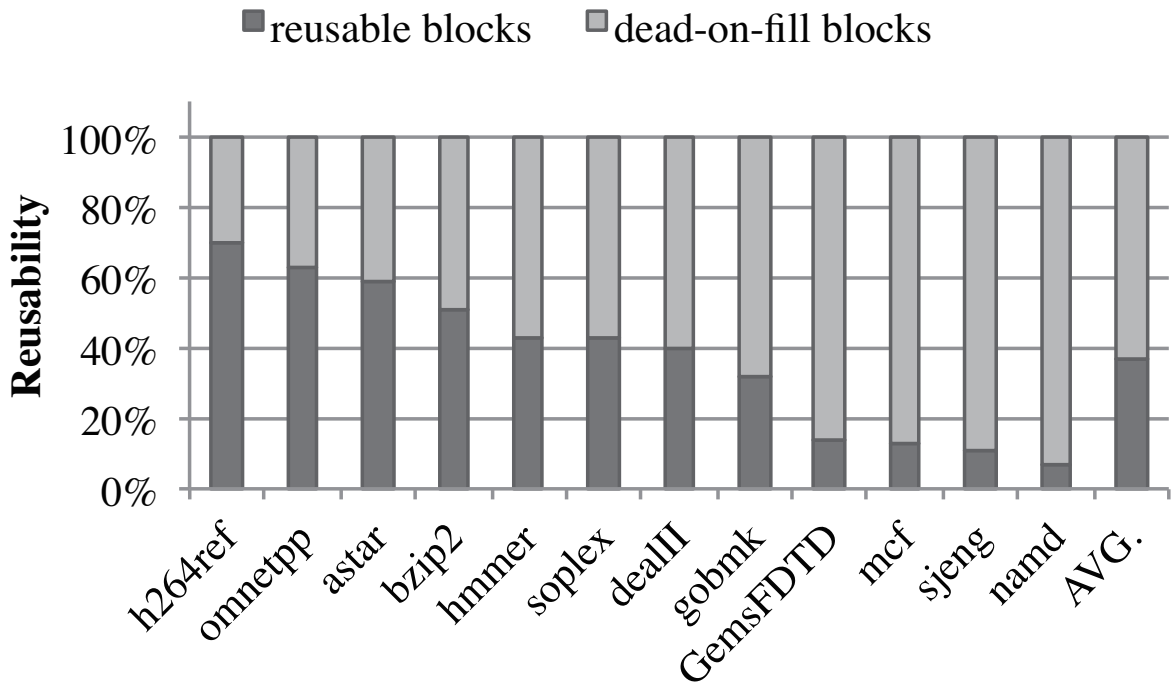


Figure 3.2: Cache block reusability.

with dead-on-fill blocks. They would decrease the benefit gained from the power-aware dynamic cache partitioning mechanism. If the ratio of dead-on-fill blocks can be reduced, the mechanism can reduce the allocated cache parts that are available for a running thread. Consequently, the data management policy needs to consider dead-on-fill blocks more proactively, and to reduce the number of them in the cache.

3.3 Dynamic LRU-K Insertion Policy

3.3.1 Policy Overview

To reduce dead-on-fill blocks in the cache, this paper proposes a new insertion policy named dynamic LRU-K insertion policy. In this policy, a new block is inserted at the K-th LRU position in the LRU chain. Moreover, the value K is dynamically adjusted for each application and its dynamic changes of reusability characteristics. As a result, the dynamic LRU-K insertion policy can evict dead-on-fill blocks earlier than the LRU policy that always inserts new blocks at the MRU position irrespective of their non-reusability. This improves the efficiency of cache usage.

The power-aware dynamic cache partitioning mechanism can enhance capacity-efficient allocation by using the proposed policy. Figure 3.3 shows the difference in behavior between the LRU policy and the proposed policy. In this figure, the insertion position in the proposed policy is fixed to the fourth LRU position ($K = 4$) as an example. To store all reusable blocks in this access stream, the LRU replacement policy requires the whole LRU chain. On the other hand, the proposed policy requires only a part of the LRU chain from the highest resident priority position to the third LRU position. This means that the LRU replacement policy requires eight ways while the proposed policy only requires six ways to keep reusable blocks, where the required length of the LRU chain is equal to the number of ways in the set-associative cache. As shown in this figure, reusable blocks can be stored using fewer ways, resulting in lowering the cache capacity activated by the power-aware dynamic cache partitioning mechanism.

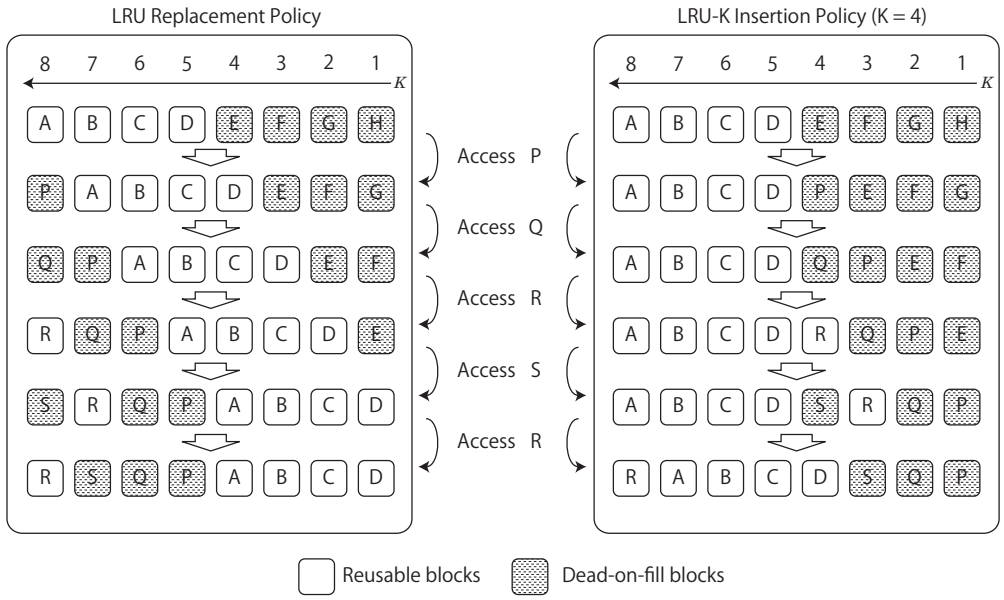


Figure 3.3: Comparing the LRU replacement policy with the LRU-K insertion policy ($K = 4$).

3.3.2 Principle of Optimal Insertion Position

In the power-aware dynamic cache partitioning mechanism, the resized cache capacity should be as small as possible unless reusable blocks are evicted. Hence, the policy has to decide the optimal insertion position under this condition. There is a trade-off when deciding the optimal insertion position. Considering early eviction of dead-on-fill blocks, value K should be as small as possible. However, to avoid eviction of reusable blocks, K should not be too small. To keep a balance between these two factors, the insertion position adjusted by K is decided based on the profile data, each of which records the position where a block is reused for the first time.

Figure 3.4 illustrates how the resident priority of a new cache block *X* changes in a set of an 8-way set-associative cache with the LRU replacement policy. First, *X* is inserted at the MRU ($K = 8$) position. After four misses occur, *X* is placed at the fourth LRU position. Then, when the first reuse of *X* occurs, the block is

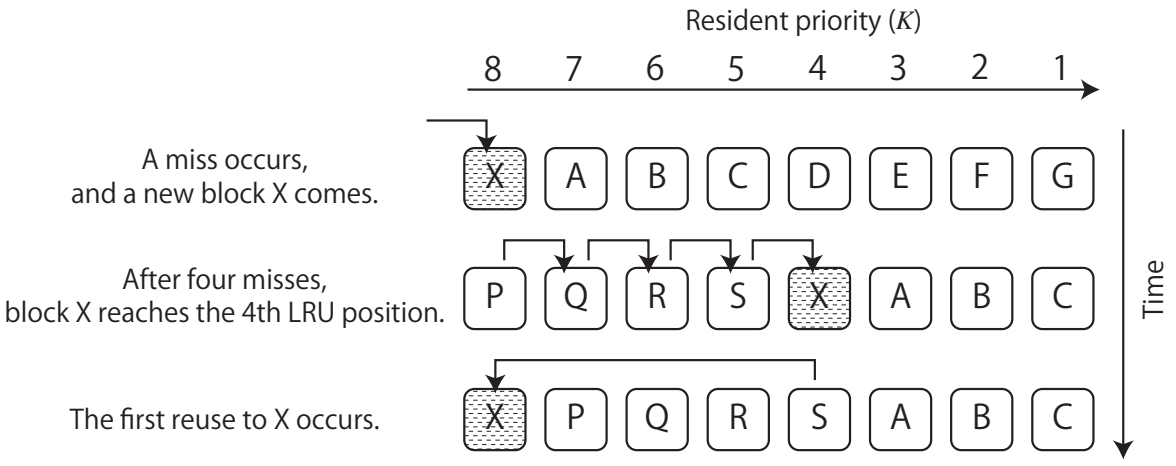


Figure 3.4: First reuse to a newly inserted block X and its resident priority change in the cache.

again moved to the MRU position. Focusing on this block, the optimal insertion position is the fifth LRU position. Even if X is inserted at the fifth LRU position, the first reuse of X occurs before X is evicted.

The position where X is reused for the first time is called the first reuse position of X. Figure 3.5 shows an example of histograms of first reuse positions. The vertical axis means the frequency of first reuses at each LRU position. The left-side histogram is a typical one in the case where a new cache block is always inserted at the MRU position. This histogram indicates that first reuses do not occur at the third, second, and first LRU positions. Therefore, a cache block that reaches the third LRU position without being reused should be evicted. In this situation, the insertion position of a new cache block should be changed to the fifth position as shown in the right-side histogram. As a result, dead-on-fill blocks are evicted earlier without evicting reusable blocks. It should be noted that the total number of first reuses in the left-side histogram is the same as that in the right-side one. This means that the newly coming reusable blocks are not evicted before being reused, even after the change of the insertion position.

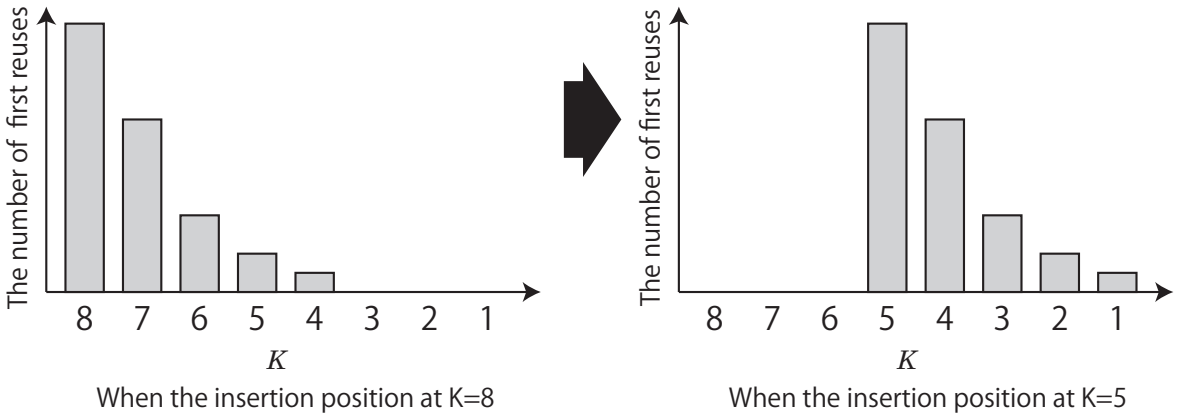


Figure 3.5: Profiling results of the number of first reuses to the newly inserted blocks in each priority position.

3.3.3 An Adjusting Mechanism of Insertion Position

A key to achieve automatic adjustment of the insertion position is to find the position of the right-most edge of the histogram of first reuse positions. At lower LRU positions after this edge position, there is no more first reuse in the histogram. As the insertion position moves to the lower LRU position, the right-most edge of the histogram also becomes closer to the LRU position. When the right-most edge reaches the LRU position, the insertion position becomes the lowest under the condition of not reducing first reuses. To find this insertion position, it is necessary to count the first reuses of blocks at the LRU position, because the number of first reuses typically decreases with K due to the temporal locality of reference.

In addition, phase changes in executing an application affect cache block reusability. This means that the shape of the histogram changes during the execution. Hence, the insertion position K should be changed at a fixed interval according to the algorithm described below.

$$K := \begin{cases} K+1 & (\text{if } C_{f,LRU} \geq A) \\ K-1 & (\text{if } C_{f,LRU} < A), \end{cases} \tag{3.1}$$

where,

$$\begin{aligned} K &= \text{(the insertion position in the LRU chain)}, \\ C_{f,LRU} &= \text{(the number of first reuses at the LRU position)}, \\ A &= \text{(the threshold for judgement of first reuses)}. \end{aligned}$$

According to Equation (3.1), the insertion position is shifted to the right by reducing K if the number of first reuses at the LRU position is small. On the other hand, the insertion position is moved to the left if the number of first reuses at the LRU position is large. As a result, the insertion position is adjusted to be the lowest under the condition of not causing significant performance degradation. Accordingly, it is expected that the power-aware dynamic cache partitioning mechanism can activate only the minimum number of ways to keep the performance.

3.4 Evaluations

3.4.1 Experimental Setup

In this section, the energy consumption and the performance of the power-aware dynamic cache partitioning mechanism with the proposed policy are evaluated. The proposed policy has been incorporated into the simulator used in Section 3.2.2. Simulation parameters are the same as shown in Table 3.1. The proposed policy changes the insertion position K every 5 million cycles, which is the same as the interval of changing the number of activated ways in the power-aware dynamic cache partitioning. Benchmark programs examined on the simulator are selected from the SPEC CPU2006 Benchmark suite [54]. Each simulation is done by executing 2 billion instructions after 1 billion instructions.

3.4.2 Evaluation Results of a Single-Core Processor

Insertion Position

To simply confirm that the policy can change the insertion position, Figure 3.6 shows the average insertion position by the proposed policy. The vertical axis indicates the average insertion position, which is calculated by normalizing average K by the average number of activated ways during execution. The metric is used in this evaluation because the maximum value of K depends on the number of allocated ways and changes during execution. As the graph bar becomes short, the insertion position becomes closer to the LRU position. Eight bars represent the average insertion positions of all the benchmarks when $A = 1, 2, 4, 8, 16, 32, 64, 128$, respectively. This figure shows that the insertion position averaged over all the benchmarks is about 0.86 when $A = 1$. This indicates

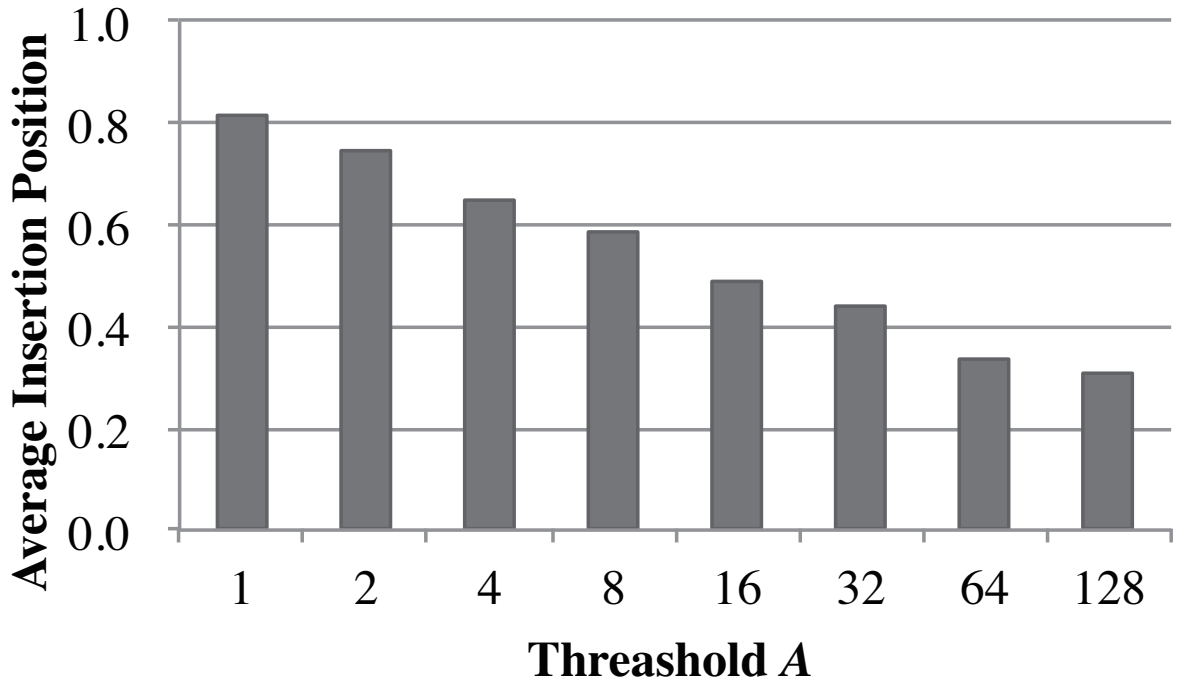


Figure 3.6: Average insertion position of all the benchmarks by the proposed policy.

that the proposed policy can move the insertion position closer to the LRU position than the LRU policy. Moreover, as the threshold A increases, the average insertion position becomes closer to the LRU position.

