Performance and Power Aware Cache Memory Architectures

TOHOKU UNIVERSITY
Graduate School of Information Sciences

Performance and Power Aware Cache Memory Architectures

**(性能と消費電力を考慮した
キャッシュメモリアーキテクチャに関する研究)**

A dissertation submitted for the degree of
Doctor of Philosophy (Information Sciences)

Department of Computer and Mathematical Sciences

by

Isao KOTERA

January 16, 2009

Performance and Power Aware Cache Memory Architectures

Isao Kotera

Abstract

In recent years, due to the dramatic evolution of the CMOS technology as Moore's law, the amount of hardware resource on a chip and the switching speed of transistors have progressed exponentially. Hereby, computer architects have been able to design high-speed and multi-functional general-purpose processors. Meanwhile, the power consumption due to driving a vast amount of hardware with high clock frequency is becoming a critical problem in high-performance processor design. The increase in the power consumption causes a high power density, and it brings thermal problem, high operating cost and performance degradation of the processors to computer architects. However, needs for the higher performance processors are still strong in several fields. Therefore, there is a strong demand for processor technology to achieve both high-performance and low-power. Especially, mobile devices require such a technology because their sizes and power supply are severely restricted. However, it is basically difficult to achieve both high-performance and low-power because there is a trade-off between them. As a general trend, high-end processors need high power, although they provide higher performance than low-power processors. To realize high-performance and low-power processors from a viewpoint of computer architecture designs, a mechanism, which can improve power-efficiently of processors has been expected.

In recent processors, the cache memory is not only an important factor that determines the processor performance but also one of the major power hungry elements of processors. Especially, a substantial part of the static power is consumed in the large on-chip cache memory. Hence, the reduction in power consumption by the caches is strongly desired to realize a high-performance and low-power processor. Therefore, this dissertation focuses on the cache memories to realize a high-performance and low-power processor.

As well as power reduction, further performance improvement is another important research issue to realize a high-performance and low-power processor. Although a single core processor had been a major architecture until recent years, it has become difficult to improve the performance by increasing the clock frequency and exploiting the instruction level parallelism (ILP), because there are a lot of technology obstacles such as a limit of the semiconductor manufacturing technology, a limit of the ILP, a limit of the design cost and so on. In response to this situation, a chip multi-processor (CMP) has recently become the common architecture to increase the performance. The CMP architecture, which has multiple processing cores on a single die, can improve the performance by simultaneous execution of multiple workloads based on the thread label parallelism (TLP).

Here, an on-chip shared cache, which generally installed in CMP, plays a key role in efficiently executing multiple workloads. However, in the shared cache, there is a shared cache conflict problem, in which one core cannot use the enough cache space to keep performance when multiple cores execute independent programs. The performance impact of each core due to the shared cache conflict depends on the cache access characteristic of the program executed on the each care, and therefore the performances of cores unfairly degrade. Consequently, it is necessary for CMP to avoid the shared cache conflict problem to improve the performance by exploiting TLP.

For these power consumption and the shared cache conflict problems in processor design, a mechanism to save cache power consumption while avoiding the shared cache conflict is required for future high-performance and low-power CMP processors. The objective of this dissertation is to establish a cache management methodology to achieve both of low-power and high-performance. To solve these problems, this dissertation develops a basic strategy that cache mechanism allocates the capacity in keeping performance according to the cache requirements of applications. The power efficiency improves by turning off the power supply to the unused cache area. The cache requirement metric is also useful to avoid the shared cache conflict problem; if the cache requirement of each core is obtained, a shared cache can be appropriately partitioned into multiple parts, each of which is exclusively accessed only by one core.

This dissertation considers following three methods: a cache requirement metric in order to assess minimum cache capacity to keep the performance of applications, a cache power control mechanism to supply power to cache area depending on the cache requirement and a cache control mechanism to avoid the shared cache conflicts in CMP. By considering these methods, this dissertation establishes a cache management methodology to improve the power efficiency, while keeping high-performance.

Firstly, Chapter 2 quantitatively analyzes cache access behaviors and introduces a cache requirement metric based on the cache access locality in order to assess a minimum cache capacity to keep the performance of application running. To analyze the cache access behaviors, the stack distance profiling method is introduced. The stack distance profiling is a method, which measures data reusability of cache lines by monitoring an LRU stack. It is possible to assess the temporal access locality and the degree of the cache requirement by analyzing stack distance distributions (SDDs), which are derived from the stack distance profiling. Additionally, to analyze and quantify the characteristic of SDD, SDD has to be approximately modeled. This dissertation hypothesizes that the SDDs obey Zipf's law, and then verify the hypothesis against the real application. The verification indicates that almost all applications have sufficiently large correlation coefficients between SDDs and Zipf's distributions. When the SDDs obey Zipf's distributions, the parameter in Zipf's law indicates characteristics of the cache access locality. To handily obtain the cache access locality and to utilize it for the evaluation of the cache requirement, this dissertation proposes and verifies the cache requirement metric D. Based on this approach, this chapter establishes the cache requirement metric for the high-performance and low-power cache mechanism. This cache requirement metric plays an important role for the cache management methodology proposed in this dissertation.

In Chapter 3, a cache power control mechanism, named dynamic way-adaptable cache is introduced to reduce the power consumption with keeping the performance for a single workload. The mechanism monitors the cache requirement introduced in Chapter 2, and then it appropriately controls the activated cache area using the cache requirement. The proposed cache mechanism defines either increasing or decreasing the number of activated cache ways based on the cache requirement metric for up-sizing or down-sizing the cache. To realize appropriate cache control, the assessment of the cache requirement uses the local and global information of the cache requirement. The local assessment of cache access behavior assesses the absolute magnitude of the cache requirement in a fixed interval and estimates up-sizing or down-sizing by comparing the cache requirement with two thresholds. Besides, the global assessment using an n-bit state machine is considered to avoid aggressive reactions to the quick change of the cache requirement in a short period. This state machine determines the up-sizing or down-sizing control. When the estimation is up-sizing, the power control hardware selects one from inactivated ways and activates it. When the estimation is down-sizing, the hardware selects one from activated ways and inactivates it after write back cached data to the main memory. Experimental results indicate that the proposed cache mechanism achieves the stable cache control to find

an appropriate trade-off between activated cache area and its achieved performance. The local assessment of the cache requirement with two thresholds achieves performance-oriented control with more activated ways to keep the higher performance, and lower-associativity-oriented control at the expense of the performance. In global assessment, an n-bit asymmetric state machine works well to appropriately control the number of activated ways in the case of the benchmarks with highly-irregular access behaviors. Accordingly, this chapter establishes the cache power control methodology to supply power to cache area depending on the cache requirement.

In Chapter 4, the way-adaptable cache mechanism introduced in the previous chapter is extended for a shared cache of a two-core CMP. To effectively execute multiple workloads in the case where multiple cores executing independent programs share a cache, the shared cache requires a mechanism to avoid performance degradation by shared cache conflicts. The approach of this dissertation to avoid the shared cache conflict and improve the cache performance is to design a cache partitioning mechanism based on the cache requirement defined in Chapter 2. The proposed shared cache mechanism consists of a way-allocation function for cache partitioning and a power control function. The way-allocation function of the proposed shared cache mechanism defines the allocated cache area by making comparison between the cache requirements of applications simultaneously executed in each core. The way-adaptable cache mechanism is applied to the power control function to conserve the power consumption by inactivating these less needed ways. The function appropriately controls the number of activated ways in the allocated cache area by the way-allocation function. The proposed shared cache mechanism searches most appropriate allocated and activated ways by iterating these two processes. The evaluation results show that the proposed shared cache mechanism is comparable to the utility-based scheme with more effective power saving. The results of all functions indicate that the proposed shared cache mechanism achieves the performance improvement, and moreover it reduces power consumption with keeping the performance by the power control mechanism. The power control policy of the mechanism can be adjusted from a performance-oriented configuration to an energy-oriented one. The proposed shared cache mechanism with a performance-oriented parameter setting can reduce an energy consumption by 20%, while keeping the performance in comparison with a conventional one. On the other hand, the cache with an energy-oriented parameter setting can reduce 55% of energy consumption with a performance degradation of 13%. Moreover, a control mechanism is designed to evaluate hardware overheads. The evaluation of the hardware overheads of the cache control circuit shows that the cache control hardware has an extremely small overhead and a small effect on the chip design. These results show that the proposed shared cache mechanism is a promising method for the high-performance and low-power cache design. Therefore, this chapter establishes a shared cache control mechanism for CMP to achieve high-performance and low-power.

In conclusion, as a result of above three approaches, this dissertation establishes the cache management methodology to achieve both of low-power and high-performance.

Acknowledgments

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

First of all, I would like to express deep gratitude to my supervisor, Professor Hiroaki Kobayashi, 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 seven years. I would like to thank Professor Takafumi Aoki and Professor Susumu Horiguchi 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/personal support and their hot encouragement. I would also like to thank Professor Emeritus Tadao Nakamura, Professor Emeritus Michael Flynn of Stanford University and Associate Professor Kenichi Suzuki of Tohoku Institute of Technology for his valuable and helpful comments.

I would like to thank Associate Professor Hideaki Goto and Associate Professor Kentaro Sano for their many support and advice to my research activities. Thanks go out to Ms. Maki Takahashi, Ms. Rikako Hasegawa, and all member of Cybersience Center of Tohoku University for their support to my comport laboratory life.

I would also like to express my appreciation to all the members of Kobayashi and Goto laboratories, Graduate School of Information Sciences, Tohoku University. Special thanks go to the members of the processor project team, Mr. Yusuke Funaya, Mr. Masayuki Sato, Mr. Gao Ye and the late Kenta Abe for the heated discussions and encouragement both officially and privately.

Finally, 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 sister, my brother-in-law, and their children for their encouragement.

January 16, 2009
Isao Kotera

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Background

The performance of general-purpose processors has been growing rapidly with development of the semiconductor manufacturing technology, integrated-circuit design, computer architecture design, compilation technology, and so on [1]. Moore's law [2], which was advocated in 1965, indicates that a transistor integration degree on a chip exponentially grows by twofold in 1.5 years. For example, Intel's Itanium2 Montecito released in 2006 contains 1.72 billion transistors on a single 21.5 mm by 27.7 mm die [3, 4], while Intel's Pentium Pro released in 1995 contains 5.5 million transistors on a die measured 17.3 mm on a side [5]. Such a rapid growth makes it possible to integrate a large amount of hardware resources on a chip, and thereby to enhance processor performance.

On the other hand, the power consumption due to driving a vast amount of hardware on a chip is becoming a critical problem in high-performance processor design. For example, Intel's Itanium2 consumes 100W at 1.8GHz [3, 4], and Intel's Xeon consumes 163W at 3.0GHz [6, 7]. An increase in the power consumption dose not only cause an increase in system operation cost, but also degrades the system reliability by heat generation problem [8]. A CMOS circuit consumes dynamic power, short-circuit power, and static power. The dynamic power and the short-circuit power are consumed for switching a transistor

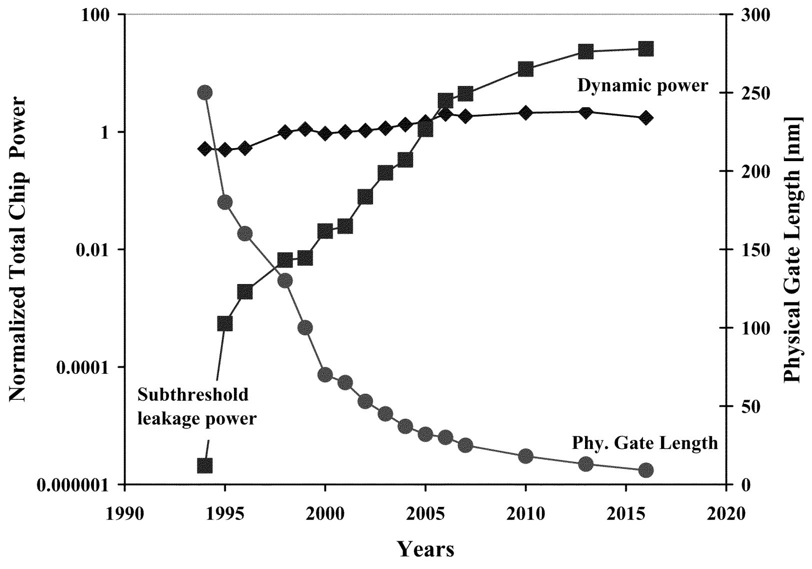


Figure 1.1: Transitions and predictions of dynamic and static power consumptions[17].

by charging or discharging the load capacity. They increase with a clock frequency because it is proportional to the number of switches per unit time. The power consumed by the short-circuit current can be kept to less than 20% (typically less than 10%) of the dynamic power [9, 10]. On the other hand, the static power consumption induced by current leakage is consumed regardless of the switching frequency. Nowadays, it becomes more dominant in the total power consumption, as the gate width becomes smaller [11, 12]. Figure 1.1 shows predicted transition of gate width, dynamic power, and static power. Unlike dynamic power consumption, static power consumption by subthreshold leakage current increases exponentially in advanced CMOS technologies. According to the International Technology Roadmap for Semiconductors (ITRS) [13], static power is expected to dominate more than 80% of the total power [14, 15]. Therefore, computer architects have to consider static power consumption to realize power-aware computing [16].

An increase in power consumption causes performance degradation, thermal problem, and high system operation cost. Therefore, there is a strong demand for processor technologies to achieve both high-performance and low-power. Especially, mobile equipments require such technologies because their sizes and power supply are severely restricted.

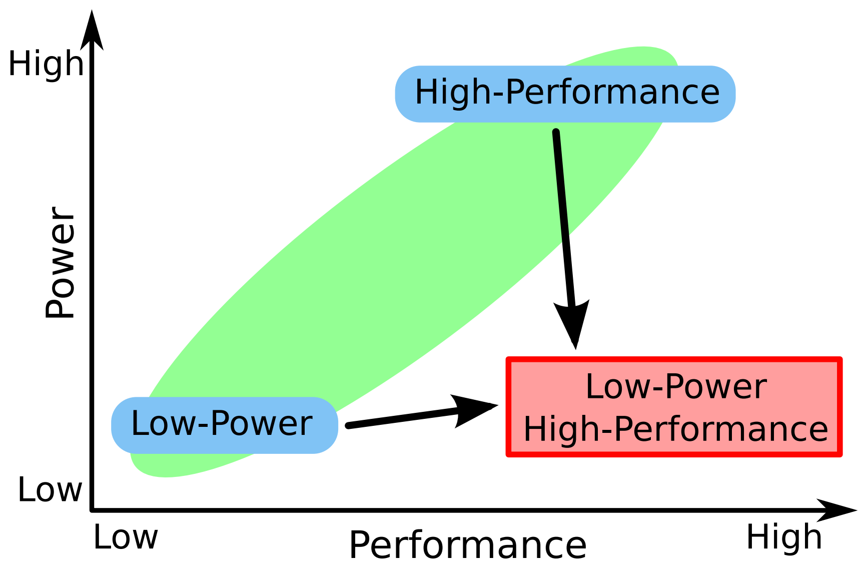


Figure 1.2: Trade-off between power and performance.

However, it is basically difficult to achieve both high-performance and low-power because there is a trade-off between them as shown in Figure 1.2. Although high-end processors such as Intel's Xeon [6, 7] need high power, they provide higher performance than low-power processors such as Intel's Atom [18]. Accordingly, an important open issue is how to improve power-efficiently of processors.

To realize such a processor, a cache memory is an indispensable hardware element. In recent processors, the cache memory is not only an important factor that determines the processor performance [19] but also one of the major power consuming elements in the on-chip hardware resources. For instance, cache and memory structures consume 30% of the total power of Alpha 21264, and 60% of StrongARM [20]. In the 300MHz bipolar ECL CPU [21, 22], 50% of the total power is consumed by caches. Especially, a substantial part of the static power is consumed in the large on-chip cache memory structures with high transistor densities. Hence, the reduction in power consumption by the caches is strongly desired to realize a high-performance and low-power processor.

As well as power saving, further performance improvement is another important research issue to realize a high-performance and low-power processor. Although a single core processor had been a major architecture until recent years, it has become difficult to improve the performance by increasing the clock frequency and exploiting the instruction level parallelism (ILP), because there are many technology obstacles as follows:

- Limit of semiconductor manufacturing technology:
  due to physical limits, it is becoming difficult to shrink the transistor size and to increase the clock frequency more [23].

- Limit of instruction level parallelism:
  the number of independent instructions and ability of ILP extraction are limited, and hence hardware investment for increases in the number of execution units and instruction window size does not lead to performance improvement [24].

- Limit of design cost:
  processor design cost is doubling every four years. It is becoming difficult to estimate how much time it will take to design and verify more complex processor, because the computational complexity of processor design rapidly increases with the hardware resources on a chip [25, 26].

In response to this situation, a chip multi-processor (CMP) [27, 28] has recently become common to increase the performance without worsening the above problems, such as Intel's Xeon [6, 7], AMD's Opteron [29], IBM's Power5 [30], Sun's Niagara [31] and so on. The CMP architecture, which has multiple processing cores on a single die, can simultaneously execute multiple workloads based on the thread level parallelism (TLP). The CMP allows core design to be smaller and simpler than single-core processors, because the CMP performance improvement relies on not only the capabilities of cores but also the number of cores. Therefore, the CMP design can inhibit the growth in the clock frequency and the circuit complexity.

In general, CMP consists of multiple cores and shared hardware resources among cores, such as the last-level cache and memory bus. Figure 1.3 shows a typical structure

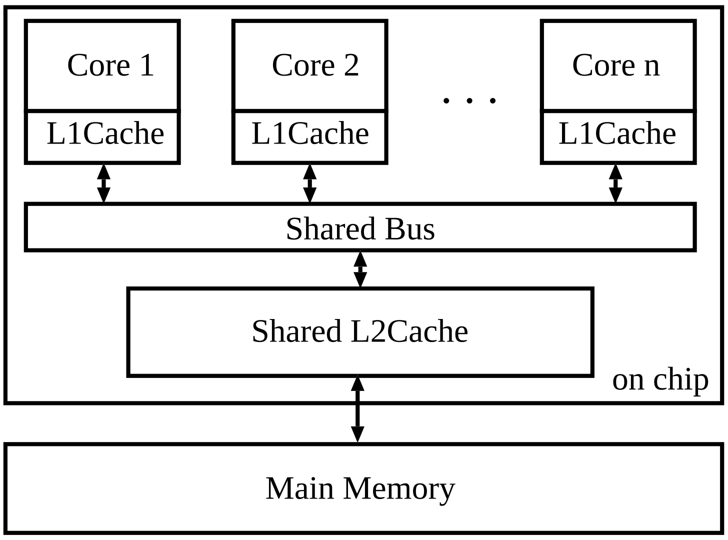


Figure 1.3: Chip multi-processor.

of CMP including the memory hierarchy. Here, an on-chip shared cache [32] plays a key role in efficiently executing multiple workloads. In the case where multiple cores executing independent programs share a cache, no core can use the entire cache capacity. If one core uses a large part of the cache capacity, the others can use only the remaining small cache space, resulting in the cache capacity shortage. This is called the *shared cache conflict* problem. The performance impact of each core due to the shared cache conflict depends on the program executed on the core, and therefore the performances of cores unfairly degrade. Consequently, it is necessary for CMP to avoid the shared cache conflict problem to improve the performance by exploiting TLP.

For these power consumption and the shared cache conflict problems in processor design, a mechanism to save cache power consumption while avoiding the shared cache conflict is required for future high-performance and low-power CMP processors.

1.2 Objective of the Dissertation

The objective of this dissertation is to establish a cache management methodology to achieve both of low-power and high-performance. The current cache memory design faces two important problems; the power consumption problem and the performance degradation problem by shared cache conflicts in CMP. To solve these problems and to achieve the objective, this dissertation develops a basic strategy that cache mechanism allocates the capacity in keeping performance according to cache requirements of applications. The power efficiency improves by turning off the power supply to the unused cache area. The cache requirement metric is also useful to avoid the shared cache conflict problem; if the cache requirement of each core is obtained, a shared cache can be appropriately partitioned into multiple parts, each of which is exclusively accessed only by one core. This dissertation considers following three issues to implement the above strategy.

- A cache requirement metric in order to assess minimum cache capacity to keep the performance for given applications
- A cache power control mechanism to supply power to cache area depending on the cache requirement.
- A cache control mechanism to avoid the shared cache conflicts in CMP

Through the discussion on these three issues, this dissertation establishes a cache management methodology to improve the power efficiency, while keeping high-performance.

1.3 Organization of the Dissertation

This dissertation is organized as follows.

Chapter 1 describes the background and objective of the dissertation. Here, two problems of current processors are pointed out: the power consumption problem and the shared resource conflict problem. Then, this chapter presents the objective of this dissertation and three issues to achieve the objective. Finally, this chapter shows an organization of this dissertation.

Chapter 2 describes a modeling method of the cache access behaviors according to Zipf's law. In addition, a cache requirement metric is discussed based on the cache access modeling. The cache requirement metric plays an important role for a power control function in Chapter 3 and a cache partitioning function in Chapter 4.

Chapter 3 proposes a way-adaptable cache mechanism to optimize the power efficiency for a single workload. The mechanism is designed in order to achieve cache power saving through dynamic control of active ways based on the cache requirement metric proposed in Chapter 2.

Chapter 4 extends the proposed cache mechanism for the shared cache in a two-core CMP. To solve unfair performance degradation caused by the cache conflict among cores, this proposed mechanism introduces the cache partitioning method based on the proposed cache requirement metric.

Finally, Chapter 5 describes concluding remarks.

Chapter 2

Modeling of Cache Access Behavior

2.1 Introduction

To realize efficient usage of a cache memory, accurate estimation and modeling of cache access behaviors is the foremost function. The objective of this chapter is to analyze the cache access behaviors and quantitatively assess the cache requirement, in order to assess minimum cache capacity to keep the performance for given applications. To this end, stack distance profiling is introduced to quantify cache access behaviors in this chapter. The stack distance distributions (SDDs), which are derived from the stack distance profiling, can assess the temporal locality of cache accesses. Additionally, SDDs have to be approximately modeled to accurately predict the behaviors. This chapter hypothesizes that the SDDs obey Zipf's law [33], and then verify the hypothesis against the real application. In addition, in this chapter, a cache requirement metric D based on the hypothesis is introduced to quantitatively measure cache size requirements of individual workloads. Finally, this chapter discusses the estimation accuracy of the metric D, by comparing the estimated distributions with the actual ones.

The rest of this chapter is organized as follows. In Section 2.2, Zipf's law is introduced to model SDDs, and then proposes a cache requirement metric. Section 2.3 shows the accuracy of the SDD model by the simulation, and also verifies the accuracy and the effectiveness of the prediction method of the cache requirement. Section 2.4 concludes the chapter with some future work.

Table 2.1: Sequence of LRU stack.

Time	Referenced Address	LRU stack	Stack distance
1	A	A	–
2	B	B A	–
3	C	C B A	–
4	C	C B A	1
5	B	B C A	2
6	D	D B C A	–
7	A	A D B C	4
8	A	A D B C	1
9	B	B A D C	3
10	A	A B D C	2

2.2 Cache Access Modeling

2.2.1 Stack Distance Profile

This section reviews the *stack distance profiling* [34, 35, 36] that can assess the temporal locality of reference. The simple set associative cache employs a replacement algorithm called *least recently used* (LRU). A naive implementation of LRU is to maintain a list of lines, called an LRU stack, to determine a cache line to be replaced. Table 2.1 shows how a simple access sequence changes the LRU stack of a 4-way set associative cache. In the table, there are ten accesses to four different lines (A, B, C and D) that belong to the same cache set. The *most recently used* (MRU) line is always moved to a top of the LRU stack. The bottom of the LRU stack is the LRU line of the set. The stack distance of a line is defined as the stack position of the line when the line is accessed. A *stack distance distribution* (SDD) can be confirmed by a histogram of the numbers of accesses to each stack distance. Figure 2.1 shows the SDD of the example in Table 2.1. C_i counts the number of accesses to the stack distance i. Therefore, counters C_1 and C_4 are used to count the numbers of accesses to MRU and LRU lines, respectively. C_m is used to count

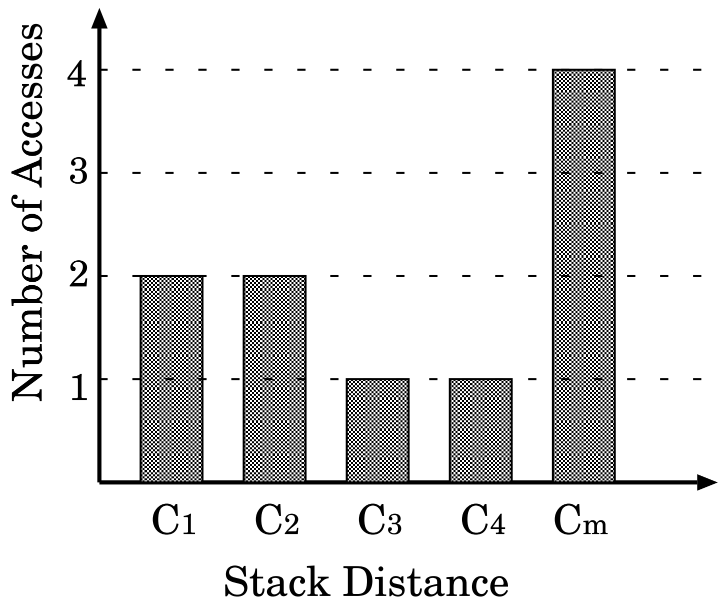
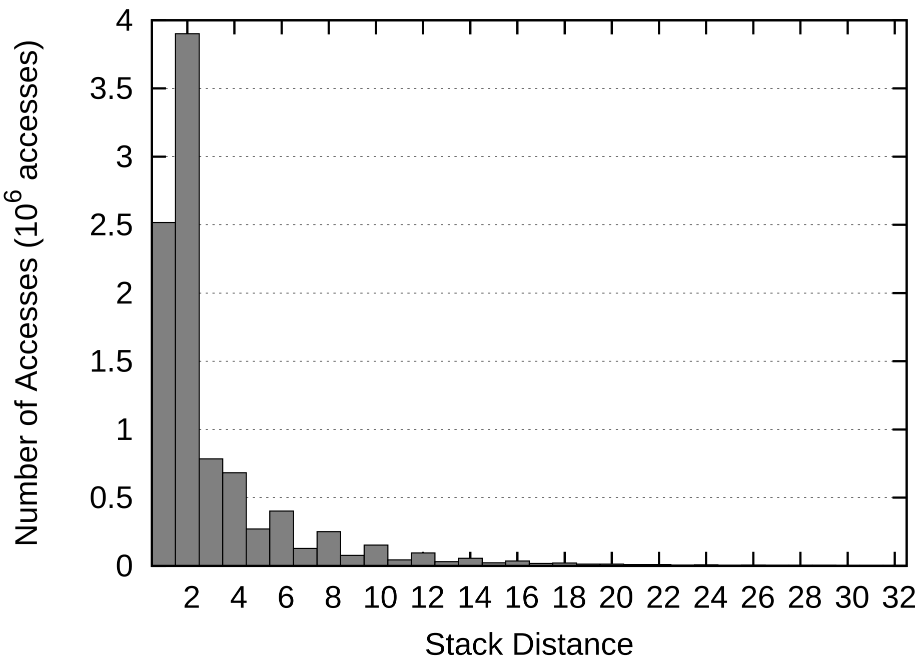


Figure 2.1: Stack distance distribution of the example.

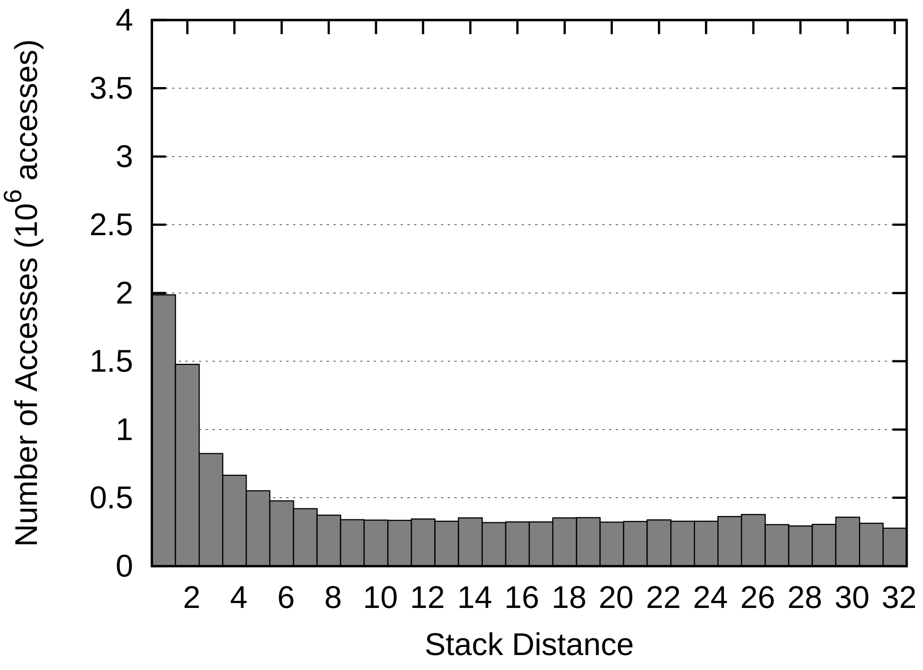
the number of cache misses. The histogram of C_i obtained by the stack distance profiling can be used to estimate whether a working set is large compared to the current cache configuration. In the case of a small working set, the cache accesses tend to concentrate on C_1. In the case of a larger working set, a distribution of cache accesses becomes flatter, resulting in relatively increasing C_4.

Figure 2.2 shows two examples of SDDs (L2 32-way cache). Figure 2.2(a) is a distribution of a workload (`aster`) that has a small working set. On the other hand, Figure 2.2(b) is a distribution of a workload (`bzip2`) with a large working set. In the case of `bzip2`, the number of accesses to the MRU lines (smaller stack distances) exponentially increases.

An approximate model of these SDDs is needed to accurately predict the cache access behaviors and to consider the cache requirement.



(a) `astar(lake)`



(b) `bzip2(liberty)`

Figure 2.2: Stack distance distributions.

2.2.2 Zipf's Law

Zipf's law states that the frequency of words in a document decays as a power function of its rank [33]. It is an empirical law formulated using mathematical statics. Zipf's law predicts that the frequency is calculated as follows:

$$f(k; s, N) = \frac{1/k^s}{\sum_{n=1}^{N} 1/n^s}, \tag{2.1}$$

where,

$$\begin{aligned} N &= \text{the number of elements,} \\ k &= \text{the rank of each element,} \\ s &= \text{the value of the exponent characterizing,} \\ &\quad\ \text{the distribution. } (s > 0) \end{aligned}$$

This law is also formulated as the following equation because it is trivial that the distribution obeying this law lies on a straight line with a double logarithmic plot.

$$\log(f(k)) = -a \log k - b, \tag{2.2}$$

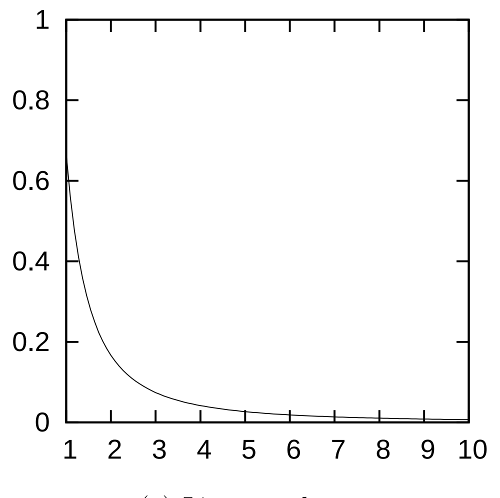
where,

$$a = s, \tag{2.3}$$

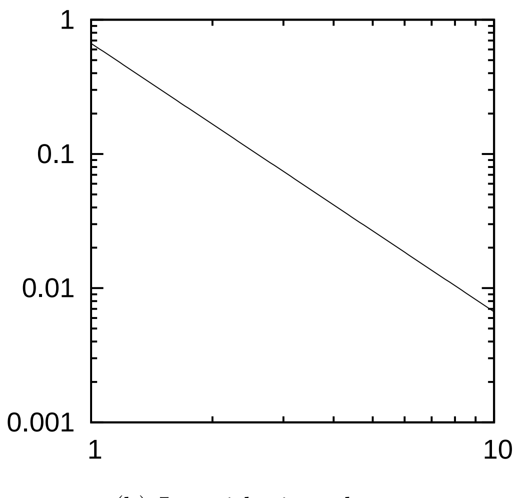
$$b = \log(\sum_{n=1}^{N} 1/n^s). \tag{2.4}$$

Thus, a distribution obeying Zipf's law can be represented with two parameters, a and b. Figure 2.3 shows the same Zipf's law probability distribution example ($a = 2$, $b = 1.5$) plotted on linear (Figure 2.3(a)) and logarithmic scales (Figure 2.3(b)) on both axes.

Zipf's law can be applied to many natural and social phenomena(e.g. many charac-



(a) Linear scales



(b) Logarithmic scales

Figure 2.3: Distribution obeying Zipf's law.

teristics of world wide web use [37, 38], Top500 supercomputers ranking [39, 40, 41], a relationship between a magnitude and total number of earthquakes [42] and so on). Characteristics of these phenomena, especially web references, are similar to the stack distance, in which the number of references to a more recently used line increases exponentially. Therefore, Zipf's law can be also hypothesized to represent the SDDs successfully.

2.2.3 Cache Requirement Metric

When the SDDs obey Zipf's distributions, the parameter s indicates characteristics of cache access locality. However, to realize a cache partitioning with a higher accurate performance prediction, cache accesses have to be monitored to obtain s with a low overhead. Here, the following metric D to approximately assess the locality of a program is proposed.

$$D = \frac{LRUcount}{MRUcount}, \tag{2.5}$$

where, $LRUcount$ and $MRUcount$ mean the numbers of LRU and MRU entries referenced in a certain period of cache accesses, respectively. Thus, if a program executed on a core has low-locality, D of the program becomes large. On the other hand, if it has high-locality, D becomes small.

By using Equations (2.5) and (2.1), the relationship between D and s can be considered as follows:

$$MRUcount = f(1) = \frac{1/1^s}{\sum_{n=1}^{N} 1/n^s}, \tag{2.6}$$

$$LRUcount = f(W) = \frac{1/W^s}{\sum_{n=1}^{N} 1/n^s}, \tag{2.7}$$

$$D = W^{-s}, \tag{2.8}$$

where W is the number of ways. Equation (2.8) indicates that D depends on parameter s. Therefore, the adjustment of D results in indirectly tuning s. The advantage of using D instead of s for cache partitioning is ease of hardware implementation; it requires an integer-divider and monitoring of MRU and LRU block accesses.

Bardine et al. have used metric D as a means to reduce the static power consumption of D-NUCA (Dynamic Non-Uniform Cache Architecture) cache without performance degradation [43].

Table 2.2: Simulation parameters.

Parameter	Value
Fetch width	8 insts
Decode width	8 insts
Issue width	8 insts
Commit width	8 insts
Inst. queue	64 insts
LSQ size	32 entries
L1 Icache	32kB, 2-way, 32B-line, 1 cycle latency
L1 Dcache	32kB, 2-way, 32B-line, 1 cycle latency
L2 cache	1024kB, 32-way, 64B-line, 14 cycle latency
Main memory	100 cycle latency
Clock frequency	1GHz

2.3 Experimental Analysis of Stack Distance Distributions

2.3.1 Experimental Setup

An experimental analysis firstly traces L2 cache accesses to obtain the stack distance profiling in this section. To this end, a cycle accurate simulator based on the M5 microprocessor architectural simulator tools [44] is developed. For the experiments, Alpha-based core with an L2 cache simulates the first two billion instructions of benchmark applications using the reference input set. The parameters used in the simulation are listed in Table 2.2. As shown in Table 2.3, 32 benchmark applications are selected from the SPEC CPU2006 suite version 1.1 [45].

Table 2.3: Benchmark applications.

INT/FP	**Name**	**Function**	**Data Set(ref)**
INT	`bzip2`	Compression	chicken, combined, html, liberty, program, source
	`gcc`	C Compiler	ctypeck, expr2, s04
	`gobmk`	Artificial Intelligence: Go	13x13, nngs, score2, trevorc, trevord
	`sjeng`	Artificial Intelligence: chess	
	`h264ref`	Video Compression	base, main, sss
	`omnetpp`	Discrete Event Simulation	
	`astar`	Path-finding Algorithms	lake, river
FP	`bwaves`	Fluid Dynamics	
	`gamess`	Quantum Chemistry	cytosine, h2ocu2, triazolium
	`milc`	Physics / Quantum Chromodynamics	
	`zeusmp`	Physics / CFD	
	`leslie3d`	Fluid Dynamics	
	`dealII`	Finite Element Analysis	
	`GemsFDTD`	Computational Electromagnetics	
	`tonto`	Quantum Chemistry	
	`lbm`	Fluid Dynamics	

2.3.2 Validity of Zipf's Law for Modeling of SDDs

Figures from 2.4 to 2.11 shows the some SDDs that are plotted on the double logarithmic scale, and their fitting results with the linear least-squares method. Almost all applications except `milc` show a high linearity. Correlation coefficients of 30/32 applications are more than 0.8 (Figure 2.12). Generally, a correlation coefficient over 0.8 is sufficiently large and its distribution has a sufficient correlation. Such a SDD, therefore, can be considered to conform to Zipf's law.

Applications can be classified into the following four groups, according to the characteristic of each distribution.

Group I `bzip2, gcc, h264ref, leslie3d, GemsFDTD`:
: This group includes 14/32 applications. Their distributions have high correlations between the logarithmic numbers of stack distances and accesses. The cache requirement can be predicted with a high accuracy for the cache partitioning, because cache accesses of these applications can easily be modeled. (Figures 2.4, 2.5, 2.6)

Group II `gobmk, sjeng, astar, bwaves, gamess (cytosine, h2ocu2), dealII, tonto`:
: This group includes 13/32 applications. Their distributions have higher correlation coefficients, however, the numbers of accesses to the MRU lines are overestimated and distant from the fitting results. These applications usually have a relatively larger a. (Figures 2.7, 2.8, 2.9)

Group III `gamess (triazolium), lbm, zeusmp, omnetpp`:
: This group includes 4/32 applications. Their distributions have relatively lower correlation coefficients. However, the distribution has higher correlation coefficients in a certain range of stack distance. In the case of `gamess (triazolium)`, an obvious change can be seen at around eight ways. To pre-

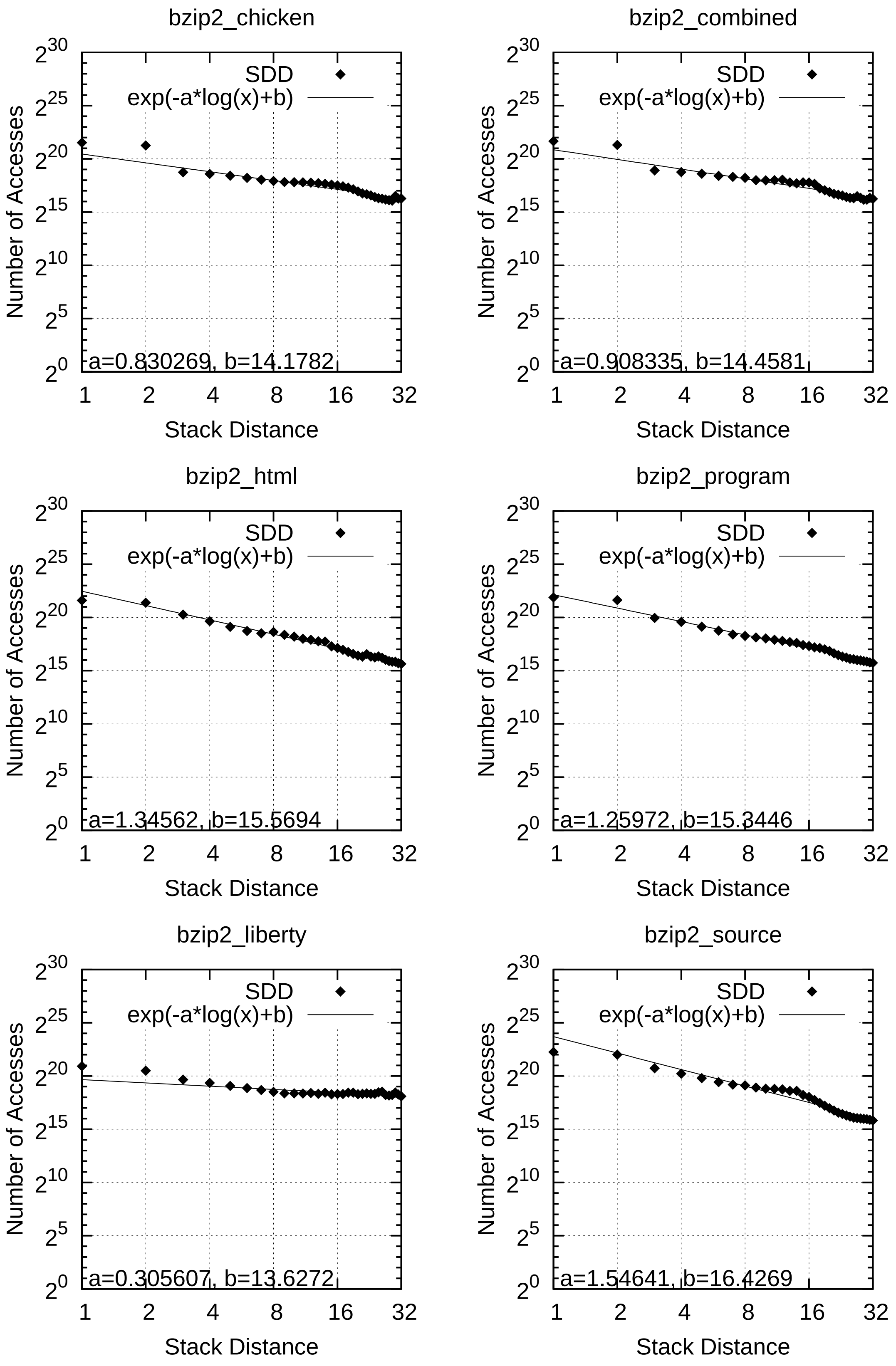


Figure 2.4: SDDs and fitting results of Group I (`bzip2(chicken, combined, html, program, liberty, source)`).

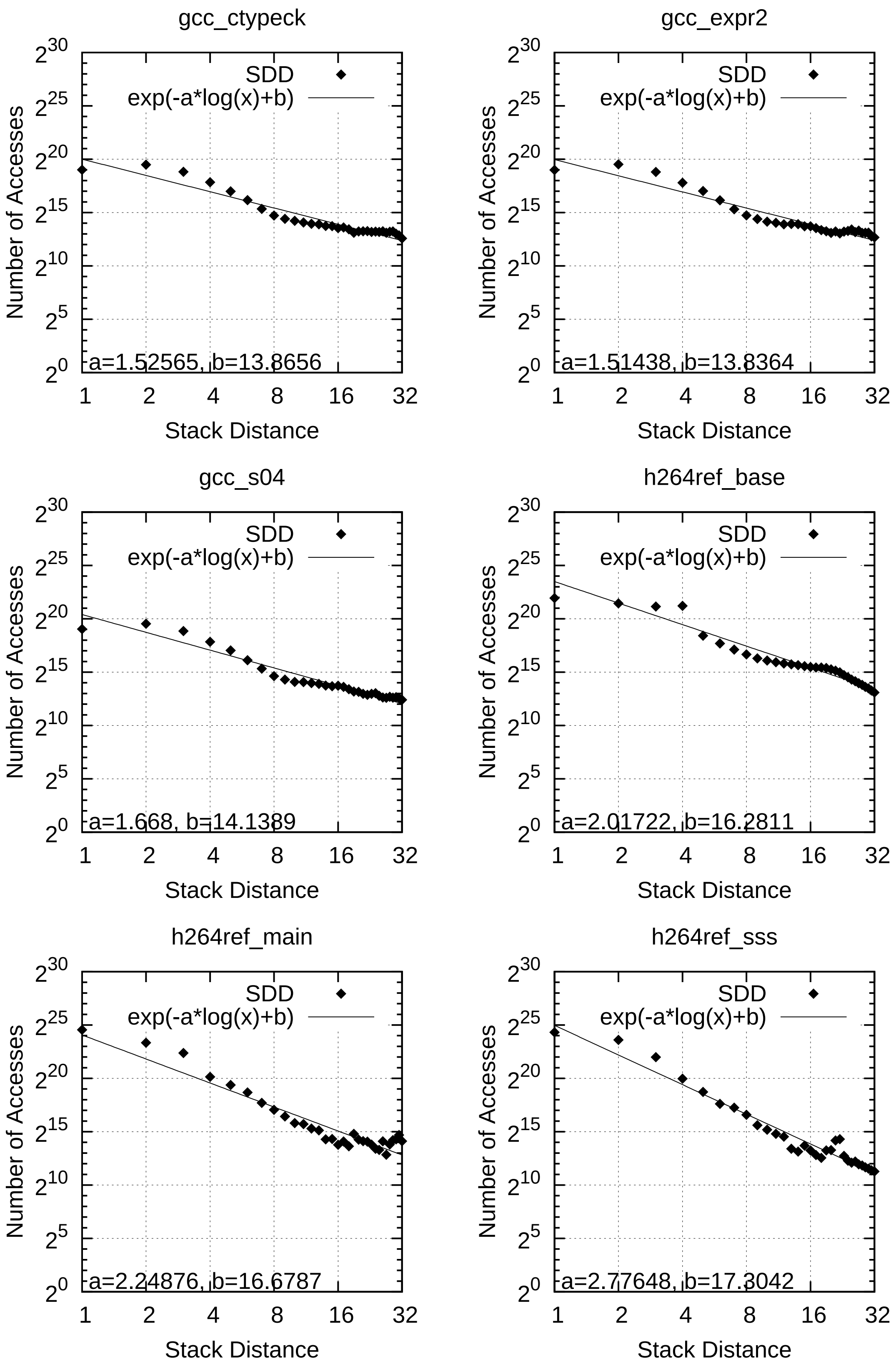


Figure 2.5: SDDs and fitting results of Group I (`gcc(ctypeck, expr2, s04, h264ref(base, main, sss))`).