Energy Consumption and Performance

In the following evaluation, the benchmark programs are classified into four categories from two viewpoints. In terms of reusability, the benchmarks are first classified into two categories, high(H) and low(L). The proposed policy is effective for the benchmarks in the L category because they have many dead-on-fill blocks. The benchmarks in each category are further classified into two classes, long(L) and short(S), from the viewpoint of first reuse distance, which is the distance between the insertion position and the first reuse position. The distance of

Table 3.2: Benchmark classification by reusability and first reuse distance.

		First reuse distance	
		Long (L)	Short (S)
Reusability	High (H)	astar hmmer bzip2	h264ref omnetpp soplex
	Low (L)	gobmk mcf sjeng	dealII GemsFDTD namd

each benchmark is assessed using the length between the insertion position and the LRU position when $A = 1$, where the distance is normalized by the number of activated ways decided by the power-aware dynamic cache partitioning mechanism with the conventional LRU replacement policy. Between these two categories, the proposed policy works better for the benchmarks in the `S` category because the insertion positions of the benchmarks in the `S` category tend to become closer to the LRU position than those in the other categories.

Hereafter, these four categories are denoted by the combination of two letters. For example, the `HL` category includes benchmarks that are included in both the high(`H`) reusability category and the long(`L`) distance category. Table 3.2 shows the classification results. From the above discussions, it is predicted that the proposal is expected to be effective especially for the benchmarks in the `LS` category.

Figure 3.7 shows the energy consumption of the L3 cache with the power-aware dynamic cache partitioning mechanism and the dynamic LRU-K insertion policy. Here, the energy consumption is normalized by that with the LRU replacement policy. The horizontal axis shows the benchmark groups as defined in the previous paragraph. In the left-side graph, eight bars in each category

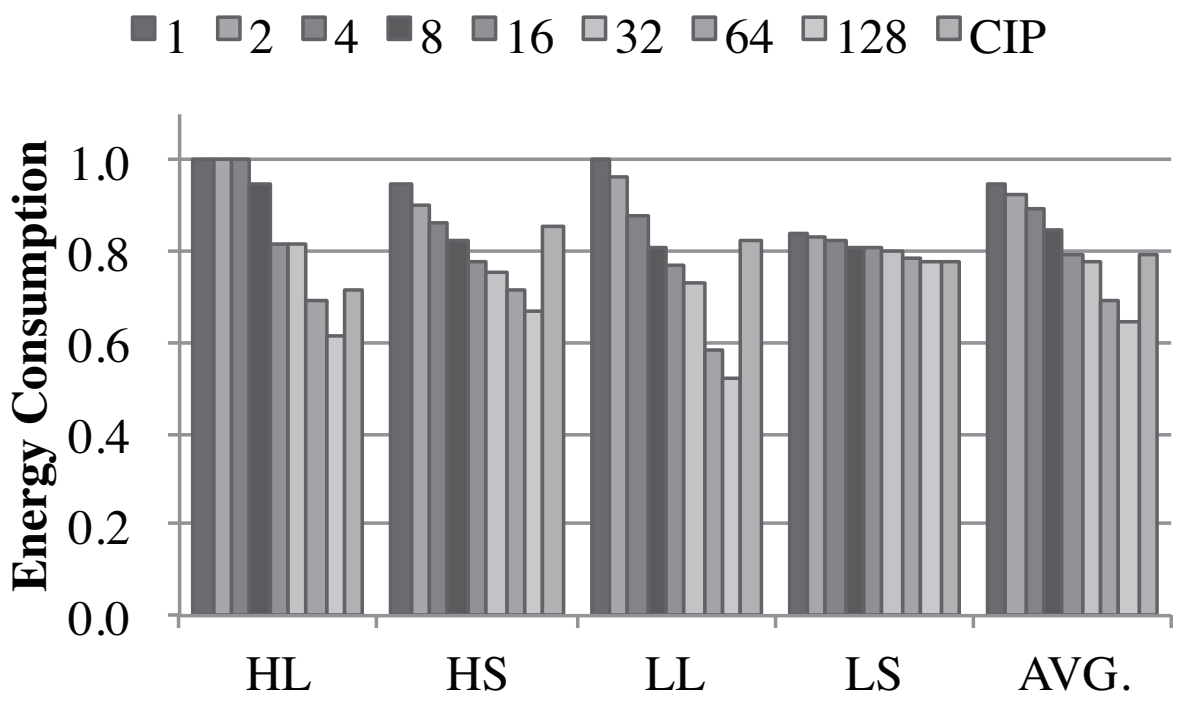


Figure 3.7: Energy consumption of the L3 cache on the single-core processor.

indicate the energy consumptions with $A = 1, 2, 4, 8, 16, 32, 64$, and 128, respectively. To compare the proposed policy with a policy whose insertion position is limited and not flexible, the right-most bar shows the results obtained with the center insertion policy (CIP), which simply inserts blocks at the middle between the LRU and MRU positions, i.e., the fourth LRU position if eight ways are activated as a result of dynamic resizing. This graph shows that the proposed policy can reduce the energy consumption of the L3 cache. When $A = 1$, the energy consumption is reduced by 6% on an average.

Figure 3.8 shows the normalized IPC to evaluate the performance of the single-thread execution, which is Instruction Per Cycle (IPC) normalized by that of the L3 cache with the power-aware dynamic cache partitioning mechanism and the LRU replacement policy. The horizontal axis shows the benchmark categories. In all the categories, the performance degradation by the proposed policy

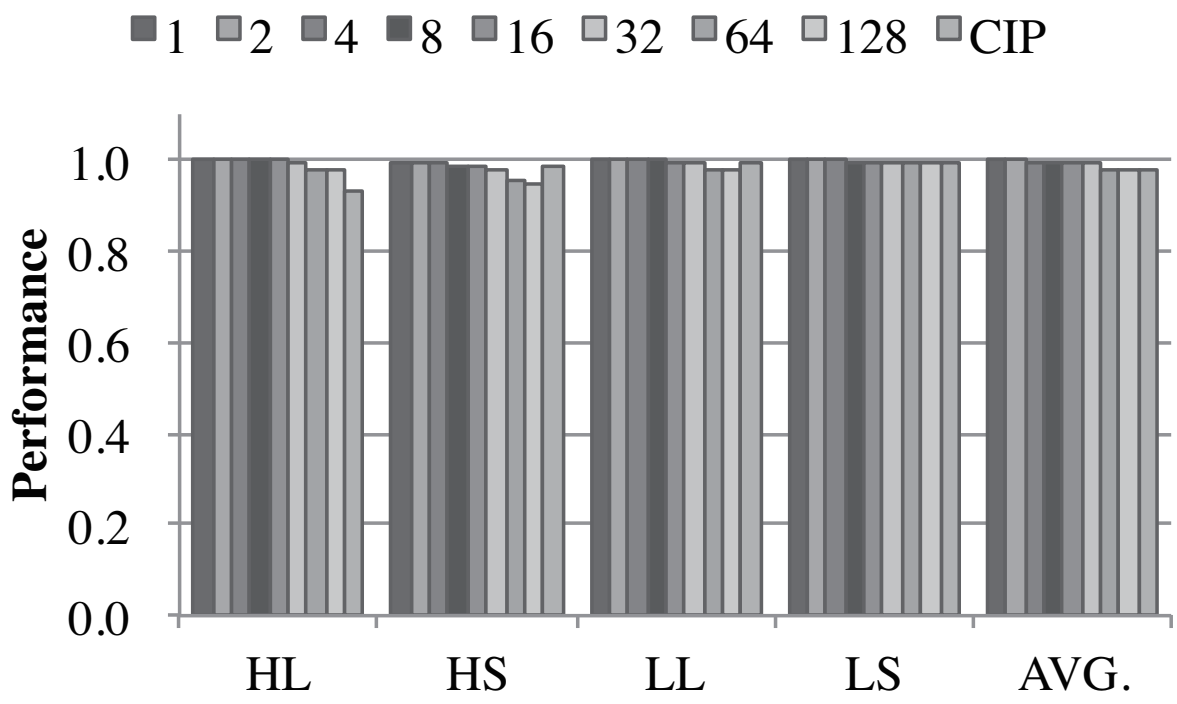


Figure 3.8: Performance on the single-core processor.

is not significant compared with the energy reduction. When $A = 1$, the maximum performance degradation is smaller than 1%. As a result, the proposed policy can enhance the energy efficiency of the power-aware dynamic cache partitioning.

As predicted at the benchmark classification, the proposed policy can reduce more energy consumption for the `HS` and `LS` categories than the `HL` and `LL` categories. Moreover, energy reduction of the `LS` benchmarks is larger than that of the `HS` benchmarks. As a result, the proposed policy is the most effective for the `LS` category. When $A = 1$, a maximum energy reduction of about 30% is achieved for `dealII`, which is one of the benchmarks in the `LS` category.

Figures 3.7 and 3.8 also indicate trade-offs decided by threshold A. If $A = 1$, the performance degradation is very small and therefore the energy reduction

is relatively small. As A increases, the performance degradation becomes indispensable while the energy reduction becomes large. When $A = 128$, the policy brings a 3% performance degradation with a 36% energy reduction on an average. These observations suggest that A is as small as possible if the performance degradation is avoided by the proposed policy. On the other hand, by increasing A, the proposed policy can prioritize a large energy reduction in compensation for a certain performance degradation.

Comparing the proposed policy with CIP, the proposed policy can achieve a higher energy efficiency. Figures 3.7 and 3.8 show that the proposed policy with $A = 64$ achieves a lower energy consumption than that of CIP even though the performances of both the proposed policy and CIP are the same. This is because the proposed policy can change the insertion position more flexibly than CIP.

Reusability

In the both evaluations, energy consumption and performance monotonically degrade as A increases. To find the optimal value of A, Figure 3.9 shows the reusability, which is the ratio of reusable blocks in the cache. In the figure, the vertical axis indicates the reusability, and the horizontal axis shows the results of the benchmark categories and the average result of all the benchmarks. The left-most bar in nine bars shows the reusability of the LRU replacement policy, the other bars shows those of the proposed policy when $A = 1, 2, 4, 8, 16, 32, 64,$ and 128, respectively.

From the average results, it is observed that the proposed policy can improve the reusability compared with the LRU replacement policy. These results indicate that the proposed policy can reduce dead-on-fill blocks. However, the effectiveness is limited when A is too large, e.g., $A = 64,$ and 128 in the figure. In

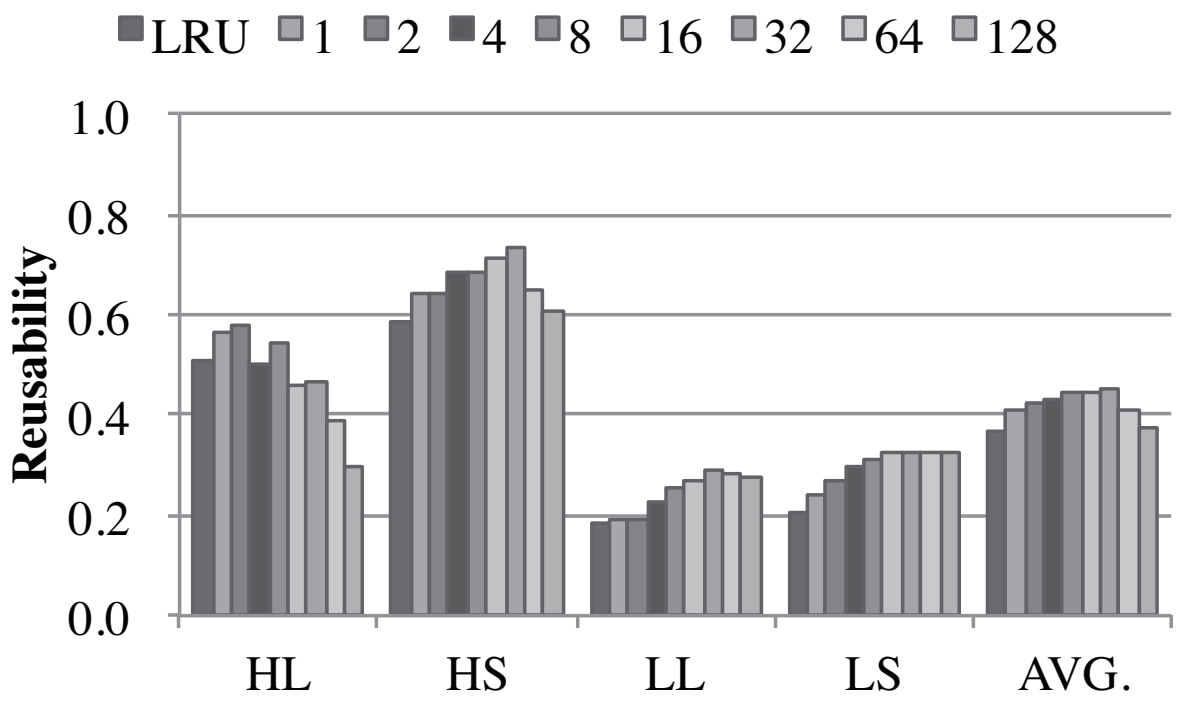


Figure 3.9: Reusability improvement in various A of the proposed policy.

these cases, the proposed policy evicts even reusable blocks earlier, and hence those blocks are changed into dead-on-fill blocks. Accordingly, A is an important parameter that should be carefully configured. This result shows that the reusability is the highest when $A = 32$ on an average. Hence, this value is the optimal from the viewpoint of reusability in the simulated architecture.

Focusing on each benchmark category, the reusability of the proposed policy is improved in the `HS`, `LL`, and `LS` benchmarks. On the other hand, the `HL` benchmarks hardly improve the reusability in higher A. The `HL` benchmarks originally have high reusability, and their first reuse distances are very long. Therefore, the proposed policy causes a lot of evictions of reusable blocks that have long distances of first reuses. However, the reusability is still comparable with the LRU replacement policy if A is smaller than 32. As a result, the performance impacts are not significant, as shown in Figure 3.8.