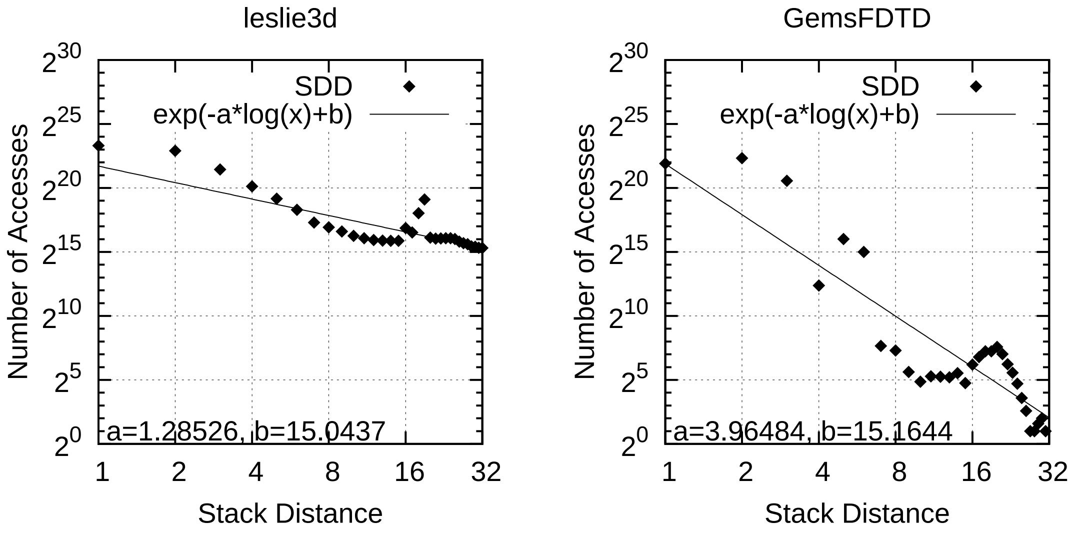


Figure 2.6: SDDs and fitting results of Group I (`leslie3d, GemsFDTD`).

dict the cache requirement, parameter a must be appropriately adjusted so as to adapt to the change.(Figure 2.10)

Group IV `milc`:

Its distribution has the lowest correlation coefficient (0.27) and does not have an appropriate fitting line. In this case, it is hard to predict the cache requirement by the stack distance profiling, but this kind of applications usually has a higher cache requirement. (Figure 2.11)

The results indicate that 17/32 SDDs in the Groups I and II of the above classifications can be modeled. The certain ranges of distributions in the Group III have higher correlation coefficients. Therefore, almost all the SDDs conform to Zipf's law.

The fitting lines in Figures from 2.4 to 2.11 are consistent with Equation (2.2). Except `milc` whose distribution has decorrelation, parameter a ranges from 0.31 to 7.79 and b ranges from 11.95 to 26.73. These results indicate that a and b can be used as parameters to predict the cache access locality and cache access frequency of a workload at each cache configuration, respectively. In the case of `milc`, a of its distribution can be approximated by zero, because it has a much higher working set.

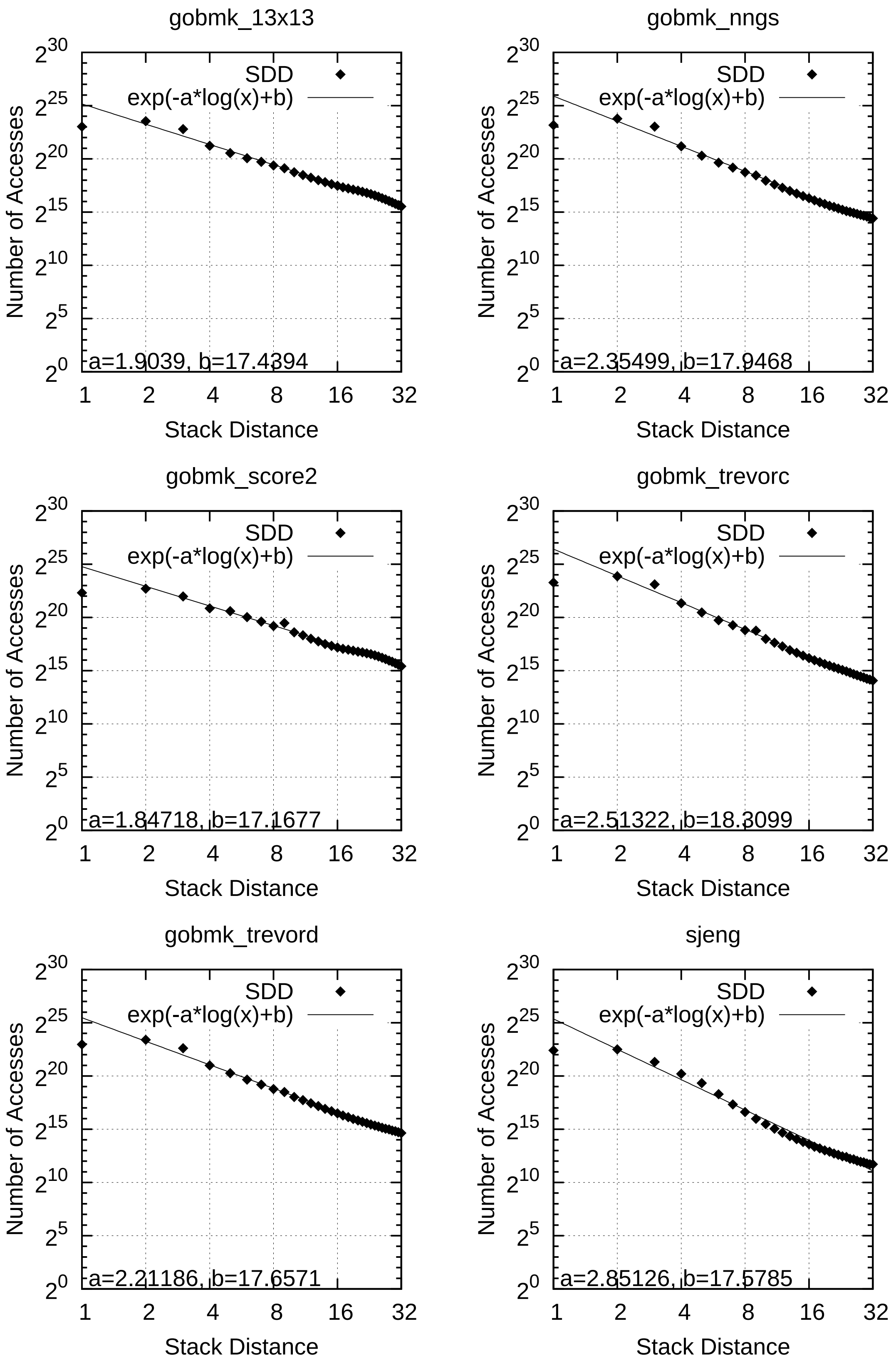


Figure 2.7: SDDs and fitting results of Group II (`gobmk(13x13, nngs, score2, trevorc, trevord)`, `sjeng`).

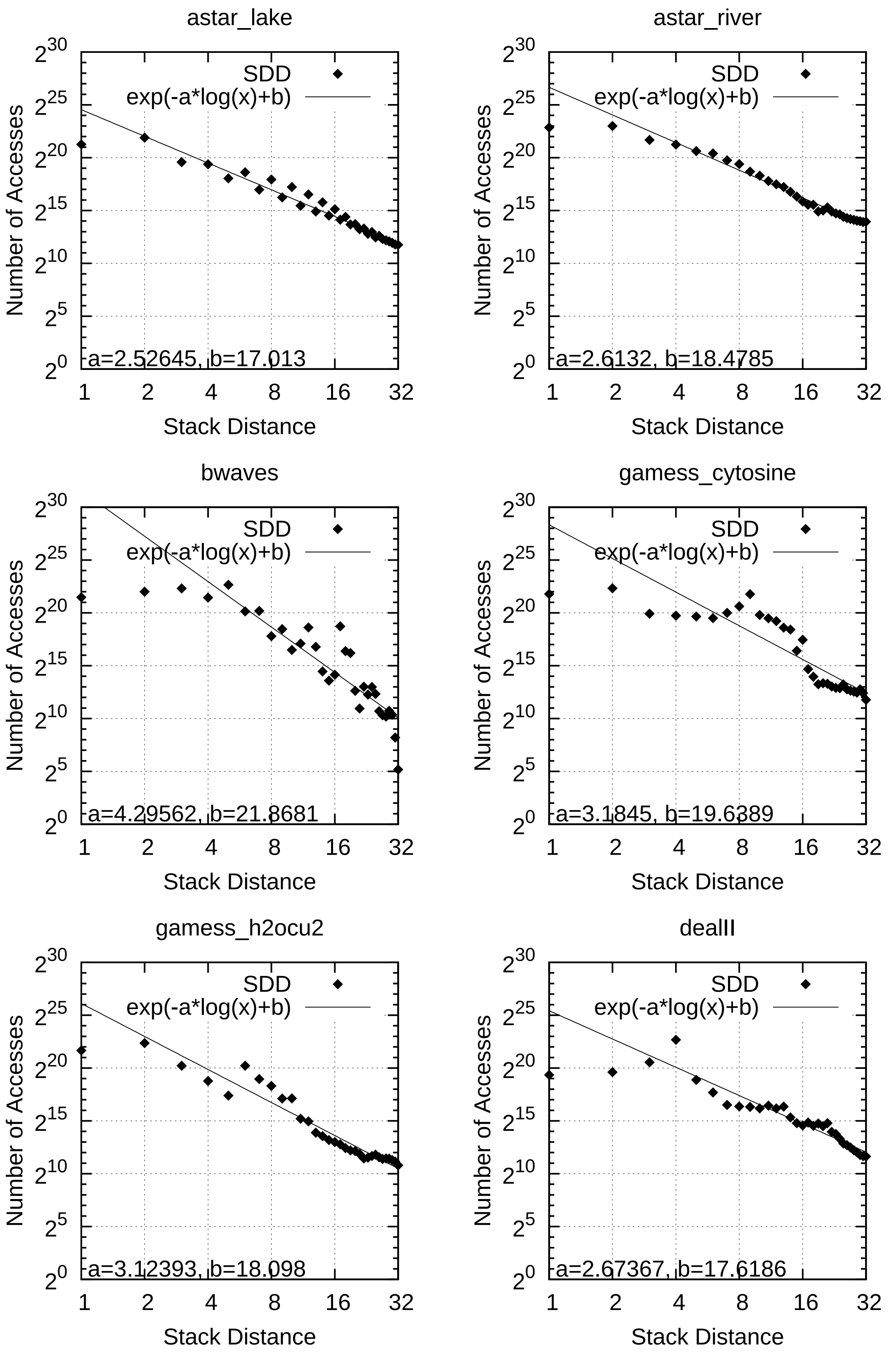


Figure 2.8: SDDs and fitting results of Group II (`astar(lake, river)`, `bwaves`, `gamess(cytosine, h2ocu2)`, `dealII`).

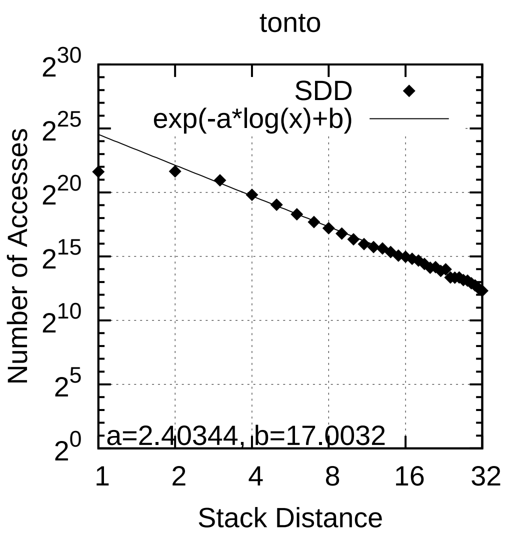


Figure 2.9: SDD and fitting result of Group II(`tonto`).

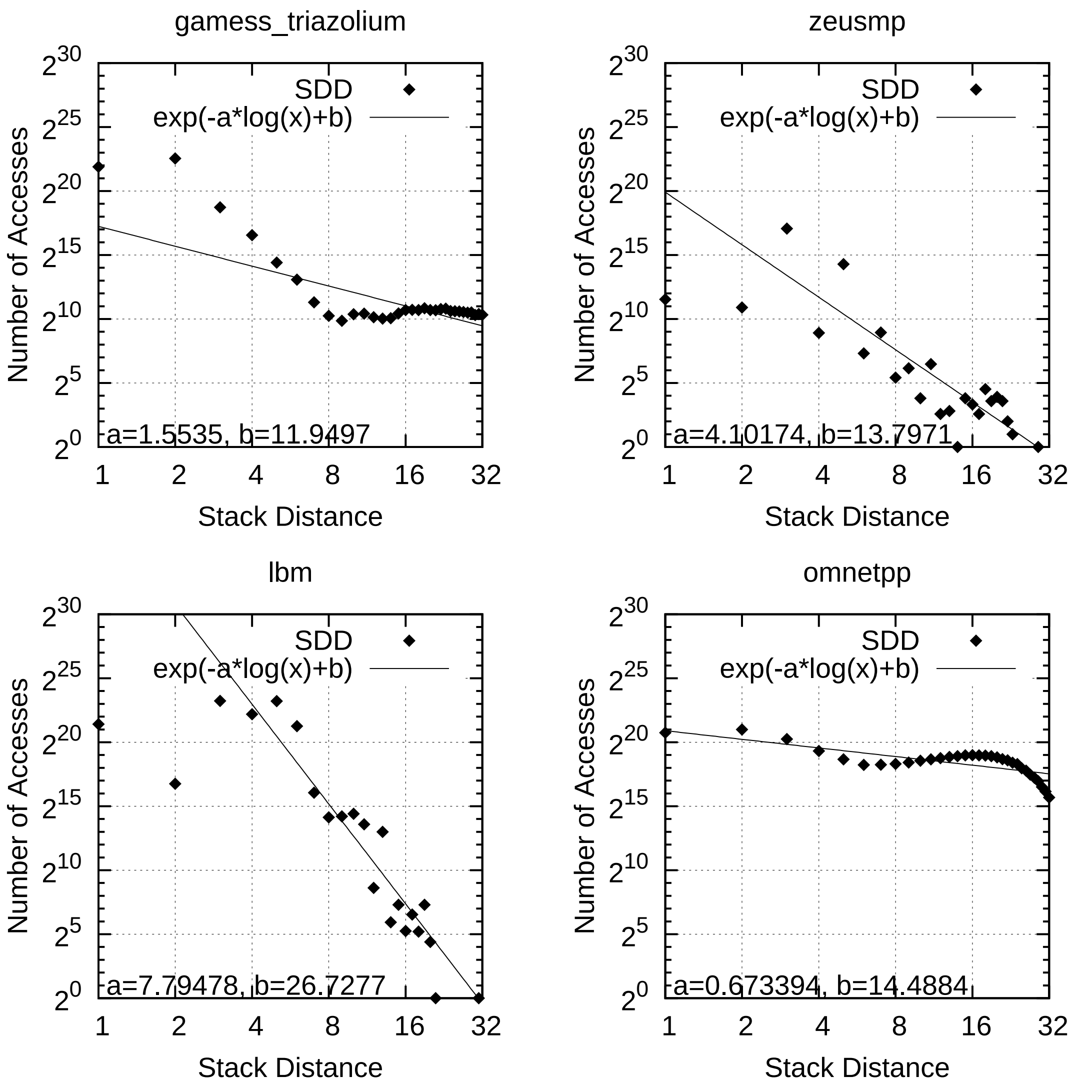


Figure 2.10: SDDs and fitting results of Group III(`gamess(triazolium), zeusmp, lbm, omnetpp`).

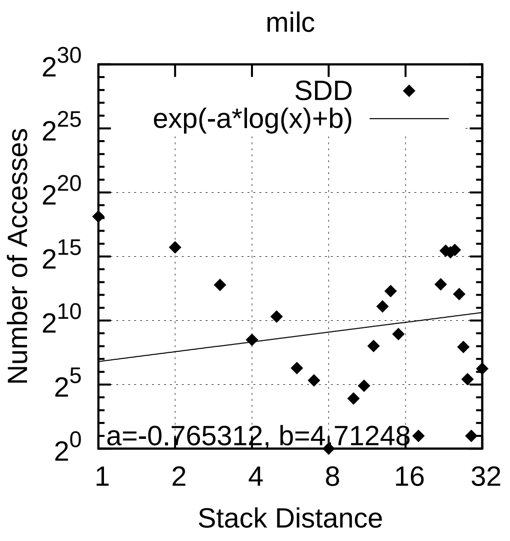


Figure 2.11: SDD and fitting result of Group IV(`milc`).

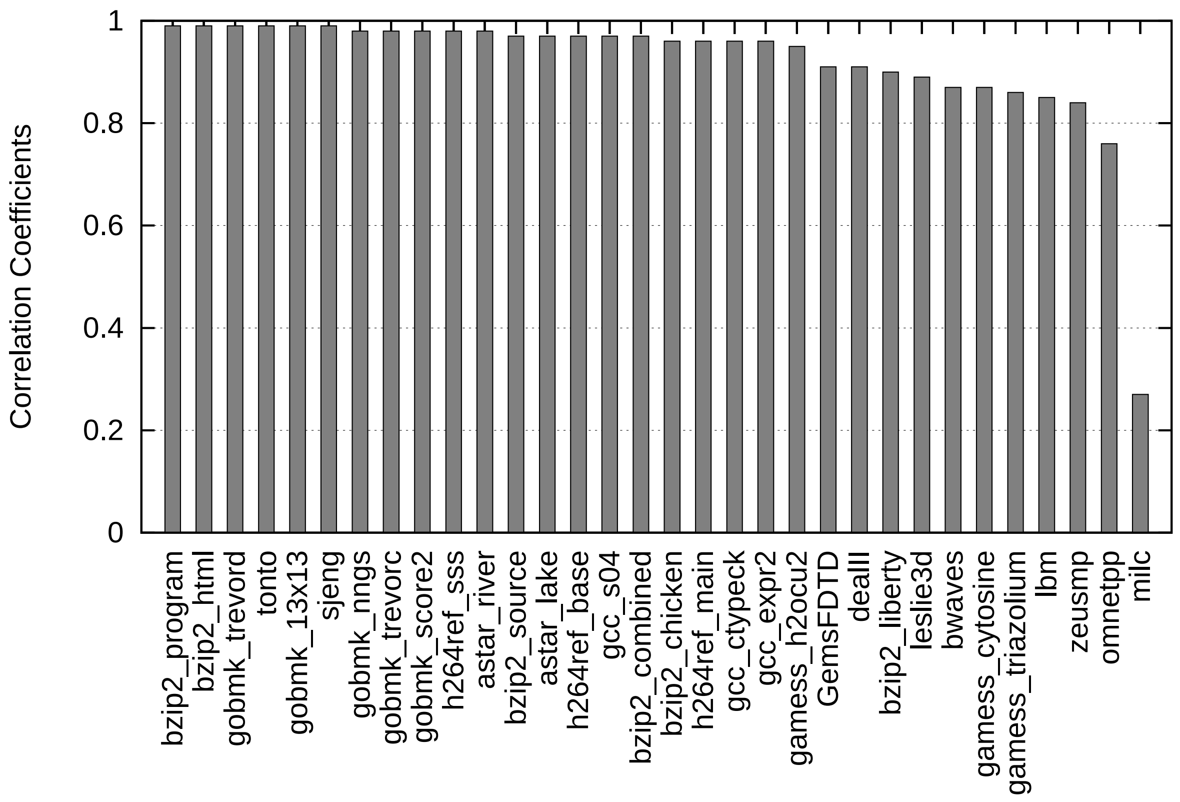


Figure 2.12: Coefficient of correlations.

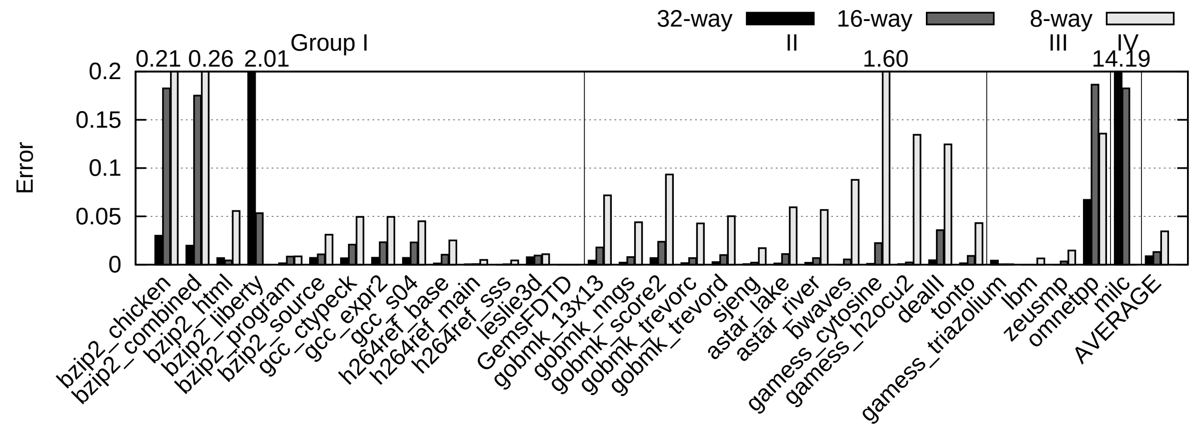


Figure 2.13: D_{error} of each application.

2.3.3 Validation of D as a Metric for Cache Requirements

To clarify the effectiveness of D, this section compares the cache requirement metric D_{actual} by Equation (2.5) with the cache requirement D_{fit} calculated by the fitting line of Equation (2.8). Figure 2.13 shows error D_{error} of each workload obtained by the following equation.

$$D_{error} = |D_{actual} - D_{fit}|, \tag{2.9}$$

Note that, in Figure 2.13, the average do not include `milc` to calculate, because it has no appropriate fitting line. Also note that each size of 32-way, 16-way and 8-way cache is 1MB, 512KB and 256KB respectively. The results show that D_{error} increases as the number of L2 cache ways allocated to a workload decreases, and its averages are sufficiently small when the cache has more than 16-way. Therefore, the cache requirement metric with more than 16-way cache can accurately predict the actual SDD. A hardware overhead does not change when the number of ways increases, because D is calculated using only the number of MRU and LRU accesses. Thus, the metric D has the efficacy as a quantification method of the cache requirement for a high associative cache.

However, some applications have a large D_{error}. D_{error} increases when the SDDs do not conform to Zipf's law (Groups IV). D_{error} are more than 0.15 in five applications (`bzip2` (`chicken`, `combined`, `liberty`), `gamess_cytosine` and `omnetpp`), in which the numbers of LRU accesses are extremely large. This is because D linearly increases with the number of LRU accesses, while D inversely increases with the number of MRU accesses. Therefore, the estimation accuracy of the cache requirement can be improved by considering the number of LRU accesses adequately. This is remained as the future work.

2.4 Conclusions

This chapter has introduced the cache requirement metric in order to assess the minimum cache capacity to keep the performance of applications. Firstly, the stack distance profiling method, which analyzes the cache locality with the histogram, has been introduced in order to assess cache access behaviors. Then, the profiling result of each application has been modeled with validating the hypothesis that the stack distance distribution obeys Zipf's law. The verification results have shown that almost all the SDDs conform to Zipf's law. Finally, this chapter has proposed and verified a metric based on the hypothesis to measure cache size requirements of individual workloads quantitatively. The results have shown that the D used to quantify cache requirements is effective to estimate the locality of an application.

According to this approach, this chapter has established the cache requirement metric for the high-performance and low-power cache mechanism. The proposed cache requirement metric plays an important role for a cache power control method in Chapter 3 and a cache partitioning method for the CMP's shared cache in Chapter 4.

Chapter 3

Dynamic Way-Adaptable Cache Mechanism

3.1 Introduction

Although cache memories play an important role to achieve high-performance processing on several kinds of microprocessors, they become the major power consumption sources due to their large capacity. As high-performance microprocessors have a large highly-associative on-chip cache, e.g., Itanium2 [46], novel low-power on-chip cache architectures are highly desired. Higher-associativity is also crucial for embedded microprocessors to cover the limited on-chip cache capacity [47]. However, not all applications need large cache capacity and higher associativity throughout their execution. The different usage of the hardware resources happens not only across applications, but also within the execution of individual applications. Therefore, if a cache is partitioned and necessary/unnecessary building blocks of the cache can be activated/inactivated so as to react to the resource requests by workloads, resource-efficient and power-efficient computing could be realized.

Several approaches have been proposed to reduce cache power consumption by adjusting the hardware resources in an on-demand fashion and to trade off a small performance penalty for large power savings. Block buffering[48], filter cache [49], and L-cache [50] achieve the power saving by adding a very small L0-cache between the processor and L1-cache. The mechanism of selective cache ways provides the functionality to statically turn on/off cache ways and is controlled by a performance metric given by users [51]. The accounting cache with dynamically resizing on-chip storage structures uses a tolerance metric to control the amount of performance degradation permissible [52, 53]. The advantage of the design is its ability to directly calculate the effect of different configurations relative to some base configurations and to protect against pathological behaviors. The way predictive set-associative caches [54, 55, 56] initially access one way, which is predicted to hold the accessed data. If the initial access does not hit, the other ways are accessed. Although the prediction successes have prospects of dynamic power saving, the misses degrade access latency. To reduce leakage current, the cache decay mechanism [57] shuts off the power supply to invalidate cache lines in a way. However, it causes the performance degradation by an increase in cache misses, since shutting off the power must discard the hold data. The drowsy cache [58] has been proposed to cover the drawback of the cache decay. It disables a part of the cache by dynamically switching to a lower-leakage mode called *drowsy mode*. The drowsy mode keeps supplying minimum power necessary to cached data to each line, even though it has a large overhead due to the high complexity of its power-gating circuits. Powell et al. have proposed the Dynamically ResIzable instruction-cache (DRI i-cache) [59, 60], which is an approach to reducing static power. The cache turns off power supply to a part of cache sets by introducing gated-Vdd transistors. The cache resizing is estimated using the number of miss and the threshold. However, the DRI i-cache can be applied only to an L1 instruction cache, which is based on a direct-mapped or low-associative cache with a small area. In order to reduce static power consumption, the mechanism has to cover not only an L1 instruction cache but also L1 data and lower-level (L2, L3) caches, which occupy a larger fraction of the chip area. Furthermore, the DRI i-cache employs a coarse-grain cache resizing mechanism by changing the number of index bits. Therefore, it is difficult to finely adapt its size to the program requirements.

These caches called *way-adaptable caches* in this chapter help to realize a good power-optimizing computing regarding on-chip cache resources, however, their control depends on absolute performance metrics such as cache hit rates and/or CPI (Clocks Per Instruction), which are application-specific metrics. For example, Yang et al. [61] proposed a resizing control mechanism that uses a threshold based on the cache miss rate. However, as the miss rates of individual applications are different even on the same hardware configuration, e.g., *"keeping miss rate less than 0.01"* is not an universal target for all of the applications on given hardware, it needs the profiling information of each application in advance to set its specific threshold for the resizing control.

This chapter presents a cache resizing control mechanism for dynamic way-adaptable caches. The mechanism uses the local and global information about the locality of cache accesses during execution. As the local information, the cache access pattern is evaluated by the cache requirement metric D based on the stack distance profiling of the cache access. If the cache accesses have relatively small D, which means that the cache accesses are concentrated on and near the MRU lines, the mechanism knows that the current cache resource allocation is excessive and there is room to decrease the number of ways activated. On the other hand, if the accesses have relatively large D, which means that the accesses are widely distributed from the MRU lines to the LRU ones, the mechanism understands that more ways are needed to keep the performance as long as the resources are available. In addition, to examine the global behavior of the locality of cache accesses, an n-bit state machine like n-bit branch predictors is introduced into the mechanism. The state machine traces and evaluates the strength and weakness of the requests for cache resources across the execution time. Therefore, the mechanism can avoid unstable actions for enabling/disabling cache ways when the locality shows the highly irregular behavior.

The rest of this chapter is organized as follows. Section 3.2 presents a cache resizing mechanism to control the dynamic way-adaptable cache. In the mechanism, n-bit state machines to appropriately decide when the way enabling or disabling should take place are used to avoid unstable behavior of way adaptation in highly irregular locality of accesses. Section 3.3 presents the performance evaluation of the proposed mechanism. The experimental results indicate that n-bit asymmetric state machines show the good performance in terms of power-optimized computing even in the case of applications with irregularity in locality of accesses across the execution time. Section 3.4 concludes this chapter.

3.2 Cache-Resizing Control Mechanism

3.2.1 Mechanism Overview

The cache-resizing control mechanism for a many-way set-associative cache is designed with a power-gating function, which can shut off the power supply to each way independently using the power-gating circuits. Figure 3.1 shows the power-gating by an NMOS transistor. An electric potential of the virtual GND line can be shifted from a potential of the GND line to one of the VDD line by the control signal. The circuit block is active when the potential of the virtual GND line is GND, and inactive when the potential is VDD.

To appropriately resize a cache, the approach of this mechanism is to determine whether it should up-size or down-size an allocated cache area based on the cache requirement metric D. In fact, the inputs of this mechanism are cache access sampling results for a fixed interval, and the output is a cache resizing request *INC* (up-sizing), *DEC* (down-sizing) or *KEEP* (keeping current configuration). The power-gating hardware activates or inactivates one way when the cache resizing is requested.

Figure 3.2 indicates the control flow chart of the proposed mechanism. After the cache access sampling phase, the mechanism calculates the cache requirement metric D. It outputs the resizing request based on the D through the assessment of local and global access behaviors. When the output request is *INC* and the cache has inactivated ways ($W\_active < W\_total$, where W_active means the number of activated ways and W_total means the number of total ways), the cache selects one way from inactivated ways and activates the way by turning on the power supply. When the request is *DEC* and the number of activated ways is more than two because the mechanism needs MRU and LRU lines to calculate D, it selects one way from activated ways. After the data included in the selected way is written back to a lower-level memory, the power-gating circuits inactivate the way by shutting off the power supply.

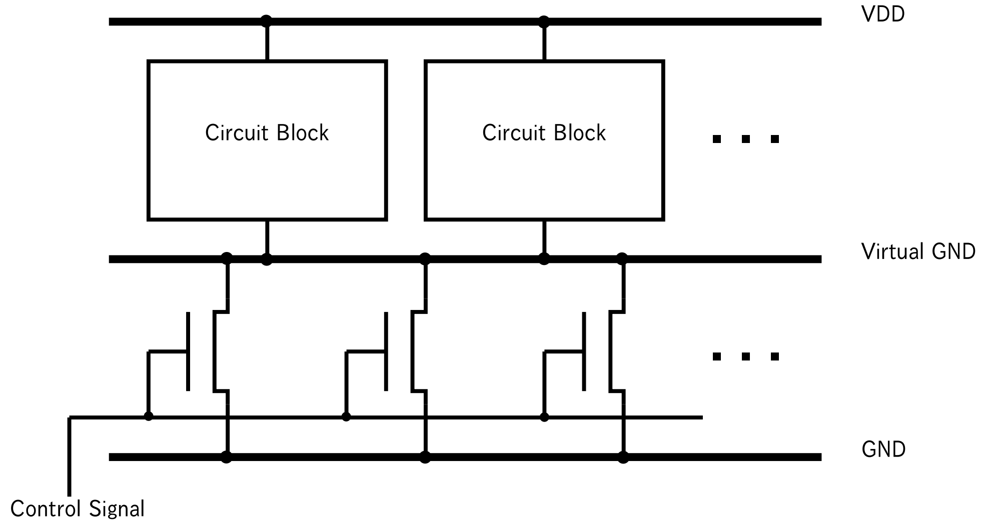


Figure 3.1: Power-gating circuit.

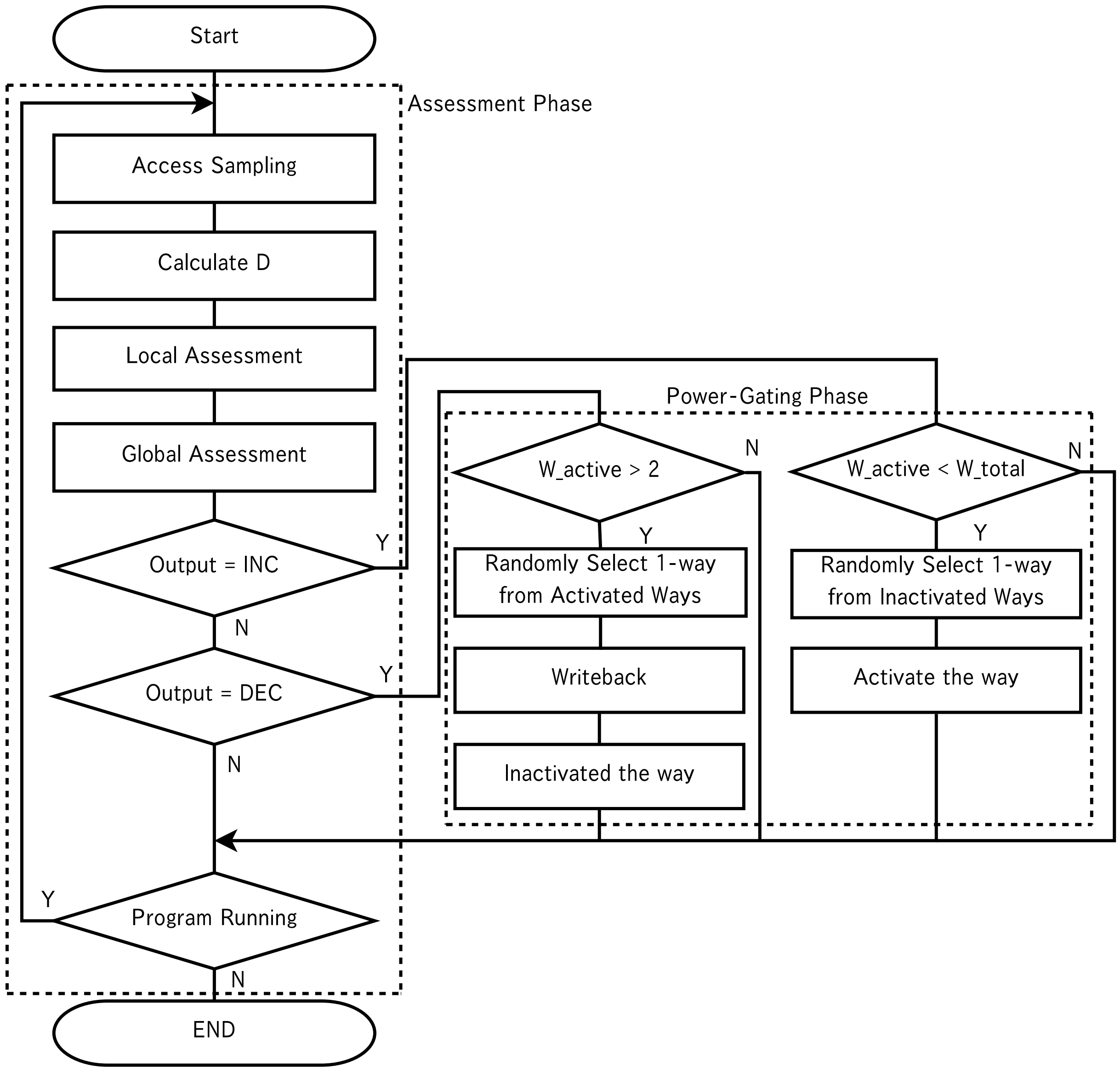


Figure 3.2: The control flow chart of the proposed mechanism.

Algorithm 1 The local assessment of the cache requirement.

```
Require: $D = LRUcount/MRUcount$
  if $D < T1$ then
    return Down-sizing request ($dec$)
  else if $D > T2$ then
    return Up-sizing request ($inc$)
  else
    return Keeping configuration request ($keep$)
  end if
```

3.2.2 Assessment of Local Behavior

The aim of the local assessment is to quantitatively assess the absolute magnitude of cache requirement in a fixed interval and to estimate to up/down-size an allocated cache size. The input of this assessment is the cache requirement metric D, and the output is a resizing signal, which is an input to the next phase: the global assessment.

To quantify the absolute magnitude of local requirement for cache resources, D is compared with two thresholds, T_1 and T_2 ($T_1 < T_2$). Algorithm 1 shows the description of the algorithm to estimate the up/down-sizing request. If D, which is obtained in execution of a program on a core, is larger than T_2, the program can be considered to have the low locality and hence to need many ways. In this situation, the resizing mechanism outputs a signal *inc* (up-sizing request) to increase the number of activated ways. On the other hand, it gives a signal *dec* (down-sizing request) to decrease the number of activated ways if D is smaller than T_1. If D is between T_1 and T_2, the mechanism outputs a signal *keep* to keep the current configuration. The mechanism tends to output *dec* if both T_1 and T_2 are relatively large. On the contrary, it tends to output *inc* if both T_1 and T_2 are relatively small. Thus smaller thresholds make the mechanism performance-oriented, and larger ones make it lower-power-oriented. As a result, the mechanism can adjust the control policy from a performance-oriented configuration to a lower-power-oriented one.

3.2.3 Assessment of Global Behavior

Although the adaptability of the cache to the change in the locality during execution is very important to realize power-optimized computing, it should be conservative when the locality is highly irregular and unstable. A cache resizing request given by local behavior assessment is not always consistent with its previous and subsequent ones; it may alternate up-sizing and down-sizing due to a temporal disturbance in memory access. If the locality of accesses becomes high and low very quickly in a short cache monitoring period, frequently enabling and disabling a cache way may happen when only using the information about the local behavior of the locality of accesses as defined in Chapter 2, resulting in the performance loss due to the large overhead without any gain. Therefore, the strategy to this situation is to wait at the current position until going into the steady state. To prevent excessively responding to such unstable requests, it is effective to utilize an n-bit state machine of cache resizing requests. The state machine judges that the requests are strong if the same resizing demands continue for a certain period. When the request is considered strong, the mechanism resizes the activated area.

Figure 3.3 shows an n-bit state machine, which uses an idea similar to the n-bit branch prediction scheme [62, 63]. As the branch predictor suggests the strength and weakness of the branch direction based on the information about the past branch activities, the proposed state machine predicts the stability of a cache resizing request in the next period using the past several evaluations of the locality by metric D.

Figure 3.3(a) shows a state transition diagram of a 3-bit symmetric state machine that moves toward the state for taking the opposite action symmetrically, and Table 3.1 shows its state transition table. The state machine will provide flexible cache control in adjusting the monitoring interval to obtain a steady state in the locality behavior. At the same time, it also makes cache resizing slow to the change in the locality. Before moving into State 000 and State 111 for generating the *INC* and *DEC* signal respectively, the machine transits to intermediate states from 001 to 110 to judge the continuity of the up-sizing and down-sizing requests. During these states, it outputs *KEEP* to keep the current cache configuration. After continuing *inc* or *dec* requests, the machine transits to State 000 or State 111 and then outputs the *INC* signal for up-sizing or *DEC* for down-sizing.

The symmetric state machine takes time to appropriately adapt the cache size to the cache requirement, because the machine conservatively works to generate signals for the up/down-sizing. However, the down-sizing operation strongly impacts the performance, since it has to write back the hold data. Therefore, to avoid the performance degradation, the power control mechanism gets the up-sizing operation to work aggressively and the down-sizing one to work conservatively.

The asymmetric state machine realizes the cache resizing control so as to react quickly to up-sizing requests and slowly to down-sizing requests, it can minimize performance degradation. Figure 3.3(b) and Table 3.2 show an asymmetric 3-bit state machine. When *inc* is given to the state machine, it outputs the 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 the down-sizing signal *DEC*. This state machine can prevent responding to temporary disturbances, and further make inactivation conservative to minimize the performance degradation induced by shortage of activated ways.

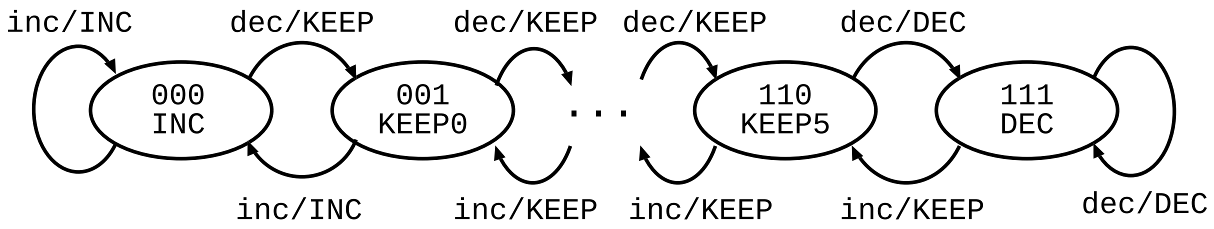
Yang et al. [61] also suggested that the effectiveness of a state machine for way-adaptable cache control although they used miss rates for cache resizing decision. However, their state machine was designed to detect repeated resizing between two adjacent sizes in a way-adaptable cache when the size of the required cache is just between realizable sizes. On the other hand, the function of the proposed state machine is to follow the dynamic behavior the locality of accesses and to evaluate the strength or weakness of a cache resizing request to judge whether taking a resizing action or not in the next period.

Table 3.1: State transition table of symmetric 3-bit state machine.

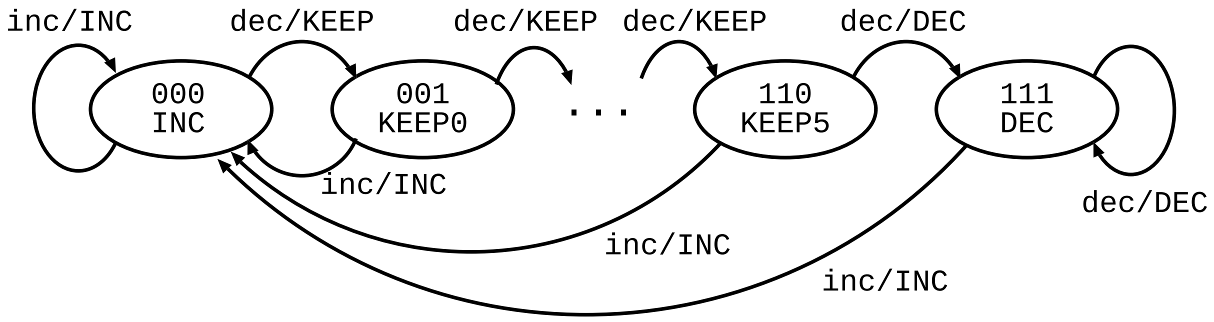
State	Input signal		
	inc	keep	dec
000	INC/000	KEEP/000	KEEP/001
001	INC/000	KEEP/001	KEEP/010
010	KEEP/001	KEEP/010	KEEP/011
011	KEEP/010	KEEP/011	KEEP/100
100	KEEP/011	KEEP/100	KEEP/101
101	KEEP/100	KEEP/101	KEEP/110
110	KEEP/101	KEEP/110	DEC/111
111	KEEP/110	KEEP/111	DEC/111

Table 3.2: State transition table of asymmetric 3-bit state machine.

State	Input signal		
	inc	keep	dec
000	INC/000	KEEP/000	KEEP/001
001	INC/000	KEEP/001	KEEP/010
010	INC/000	KEEP/010	KEEP/011
011	INC/000	KEEP/011	KEEP/100
100	INC/000	KEEP/100	KEEP/101
101	INC/000	KEEP/101	KEEP/110
110	INC/000	KEEP/110	DEC/111
111	INC/000	KEEP/111	DEC/111



(a) Symmetric 3-bit state machine



(b) Asymmetric 3-bit state machine

Figure 3.3: 3-bit state machines for cache control.

3.3 Performance Evaluation

3.3.1 Simulation Model

A way-adaptable cache is a set-associative cache, in which each way can independently be enabled or disabled. Like a conventional set-associative cache with the LRU policy for replacement, each entry in the cache has an LRU state. To evaluate the performance of the proposed control mechanism, a cycle accurate simulator is developed using SimpleScalar tool set version 3 [64]. Table 3.4 shows the parameters of the simulator designed based on Alpha 21264 [65]. This section examines the performance of way-adaptable caches that can change the number of ways from 2 to 32. The local and global evaluations of the locality for resizing are performed every 100K memory accesses. SPECint/fp [45], MediaBench [66], anagram, and dhrystone listed in Table 3.3 are used as the benchmark applications for the performance evaluation. For `art`, `equake`, and `mpeg2encode`, one billion instructions are executed to reduce simulation time, while skipping the first 500M instructions to avoid temporal effects in the initial phase of the execution on the performance. For the other applications, the entire instructions are simulated. In the following discussion, this section examines the performance in CPI and the number of activated ways obtained by averaging the results of L1 instruction and data caches if not specified.

Table 3.3: Benchmarks applications.

Suites	**Benchmarks**	**Functions**	**Instruction Executed**	**Cache References**
SPECint95	`go`	go program	100M	29M
	`gcc`	Based on the GNU C Compiler	100M	37M
	`compress`	UNIX utility program	78M	3M
	`perl`	Programming language	19M	7M
SPECfp2000	`ammp`	Computational Chemistry	19M	8M
	`art`	Image Recognition / Neural Networks	100M	14M
	`equake`	Seismic Wave Propagation Simulation	100M	37M
MediaBench	`mpeg2encode`	Mpeg2 encoder	100M	24M
	`mpeg2decode`	Mpeg2 decoder	100M	18M
	`g721-encode`	Voice compression encoder	100M	46M
	`g721-decode`	Voice compression decoder	100M	46M
	`rasta`	Speech recognition	17M	6M
Misc	`anagram`	Puzzle	15M	5M
	`dhrystone`	Synthetic benchmark	100M	33M

Table 3.4: Simulation parameters.

Parameter	**Value**
Fetch queue	4 entries
Branch predictor	comb(bimodal, 2-level gshare)
	bimodal - 2048 entries
	gshare Level1 1024(hint. 10)
	Level2 4096(global)
	Combining pred. 1024 entries
	RAS entries-32; BTB-1024 $\times$ 2ways
Branch mispred. latency	10 cycles
Decode width	4 instructions
Issue width	4 instructions
Commit width	4 instructions
Register update unit	16 entries
Load/Store queue	32 entries
Instruction TLB	64$\times$4-way, 8K pages, 30 cycles
Data TLB	128$\times$ 4-way, 8K pages, 30 cycles
Memory latency	80 cycles
Memory access bus	32 entries
Functional Units	Int 4, FP 2
L1 I-Cache	64KB, 32-way, 32B line, 2 cycles
L1 D-Cache	64KB, 32-way, 32B line, 2 cycles
L2 unified-Cache	1MB, 32-way, 64B line, 12 cycles

3.3.2 Experimental Results and Discussion

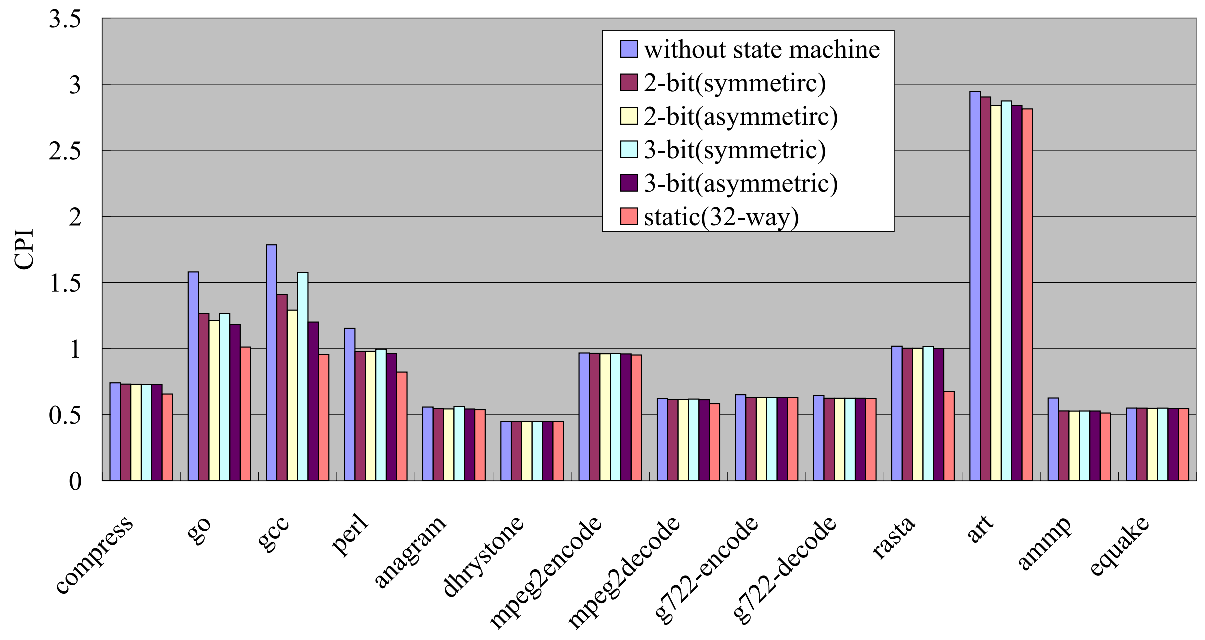
Figure 3.4 graphs the performance of several configurations of the proposed control mechanism as a function of the number of bits for the state machines. Here, $T1$ and $T2$ are set to 0.005 and 0.02, respectively. In the case of the SPECfp and MediaBench benchmarks with the regular and high locality, the number of bits does not play an important role in cache reconfiguration. Even without the state machine, the cache can be reconfigured very compactly while keeping almost the same performance that can be achieved by a 32-way cache. Therefore, only the local evaluation of the locality based on metric D at each time is sufficient to control the cache in this case. However, in the case of the SPECint95 benchmarks, which have highly irregular and time-variant behavior of the locality, e.g., `gcc`, more bits can provide more appropriate cache configurations in terms of performance/resource-usage. In addition, the control using the asymmetric state machines presents better performance compared with that using symmetric ones. Therefore, these results clearly indicate that analyses of both local and global behaviors of the locality are quite effective for stable control of way-adaptable caches.