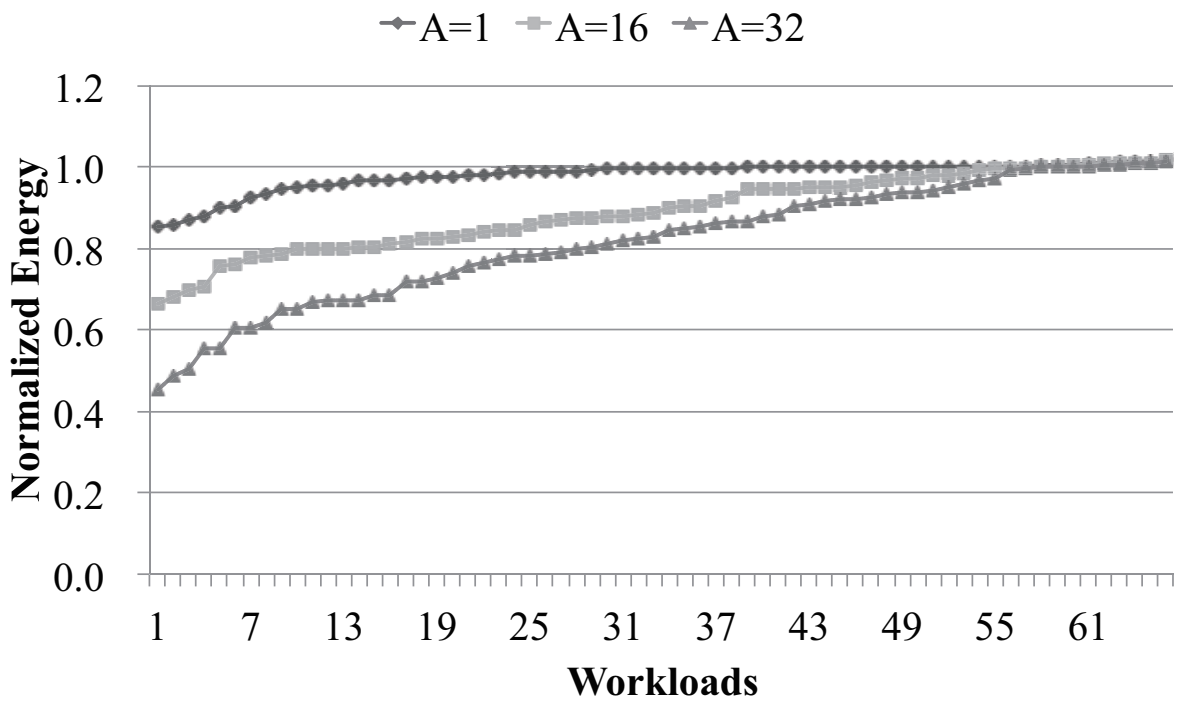


Figure 3.10: Energy consumption of the L3 cache on the 2-core CMP.

3.4.3 Evaluation Results of a 2-core CMP

This subsection evaluates the energy consumption and the performance of a 2-core CMP. The simulation is performed by the 2-thread workloads, each of which consists of 2 benchmarks. Hence, the number of workloads is 66 ($= {}_{12}C_2$). Each simulation is done by executing first 2 billion cycles of the simulated microprocessor so that the experimented periods of all benchmarks do not change as much as possible.

Figure 3.10 shows the energy consumption of the L3 cache with the proposed policy. In the figure, the vertical axis indicates the energy consumption, which is normalized by that of the power-aware dynamic cache partitioning mechanism with the LRU replacement policy. The horizontal axis shows the workloads. Here, the workloads are arranged in ascending order of the normalized energy.

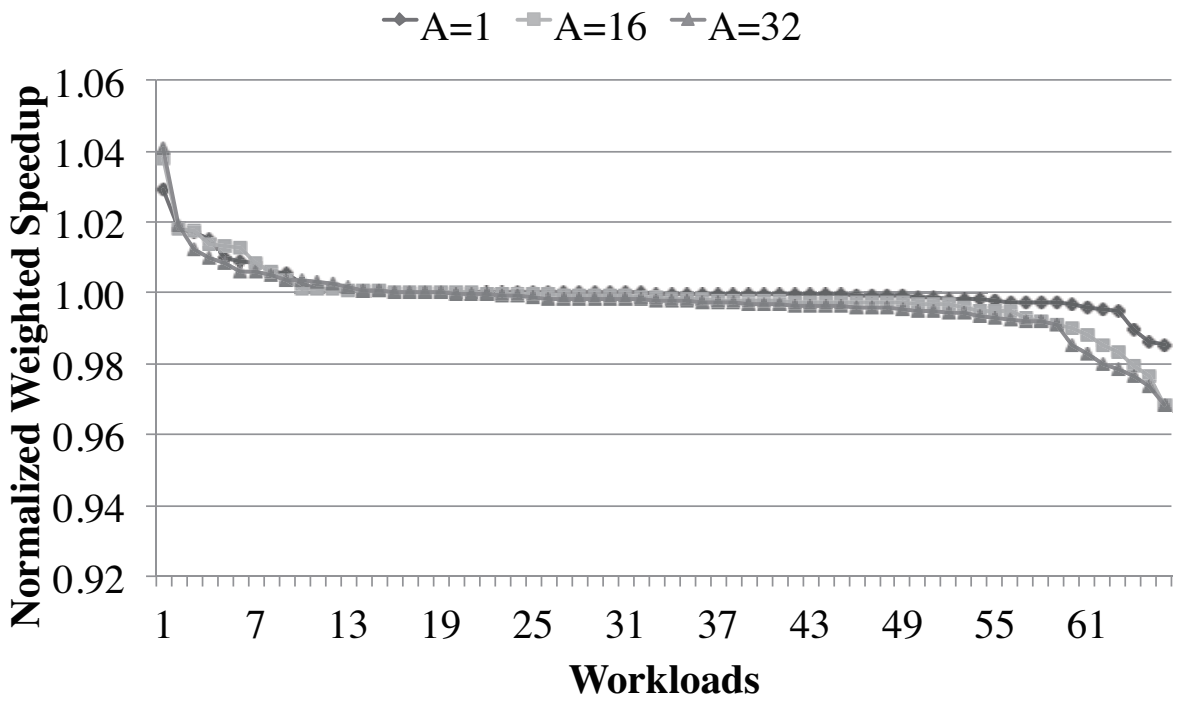


Figure 3.11: Performance on the 2-core CMP.

The three lines show the results of $A = 1, 16,$ and $32,$ respectively. The results with $A = 1$ indicate the normalized energy consumptions achieved when eviction of reusable blocks is avoided as much as possible. The results with $A = 16$ or 32 show the normalized energy consumptions when the eviction of some reusable blocks is allowed and the reusability becomes the higher, as shown in Figure 3.9. In Figure 3.10, it is observed that the proposed policy can reduce the energy consumption of the L3 cache with the power-aware dynamic cache partitioning mechanism by about 2% on an average when $A = 1$. If A becomes larger, the proposed policy can reduce more energy consumption. The proposed policy can reduce the energy consumption by 11% and 19% on an average when $A = 16$ and $32,$ respectively. Furthermore, the number of workloads whose energy consumption is obviously decreased increases with A.

Figure 3.11 shows Weighted Speedup [55] to evaluate the performances of

the multi-thread workloads. The performances are normalized by those of the power-aware dynamic cache partitioning mechanism with the LRU replacement policy. In Figure 3.11, the vertical axis shows the performance, and the horizontal axis indicates the workloads. Three lines show the results with $A = 1, 16,$ and $32,$ respectively. This figure indicates that the significant performance degradations do not occur in any workloads. The maximum performance degradations are 2%, 3%, and 3% when $A = 1, 16,$ and $32,$ respectively.

Additional performance improvements are obtained in some workloads. Especially, when $A = 32$, the maximum performance improvement of about 4% is observed in the combination of `bzip2` and `hmmer`, which is at the highest performance workload when $A = 32$ in Figure 3.11. At the same time, the energy consumption does not change. This combination is the 57-th lowest energy workload when $A = 32$ in Figure 3.10. In the case of the combination of `astar` and `hmmer` when $A = 32$, both the performance improvement and the energy reduction are achieved. The performance improvement is about 2%, the second highest performance workload of $A = 32$ in Figure 3.11. At the same time, the energy consumption is reduced by about 24%, the 22-th lowest energy workload when $A = 32$ in Figure 3.10. These results indicate that the reduction in dead-on-fill blocks can improve the performance while the energy consumption is reduced.

From these observations, it is clear that the proposed policy can contribute to the reduction in dead-on-fill blocks and the improvement of cache block reusability. As a result, the proposed policy can achieve energy reduction of the power-aware dynamic cache partitioning without significant performance degradation, and can achieve remarkable performance improvement in some cases.

3.4.4 Hardware Overhead

The implementation overhead of the proposed policy is discussed. The proposed policy needs one bit per cache line to distinguish whether accesses to blocks are first reuses or not. However, in a cache with 64-byte (512-bit) blocks, the additional bit cost is less than 0.2% of all the bits of the L3 data array in the experimented cache configuration. In addition, an access counter used to count $C_{first,1}$, and a register specifying the insertion position K are required. These hardware costs are negligible compared with the overall cache hardware. The above discussions indicate that the proposed policy will be implemented at a low overhead.

However, the power-aware dynamic cache partitioning mechanism and some other dynamic cache resizing mechanisms need stack distance profiling [49] based on the LRU replacement policy to decide the cache size to be allocated to threads. In our experiments, the power-aware dynamic cache partitioning mechanism stores the resident priority of the LRU chain of the LRU replacement policy in addition to the resident priority of the proposed policy. This will cause a non-ignorable overhead if this mechanism is naively implemented, i.e., additional five bits per line. The overhead is 1% of all the bits of the L3 data array in the experimented cache configuration. In the future, we will develop an approximate approach to stack distance profiling by using the LRU chain of the proposed policy, and reduce the overhead to store the information of the LRU chain of the LRU replacement policy.

3.5 Conclusions

In recent years, various applications have become more complex and its data set becomes larger. As a result, some applications cause dead-on-fill blocks. Dead-on-fill blocks may occupy the cache capacity while they do not contribute to performance. This is because they are not reused after their insertion to cache memories. Due to their existence, the power-aware dynamic cache partitioning mechanism must activate much more cache area to maintain both reusable and dead-on-fill blocks. As a result, the effect of the mechanism on energy efficiency degrades.

To reduce dead-on-fill blocks in the cache, this paper has proposed the dynamic LRU-K insertion policy. The proposed policy can flexibly insert newly coming blocks into any resident priority positions to keep a balance between early eviction of dead-on-fill blocks and retainment of reusable blocks. Therefore, the proposed policy enables the power-aware dynamic cache partitioning mechanism to do efficient allocation without performance degradation by reducing only dead-on-fill blocks. Experimental results show that energy consumption of the cache is reduced by up to 30%, and 6% on an average without significant performance degradation in the single-core processor. Moreover, the simulation results show that the proposed policy is also effective for the power-aware dynamic cache partitioning mechanism in the CMP.

Chapter 4

A Thread Scheduling Method based on Working Set Assessment for CMPs

4.1 Introduction

This chapter focuses on inter-thread cache conflicts. In many CMPs, multiple threads share a single cache, and CMP suffers from inter-thread cache conflicts. Inter-thread cache conflicts have two causes, inter-thread kickouts (ITKOs) and capacity shortage. ITKOs occur when newly coming data of a thread replaces data of other threads in the cache. Capacity shortage occurs when the sum of working set sizes of the threads becomes larger than the cache capacity.

Against ITKOs, the power-aware dynamic cache partitioning mechanism [24, 25, 26] discussed in the previous chapters has been proposed. However, one of the problems in the mechanism is that it cannot avoid capacity shortage. The mechanism should allocate the ways so that the working sets can be stored in

the cache. However, the mechanism cannot do it in the case where threads with large working sets share a cache, in which the sum of working set sizes of the threads becomes larger than the cache capacity. As a result, performance does not improve by the power-aware dynamic cache partitioning mechanism.

To avoid capacity shortage in the power-aware dynamic cache partitioning mechanism, this chapter focuses on differences of working set sizes among threads. Since the working set sizes of threads highly depends on the cache access characteristics of threads, the sum of working set sizes of the thread combinations that share a cache may be different from one thread combination to another. If the threads with large working set sizes share a cache, capacity shortage occurs and degrades the performance. However, if the threads with small working set sizes share a cache, capacity shortage does not occur. The results of the preliminary evaluation show that performance degradation by capacity shortage becomes larger as the sum of working set sizes increases. Especially, performance degradation becomes significant when the sum of working set sizes of threads exceeds cache capacity.

In this chapter, to solve the cache capacity shortage problem, a thread scheduling method based on working set assessment for CMPs is proposed. Recently, a CMP includes multiple cache memories, each of which is shared by multiple cores. In this case, the thread combinations sharing a cache memory can be flexibly changed. Hence, the proposed thread scheduling method changes the thread combinations so that the capacity shortage problem is alleviated. The proposed scheduling method consists of two stages, assessment of working set sizes of threads and decision of thread assignments to cores by a scheduling algorithm. The algorithm decides the assignments so that threads with large working sets do not share a same cache, and a thread with the largest working set is coupled

with a thread with the smallest working set. As a result, the power-aware dynamic cache partitioning mechanism can allocate the enough number of ways to the threads, resulting in performance improvement.

The rest of this chapter is organized as follows. In Section 4.2, the motivation of this chapter from the viewpoint of inter-thread cache conflicts and related work are described. Section 4.3 proposes the thread scheduling method based on working set assessment. In Section 4.4, the proposed method is evaluated. Finally, Section 4.5 concludes this chapter.

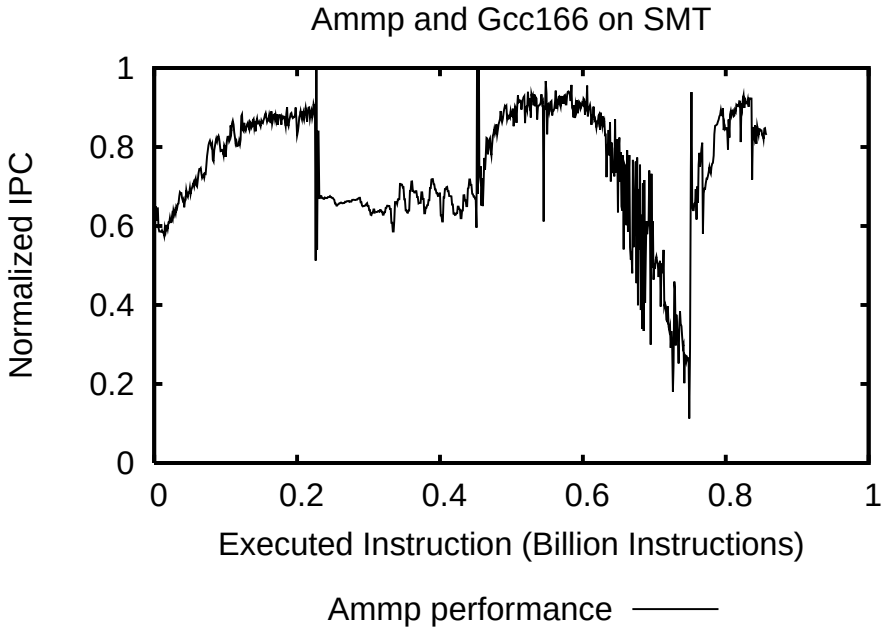


Figure 4.1: Performance of `Ammp` when simultaneously executed with `Gcc166` on a SMT processor.

4.2 Motivation and Related Work

4.2.1 Inter-Thread Cache Conflicts

As discussed in Section 4.1, inter-thread cache conflicts have a major impact to the performance of multi-thread execution. To clarify the effects of inter-thread cache conflicts, preliminary evaluations are carried out. In these evaluations, a simultaneous multithreading (SMT) processor executing two threads is simulated. The processor has a two-level cache hierarchy. The threads used in the simulations are selected from the SPEC CPU2000 benchmark suite [85].

First, simultaneous execution of `ammp` and `gcc166` are examined. Figure 4.1

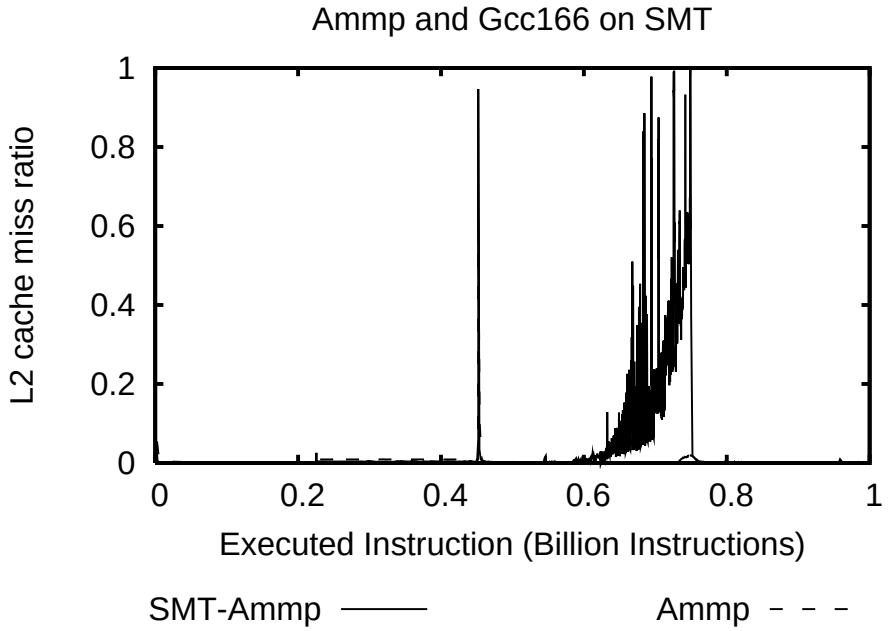


Figure 4.2: L2 cache miss ratio of `Ammp` when simultaneously executed with `Gcc166` on a SMT processor.

depicts the performance of `ammp` in multi-thread execution by measured in Instruction Per Cycle (IPC), which is normalized by the performance of single-thread execution. From the figure, it is observed that the performance of multi-thread execution gradually and significantly degrades in the period from 0.6 to 0.8 billion instructions. Figure 4.2 shows the miss ratio for `ammp` in the L2 cache. The executed period of Figure 4.2 is the same as that of Figure 4.1. Figure 4.2 indicates the miss ratio of the L2 cache gradually and significantly increases in the same period as with the performance degradation focused in Figure 4.1. During the same period, the miss ratio of the L1 cache does not change compared with that of single-thread execution. Therefore, the increase in the miss ratio of the L2 cache occurs by inter-thread cache conflicts, which degrade the performance of `ammp`.

Second, simultaneous execution of `twolf` and `mcf` is discussed. Figure 4.3

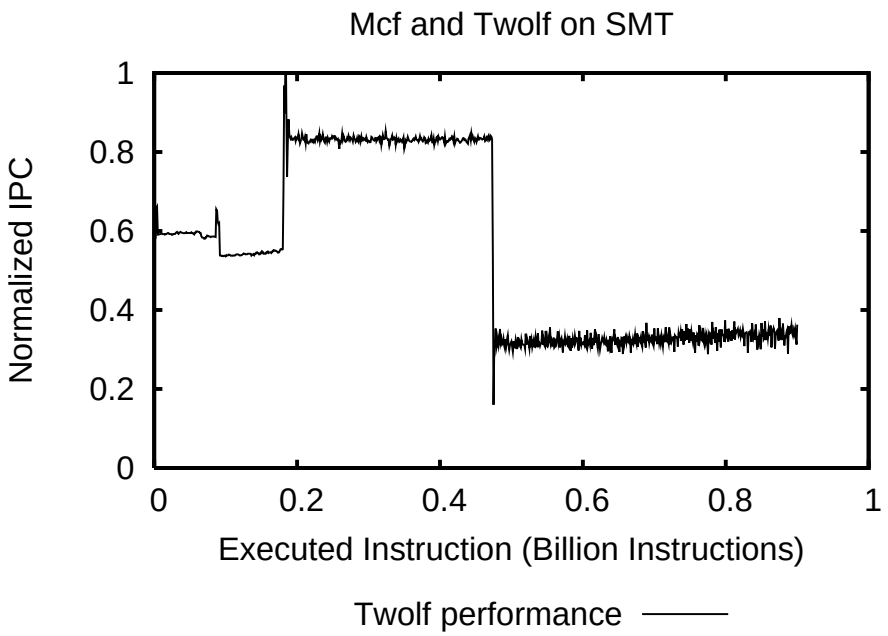


Figure 4.3: Performance of `twolf` when simultaneously executed with `mcf` on a SMT processor.

indicates the performance of `twolf` in multi-thread execution in IPC, which is normalized by IPC of single-thread execution. From the figure, it is observed that the performance significantly degrades after executing 0.5 billion instructions. Figure 4.5 shows the L2 cache miss ratio of `twolf`. As with `ammp`, the performance degrades as the miss ratio increases. On the other hand, Figure 4.4 shows the L1 cache miss ratio. Figures 4.3 and 4.4 indicate that the increase in the L1 cache miss ratio in the period from 0.2 to 0.5 billion executed instructions hardly affects the performance. Hence, the L2 cache has a larger impact on the performance than the L1 cache.

From the above, it is observed that inter-thread cache conflicts cause the increase in the miss ratio in the shared caches, and cause significant performance degradation. Moreover, last-level caches have larger impacts on the performance than upper-level caches.

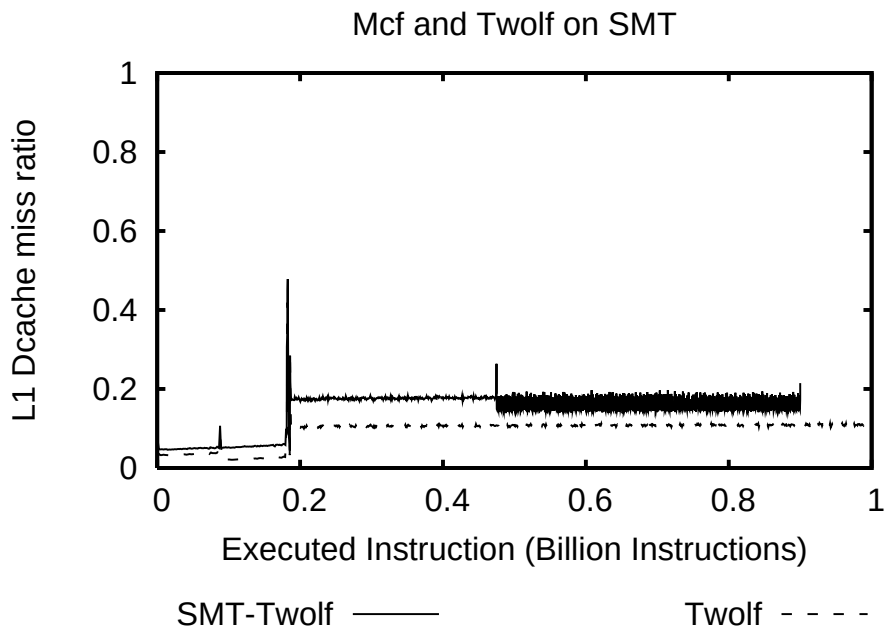


Figure 4.4: L1 data cache miss ratio of `twolf` when simultaneously executed with `mcf` on a SMT processor.

There are some other papers that indicate the harmful effect of inter-thread cache conflicts on the performance of multi-thread execution. Hily et al. described that ignoring the effect of inter-thread cache conflicts causes over-estimation of the performance of SMT processors [86]. In addition, Thekkath et al. investigate the effectiveness for microprocessors to support multiple hardware contexts and conclude that the effect of inter-thread cache conflicts becomes larger as the number of threads increases [87].

4.2.2 Dynamic Cache Resizing Mechanisms

As previously discussed, one of the causes of inter-thread cache conflicts is ITKO. To avoid ITKOs, dynamic cache resizing mechanisms, e.g., the power-aware dynamic cache partitioning mechanism, have been proposed. The mechanism can prevent ITKOs and achieve energy saving. As another approach that can avoid

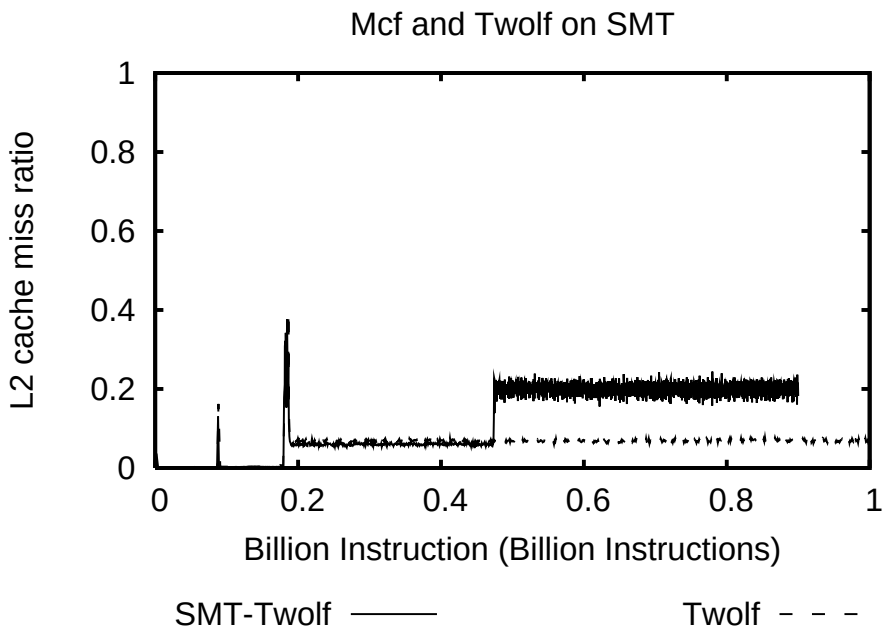


Figure 4.5: L2 cache miss ratio of `twolf` when simultaneously executed with `mcf` on a SMT processor.

ITKO, Lin et al. proposed a software-controlled dynamic cache partitioning by using virtual memory address mapping of the operating system [88]. It is the interesting case that dynamic cache partitioning mechanisms are applied to a real machine. Compared with the past researches of dynamic cache partitioning mechanisms by simulation experiments, performance improvement in the real machines is significant.

However, the problem in the power-aware dynamic cache partitioning mechanisms is that their performance improvements are limited when threads with large working sets share a cache. Moreto et al. have reported that performance degradation in dynamic cache partitioning becomes larger if the sum of the sufficient numbers of ways, which is for maintaining 90% performance on each individual thread, exceeds the total number of ways in the cache [89].

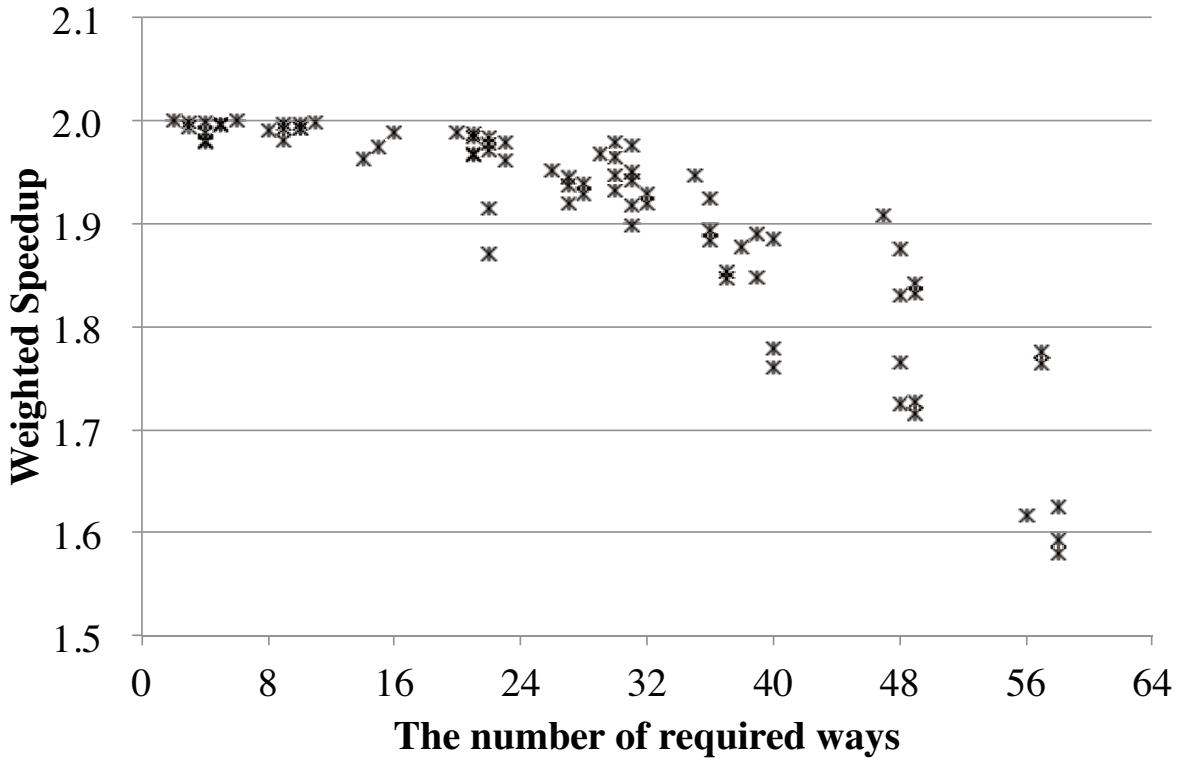


Figure 4.6: Relationship between the number of required ways and the performance when two threads are simultaneously executed.

To highlight this problem more clearly, a preliminary experiment is performed. This experiment assumes a 2-core CMP with a last-level 32-way 1MB L2 shared cache in which the power-aware dynamic cache partitioning mechanism [26] is applied. On this CMP, various combinations of two threads are executed. These two threads are selected from threads with various working set sizes in the SPEC CPU2000 benchmark suite [85]. From the execution results, relationship between the sum of working set sizes of threads and performance degradation is revealed. Here, the working set size of the benchmark is defined as the number of ways that the thread requires for achieving 95% performance compared with the performance when the ways in the cache are fully allocated.

Figure 4.6 shows the relationship between the sum of working set sizes of

threads sharing the L2 cache and their performance. The horizontal axis shows the sum of working set sizes of threads in the unit of the number of ways. The vertical axis shows the weighted speedup [55]. Figure 4.6 indicates that the performance degradation becomes larger as the sum of working set sizes increases. Especially, the performance degradation is significant if the sum of working set sizes exceeds 32 ways, while not significant if the sum is less than 32 ways. From these results, it is clear that the sum of working set sizes should not exceed the shared cache capacity of CMPs to avoid severe performance degradation.

To overcome the performance degradation due to capacity shortage, this chapter proposes a thread scheduling method, which assigns the threads to cores so that the threads with large working sets do not share a same cache.

4.2.3 Thread Scheduling Methods

There are several researches about thread scheduling methods considering inter-thread cache conflicts. Settle et al. indicated that the main cause of performance degradation in simultaneous multithreading (SMT) processors [90] is inter-thread cache conflicts. Furthermore, they proposed the cache monitoring scheme to find out cache sets that cause ITKOs [91], and a thread scheduling method to avoid ITKOs [92]. Knauerhase et al. focused on threads with high cache miss rates, and scheduled such threads to different phases or cores [93]. Banikazemi et al. have proposed a thread scheduling method based on the performance prediction model considering cache line occupancy, miss rate, and cycle per instruction of threads [94].

These previous researches focused on reducing ITKOs by thread scheduling. On the other hand, the thread scheduling method proposed in this chapter also

alleviates the capacity shortage problem in shared caches. In addition, the proposed scheduling method reduces ITKOs by cooperating with the power-aware dynamic cache partitioning mechanisms. As a result, this approach considers the two causes of inter-thread cache conflicts, ITKOs and capacity shortage, and further performance improvement is expected. Suh et al. proposed memory-aware scheduling, shown in detail in [95], and cache partitioning using *marginal gain*, which is the derivative of the miss-rate curve [96]. However, they do not consider the combination of thread scheduling and cache partitioning.

Thread scheduling methods focusing on other kinds of resource, e.g. execution cores of SMT processors, have been also proposed. This dissertation considers a cache memory as the most important component in which conflicts should be reduced. In addition, inter-thread conflicts in execution cores do not occur on the supposed CMP in this dissertation because resource in the cores is not shared by multiple threads. Therefore, other kinds of resource and their conflicts are beyond the scope of this dissertation. El-Moursy et al. [97] have proposed the scheduling method based on the ready to in-flight instruction ratio (RIR). If this ratio is large, the thread is sensitive to resource conflicts in the SMT processor. Snavely et al. [55] have also proposed the thread scheduling method considering the conflicts on execution units, instruction queues, and caches, exhaustively. Other researches have considered resource conflicts on SMT processors from the viewpoint of QoS [98], and intra-core resource partitioning [99].