To examine the way-enabling/disabling behavior in detail, Figure 3.5 shows the number of activated ways of the L1 instruction cache as a function of elapsed time in `gcc`. In the case of cache resizing control without the state machine, no global information about the locality is considered. Therefore, the cache sensitively reacts to the very quick change in the locality as shown in the figure. As a result, the number of average activated ways becomes very small and its achieved CPI reaches an unacceptably poor level. On the other hand, the consideration of the global behavior of the locality by using 2- and 3-bit state machines appropriately relaxes the sensitivity to the change in the degree of the locality during execution and realizes the stable cache control.

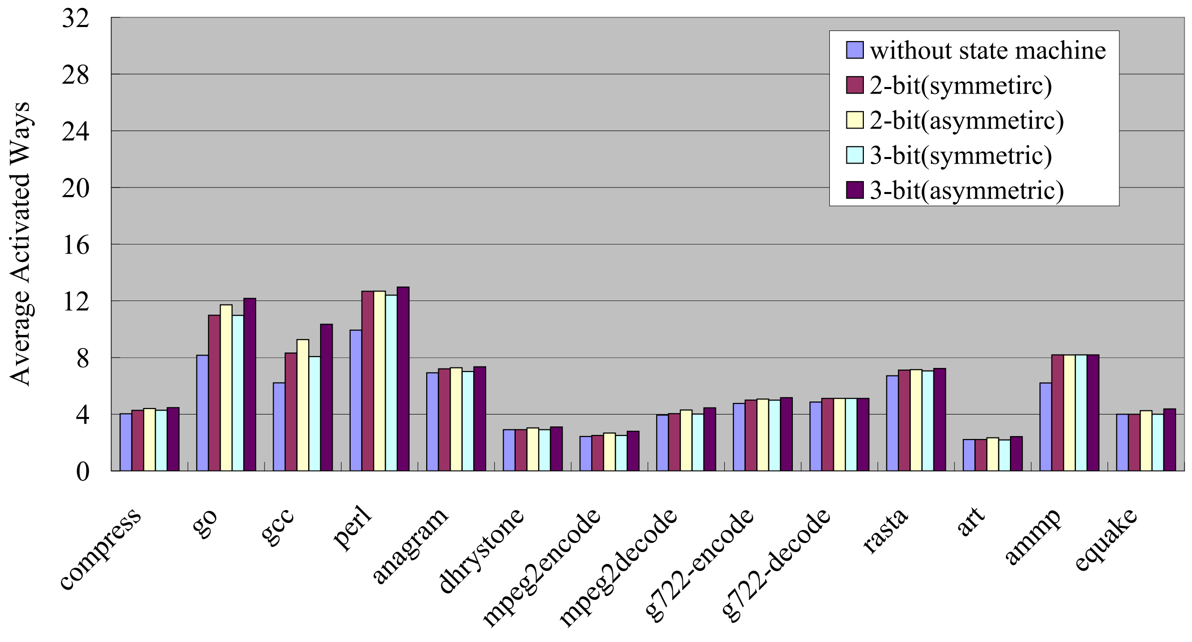
Cache resizing control using the n-bit symmetric state machine tends to keep the number of average activated ways small compared with the n-bit asymmetric state machine case. This is because once the number of average activated ways becomes small, it needs several consecutive requests for way-increasing to return back to the state for taking the way-increasing actions. Therefore, the cache slowly responds to the requests for increasing ways, resulting in a lower CPI, even though the number of activated ways is also saved. The n-bit asymmetric state machines can solve this problem, because it is very conservative to reduce the cache ways and sensitive to the quick change toward the lower locality. Consequently, the excessive reduction of cache ways is avoided while still providing a certain degree of saving in cache resource usage.

Figure 3.6 shows the effects of the number of bits on CPI and the time-variant change in activated ways of the L1 instruction cache. Figure 3.7 presents the number of reconfigurations of cache ways as a function of the number of bits for the asymmetric state machine. The figures suggest that although three or more bits do not show clear improvement in term of CPI performance, the number of activations/inactivations of the cache ways decreases as the number of bits increases. Therefore, from the viewpoint of the lower overheads in controlling the cache ways, increasing bits is preferable. On the other hand, the larger number of bits makes the cache resizing adaptability low. Three bits should realize a good trade-off under this environment, although the precise evaluation of time and space overheads in reconfiguration of ways is required to find the appropriate bit size, namely, the number of states in the state machine to define the degree of conservativeness to requests for deactivating cache ways. This is addressed as future work.

The values of $T1$ and $T2$ affect the performance of the cache mechanism. Figure 3.8 explores the CPI space as a function of $T1$ and $T2$. Here, the 3-bit asymmetric state machine is used. The figure also shows the dynamic behavior of the activated ways in L1 instruction and data caches. As the figure shows, a pair of smaller $T1$ and $T2$, (0.001, 0.002) keeps higher associativity in both L1 instruction and data caches, and results in a lower CPI, i.e., performance-oriented control to the cache. On the other hand, larger $T1$ and $T2$, e.g., (0.029, 0.030) achieve lower-power-oriented control at the expense of the performance. These are also confirmed in the other integer benchmarks. Therefore, these parameters should be adjustable at user-/system-levels to satisfy various requests



(a) CPI



(b) Average activated ways

Figure 3.4: Performance of the n-bit state machines.

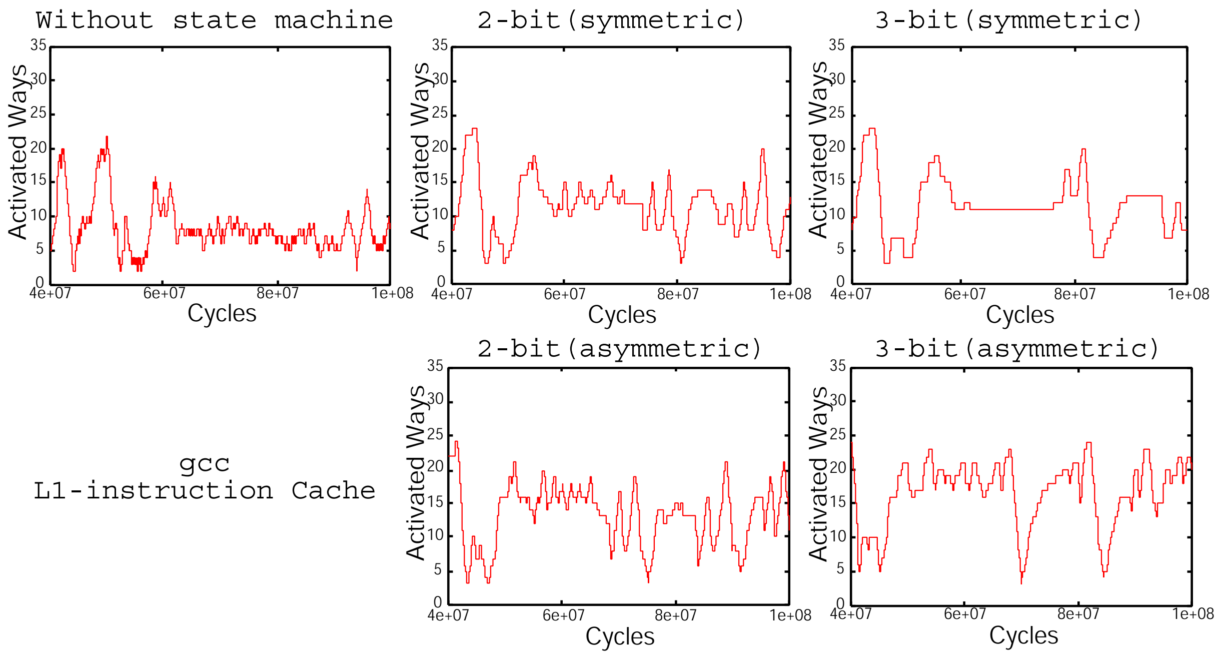


Figure 3.5: Way behavior vs. Elapsed time (gcc, L1 Instruction Cache).

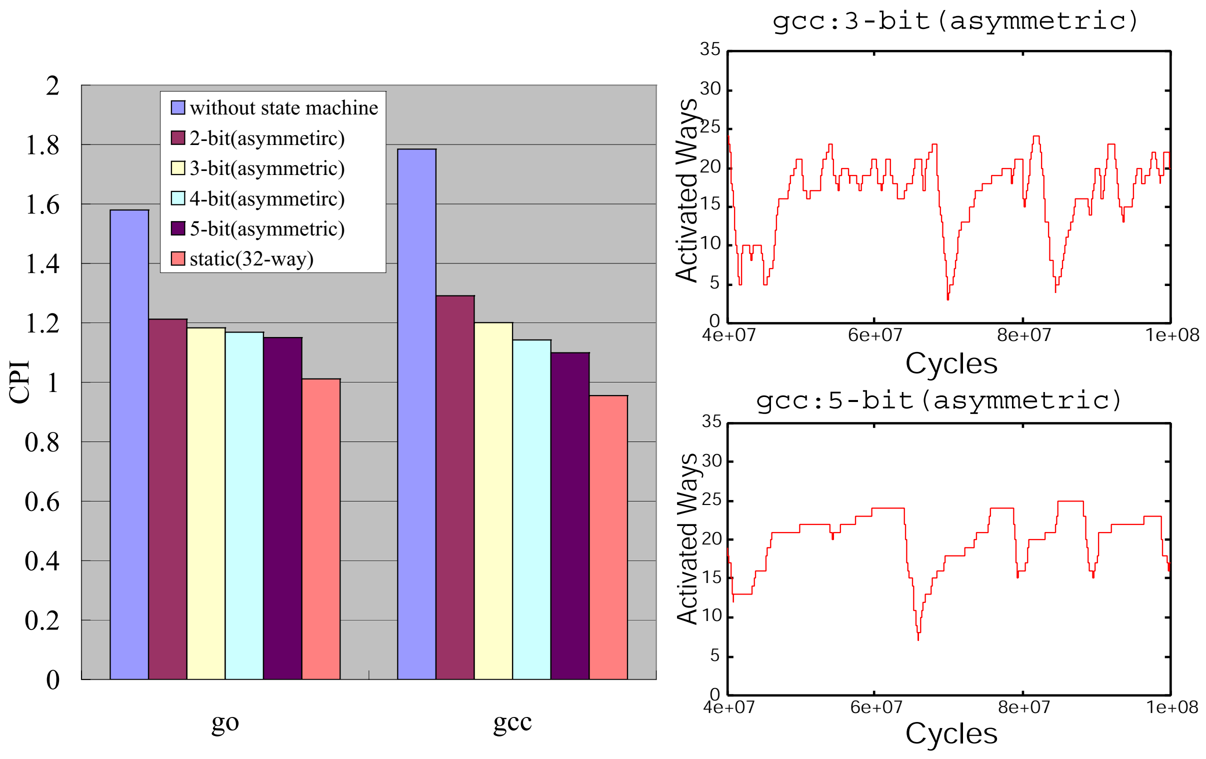


Figure 3.6: Effects of the number of bits on cache way control and CPI.

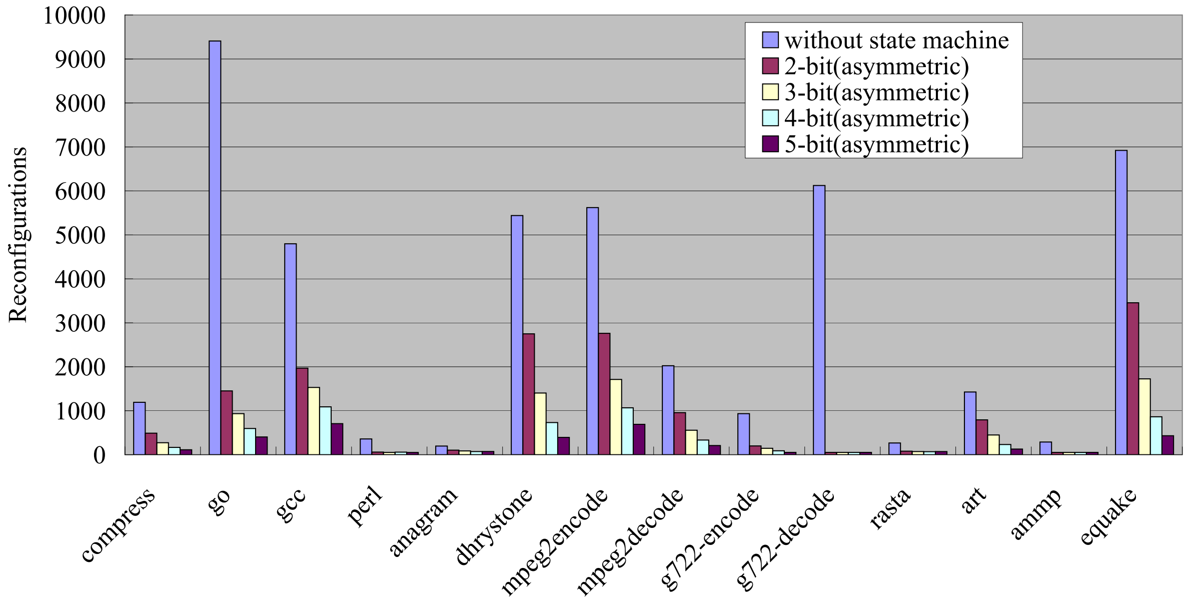


Figure 3.7: Number of reconfigurations.

regarding trade-offs between performance and resource-usage.

Since D in the range between $T1$ and $T2$ does not request any cache resizing, making the range wider may lead to stable control in unstable behavior of access locality. However, wider ranges between $T1$ and $T2$ also make the adaptability of the cache dull even in the case where strong and continuous resizing requests occur. The state machine achieves an effect similar to appropriate dynamic-adjusting of the range between $T1$ and $T2$; it makes the cache adaptability dull in the unstable situation and sensitive in the stable situation. Therefore, the combination of the local and global analysis of the locality for cache resizing decision is very important for dynamic way-adaptable cache control.

As shown in Figure 3.9, `rasta` presents very interesting stack distance distributions over the L1 instruction cache with 32 ways. In Figure 3.9, LRUstate0 (LRU0) means the count of the accesses to cache entries with the most recently used (MRU) state, and LRUstate31 (LRU31) means the count of the LRU state entries accessed. As the statistics concerning the cache accesses are measured every 100K memory accesses, "time t" on the time-axis means the t-th time interval measured in the unit of 100K memory accesses. Its behavior is highly regular, but periodically it needs to access old cache entries with LRU or nearby-LRU states, while almost no access to the entries with intermediate LRU states. In this case, as the LRU replacement policy does not work well on caches without enough associativity, metric D cannot figure out the locality of accesses. Therefore, another intelligent replacement is required to reduce the distribution of accesses on a smaller number of cache ways. A self-tuning replacement policy [67], instead of the LRU policy, might be promising for this purpose.

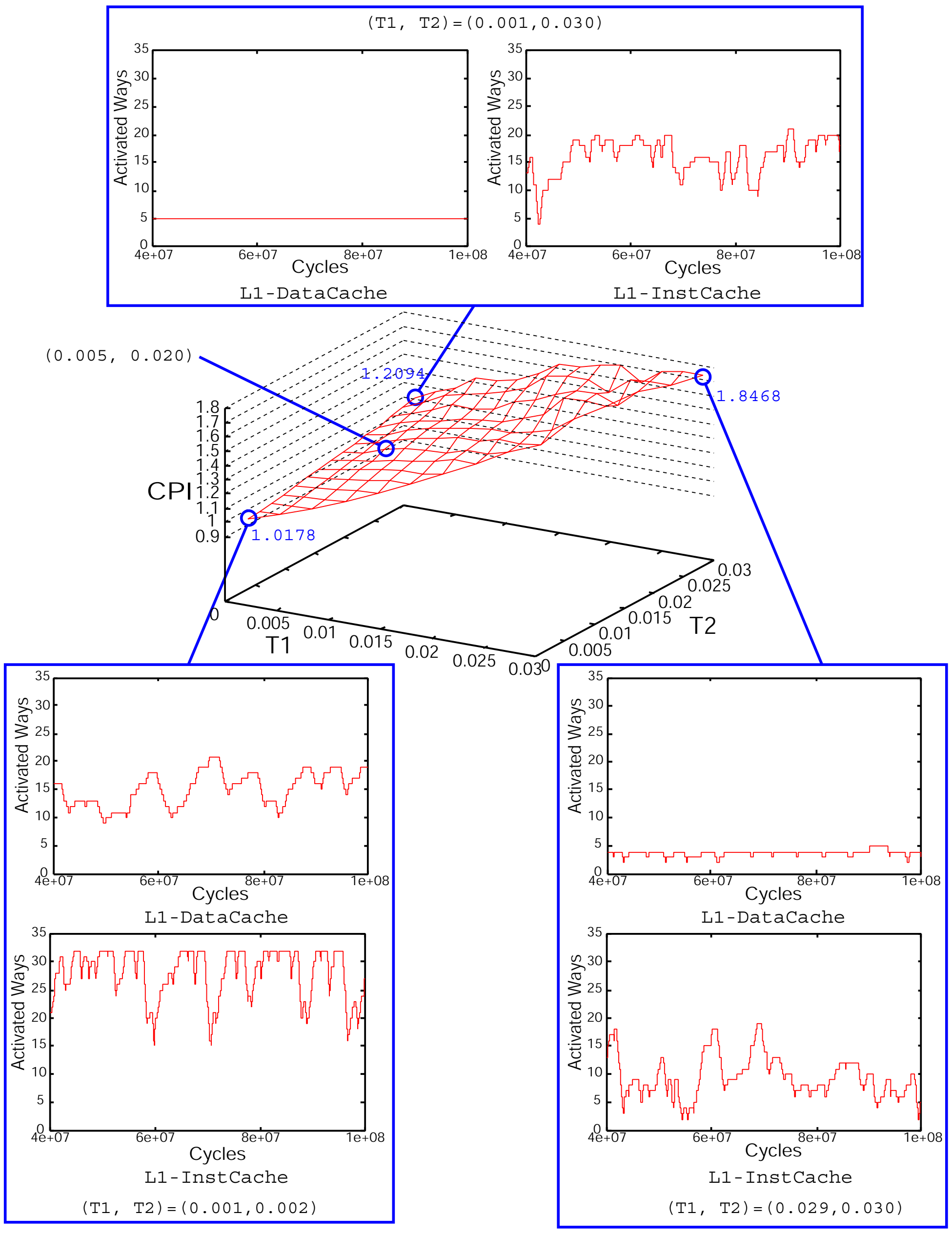
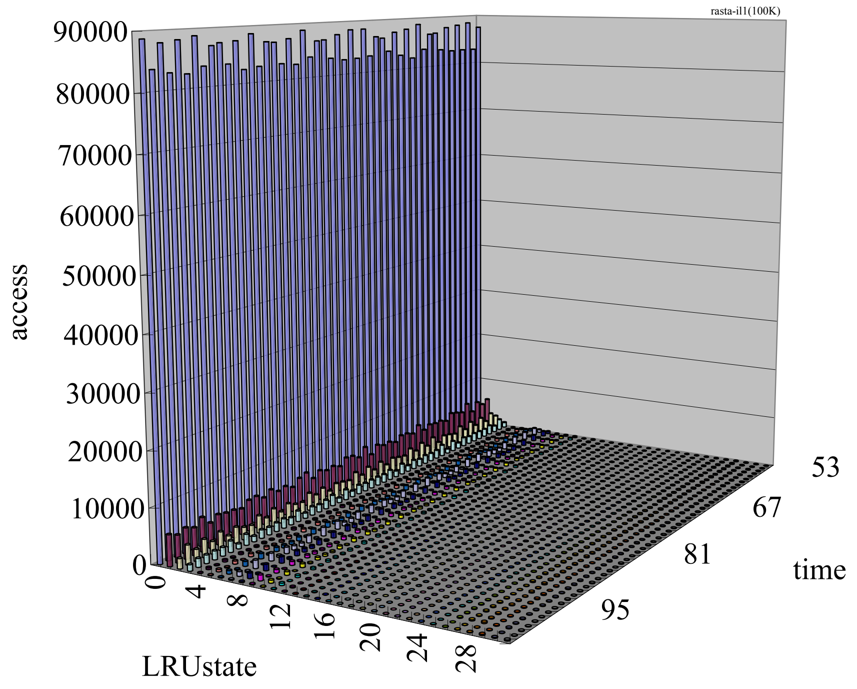
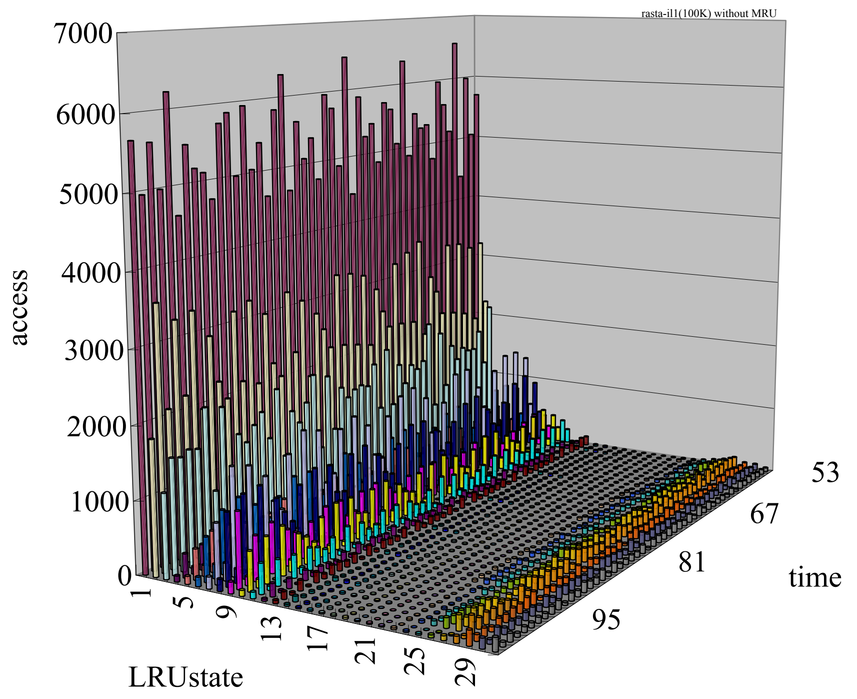


Figure 3.8: CPI space as a function of $T1$ and $T2$ in the `gcc` execution.