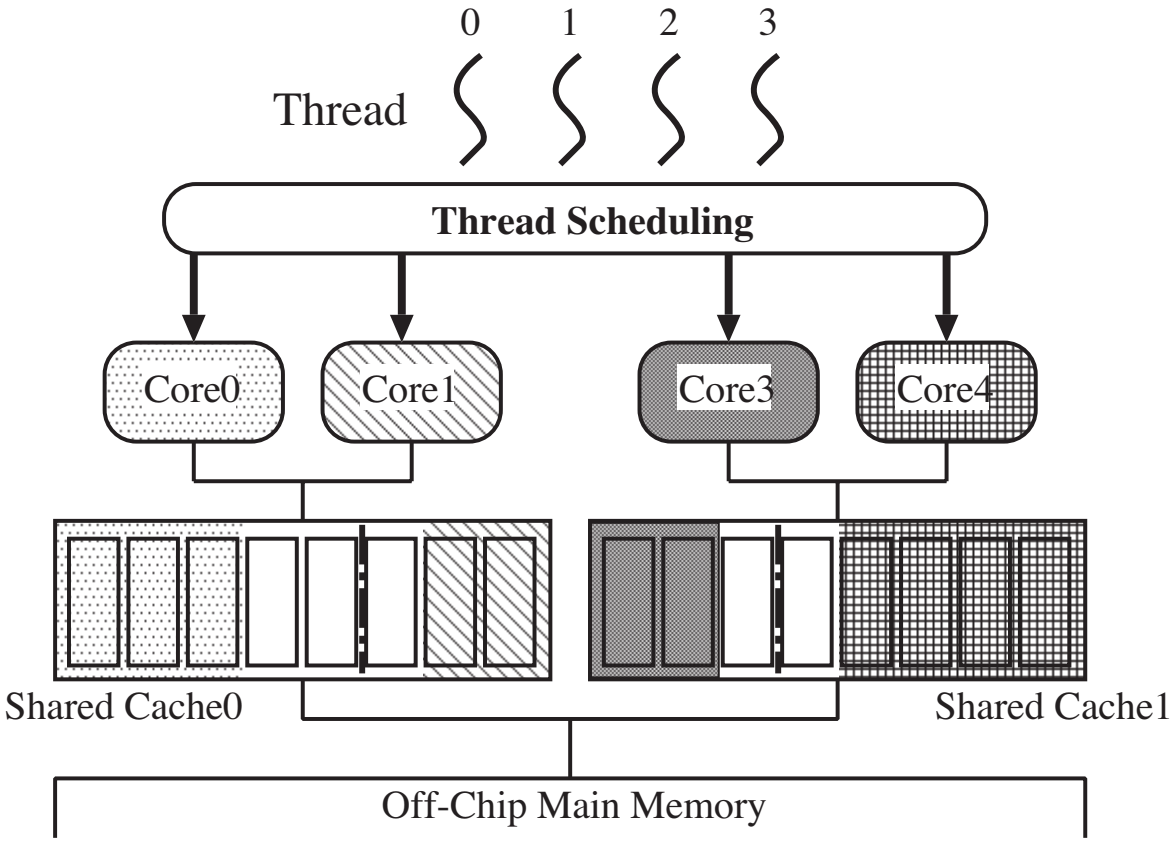


Figure 4.7: Overview of the proposal.

4.3 A Thread Scheduling Method based on Working Set Assessment

4.3.1 Method Overview

To avoid the performance degradation by capacity shortage when using the power-aware dynamic cache partitioning, this chapter proposes a thread scheduling method based on working set assessment for CMPs. Figure 4.7 shows the CMP architecture assumed in the proposed method. The CMP has four cores and two shared caches. The structure of the CMP is also employed in real microprocessors, Intel Core 2 Quad [100] and Intel Xeon [101]. The power-aware

dynamic cache partitioning mechanisms are applied to these two caches. On this architecture, the proposed method equalizes the sums of working set sizes of threads among *Shared Cache 0 and 1* and alleviates the capacity shortage problem.

In the proposed method, the working set sizes of the threads are assessed. Then, based on the assessment results, combinations of threads that should share a same cache are decided by a scheduling algorithm. The details of these processes are shown in the following sections.

4.3.2 Working Set Assessment

In the proposed scheme, an assessment of working set sizes of threads is performed by stack distance profiling [49] and locality assessment metric *D* as previously discussed in Chapter 2.

Here, the conceptual figure of stack distance profiling is shown again in Figure 4.8. The C_x means the number of accesses to blocks with the x-th LRU position ($1 < x < N$ in the case that N ways are allocated by the power-aware dynamic cache partitioning mechanism). In the power-aware dynamic cache partitioning mechanism, the locality assessment metric $D(= C_1/C_N)$ is used for deciding the number of allocated ways. If $D < t1$, the number of allocated ways should be increased. If $D > t_2$, the number of allocated ways is decreased. If $t_1 < D < t_2$, the number of allocated ways is not changed. This means that the number of allocated ways is sufficient when $t_1 < D < t_2$. Therefore, working set size *W* assessed in the proposed method is defined as follows.

$$W = x \text{ where } t_1 < \frac{C_1}{C_x} < t_2 \tag{4.1}$$

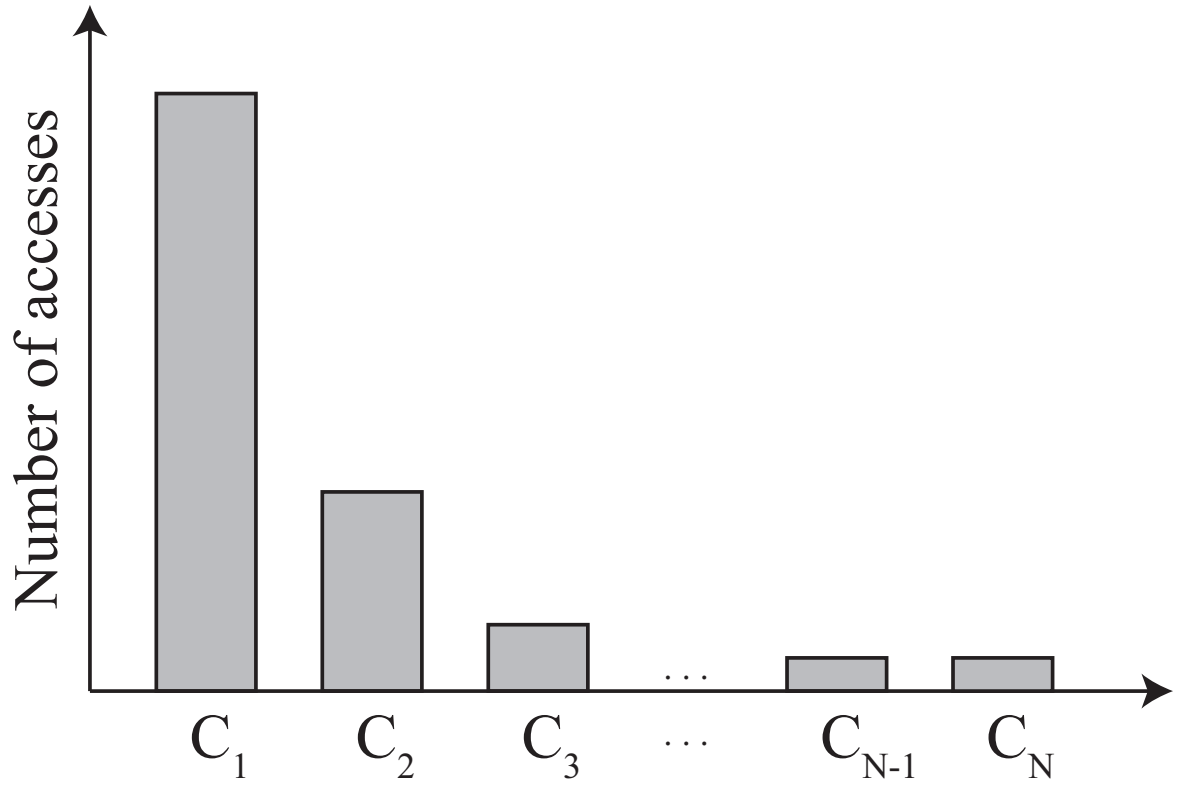


Figure 4.8: Conceptual figure of stack distance profiling.

4.3.3 A Scheduling Algorithm

The proposed method needs a scheduling algorithm to decide assignments of threads to cores so that threads with large working sets do not share a same cache. In this section, the scheduling algorithm is described in detail as bellow.

The objective of this algorithm is to make thread combinations that should share a cache, so that the sum of working set sizes of a shared cache becomes equal to that of other shared cache as much as possible. This approach avoids the situation where threads with large working set sizes share a same cache.

Let assume a k-core CMP with k/n shared caches, in which n cores share a cache. On this CMP, the proposed method assigns k threads to k cores. The flow chart of the algorithm is shown in Figure 4.9. At the initial state, there are the

list T that contains k threads, and the list G that consists of k/n groups. At the end of the algorithm, each group includes n threads. Assignments of threads to cores are decided so that these n threads share a same cache.

In Step 1 of the algorithm, threads in T are sorted according to their working set sizes in descending order. In Figure 4.9, W_i denotes the working set size of the i-th thread in T. In Step 2, each of top k/n threads in T is allocated into one of groups in G. As a result of Step 2, the i-th thread in T is allocated to the i-th group. In Step 3, the groups in G are sorted according to the sum of working set sizes of the allocated threads in ascending order. In Figure 4.9, $W_{all,j}$ is the sum of working set sizes of threads allocated to the j-th group. This step is required to allow a thread with a larger W_i to be added to a group with smaller $W_{all,j}$ in Step 2 of the next iteration. Finally, the scheduling algorithm finishes if T is empty. Otherwise, go to Step 2.

4.3.4 Cooperation between the Thread Scheduling Method with Dynamic Cache Resizing Mechanisms

The cooperation between the proposed thread scheduling based on working set assessment and the power-aware dynamic cache partitioning mechanism has two advantages.

The first advantage is avoidance of both capacity shortage and ITKOs. Without the power-aware dynamic cache partitioning mechanism, ITKOs cannot be avoided. Therefore, thread scheduling methods must consider both capacity shortage and inter-thread kickouts. In the proposed method, the scheduling algorithm only has to concentrate on the capacity shortage, and the power-aware dynamic cache partitioning mechanism avoids ITKOs. Based of these effects, the performance of the multi-thread execution improves.

The second advantage is that this cooperation can realize the flexible way-allocation and energy saving of the power-aware dynamic cache partitioning mechanism. Under the situation that a thread with large working set and that with small working set share a cache without capacity shortage, the larger capacity should be allocated to the former thread. As a result, the thread with large working set can obtain sufficient performance. In addition, in the case of the sum of working set sizes of the threads is smaller than the capacity of a cache, the power-aware dynamic cache partitioning mechanism can disable excessive ways and reduce energy. As a result, the proposed cache system can contribute to energy saving as well as performance improvement.

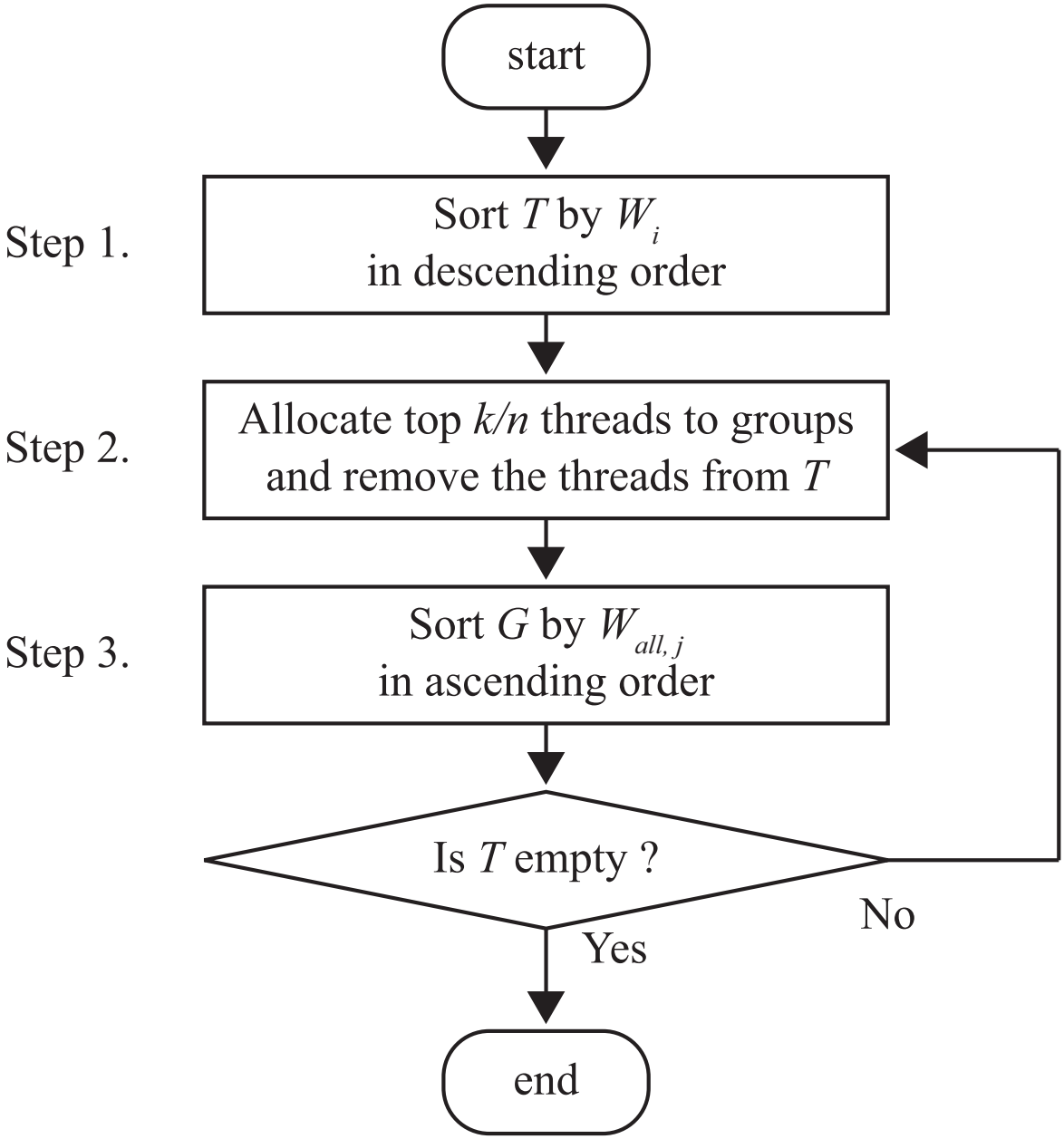


Figure 4.9: Flow chart of the scheduling algorithm.

Table 4.1: Simulation parameters for the CMP.

Architectural parameters	**Specifications**
Core	8-issue out-of-order
L1 I-cache	32kB, 2-way, 32B-line, 1 cycle latency
L1 D-cache	32kB, 2-way, 32B-line, 1 cycle latency
L2 cache	1MB, 32-way, 64B-line, 14-cycle latency
Main memory	100-cycle latency
Frequency	1 GHz
Technology	70 nm
Vdd	0.9V

4.4 Evaluations

4.4.1 Experimental Setup

This section evaluates the effectiveness of the proposed scheduling method. A simulator is developed based on the M5 simulator [50] and CACTI 4.2 [51, 102] for this evaluation. Detailed parameters are shown in Table 4.1. This CMP has four cores and two last-level L2 shared caches, each of which is shared by two cores. The power-aware dynamic cache partitioning mechanisms is applied to each of the L2 cache. In the simulations, four threads are simultaneously executed on the CMP, and one core executes one thread. Each simulation is done by 1 billion cycles.

The experimental procedure is as follows. Firstly, the proposed method has to know the working set sizes of the threads. In the evaluations, the working set sizes are assessed by the single-thread execution of the power-aware dynamic cache partitioning mechanisms with the parameters $(t_1, t_2) = (0.001, 0.005)$. The number of allocated ways obtained by each execution is used as the working set

Table 4.2: Experimented representative benchmarks.

Benchmark	Contents	Utility	Working set (way)
vpr_place	FPGA Circuit Placement	H	32
twolf	Place and Route Simulator	H	32
equake	Wave Propagation Simulation	S	18
mesa	3-D Graphics Library	S	11
wupwise	Quantum Chromodynamics	L	9
applu	Partial Differential Equation	L	7

size. The detailed definition is as follow.

$$W_p = \frac{\sum_{t=t_0}^{t_1} w(t)}{t_1 - t_0}. \tag{4.2}$$

Here,

$$W_p = \text{(the working set size of thread)}, \tag{4.3}$$

$$w(t) = \text{(the number of currently allocated ways at time } t\text{)}, \tag{4.4}$$

$$t_0 = \text{(the time at that the profiling starts)}, \tag{4.5}$$

$$t_1 = \text{(the time at that the profiling ends)}. \tag{4.6}$$

After the working set sizes of threads are assessed, the scheduling algorithm decides the assignment of the threads to the cores.