(a) With MRU accesses



(b) Without MRU accesses

Figure 3.9: Stack distance distribution on L1 instruction cache (`rasta`).

3.4 Conclusions

This chapter has designed a low-power cache control mechanism for the way-adaptable cache using power-gating scheme. To control the cache, the local and global behaviors of the locality of cache accesses on the cache are evaluated. The local evaluation of the locality is carried out using the statistics of LRU states of accessed entries. Besides, the global behavior of the locality monitored using n-bit state machines is considered to avoid aggressive reactions to the quick change in the degree of the locality in a short period. By considering the locality of accesses locally and globally, the on-line estimation of relative contribution of each way to the performance becomes possible, and therefore, it can avoid the use of absolute performance metrics such as miss rates and CPI for making resizing decision, which highly depend on individual applications.

Experimental results have indicated that the proposed cache mechanism achieves a stable cache control to find an appropriate trade-off between activated cache area and its achieved performance. The local assessment of cache requirement with two thresholds has achieved performance-oriented control with more activated ways to keep the higher performance, and lower-associativity-oriented control at the expense of CPI. In the global assessment, an n-bit asymmetric state machine works well to appropriately control the number of activated ways in the case of the benchmarks with highly-irregular access behaviors. Accordingly, this chapter has established the cache power control methodology to supply power to cache area depending on the cache requirement.

The proposed cache mechanism has assumed just a private cache in this chapter. In the following chapter, the proposed cache mechanism will be extended for the cache partitioning for CMP, which is current major processor architecture.

Chapter 4

Power-Aware Dynamic Cache Partitioning for CMPs

4.1 Introduction

Recently, a Chip Multi-Processor (CMP) has become a popular architecture for current general-purpose processors. The CMP is a promising architecture to effectively utilize a large amount of hardware budget on a chip and to enhance the performance by using thread-level parallelism [28]. Here, an on-chip shared cache among cores is the key elements to efficiently execute multiple workloads. However, the cache sharing causes the *shared cache conflicts*, and thereby it degrades the CMP performance. Therefore, to optimize the power efficiency and to achieve the high-performance and low-power cache for CMP, the cache mechanism has to solve the cache conflict problem.

The objective of this chapter is to establish a cache management methodology to avoid the shared cache conflicts in the two-core CMP with high-performance and low-power consumption. This chapter introduces a power-aware cache partitioning mechanism for the shared cache in order to avoid the shared cache conflict problem and to enhance the power efficiency in CMPs. The cache mechanism, which is an extension of the way-adaptable cache, has two functions: a way-allocation function for cache partitioning and a power control function for power-aware computing. The way-allocation function allocates ways of a set-associative cache to each core based on their contributions to the effective performance; each core can use the allocated ways as its private memory space as with the L1 cache. Ways are allocated to each core in proportion to the degree of their locality. As a result, the L2 cache is shared among cores without conflicts, and hence the mechanism allows each core to exclusively use an appropriately-sized cache area. The function also helps to find out less needed cache ways, which hardly contribute to the performance. The way-adaptable cache mechanism is applied to the power control function to conserve the power consumption by disabling these less needed ways. In this chapter, the performance of the proposed mechanism is evaluated in terms of three metrics. First, the weighted speedup is used to assess the performance improvement by the proposed way-allocation function. Then, the cache energy consumption is evaluated to show the power saving capability of the proposed power-aware cache partitioning mechanism. Finally, the hardware overhead to implement the control unit of the proposed mechanism is evaluated.

The rest of this chapter is organized as follows. Section 4.2 describes the shared cache conflict problem and its solutions with simple examples. Section 4.3 describes details on the proposed cache partitioning and power-aware cache mechanism under some assumptions. In particular, the way-allocation and power control functions are discussed. Section 4.4 shows the performance evaluation of the proposed mechanism. Section 4.5 concludes the chapter.

4.2 Shared Cache Conflict Problem

4.2.1 Problem Overview

CMPs usually have multilevel caches including a shared last-level cache among cores as shown in Figure 1.3. The cache sharing enables CMP to simultaneously execute multiple threads or programs. In such CMPs, threads or programs running on different cores can share data at a low latency. They can also utilize the large cache capacity because all cores can access the entire cache space. However, the cache sharing causes a problem of unfair performance degradation among cores. Data used by a specific program might occupy most of the cache space if the program has a higher access frequency or a larger working set than the other programs. As a result, the others cannot sufficiently use the cache space and therefore their cache misses considerably increase. In this paper, this unfair situation in the shared cache is named a *shared cache conflict*.

Unfair performance degradation by the shared cache conflict critically impacts its throughput and the effectiveness of an operating system scheduler [68]. Moreover, it also reduces the ability to enforce priorities and to provide Quality-of-Service (QoS) [69]. The fair cache sharing is required in order to improve not only the processor performance but also the system performance.

4.2.2 Conflict Mechanism

This section describes the basic example of the unfair performance degradation using the LRU stack as shown in Section 2.2.1. The following examples assume 4-way set associative cache shared by two cores. Applications (App.0, App.1) simultaneously executed in each core do not share their address space. This example focuses on accesses to any cache set. A, B and C are block addresses accessed by App.0 in the same cache set and X, Y and Z are by App.1. The following two examples describe the cases of the non-conflict access and the conflict access.

When the applications executed simultaneously have the same access frequency and the same access locality, the unfair performance degradation by the shared cache conflicts does not occur. Table 4.1 shows the changes of the LRU stack while the shared cache is alternately accessed from both of the applications as follows:

- App.0: A A A B B B C C C
- App.1: X X X Y Y Y Z Z Z

In Table 4.1, the "Miss" column indicates the timing and the kind of misses by each application (I: Compulsory (Initial) miss). In this case, every applications have just three compulsory misses and these miss rates are even (0.33), respectively. Therefore, the performance based on the miss rate does not unfairly degrade among cores.

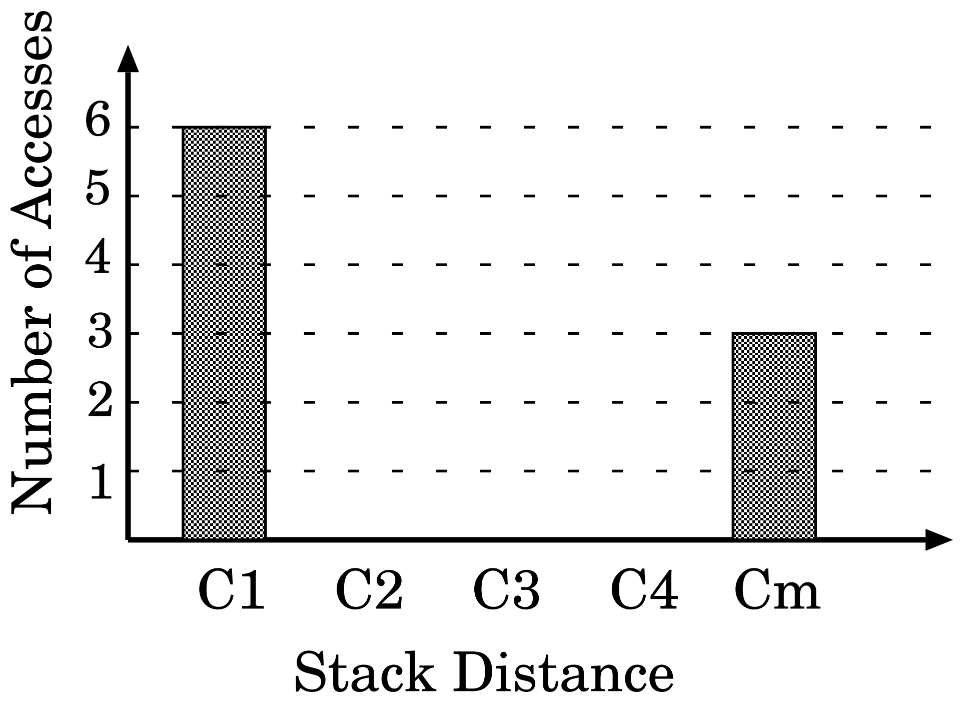
The next example shows the case of unfair performance degradation. The one-sided performance degradation occurs by getting rid of reusable data according to the LRU replacement policy. The unwished replacements are caused by the difference of access locality. This case assumes the combination of applications, which have different access localities. Each application access is indicated as follows:

- App.0: A A A B B B C C C
- App.1: X Y Z X Y Z X Y Z

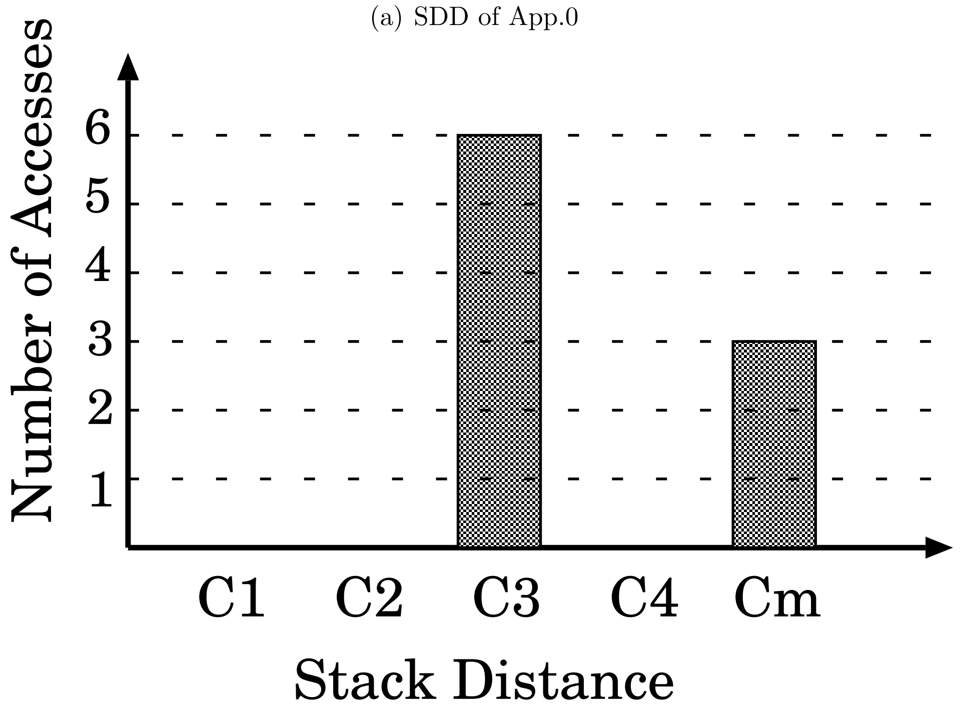
Table 4.1: LRU stack sequence of fair accesses.

Referenced Address	**LRU Stack**	**Miss**	
		App. 0	**App. 1**
A	A	I	
X	X A		I
A	A X		
X	X A		
A	A X		
X	X A		
B	B X A	I	
Y	Y B X A		I
B	B Y X A		
Y	Y B X A		
B	B Y X A		
Y	Y B X A		
C	C Y B X	I	
Z	Z C Y B		I
C	C Z Y B		
Z	Z C Y B		
C	C Z Y B		
Z	X C Y B		
	Num. of Misses	3	3
	Miss Rate	.33	.33
	Total Miss Rate	.33	

From stack distance distributions of each application as shown in Figure 4.1, App.0 has the high locality, which mostly accesses to the MRU line, and App.1 has lower locality than App.0. As Table 4.1 and 4.2 show the changes of the LRU stack while the cache is alternately accessed. Although misses by App.0 are only three initial misses (I), miss by App.1 are three initial misses and four conflict misses (C). The miss rate of App.1 is also larger than that of App.0 because the access frequencies are equal. Moreover, the total miss rate increases. Therefore, the different access locality causes unfair performance degradation among cores, and the throughput degrades compared to the case of fair access behavior.



(a) SDD of App.0



(b) SDD of App.1

Figure 4.1: Stack distance distribution of each application in example.

Table 4.2: LRU stack sequence of shared cache conflict.

Referenced Address	**LRU Stack**	**Miss**	
		App. 0	**App. 1**
A	A	I	
X	X A		I
A	A X		
Y	Y A X		I
A	A Y X		
Z	Z A Y X		I
B	B Z A Y	I	
X	X B Z A		C
B	B X Z A		
Y	Y B X Z		C
B	B Y X Z		
Z	Z B Y X		
C	C Z B Y	I	
X	X C Z B		C
C	C X Z B		
Y	Y C X Z		C
C	C Y X Z		
Z	Z C Y X		
	Num. of Misses	3	7
	Miss Rate	.33	.78
	Total Miss Rate	.56	

4.2.3 Solutions to the Shared Cache Conflict Problem

One of simple ways to solve the problem is increasing the cache size. However, this approach leads to an increase in power consumption and an inefficient use of a large cache for most applications. To solve the performance degradation due to the cache conflicts, several cache partitioning methods have been proposed [70]. The cache partitioning method exclusively allocates cache resource to cores/threads. Appropriate cache partitioning can avoid the shared cache conflict, and thereby can avoid the unfair performance degradation, because cores/threads can exclusively use the allocated cache area. Table 4.3 shows an example of avoiding the shared cache conflict by cache partitioning in the case of access pattern indicated as the cache conflict in Table 4.2. In this case, one line and three lines are respectively allocated to each application by cache partitioning appropriately. Each allocated area manages LRU stacks (LS0 and LS1). The number of misses by App.1 is less than one of Table 4.2, and both of applications have equal miss rate, because conflict misses do not occur by cache partitioning. Therefore, appropriate cache partitioning can be considered as one solution to protect CMPs from the shared cache conflict. However, the cache partitioning needs an algorithm, which fairly allocates cache area.

So far, many control methods for the cache partitioning have been proposed. Suh et al. have investigated the dynamic partitioning method for a shared cache [71] to solve the performance degradation problem. They proposed a marginal gain based cache partitioning algorithm with a low-overhead control scheme. Chandra et al. have also studied the performance impact of L2 cache sharing by threads on a CMP architecture, and proposed three models in order to accurately predict the performance using the stack distance and a circular sequence profile of each thread [36, 68]. The utility-based cache partitioning method [72] proposed by Qureshi et al. gains a high performance benefit with a low-overhead hardware configuration. In addition, they described that their partitioning algorithm has a high scalability. Iyer et al. have also proposed a QoS-enabled memory architecture [73] for CMP platforms. This architecture allocates more shared

Table 4.3: LRU stack sequence of cache partitioning.

Referenced	**LRU Stacks**		**Miss**	
Address	**LS0**	**LS1**	**App. 0**	**App. 1**
A	A		I	
X	A	X		I
A	A	X		
Y	A	Y X		I
A	A	Y X		
Z	A	Z Y X		I
B	B	Z Y X	I	
X	B	X Z Y		
B	B	X Z Y		
Y	B	Y X Z		
B	B	Y X Z		
Z	B	Z Y X		
C	C	Z Y X	I	
X	C	X Z Y		
C	C	X Z Y		
Y	C	Y X Z		
C	C	Y X Z		
Z	C	Z Y X		
	Num. of Misses		3	3
	Miss Rate		.33	.33
	Total Miss Rate		.33	

memory resources to high priority applications based on guidance from the operating system. However, these studies have not discussed the power consumption well.

This chapter introduces a cache partitioning method by cache ways to apply the proposed dynamic way-adaptable cache mechanism to CMPs. It is because CMPs need to avoid the performance degradation by the shared cache conflict to improve the power efficiency of CMPs. This chapter proposes a power-aware cache partitioning method to achieve the high-performance and low-power cache for CMP.

4.3 Cache Control Mechanism

4.3.1 Assumptions

The proposed power-aware cache partitioning mechanism is designed under the following assumptions.

- Two cores sharing an L2 unified cache on a chip configure a building block for CMPs. Each core has L1 private data/instruction caches.
- The L2 shared cache is a large, highly-associative on-chip cache, in which the power supply to each way can be controlled independently for power control using the power-gating circuits. Each core can access each way exclusively; the cache includes a mechanism that permits a core to access a way. In addition, the mechanism introduces an access monitoring unit to the L2 cache. This unit counts the numbers of accesses to MRU and LRU lines based on the true-LRU replacement policy.
- Co-scheduled threads on different cores are spawned from different applications. Therefore, they do not share any memory space, and each core executes a different application in the evaluations. The cache mechanism assumes that the operating system, which manages the thread scheduling, is responsible for switching the partitioning mode and the non-partitioning mode. Both context switch and thread migration between cores are also not considered in this chapter.

4.3.2 Mechanism Overview

A power-aware cache partitioning mechanism is introduced under the assumptions in the previous section. Figure 4.2 shows the basic concept of the proposed cache mechanism. Each way is allocated to one of two cores. A virtual partition defined as the boundary between two areas, each of which is allocated to one core, dynamically moves over the L2 cache during executions. Some ways are activated according to locality of memory reference in each area, and the other ways are inactivated for power saving. Each core can access allocated and activated ways only.

Figure 4.3 shows a control flow graph for an L2 cache shared with two cores. The first step is for cache access sampling to obtain statistics used in calculation of D. This step is carried out at fixed intervals, e.g., every 100,000 L2 accesses. The cache mechanism has a way-allocation function and a power control function. The former function decides which ways each core can use, and the latter decides how many ways are inactivated for power saving. In the case where both of the two cores have one or more inactivated ways, the way-allocation function is not performed, because way-allocation in such a situation does not decrease conflicts at all but causes a certain overhead.

The mechanism can randomly select a cache way to be reallocated or inactivated. Before inactivation, the data on the selected way are written back to the main memory for data coherency.

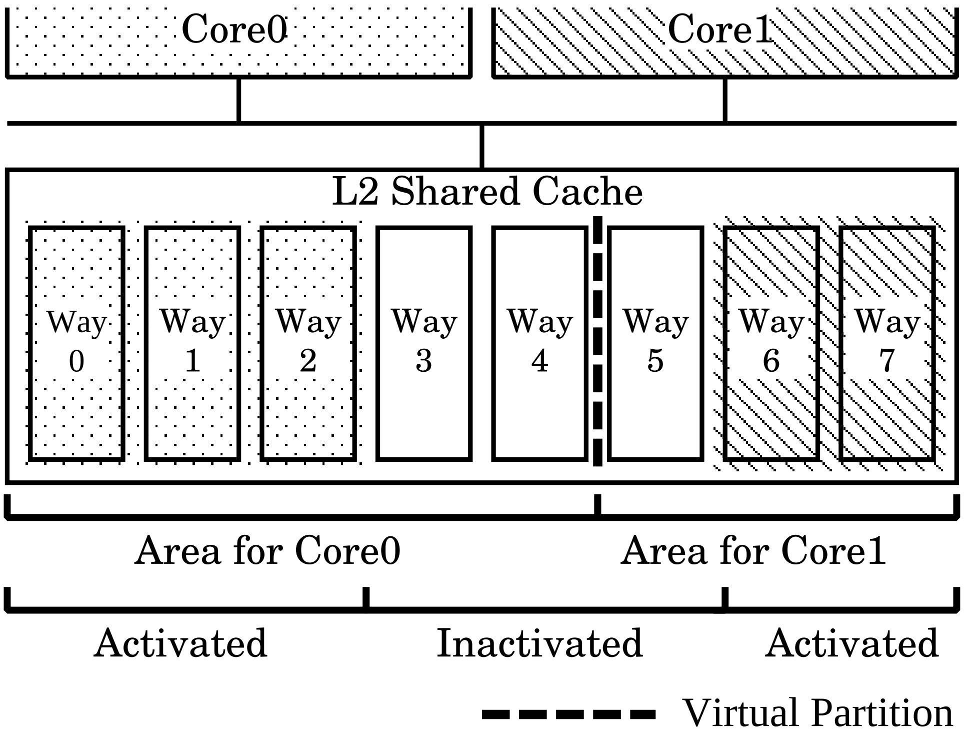


Figure 4.2: Basic concept of way-allocation and power control (in the case of two cores with an L2 shared 8-way cache).

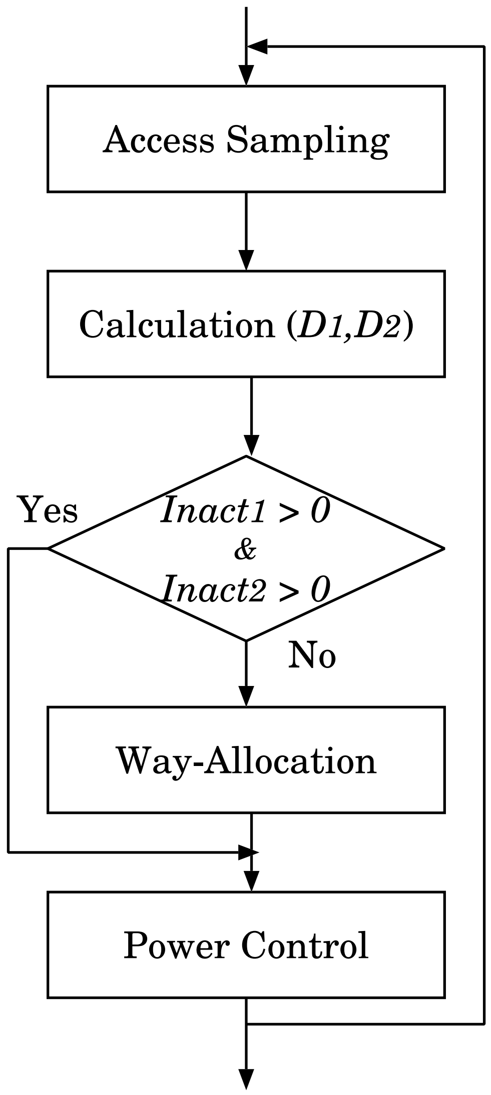


Figure 4.3: Control flow graph.

Algorithm 2 Determination of the number of allocated ways.

```
Require: $D_i = LRUcount_i/MRUcount_i$
  if $D_0 > D_1$ then
    $Alloc_0 += 1$
    $Alloc_1 -= 1$
  else if $D_0 < D_1$ then
    $Alloc_0 -= 1$
    $Alloc_1 += 1$
  else
    keep current configuration
  end if
```

4.3.3 Way-Allocation Function

The way-allocation function allocates each way of a set-associative cache to a core; each core can use the allocated ways as its private memory space like the L1 cache.

This section proposes an allocation method that considers the cache reference locality using D. Algorithm 2 shows the way-allocation algorithm. After calculating D_i from the number of cache accesses of core $i(i = 0, 1)$, the function determines whether the number of ways allocated to core i should be increased or decreased from the comparison between D_0 and D_1. Here, $Alloc_i$ is the number of ways that are allocated to core i, and satisfies the following conditions.

$$Alloc_{all} = \sum Alloc_i, \tag{4.1}$$

where $Alloc_{all}$ denotes the cache associativity. If $D_0 = D_1$, way-allocation is not performed.

4.3.4 Power Control Function

The power control function used in the proposed cache mechanism is based on the way-adaptable cache control proposed in Chapter 3. The number of activated ways increases so as to keep D_i between two thresholds to prevent unacceptable performance degradation.

To avoid iterating activation and inactivation of a way, the both local and global observation of the access behavior in memory accesses are required. For locally observing the behavior, D defined by Equation (2.8) is again employed. In addition, for globally observing its local behavior, the mechanism adopts an n-bit state machine.

Table 4.4: Simulation parameters.