The threads used for the simulations are selected from the SPEC CPU2000 benchmark suite [85]. From the suite, the six representative benchmark programs are selected based on the characteristics of cache accesses, and used as the threads. By using these threads, thread combinations are representative of all thread combinations from the viewpoint of working set sizes. This process can reduce the number of experimented combinations without losing generality.

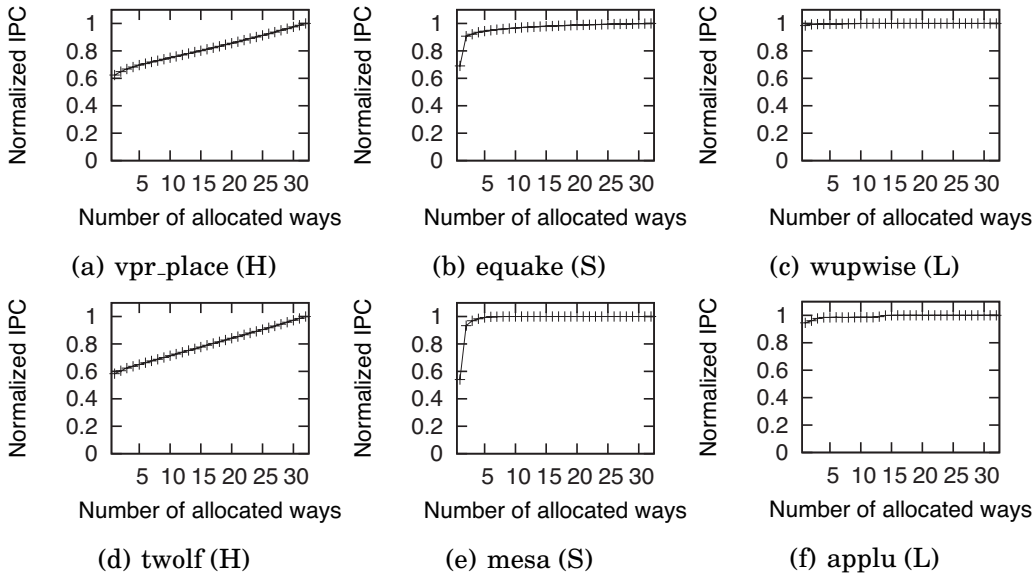


Figure 4.10: Utility graphs of the representative benchmarks.

In the selection, all the benchmarks are categorized into three classes based on their utility graphs [37]. As a result of the classification, the benchmark programs are classified into *high-utility*, *saturating-utility*, and *low-utility*. By selecting two representative benchmarks from each class, all the six benchmarks are selected, as shown in Table 4.2. Figure 4.10 shows the utility graphs of the selected benchmarks. In this figure, the performance is evaluated by IPC normalized by that when 32 ways are fully allocated to the thread. `VprPlace` and `Twolf` are categorized into high-utility benchmarks. High-utility benchmarks increase their performances gradually as the number of ways increases. `Mesa` and `Equake` are saturating-utility benchmarks. Their performances are drastically improved by increasing the number of ways. However, their performances are not improved when the number of allocated ways is large. `Wupwise` and `Apple` are low-utility benchmarks. The performances of low-utility benchmarks are not improved even if the number of ways increases. From the above

Table 4.3: Experimented benchmark combinations.

Combinations	Benchmarks			
HHSS	vpr_place	twolf	equake	mesa
HHSL	vpr_place	twolf	equake	wupwise
HHLL	vpr_place	twolf	wupwise	applu
HSSL	vpr_place	equake	mesa	wupwise
HSLL	vpr_place	equake	wupwise	applu
SSLL	equake	mesa	wupwise	applu

observations, it can be expected that high- , saturating- , and low-utility benchmarks have large, middle, and small working set sizes, respectively. This fact can be confirmed by the working set sizes in Table 4.2. Hence, the profiling method used in the proposed scheduling mechanism can appropriately estimate the working set sizes of these benchmarks.

With these benchmarks, six combinations of benchmarks are generated for performance evaluation. Although 15 combinations can be constructed from six benchmarks (${}_6C_4 = 15$), the number of benchmark combinations is reduced by removing the combinations having the same characteristics. For example, both of [`VprPlace`, `Equake`, `Wupwise`, `Applu`] and [`Twolf`, `Mesa`, `Wupwise`, `Applu`] consist of one high-utility, one saturating-utility, and two low-utility benchmarks. In this case, only the former benchmark combination is used for the evaluation, and the combination denoted `HSLL` (the thread combination of High, Sat, Low, and Low utility). Table 4.3 shows characteristics of six combinations and given labels for identification.

4.4.2 Evaluation Results of Overall Performance

To know the performance improvements by the proposed method, overall performances of CMPs are evaluated in this section. In the evaluations, Weighted

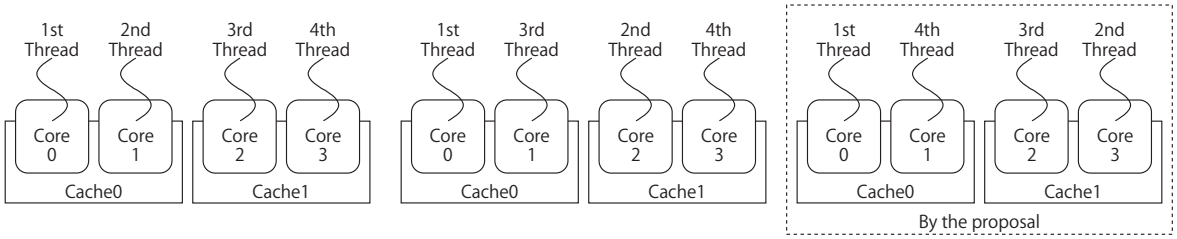


Figure 4.11: Possible scheduling cases on the CMP from the viewpoint of cache sharing of threads.

Speedup [55] is used as a performance metric.

Before showing the results, the detail of the evaluation method is described as follows. When four threads are executed on the CMP, three possible combinations of threads can be considered at the scheduling as shown in Figure 4.11, in which the combination selected by the proposed method is included. Among the three combinations, the combination with the highest performance is defined as the best case. Therefore, it is desirable that the best case is selected by the proposed method. The average performance over all the three combinations is called the average case. The performance of the average case is used as the expected performance if the scheduling method does not consider the cache sharing, e.g. a random scheduling method. The combination with the lowest performance is called the worst case. This combination induces capacity shortage. Finally, the combination selected by the proposed scheduling method is called the proposed case. In this evaluation, the proposed case is compared with the above three cases and evaluated.

Figure 4.12 shows the performances of all the cases. Four bars show the performance of the worst case, the average case, the best case, and the proposed case, respectively. The figure shows that the proposed case can get averagely 4% higher performance, and up to 8% compared with the worst case. Therefore, the proposed scheduling method can avoid the performance degradation caused

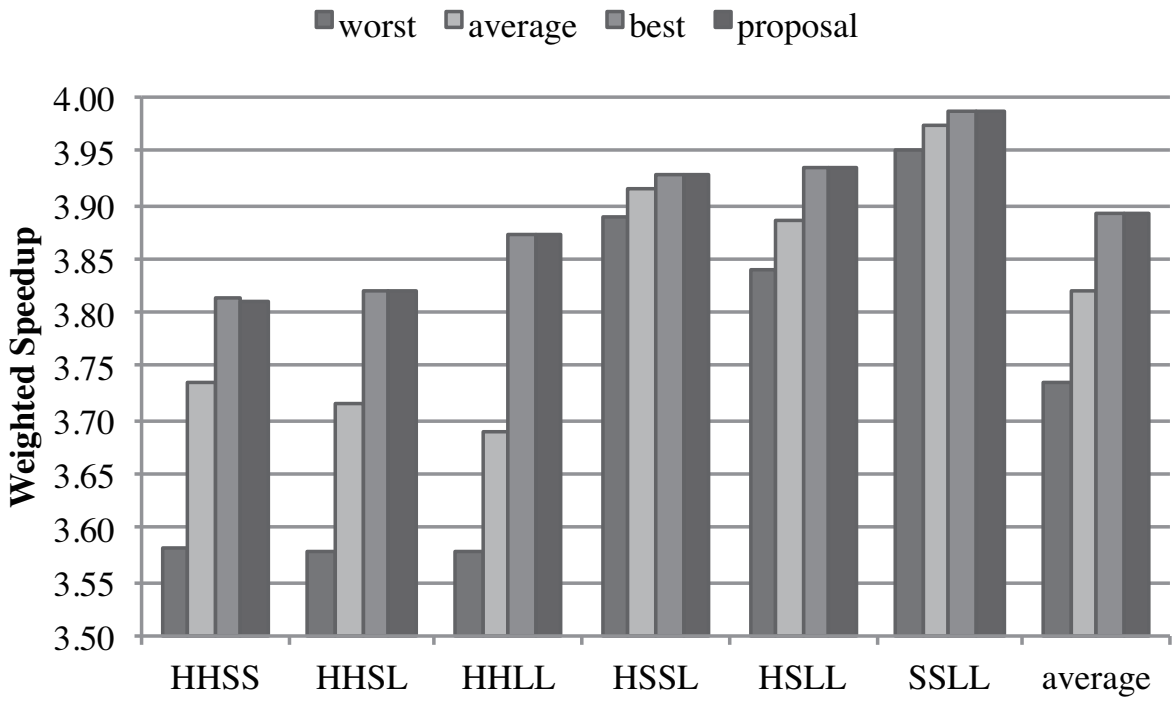


Figure 4.12: Comparing the performances of the worst, the average, the best, the proposed cases.

by selecting the worst case of scheduling. In addition, the performance of the proposed case is 1.9% higher than that of the average case. Furthermore, in the four thread combinations, the proposed case is equal to the best case. In the other two combinations, the performance difference between the proposed case and the best case is very small. Accordingly, the proposed method can assign threads to the cores to obtain the suboptimal performances.

The thread combination of `HHLL`, the performance improvement by the proposed case compared with the worst case of scheduling becomes the largest among the experimented thread combinations. This is because the combination consists of two threads with the largest working sets and two threads with the smallest working sets. In the worst case for this combination, the threads with large working sets share a same cache. Therefore, capacity shortage occurs

in the cache, and the performance degrades. On the other hand, in the proposed scheduling method, a thread with the largest working set and that with the smallest working set share a same cache. In this case, the capacity shortage is drastically alleviated, resulting in the performance improvement.

The combination of SSLL does not improve the performances even in any cases, compared with the other combinations. This is because that the combination includes the four threads with smallest working sets. In this combination, the power-aware dynamic cache partitioning mechanisms can allocate the ways sufficiently in any cases, and significant capacity shortage and its performance degradation do not occur.

In the combinations of HHSS and HHLL, the proposed case is not the best case. Hence, the proposed scheduling method cannot select the best assignment of threads to cores. This is because the high-utility benchmarks vpr_place and twolf included in these combinations have the same working set size, 32 ways, which is the number of ways when fully allocated. Hence, the proposed method cannot distinguish which the thread has a larger working set than the other thread. In addition, the objective of the working set assessment is originally to know the number of required ways, not to estimate the degree of the performance degradation when the number of ways is limited. Therefore, if the threads are coupled and actually executed, unpredictable performance degradations sometimes occur. In such cases, the proposed method cannot always select the best assignments of threads to cores. However, the performance difference between the best case and the proposed case is still little in HHSS and HHLL combinations, and the proposed scheduling method can achieve the suboptimal performances.

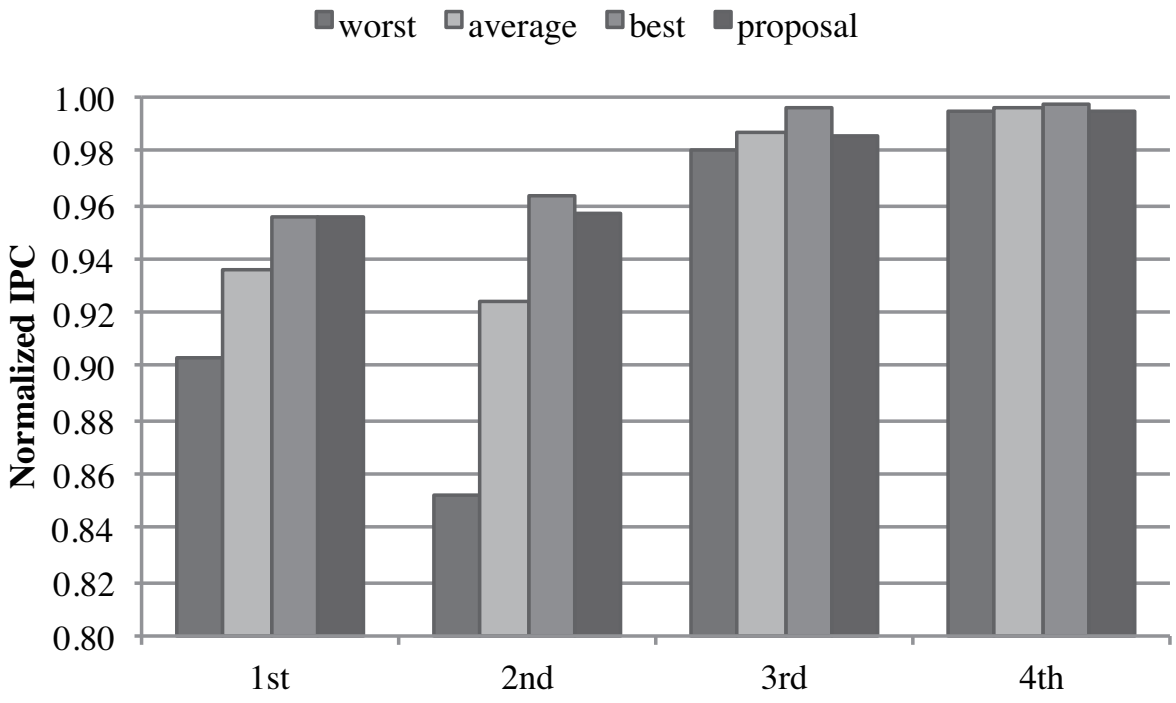


Figure 4.13: Average of normalized IPC of the threads in the combinations.

4.4.3 Evaluation Results of Individual Threads

To show the mechanism of the performance improvement by the proposed method in detail, this section evaluates performances of individual threads. Normalized IPC is used as the performance metric of a particular thread.

Figure 4.13 shows the performance of individual threads. The horizontal axis indicates the order in working set sizes. For example, *1st* means the threads with the first largest working sets in the combinations. In a similar fashion of Figure 4.12 in the previous section, Figure 4.13 includes the results of the worst case, the average case, the best case, and the proposed case.

For example, the proposed case achieves 2% and 3.5% performance improvements in the first and the second threads compared with the average case, respectively. Moreover, compared to the worst case, the proposed case realizes 5% and 12% performance improvements, respectively. From these results, it is

clear that the proposed method can improve the performances of the two threads with the larger working set sizes. In addition, performance improvements of the threads with the second largest working sets are larger than those of the threads with the first largest working sets. This is because the thread with the second largest working set cannot get enough cache capacity compared with the threads with the first largest working sets if the former and the latter share a same cache. The proposed method can solve this confused situation, and the performance improvement of the threads with the second largest working sets becomes larger.

On the other hand, the proposed case reduces 0.1% of performance on an average in the third and the fourth threads. Compared with the best case, the proposed method reduces 1% and 0.3% of performance on an average in the third and the fourth threads, respectively. This is because these threads become coupled with the threads with larger working sets by the proposed method. As a result, the number of allocated ways for the third and fourth threads decreases. However, the performance improvements of the first and the second threads are significant. Therefore, the overall performance of the CMP improves, even if the third and fourth threads slightly degrade their performances.

In addition, fairness of the performances among the threads improves by the proposed method. In the worst case, the performance degradations of the first and second threads are large. However, by the proposed method, their performances significantly improve while the performances of the third and fourth threads slightly degrade. As a result, the difference of the performances among the four threads becomes small.

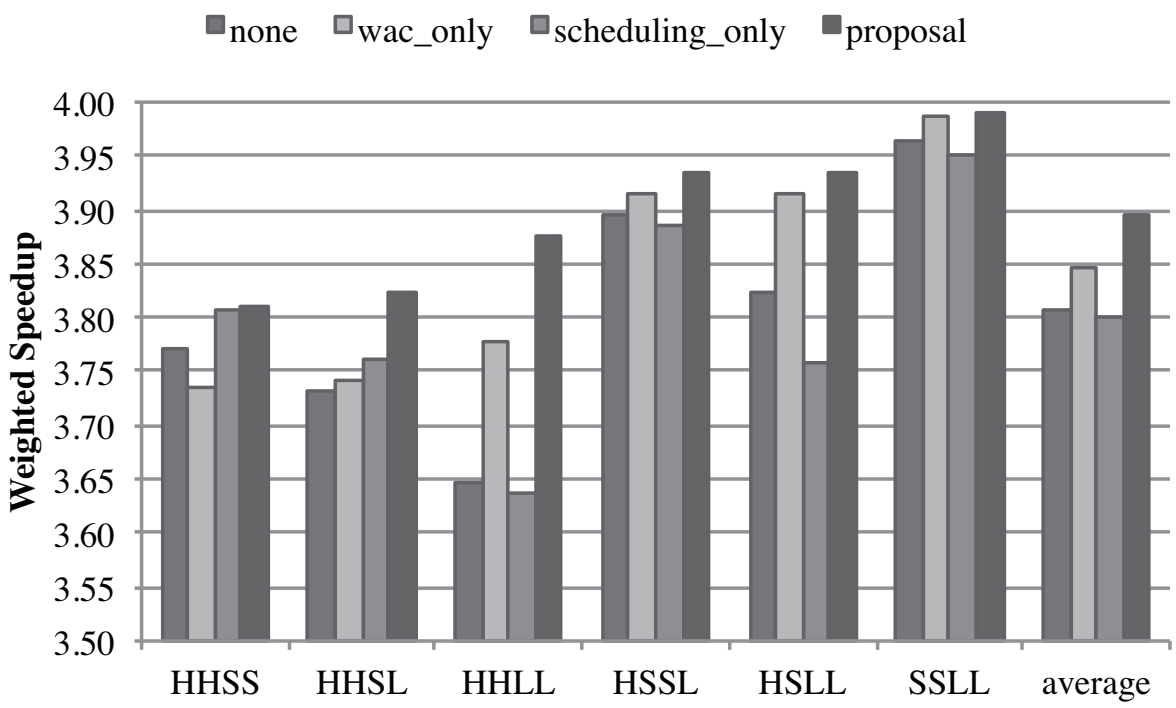


Figure 4.14: Comparing the performances of the average/proposed scheduling policies with/without the power-aware dynamic cache partitioning mechanism.

4.4.4 Performance Impact of Cooperation with the Power-Aware Dynamic Cache Partitioning

The proposed method can achieve the performance improvements by assuming the reduction of ITKOs by the power-aware dynamic cache partitioning mechanism. Furthermore, the better performances are obtained compared with the other scheduling method that does not consider both ITKOs and capacity shortage simultaneously. To show these facts, the performance impact of the power-aware dynamic cache partitioning mechanism for the proposed method is discussed.

Figure 4.14 shows the performances of the average case and the proposed case, with/without the power-aware dynamic cache partitioning mechanism. In the figure, four bars in each benchmark represent the average case without

the mechanism (*none*), the average case with the mechanism (*wac_only*), the proposed case without the mechanism (*scheduling_only*), and the proposed case with the mechanism (*proposal*).

Figure 4.14 indicates that the highest performance is achieved by the proposed case with the mechanism. This indicates the effectiveness of the proposed scheduling method. In addition, comparing *none* and *scheduling_only*, there are no performance impacts of the proposed scheduling method without the mechanism. Focusing on each thread combination, the performance impact of the proposed scheduling method is not steady without the power-aware dynamic cache partitioning mechanism. For example, the proposed scheduling method without the mechanism can achieve the performance improvements in `HHSS` and `HHSL`. On the other hand, the proposed method degrades the performances in `HHLL`, `HSSL`, `HSLL`, and `SSLL`. These results indicate that the performance degradation by ITKOs occurs without the mechanism, even in the proposed case that considers capacity shortage. Therefore, it is important to consider both capacity shortage and ITKOs in inter-thread cache conflicts.

4.4.5 Energy Impact of the Thread Scheduling Method

To understand the energy impact of the proposed method, the energy consumption by the cache memories is shown in Figure 4.15. The metric of energy consumption is normalized energy. Here, energy consumption with the energy reduction control is normalized by that without the energy reduction control. The vertical axis of the figure shows normalized energy. Four bars in each thread combination represent the worst, average, best, and proposed cases from the viewpoint of energy consumption, respectively.

From Figure 4.15, it is observed that the energy reduction control of the

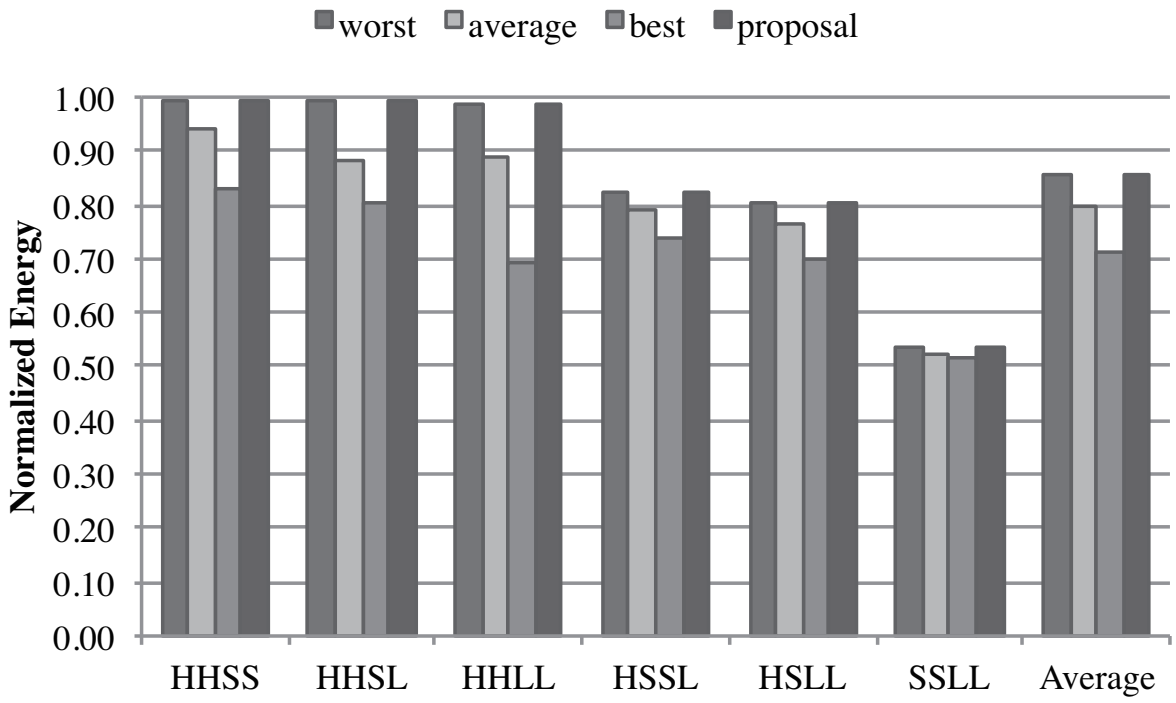


Figure 4.15: Energy consumption of the cache memories.

power-aware dynamic cache partitioning mechanism can reduce the energy even in the proposed case. However, compared with the other cases, the energy reduction of the proposed case is not significant, and equal to the worst case. This is because the proposed method contributes to an increase in cache utilization. By the proposed method, threads with larger working sets can get more capacity by sharing a same cache with the threads with smaller working sets. In this case, the power-aware dynamic cache partitioning mechanism increases the number of allocated ways for the threads with larger working sets. As a result, the energy consumption increases.

Observing the overall trends in Figure 4.15, it is clear the thread combinations of `HHSS`, `HHSL`, and `HHLL` do not reduce the energy consumption significantly. These combinations need a large cache capacity because the threads with large working sets are included. Hence, the number of allocated ways cannot be

decreased by the energy reduction control of the power-aware dynamic cache partitioning mechanism. On the other hand, the proposed method reduces the energy consumption of the combinations of `HSSL`, `HSLL`, and `SSLL`, especially by up to 86% in the combination of `SSLL`. These combinations include many threads with small working sets. Hence, a lot of ways that do not contribute to the performance can be inactivated. As a result, the energy consumption decreases.

From all the discussions above, it is clear the proposed method can achieve performance improvement because the method can increase the number of allocated ways for the threads with large working sets. On the other hand, an increase of the number of allocated ways causes the additional energy consumption. Therefore, the proposed method can improve the utilization of the cache memories at the cost of the energy growth. However, the energy consumption is still reduced even in the proposed method compared with the case without the energy reduction control of the power-aware dynamic cache partitioning mechanisms. Hence, the cooperation between the proposed method and the power-aware dynamic cache partitioning mechanisms can also contribute the energy reduction.

4.5 Conclusions

CMPs have become major in modern microprocessors. CMPs can simultaneously execute multiple threads using multiple cores on a chip. However, simultaneous execution of multiple threads causes inter-thread cache conflicts, which are ITKOs and capacity shortage. If inter-thread cache conflicts occur, CMPs cannot exploit their potential and the performances are limited. The power-aware dynamic cache partitioning mechanism can reduce ITKOs and improve the performances of multi-thread execution on CMPs. However, if multiple threads with large working sets share a same cache on CMPs, capacity shortage occurs in the cache, resulting in performance degradation.

To overcome this problem, the thread scheduling based on working set assessment is proposed in this chapter. The proposed method supposes that there are multiple shared caches with the power-aware dynamic cache partitioning mechanisms. In this case, the thread combinations sharing a single cache can be flexibly changed. Hence, the proposed method changes the thread combinations sharing a cache so that the capacity shortage problem is alleviated. At first, the working set sizes of threads are assessed using the locality assessment metric. Second, the assignments of the threads to the cores are decided by the scheduling algorithm based on working set sizes. The algorithm prevents the threads with the largest working sets from sharing a same cache. As a result, the thread scheduling method alleviates inter-thread cache conflicts by capacity shortage. The simulation results show the average performance of the proposed method is 1.9% higher than that of the average of all the case of scheduling, and 8.1% higher than that of the worst case of scheduling.

Chapter 5

Conclusions

The advance in CMOS process technology and the innovations on computer architecture design are important for microprocessors to achieve high performance and low energy consumption. However, it has been becoming difficult for the CMOS process technology to contribute to performance improvement and energy reduction. Under this situation, it is required to further innovate computer architecture design in the future.

This dissertation focuses on cache memories. Although cache memories are important for performance improvement of microprocessors, their energy consumption and inter-thread cache conflicts on CMPs are not ignorable. To overcome these problems, the power-aware dynamic cache partitioning mechanism [24, 25, 26] is proposed. However, one of the main causes of inter-thread cache conflicts, capacity shortage, is still the problem on the mechanism. Therefore, the objective of this dissertation is to improve utilization efficiency of cache resource, resulting in alleviation of the capacity shortage problem and achievement of high performance and low power cache memories. To this end, this dissertation has proposed three approaches, and incorporated them into a hardware-software co-designed cache memory system.

In Chapter 2, estimation accuracy of the working set sizes of the threads in the power-aware dynamic cache partitioning mechanism has been discussed. It is important to accurately estimate the number of ways required by a thread, because inaccurate estimation causes performance degradation and energy growth. Chapter 2 figures out that exceptional cache access behaviors cause inaccurate allocation of the ways. To reduce the negative effect of the exceptional cache access behaviors, Chapter 2 proposed a voting-based working set assessment scheme. The proposed scheme decides the number of allocated ways based on majority voting of the finer-grain working set assessment than the previous scheme. The evaluation results show that the proposed scheme can decrease the energy consumption by up to 24% and 10% on average without significant performance degradation. Consequently, by reducing the effect of exceptional cache access behaviors, the power-aware dynamic cache partitioning mechanism can reduce the energy consumption.

In Chapter 3, data management for the power-aware dynamic cache partitioning mechanism has been examined. In cache memories, dead-on-fill blocks may occupy the cache capacity, even though they do not contribute to performance. Due to the existence of them, the power-aware dynamic cache partitioning mechanism must allocate a large cache size to threads to maintain both reusable and dead-on-fill blocks in the cache. To evict dead-on-fill blocks from the cache, Chapter 3 proposed the dynamic LRU-K insertion policy. This policy evicts dead-on-fill blocks earlier, and helps the power-aware dynamic cache partitioning mechanism achieve efficient allocation of the cache capacity. By applying the proposed policy to the cache with the power-aware dynamic cache partitioning mechanism, energy consumption of the cache is reduced by up to

30% and 6% on an average without significant performance degradation. Therefore, Chapter 3 revealed that the proposed policy can evict dead-on-fill blocks earlier, and as a result, the power-aware dynamic cache partitioning mechanism can improve the efficiency.

In Chapter 4, thread scheduling to avoid inter-thread cache conflicts has been considered. The simultaneous execution of multiple threads with large working set sizes cause capacity shortage, resulting in performance degradation. In this case, the CMP cannot achieve performance improvement as expected. To against this, Chapter 4 proposed a thread scheduling method based on working set assessment. The proposed method can avoid capacity shortage by scheduling threads based on the working set sizes of the threads, and helps the power-aware dynamic cache partitioning mechanisms allocate more ways to the threads with larger working set sizes. The evaluation results show that the performance by the proposed method is 1.9% higher than that of the average of all the cases, and 8.1% higher than that of the worst case of scheduling. Consequently, Chapter 4 indicated that the thread scheduling method can avoid capacity shortage and improve the performance.

From the above, this dissertation concludes that the proposed three approaches can contribute to performance improvement and energy reduction of the hardware-software co-designed cache memory system.

Finally, long-term future work is described as follows.

- To show the generality of the scheme proposed in Chapter 2, the scheme should be applied to other dynamic cache resizing mechanisms.

- The data management policy proposed in Chapter 3 does not consider dead blocks. The data management policy that reduces not only dead-on-fill

blocks but also dead blocks should be considered to achieve higher utilization efficiency. Furthermore, to reduce dead-on-fill blocks more aggressively, cooperation with cache bypassing should be considered.

- To make the thread scheduling method proposed in Chapter 4 more practical, dynamic thread scheduling and its implementation should be considered.

Bibliography

[1] Gordon E. Moore. Cramming More Components Onto Integrated Circuits. *Electronics*, pages 114–117, January 1965.

[2] Steven R. Kunkel and James E. Smith. Optimal Pipelining in Supercomputers. *ACM SIGARCH Computer Architecture News*, 14(2):404–411, June 1986.

[3] Robert M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.

[4] James E. Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148, May 1981.

[5] Robert T. Short and Henry M. Levy. A Simulation Study of Two-Level Caches. *ACM SIGARCH Computer Architecture News*, 16(2):81–88, May 1988.

[6] Shekhar Borkar. Design Challenges of Technology Scaling. *IEEE Micro*, 19(4):23–29, 1999.

[7] Wei Huang, Shougata Ghosh, Siva Velusamy, Karthik Sankaranarayanan, Kevin Skadron, and Mircea R. Stan. HotSpot: A Compact Thermal Modeling Methodology for Early-Stage VLSI Design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(5):501–513, May 2006.

[8] Tilak Agerwala and Siddhartha Chatterjee. Computer Architecture: Challenges and Opportunities for the Next Decade. *IEEE Micro*, 25(3):58–69, May 2005.

[9] Mircea R. Stan and Kevin Skadron. Power-Aware Computing. *IEEE Computer*, 36(12):35–38, December 2003.

[10] Keith Boland and Apostolos Dollas. Predicting and Precluding Problems with Memory Latency. *IEEE Micro*, 14(4):59–67, 1994.