Parameter	Value
Fetch width	8 insts
Decode width	8 insts
Issue width	8 insts
Commit width	8 insts
Inst. queue	64 insts
LSQ size	32 entries
L1 Icache	32kB, 2-way, 32B-line, 1 cycle latency
L1 Dcache	32kB, 2-way, 32B-line, 1 cycle latency
L2 shared cache	1024kB, 32-way, 64B-line, 14 cycle latency
Main memory	100 cycle latency
Frequency	1GHz
Technology	70nm
Vdd	0.9V

4.4 Performance Evaluation

4.4.1 Methodology

For the architectural cache simulation, a cycle accurate simulator was developed based on the M5 microprocessor architectural simulator tools [44] and CACTI version 4.2 [74]. The experiments examine a CMP of two Alpha-based cores with an L2 shared cache. The parameters used in the simulation are listed in Table 4.4. The simulation executes the first 500 million instructions using the reference input set. The sampling span of the proposed cache is 100,000 L2 cache accesses.

The simulation uses 15 workloads that consist of combinations of six benchmarks. Table 2.3 shows the benchmarks selected from the SPEC CPU2006 suite [45] for performance evaluation. Each core runs one independent benchmark program. In order to evaluate the proposal fairly, various benchmarks are selected based on their utility graphs [72]. The utility graphs of the benchmark programs are shown in Figure 4.4. These graphs indicate their performance in IPC as a function of the number of activated ways. Based on the utility graphs, the benchmarks are classified into three groups: *high-utility* (High), *saturating utility*(Sat) and *low-utility* (Low). The applications that have *high-utility* benefit from an increase in activated ways (e.g. `gcc`, `bzip2`). These applications have a lower access locality. The applications with *saturating utility* have a smaller working set than the applications with *high-utility*; giving more than eight ways dose not significantly improve their performance (e.g. `dealII`, `sjeng`). The applications that have *low-utility* do not benefit significantly from an increase in activated ways (e.g. `mcf`, `cactusADM`). These applications have a higher access locality. Two applications are selected from each group as shown in Table 2.3.

This experiment evaluates the proposed cache with the *weighted speedup* and its energy consumption. The weighted speedup is used as a metric for quantifying the performance of parallel processing, in which multiple applications execute in parallel on different cores. Let $SingleIPC_i$ be the IPC of the i-th application when it is executed on a single core and can exclusively use the entire resource of the CMP, and IPC_i be the IPC of an application when running with another application on the CMP. The weighted speedup is given by:

$$Weighted\ Speedup = \sum\left(\frac{IPC_i}{SingleIPC_i}\right). \tag{4.2}$$

With the information from M5 and CACTI, the cache energy consumption is calculated as follows:

$$E_{total} = E_d + E_s, \tag{4.3}$$

$$E_d = \int (E_{d\_data\_array} \times \frac{W_{active}}{W_{total}} \times A_{L2})dt + E_{d\_data\_other} + E_{d\_tag}, \tag{4.4}$$

$$E_s = \int (P_{s\_data\_array} \times \frac{W_{active}}{W_{total}})dt + E_{s\_data\_other} + E_{s\_tag}, \tag{4.5}$$

Table 4.5: Benchmark applications.

Name	Function	Utility
`gcc`	C Compiler	High
`bzip2`	Compression	High
`dealII`	Finite Element Analysis	Sat
`sjeng`	Artificial Intelligence: chess	Sat
`mcf`	Combinatorial Optimization	Low
`cactusADM(cactus)`	General Relativity	Low

where,

$$E_{d\_data\_array} = \text{dynamic energy consumed at bit lines in a data array when all ways are activated,} \tag{4.6}$$

$$P_{s\_data\_array} = \text{static power consumed at word lines when all ways are activated,} \tag{4.7}$$

$$W_{active} = \text{the number of activated ways,} \tag{4.8}$$

$$W_{total} = \text{the total number of ways in the L2 cache,} \tag{4.9}$$

$$A_{L2} = \text{the number of L2 cache accesses,} \tag{4.10}$$

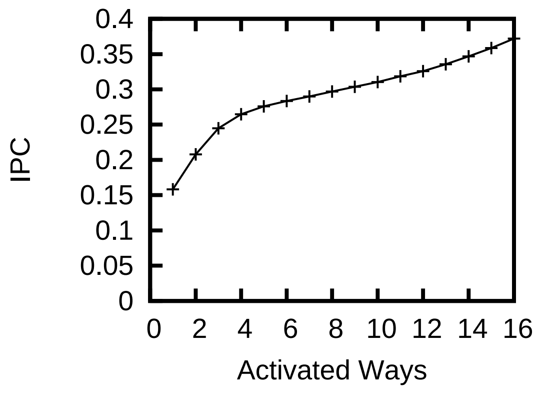
$$E_{d\_data\_other} = \text{the dynamic energy consumption at the other elements in a data array,} \tag{4.11}$$

$$E_{d\_tag} = \text{the dynamic energy consumption at the others,} \tag{4.12}$$

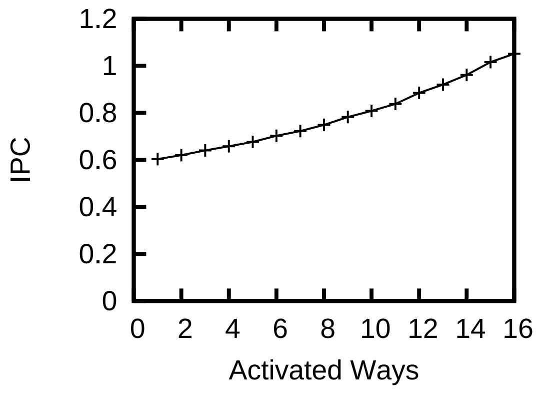
$$E_{s\_data\_other} = \text{the static energy consumption at the other elements in a data array, and} \tag{4.13}$$

$$E_{s\_tag} = \text{the static energy consumption at the others.} \tag{4.14}$$

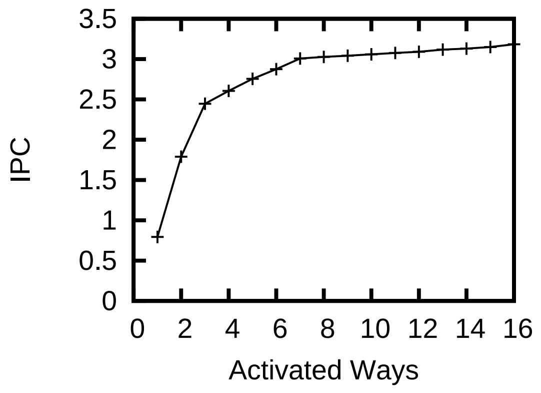
The total cache energy consumption, E_{total}, is the sum of E_d and E_s that are the dynamic energy consumption for transistor switching and the static energy consumption due to leakage current, respectively. This section also estimates E_d and E_s from the product of the array energy consumption and the proportion of the activated area.



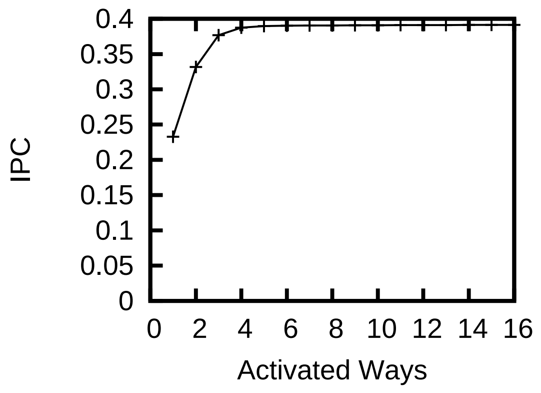
(a) `gcc` (High)



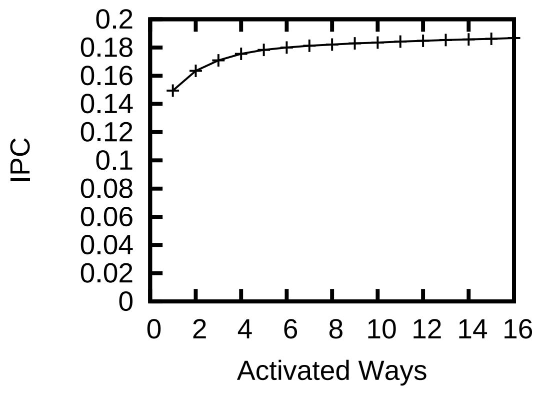
(b) `bzip2` (High)



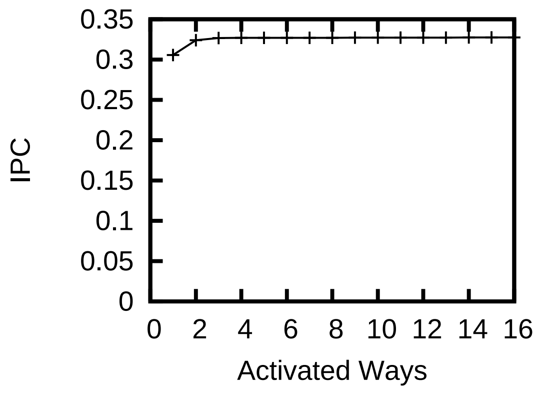
(c) `dealII` (Sat)



(d) `sjeng` (Sat)



(e) `mcf` (Low)



(f) `cactusADM` (Low)

Figure 4.4: Activated ways vs. IPC.

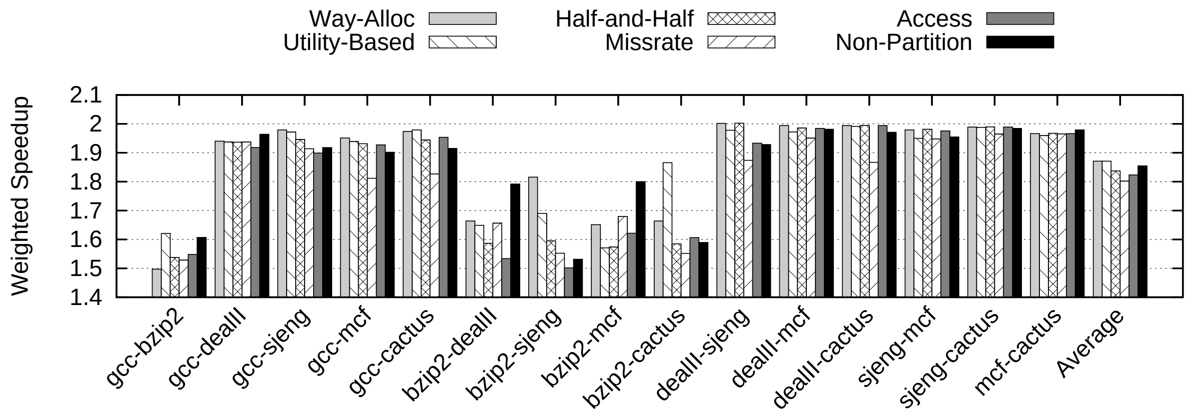


Figure 4.5: Performance of way-allocation.

4.4.2 Evaluation of Way-Allocation Function

To evaluate just the way-allocation function, this section compares the performance of the way-allocation function with five cache mechanisms: the utility-based cache partitioning method [72] (*Utility-Based*) which is representative of current partitioning schemes, *Half-and-Half* which is static equal partitioning between two cores, *Missrate* and *Access* which are the proposed way-allocation scheme using miss rate and the number of accesses respectively as the cache requirement metric D, and the non-partitioning shared cache (*Non-Partition*).

Figure 4.5 shows the evaluation results of the proposed way-allocation function (*Way-Alloc*) and the other five cache mechanisms. The horizontal axis in Figure 4.5 shows the combination of the benchmarks. The vertical axis shows the weighted speedups of CMPs. The bars labeled *Average* indicate the geometric means of weighed-speedups of each cache mechanism for individual benchmarks.

The weighted speedup of the proposed way-allocation outperforms the non-partitioning cache for 10/15 combinations listed in Figure 4.5. Especially, in the combinations which are improved the performance with the *Utility-Based* method, the proposed mechanism significantly improves the performance (for instance `bzip2-sjeng`). Therefore, it is obvious that the proposed cache mechanism can avoid cache conflicts and appropriately allocate cache ways. As a result, the proposed way-allocation based on locality assessment can prevent the performance degradation.

The weighted speedups of *Way-Alloc* and *Utility-Based* are the same as or more than that of *Half-and-Half* in almost all workloads. The *Half-and-Half* cannot achieve the performance improvement. Therefore, the results indicate that cache partitioning with the adaptive mechanism is beneficial. Moreover, the results also indicate that the proposed way-allocation has a performance comparable to the utility-based cache partitioning scheme. In addition, the proposed scheme can save the power consumption, while keeping a certain level performance by adding the power control function.

The figure also shows that the performance of *Missrate* and *Access* are lower than one of the non-partitioning cache on average. The results indicate that the way-allocations by the cache requirement cannot achieve appropriate cache partitioning. It is because that metrics *Access* and *Missrate* do not consider the access locality, and it is difficult to evaluate the appropriate cache size using them. Therefore, the proposed metric D based on the cache access locality has superiority in the cache partitioning ability to the metrics based on the miss rate or the number of accesses.

The weighted speedup obtained by the proposed cache mechanism outperforms the non-partitioning cache for almost all combinations without `bzip2`. Especially, the speedup improves in every combinations with `sjeng`. Therefore, it is obvious that the way-allocation function of the proposed cache mechanism is adequate for execution of the applications including the higher cache access locality: saturating or low utility. The validity of the proposed locality assessment model is confirmed by these results.

Every benchmark pair including `bzip2` leads to either significant performance improvement or degradation. As the utility of `bzip2` does not saturate, its performance improves in proportion to the number of activated ways as shown in Figure 4.4(b). Therefore, it is difficult to define the appropriate position of the virtual partition. A solution to this problem remains as the future work.

4.4.3 Evaluation of Way-Allocation with Power Control

Performance and Power Consumption

The proposed cache mechanism is evaluated and discussed in terms of the performance and the power consumption. Three different $(t1, t2)$-threshold settings for power control, $(0.1, 0.5)$, $(0.01, 0.05)$ and $(0.001, 0.005)$ are examined. An asymmetric 3-bit state machine is used to observe the global behavior in memory accesses. These experiments consider a write-back overhead that is needed to keep data coherency when ways are inactivated, but do not consider an overhead in power-gating for cache ways. Figures 4.6 and 4.7 show the weighted speedup and the energy consumption in all the benchmark combinations, respectively. The values of Figures 4.6 and 4.7 are normalized by the conventional cache. Figures 4.6 and 4.7 indicate that both of the weighted speedup and the energy consumption become their maximum values when the thresholds are $(0.001, 0.005)$ on almost all benchmark combinations. In contrast, when the thresholds are $(0.1, 0.5)$, both of them indicate their minimum values. When both benchmarks in the combinations have saturating or low utility (e.g. `sjeng-cactus`), the weighted speedup and the energy consumption are not sensitive to the configurations, and achieve high performance and low energy consumption. For example, in the case of a smaller threshold configuration $(0.001, 0.005)$ with the `sjeng-cactus` benchmarks, it can reduce 48% of the energy consumption while keeping the weighted speedup.

On average, the proposed cache can reduce energy consumption by 20% while keeping the performance, when the thresholds are smaller such as $(0.001, 0.005)$. On the other hand, in the case of larger thresholds such as $(0.1, 0.5)$, it can reduce a 55% energy consumption with a performance degradation of 13%. This experiment has confirmed that the values of the thresholds can decide the degree of the control policy between the performance-oriented and the energy-oriented configurations.

Figure 4.8 shows the dynamic behavior of the number of allocated and activated ways across time, when `gcc` and `sjeng` are executed in parallel. Most of the time, the

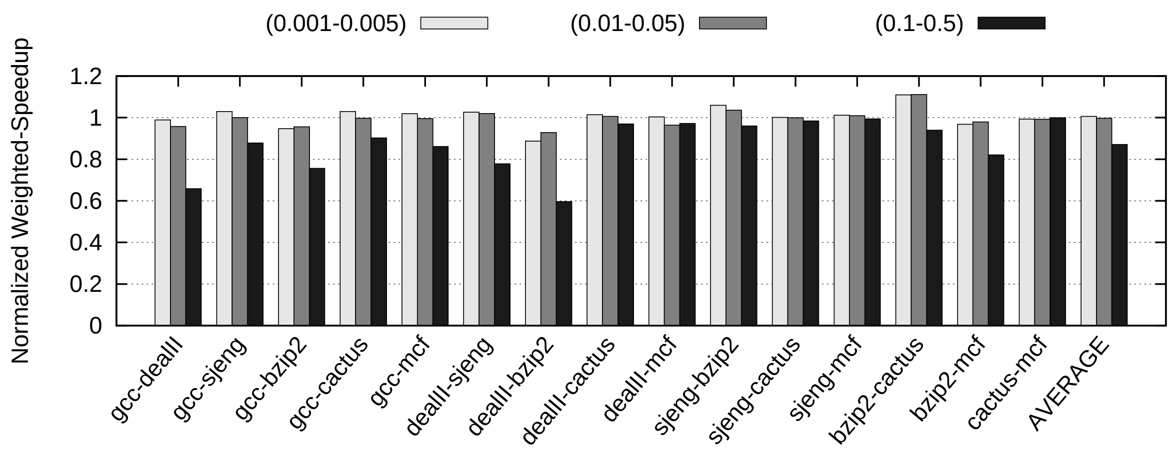


Figure 4.6: Effects of t_1 and t_2 on weighted speedup.

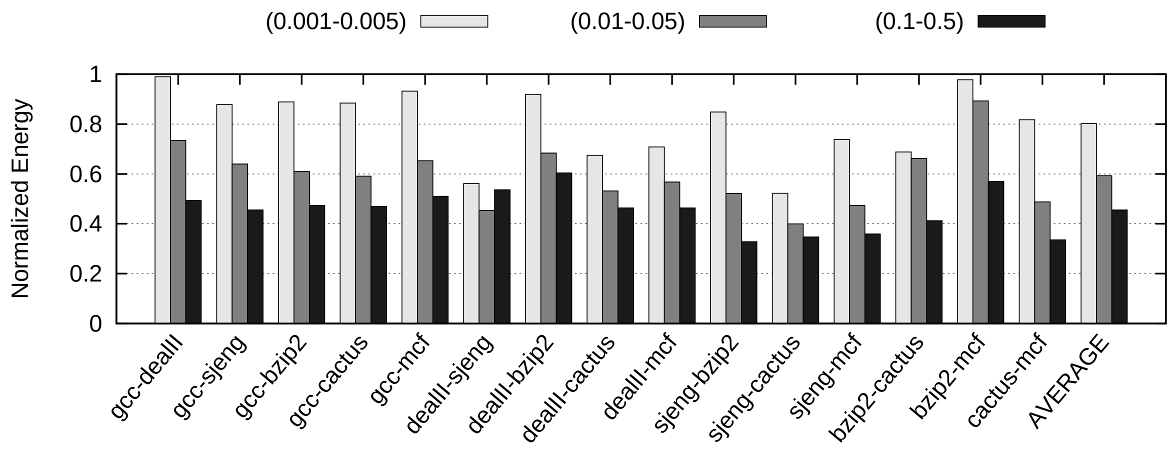
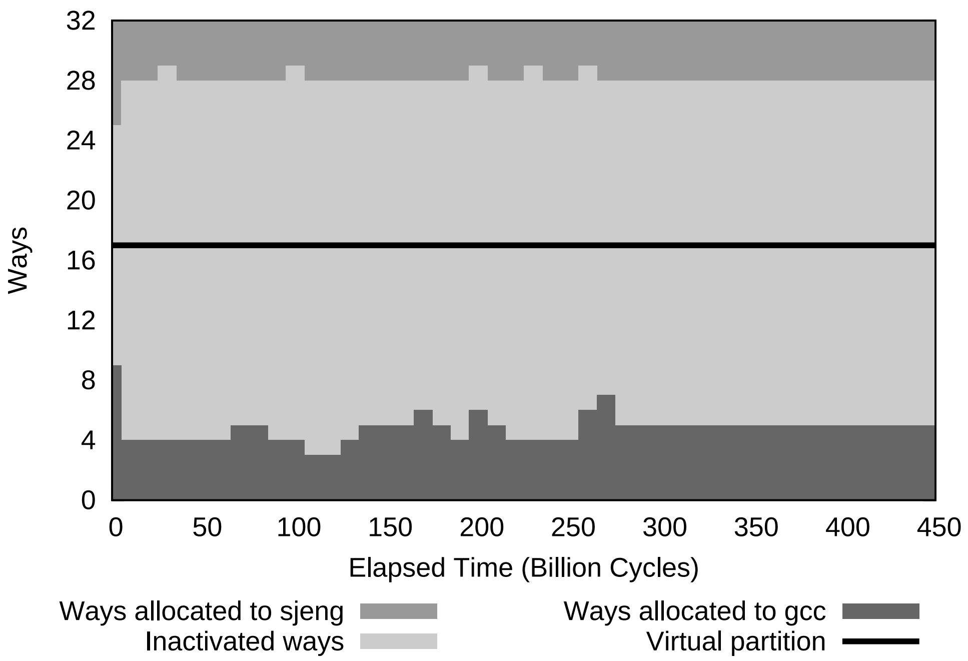


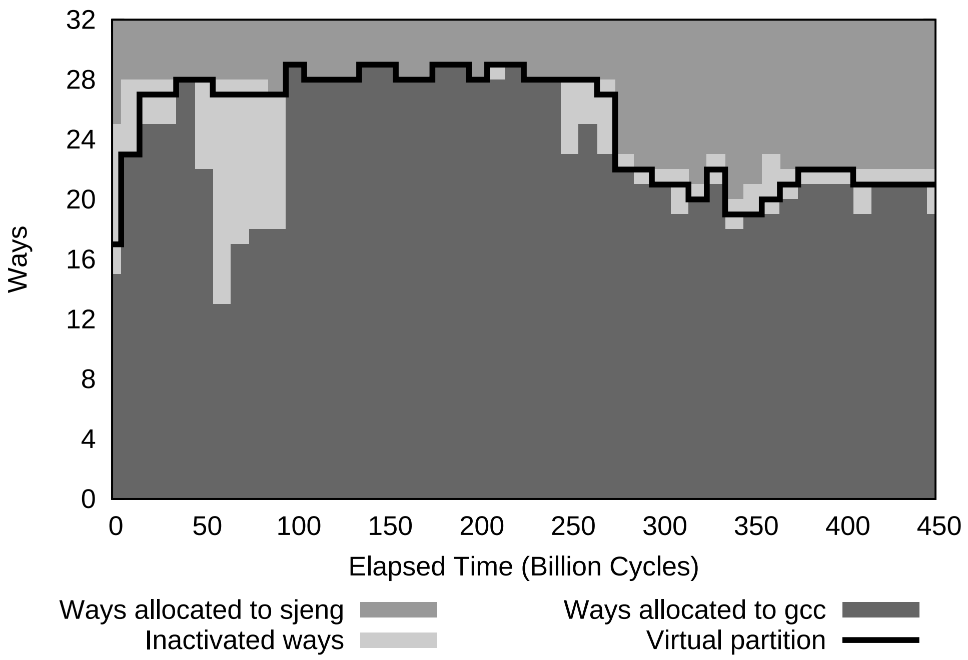
Figure 4.7: Effects of t_1 and t_2 on energy consumption.

allocation function is skipped, because most ways are inactivated for the energy-oriented configuration $(0.1, 0.5)$ as shown in Figure 4.8(a). There is no significant difference in the number of activated ways between two cores. On the other hand, with the performance-oriented configuration $(0.001, 0.005)$ as shown in Figure 4.8(b), almost all of ways are activated, and the way-allocation function effectively works. Especially, the activated ways of `gcc` increases significantly, because `gcc` have the lower access locality than `sjeng`. Therefore, the proposed mechanism appropriately allocates cache ways, and controls the activated cache area based on the access locality.