[11] Srikanth T. Srinivasan and Alvin R. Lebeck. Load Latency Tolerance In Dynamically Scheduled Processors. In *Proceedings of the 31st annual ACM/IEEE International Symposium on Microarchitecture*, pages 148–159, November 1998.

[12] Wm. A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, March 1995.

[13] Nam Sung Kim, Krisztian Flautner, David Blaauw, and Trevor Mudge. Circuit and Microarchitectural Techniques for Reducing Cache Leakage Power. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(2):167–184, February 2004.

[14] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of 27th International Symposium on Computer Architecture*, volume 28, pages 83–94. Acm, 2000.

[15] Srilatha Manne, Artur Klauser, and Dirk Grunwald. Pipeline Gating: Speculation Control For Energy Reduction. In *Proceedings. 25th Annual International Symposium on Computer Architecture*, pages 132–141. IEEE Comput. Soc, 1998.

[16] James Montanaro, Richard T. Witek, Krishna Anne, Andrew J. Black, Elizabeth M. Cooper, Dniel W. Dobberpuhl, Paul M. Donahue, Jim Eno, Gregory W. Hoeppner, David Kruckemyer, Thomas H. Lee, Peter C. M. Lin, Liam Madden, Daniel Murray, Mark H. Pearce, Sribalan Santhanam, Kathryn J. Snyder, Ray Stehpany, and Stephen C. Thierauf. A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. *IEEE Journal of Solid-State Circuits*, 31(11):1703–1714, November 1996.

[17] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–480, 2009.

[18] David W Wall. Limits of Instruction-Level Parallelism. *ACM SIGARCH Computer Architecture News*, 19(2):176–188, 1991.

[19] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-Machine Multicomputer. In *Proceedings of the 28th Annual International Symposium on*

Microarchitecture, pages 146–156. IEEE Comput. Soc. Press, 1995.

[20] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. A Single-Chip Multiprocessor. *IEEE Computer*, 30(9):79–85, 1997.

[21] Jack L. Lo, Joel S. Emer, Henry M. Levy, Rebecca L. Stamm, Dean M. Tullsen, and Susan J. Eggers. Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading. *ACM Transactions on Computer Systems*, 15(3):322–354, August 1997.

[22] Basem A. Nayfeh and Kunle Olukotun. Exploring the Design Space for a Shared-Cache Multiprocessor. In *Proceedings of 21 International Symposium on Computer Architecture*, pages 166–175. IEEE Comput. Soc. Press, 1994.

[23] Joshua Kihm, Alex Settle, Andrew Janiszewski, and Daniel A. Connors. Understanding the Impact of Inter-Thread Cache Interference on ILP in Modern SMT Processors. *The Journal of Instruction-Level Parallelism*, 7:1–28, 2005.

[24] Hiroaki Kobayashi, Isao Kotera, and Hiroyuki Takizawa. Locality Analysis to Control Dynamically Way-Adaptable Caches. *ACM SIGARCH Computer Architecture News*, 33(3):25–32, 2005.

[25] Isao Kotera, Ryusuke Egawa, Hiroyuki Takizawa, and Hiroaki Kobayashi. A Power-Aware Shared Cache Mechanism Based on Locality Assessment of Memory Reference for CMPs. In *Proceedings of ACM PACT'07 Workshop on Memory Performance, Dealing with Applications, Systems and Architectures (MEDEA'07)*, pages 121–127, 2007.

[26] Isao Kotera, Kenta Abe, Ryusuke Egawa, Hiroyuki Takizawa, and Hiroaki Kobayashi. Power-Aware Dynamic Cache Partitioning for CMPs. *Transaction on High-Performance Embedded Architectures and Compilers*, 3(2):149–167, 2008.

[27] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T. N. Vijaykumar. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design - ISLPED '00*, pages 90–95, New York, New York, USA, August 2000. ACM Press.

[28] Mainak Chaudhuri. Pseudo-LIFO: The Foundation of a New Family of Replacement Policies for Last-level Caches. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 401–412. ACM, 2009.

[29] David H Albonesi. Selective Cache Ways: On-Demand Cache Resource Allocation. In *Proceedings of 32nd Annual International Symposium on Microarchitecture*, pages 248–259, 1999.

[30] Se-Hyun Yang, Michael D. Powell, Babak Falsafi, and T. N. Vijaykumar. Exploiting Choice in Resizable Cache Design to Optimize Deep-Submicron Processor Energy-Delay. In *Proceedings of The Eighth International Symposium on High-Performance Computer Architecture*, 2002.

[31] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T. N. Vijaykumar. Reducing Leakage in a High-Performance Deep-Submicron Instruction Cache. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(1):77–89, February 2001.

[32] Gilles Pokam and Francois Bodin. Energy-efficiency potential of a phase-based cache resizing scheme for embedded systems. In *Eighth Workshop on Interaction between Compilers and Computer Architectures, 2004. INTERACT-8 2004.*, pages 53–62. IEEE, 2004.

[33] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 30, pages 211–222, December 2002.

[34] Alessandro Bardine, Manuel Comparetti, Pierfrancesco Foglia, Giacomo Gabrielli, and Cosimo Antonio Prete. Way adaptable D-NUCA caches. *International Journal of High Performance Systems Architecture*, 2(3/4):215, August 2010.

[35] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proceedings of 13th International Conference on Parallel Architecture and Compilation Techniques*, pages 111–122, 2004.

[36] Gookwon E. Suh, Larry Rudolph, and Srinivas Devadas. Dynamic Partitioning of Shared Cache Memory. *Journal of Supercomputing*, 28(1):7–26, 2004.

[37] Moinuddin K. Qureshi and Yale N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432, 2006.

[38] Alex Settle, Daniel A. Connors, Enric Gibert, and Antonio Gonzáles. A Dynamically Reconfigurable Cache for Multithreaded Processors. *Journal of Embedded Computing*, 2(2):221–233, 2006.

[39] Shekhar Srikantaiah, Mahmut Kandemir, and Mary Jane Irwin. Adaptive Set Pinning: Managing Shared Caches in Chip Multiprocessors. *ACM SIGARCH Computer Architecture News*, 36(1):135–144, 2008.

[40] Tsung Lee and Hsiang-Hua Tsou. A novel cache mapping scheme for dynamic set-based cache partitioning. In *Proceedings of 2009 IEEE Youth Conference on Information, Computing and Telecommunication*, pages 459–462. IEEE, September 2009.

[41] Jichuan Chang and Gurindar S. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In *Proceedings of the 21st annual international conference on Supercomputing - ICS '07*, page 242, New York, New York, USA, June 2007. ACM Press.

[42] Haakon Dybdahl and Per Stenstrom. An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors. In *Proceedings of 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 2–12. IEEE, February 2007.

[43] Jaehyuk Huh, Changkyu Kim, Hazim Shafi, Lixin Zhang, Doug Burger, and Stephen W. Keckler. A NUCA Substrate for Flexible CMP Cache Sharing. *IEEE Transactions on Parallel and Distributed Systems*, 18(8):1028–1040, August 2007.

[44] Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, and Mateo Valero. Online Prediction of Applications Cache Utility. In *Proceedings of the*

International Conference on Embedded Computer Systems: Architecture, Modeling and Simulation, pages 169–177, 2007.

[45] Richard West, Puneet Zaroo, Carl A. Waldspurger, and Xiao Zhang. Online Cache Modeling for Commodity Multicore Processors. *ACM SIGOPS Operating Systems Review*, 44(4):19–29, December 2010.

[46] David K. Tam, Reza Azimi, Livio B. Soares, and Michael Stumm. RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimization. In *ACM SIGPLAN Notices*, volume 44, pages 121–132, February 2009.

[47] Richard L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[48] Mark D. Hill and Alan Jay Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, 38(12):1612–1630, 1989.

[49] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 340–351, 2005.

[50] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006.

[51] Steven J E Willton and Norman P. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, 1996.

[52] Shyamkumar Thoziyoor, Naveen Muralimanohar, and Norman P Jouppi. CACTI 5.0. Technical report, HP Labs, 2007.

[53] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. CACTI 6.0: A Tool to Model Large Caches. Technical Report HPL-2009-85, HP Labs, 2009.

[54] John L. Henning. SPEC CPU2006 Benchmark Descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[55] Allan Snavely and Dean M Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreading Processor. *ACM SIGPLAN Notices*, 35(11):234–244, 2000.

[56] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel Emer. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). In *Proceedings of the 37th annual International Symposium on Computer Architecture*, volume 38, pages 60–71, 2010.

[57] R. Karedla, J. S. Love, and B. G. Wherry. Caching Strategies to Improve Disk System Performance. *Computer*, 27(3):38–46, March 1994.

[58] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 381–391, 2007.

[59] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely Jr., and Joel Emer. Adaptive Insertion Policies for Managing Shared Caches. In *Proceedings of the 17th International Conference on Parallel Architecture and Compilation Techniques*, pages 208–219, 2008.

[60] Samira Khan and Daniel A. Jimenez. Insertion Policy Selection using Decision Tree Analysis. In *Proceedings of IEEE International Conference on Computer Design*, pages 106–111. IEEE, October 2010.

[61] Inside the Intel Itanium 2 processor: an Itanium Processor Family Member for Balanced Performance over a Wide Range of Applications. In *White paper, Hewlett Packard*, number July, 2002.

[62] UltraSPARC T2 Supplement to the UltraSPARC Architecture 2007. Technical report, Sun Microsystems, Inc., 2007.

[63] Yuejian Xie and Gabriel H. Loh. PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches. *ACM SIGARCH Computer Architecture News*, 37(12):174–183, 2009.

[64] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Transactions on Computers*, 50(12):1352–1361, December 2001.

[65] Kaushik Rajan and Govindarajan Ramaswamy. Emulating Optimal Replacement with a Shepherd Cache. In *Proceedings of 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 445–454. Ieee, December 2007.

[66] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[67] Wayne A. Wong and Jean-Loup Baer. Modified LRU Policies for Improving Second-level Cache Behavior. In *Proceedings of Sixth International Symposium on High-Performance Computer Architecture*, pages 49–60. IEEE Comput. Soc, 2000.

[68] Nirmod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of FAST '03: 2nd USENIX Conference on File and Strage Techniques*, pages 115–130, 2003.

[69] Ranjith Subramanian, Yannis Smaragdakis, and Gabriel H. Loh. Adaptive Caches: Effective Shaping of Cache Behavior to Workloads. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 385–396. IEEE, December 2006.

[70] Gabriel H. Loh. Extending the Effectiveness of 3D-Stacked DRAM Caches with an Adaptive Multi-Queue Policy. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 201–212, 2009.

[71] Douglas C. Burger, James R. Goodman, and Alain Kagi. The Declining Effectiveness of Dynamic Caching for General-Purpose Microprocessors. Technical report, Univ. of Wisconsin-Madison Computer Sciences Dept., 1995.

[72] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. Cache Decay: Exploiting Generational Behavier to Reduce Cache Leakage Power. *ACM SIGARCH Computer Architecture News*, 29(2):240–251, 2001.

[73] Jaume Abella, Antonio González, Xavier Vera, and Michael F. P. O'Boyle. IATAC: A Smart Predictor to Turn-off L2 Cache Lines. *ACM Transactions on Architecture and Code Optimization*, 2(1):55–77, March 2005.

[74] Huiyang Zhou, M.C. Toburen, Eric Rotenberg, and T.M. Conte. Adaptive mode control: A static-power-efficient cache design. *ACM Transactions on Embedded Computing Systems (TECS)*, 2(3):347–372, 2003.

[75] Tomoaki Ukezono and Kiyofumi Tanaka. Reduction of leakage energy in low level caches. In *Proceedings of International Conference on Green Computing*, pages 537–544. IEEE, August 2010.

[76] Krisztian Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy Caches: Simple Techniques for Reducing Leakage Power. In *Proceedings of 29th Annual International Symposium on Computer Architecture*, pages 148–157, 2002.

[77] Masamichi Takagi and Kei Hiraki. Inter-Reference Gap Distribution Replacement : An Improved Replacement Algorithm for Set-Associative Caches. In *Proceeding of ICS '04 Proceedings of the 18th annual international conference on Supercomputing*, pages 20–30, 2004.

[78] Mazen Kharbutli and Yan Solihin. Counter-Based Cache Replacement and Bypassing Algorithms. *IEEE Transactions on Computers*, 57(4):433–447, April 2008.

[79] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing

Cache Efficiency. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, volume 36, pages 222–233. IEEE, November 2008.

[80] Gary Tyson, Matthew Farrens, John Matthews, and Andrew R. Pleszkun. A Modified Approach to Data Cache Management. In *Proceedings of the 28th annual International Symposium on Microarchitecture*, pages 93–103, 1995.

[81] Teresa L. Johnson, Daniel A. Connors, Matthew C. Merten, and Wen-Mei W. Hwu. Run-Time Cache Bypassing. *IEEE Transactions on Computers*, 48(12):1338–1354, 1999.

[82] Andreas Sandberg, David Eklöv, and Erik Hagersten. Reducing Cache Pollution Through Detection and Elimination of Non-Temporal Memory Accesses. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, number November, 2010.

[83] Family 10h AMD Phenom II Processor Product Data Sheet. *Technical Documents of Advanced Micro Devices*, 2009.

[84] David Eklov and Erik Hagersten. StatStack: Efficient Modeling of LRU Caches. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems & Software*, pages 55–65. IEEE, March 2010.

[85] John L Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *IEEE Computer*, 33(7):28–35, 2000.

[86] Sébastien Hily and André Seznec. Contention on 2nd Level Cache May Limit the Effectiveness of Simultaneous Multithreading. Technical Report PI-1086, INRIA, 1997.

[87] Radhika Thekkath and Susan J. Eggers. The Effectiveness of Multiple Hardware Contexts. *ACM SIGOPS Operating Systems Review*, 28(5):328–337, December 1994.

[88] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, and P Sadayappan. Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems. In *Proceedings of IEEE 14th International Symposium on High-Performance Computer Architecture*, pages 367–378, 2008.

[89] Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, and Mateo Valero. Explaining Dynamic Cache Partitioning Speed Ups. *IEEE Computer Architecture Letters*, 6(1):1–4, 2007.

[90] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous Multithreading: A Platform for Next-Generation Processors. *IEEE Micro*, 17(5):12–19, 1997.

[91] Joshua L. Kihm and Daniel A. Connors. Implementation of Fine-Grained Cache Monitoring for Improved SMT Scheduling. In *Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 326–331, 2004.

[92] Alex Settle, Joshua Kihm, Andrew Janiszewski, and Daniel A. Connors. Architectural Support for Enhanced SMT Job Scheduling. In *Proceedings*

of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT'04), pages 63–73, 2004.

[93] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro*, 28(3):54–66, 2008.

[94] Mohammad Banikazemi, Dan Poff, and Bulent Abali. PAM: a novel performance/power aware meta-scheduler for multi-core systems. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[95] Gookwon E. Suh, Larry Rudolph, and Srinivas Devadas. Effects of Memory Performance on Parallel Job Scheduling. In *Proceedings of the 7th International Workshop on JobScheduling Strategies for Parallel Processing*, pages 116–132, June 2001.

[96] Gookwon E. Suh, Srinivas Devadas, and Larry Rudolph. A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. In *Proceedings of The Eighth International Symposium on High-Performance Computer Architecture*, pages 117–128, 2002.

[97] Ali El-Moursy, Rajeev Garg, David H. Albonesi, and Sandhya Dwarkadas. Compatible Phase Co-Scheduling on a CMP of Multi-Threaded Processors. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium IPDPS 2006*, 2006.

[98] Francisco J. Cazorla, Peter M. W. Knijnenburg, Rizos Sakellariou, Enrique Fernandez, Alex Ramirez, and Mateo Valero. Predictable Performance in SMT Processors: Synergy between the OS and SMTs. *IEEE Transactions on Computers*, 55(7), 2006.

[99] Steven E. Raasch and Steven K. Reinhardt. The Impact of Resource Partitioning on SMT Processors. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)*, page 15, 2003.

[100] Intel Core 2 Quad Processor New version now available based on Intel's 45nm technology. In *Intel Product Brief*. Intel Corporation, 2007.

[101] Quad-Core Intel Xeon Processor 5300 Series. In *Intel Product Brief*, 2007.

[102] David Tarjan and Shyamkumar Thoziyoor. CACTI 4.0. Technical report, HP Labs, 2006.