In the cases of `dealII-sjeng`, the energy consumption is not minimum when thresh-

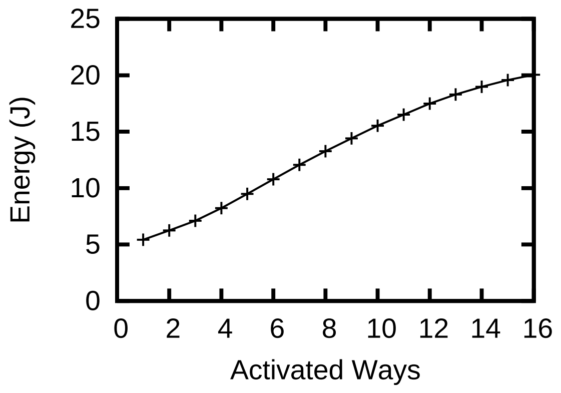


(a) $(t_1, t_2) = (0.1, 0.5)$

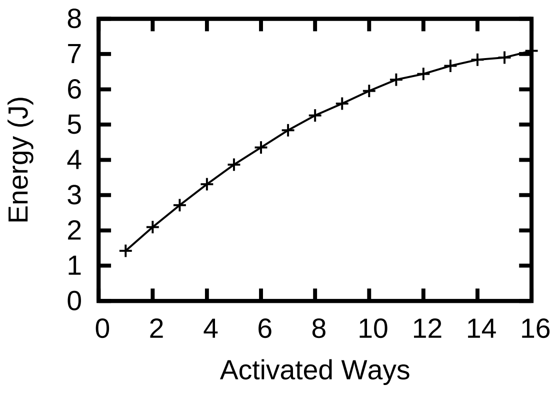


(b) $(t_1, t_2) = (0.001, 0.005)$

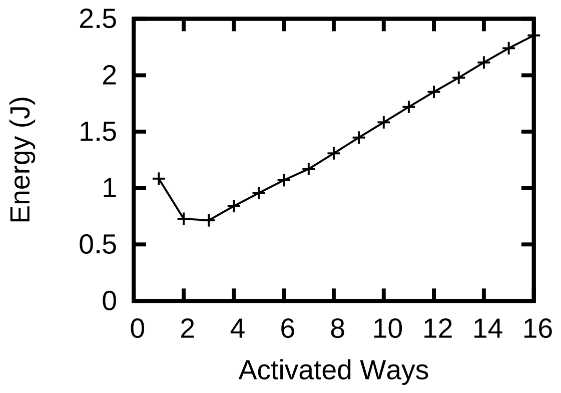
Figure 4.8: Elapsed time vs. Allocated ways (`gcc-sjeng`).



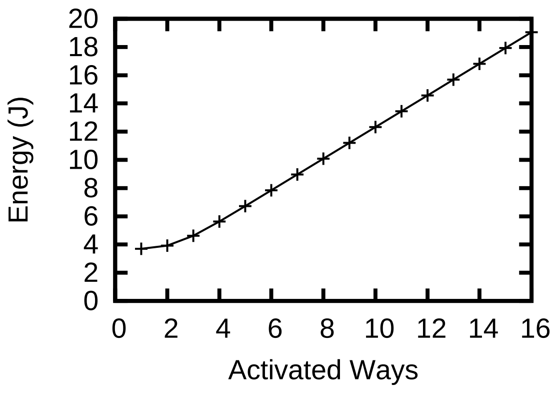
(a) gcc



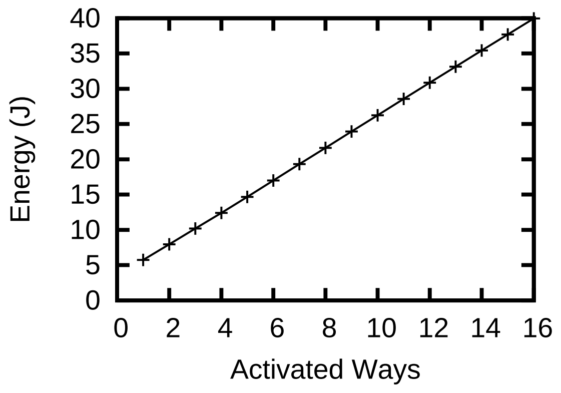
(b) bzip2



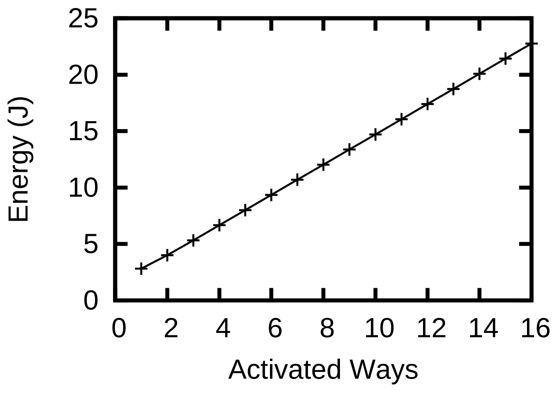
(c) dealII



(d) sjeng



(e) mcf



(f) cactusADM

Figure 4.9: Activated ways vs. Energy consumption.

olds are energy-oriented. This is because the execution time increases rapidly when the number of activated ways decreases. Figure 4.9 shows the energy consumptions of each benchmark with the number of activated ways. All but `dealII` show a monotonic increase with the number of activated ways. However, in the case of combinations with `dealII`, the minimum energy consumption is achieved when three ways are activated. Therefore, the proposed cache may not decrease the energy consumption on the energy-oriented configuration when the benchmarks like `dealII` are executed.

In the case of the combinations with `bzip2` (e.g. `gcc-bzip2`), the maximum weighted speedup is achieved at medium thresholds $(0.01, 0.05)$. This is attributed to the fact that the benchmark has the high-utility and the large working set as shown in Figure 4.4. The behavior analysis of combinations including high utility applications will be discussed in the future work.

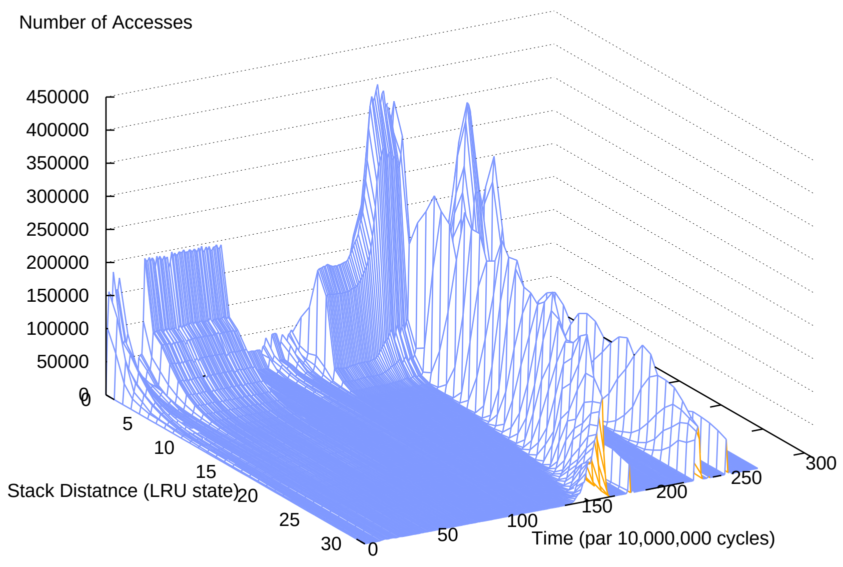


Figure 4.10: Elapsed time vs. Stack distance distribution (`gcc`).

Effects of Sampling Intervals

The dynamic behavior of the stack distance profiling depends on a characteristic of the program. Some programs frequently and drastically change a cache access behavior, and some programs hardly change it. Figures 4.10 to 4.15 show the dynamic behavior of the SDD for L2 cache, when each benchmark program is executed as a single workload. Note that the sampling interval of each SDD is 10 million cycles. From these figures, the SDD of `gcc` drastically changes and `mcf` and `sjeng` are relatively stable. Therefore, to appropriately control the cache power and partitioning according to the cache requirement, it is important to evaluate the sampling interval.

Let the parameter N be a bit width of the sampling counter, which defines the maximum sampling interval. The proposed control mechanism works after 2^N L2 cache accesses. When N is too small, the way-allocation and the power control are performed too often, resulting in an increase in the write-back overheads. On the other hand, the mechanism cannot control the number of ways on demand, when it is too large.

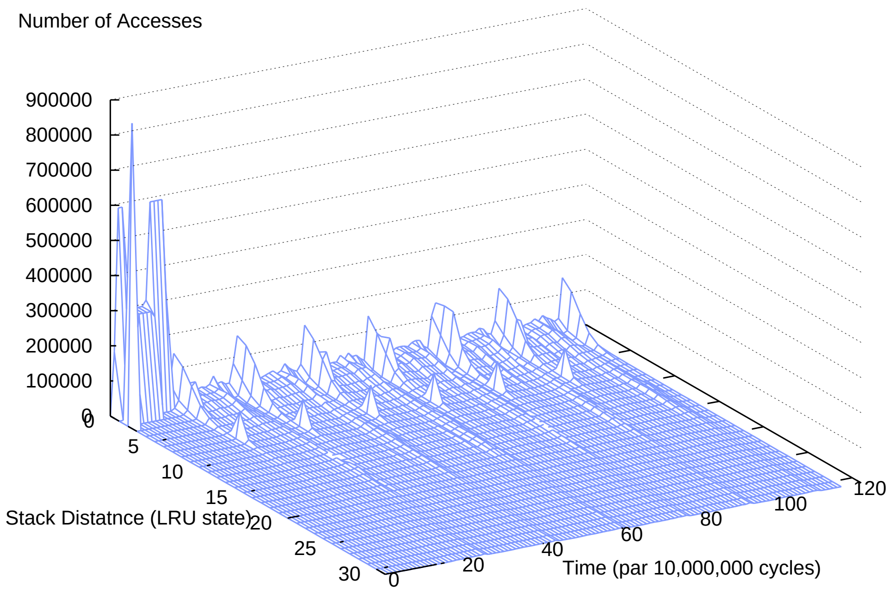


Figure 4.11: Elapsed time vs. Stack distance distribution (`bzip2`).

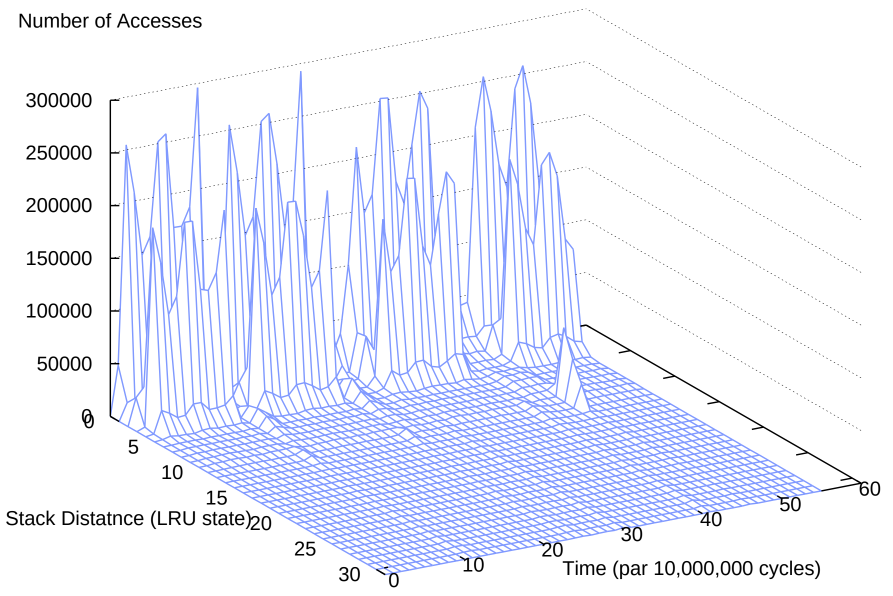


Figure 4.12: Elapsed time vs. Stack distance distribution (`dealII`).

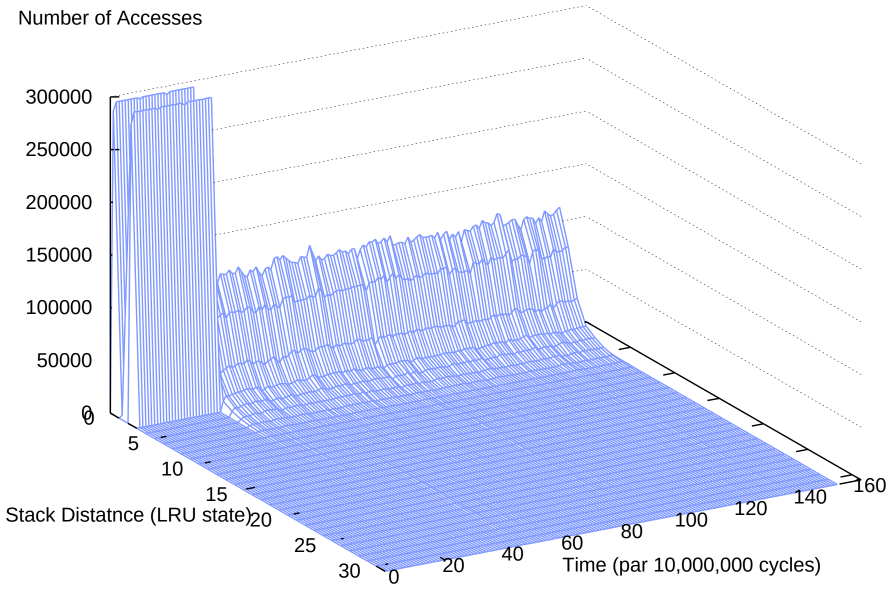


Figure 4.13: Elapsed time vs. Stack distance distribution (`sjeng`).

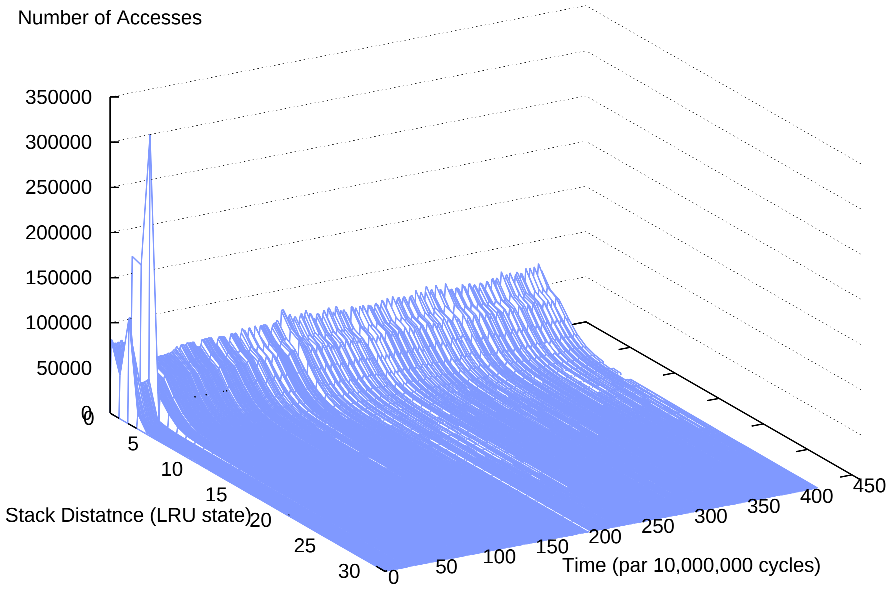


Figure 4.14: Elapsed time vs. Stack distance distribution (`mcf`).

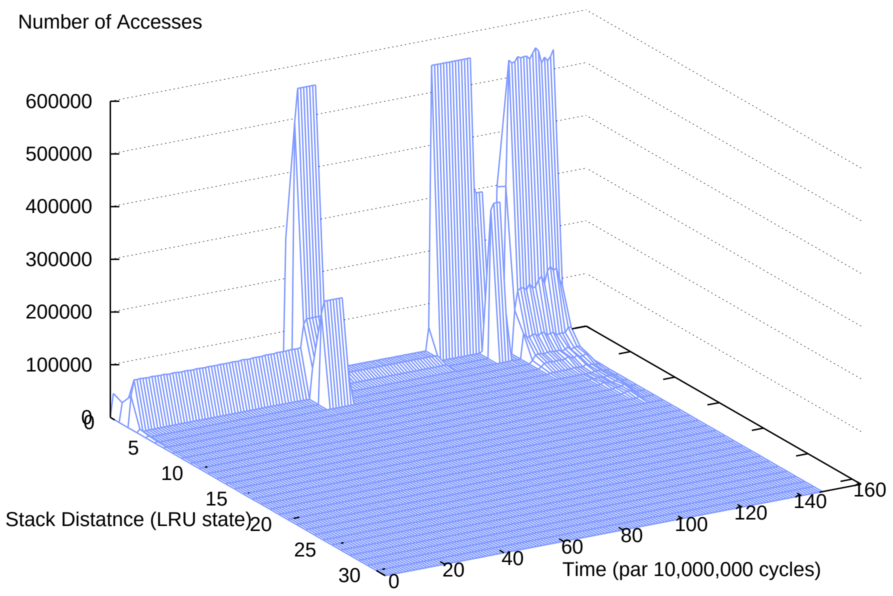


Figure 4.15: Elapsed time vs. Stack distance distribution (`cactus`).

Figure 4.16 compares the weighted speedups of four interval configurations (N=8, 12, 16, 20; each sampling interval is 2^8, 2^{12}, 2^{16}, or 2^{20} L2 cache accesses.) and the conventional shared cache. This experiment uses the performance-oriented configuration $(t1, t2) = (0.001, 0.005)$. The results indicate that decrease in the number of bits of the counter leads to a performance degradation. The results of the 8-bit configuration fall below the conventional cache in all workloads. Moreover, each workload has its own optimal control interval. A majority of workloads reaches their perk performances in the cases of 12-bit or 16-bit intervals. Hence, the control interval is an important parameter that decides the performance of the cache control mechanism. The cache mechanism needs an algorithm, which dynamically optimized the control interval for running applications.

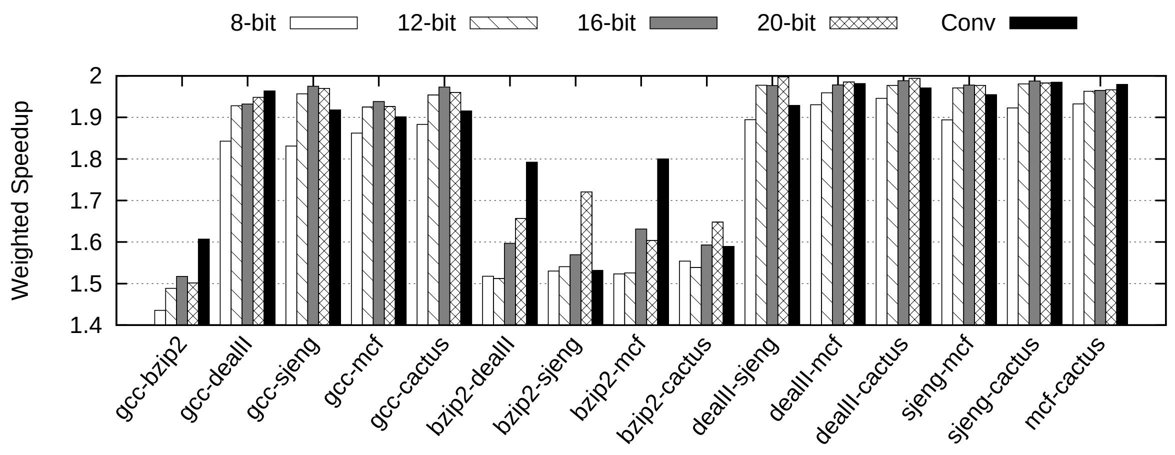


Figure 4.16: Effects of interval length.

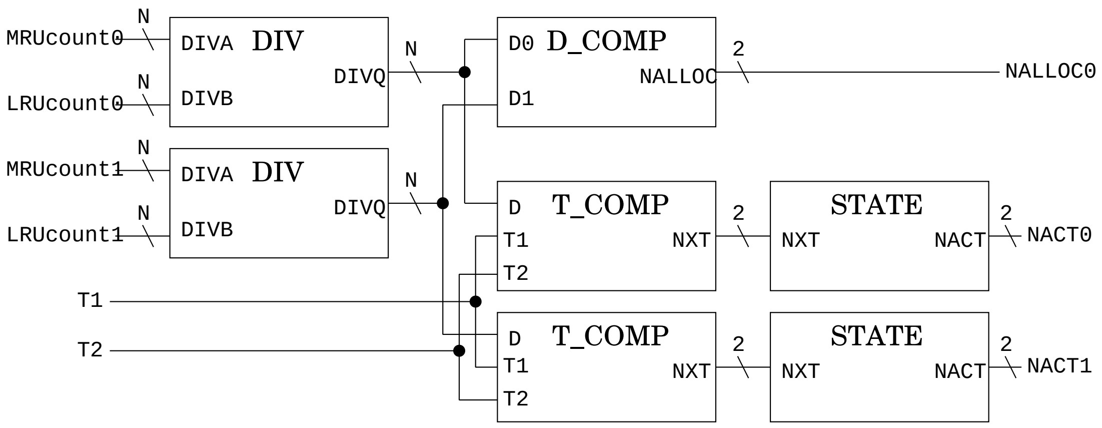


Figure 4.17: Cache control block diagram.

4.4.4 Evaluation of Hardware Overhead

To evaluate hardware overheads of the proposed cache control mechanism, this section designs a circuit as shown in Figure 4.17. The designed circuit consists of two dividers (DIV), three comparators (D_COMP and T_COMP), and two state machines (STATE). Four configurations (N = 8, 12, 16, 20) were designed by Rohm 0.18 μm CMOS Technology, using Synopsys EDA tools.

Table 4.4.4 shows the design results of the four configurations. The delay time is small enough compared with the sampling interval; therefore, the delay of the proposed mechanism hardly affects the way-allocation and power saving capability. Moreover, the circuit areas are extremely small compared with cache area. For comparison, this section estimates the area of a 1MB 32-way cache by 0.18 μm CMOS Technology using CACTI [74]. The proposed mechanism consumes only 0.1% of the cache memory area. The hardware overheads of the cache control mechanism, therefore, have an extremely small effect on the chip design.

Table 4.6: Hardware overheads.

	8-bit	12-bit	16-bit	20-bit
Delay (ns)	21.16	35.38	52.61	84.96
Area(μm^2)	27221	56314	92100	149831

4.5 Conclusions

This chapter has proposed a power-aware cache partitioning mechanism. This chapter has focused on the performance improvement by the function enhancement of the way-adaptable cache mechanism for CMP's shared cache. The mechanism has a way-allocation function based on the cache partitioning method and a power control function based on the way-adaptable cache mechanism. To solve the unfair performance degradation problem in CMP, the way-allocation function can decide the percentage of cache resources allocated to each core. The power control function decides the cache resources necessary for keeping the current performance. That is, the former is for a relative evaluation by comparing cores' demands for cache resources, and the latter is for an absolute evaluation by estimating the magnitude of the demand by each core.

The evaluation results have shown that the mechanism can save the power consumption, although the proposed cache mechanism and the utility-based scheme are comparable in the cache partitioning performance. The evaluation results also have shown that the power control policy of the mechanism can be adjusted from a performance-oriented configuration to an energy-oriented one. The way-allocation cache with a performance-oriented parameter setting can reduce an energy consumption by 20%, while keeping the performance in comparison with a conventional one. On the other hand, the cache with a energy-oriented parameter setting can reduce 55% energy consumption with a performance degradation of 13%. Moreover, the control mechanism has been designed to evaluate hardware overheads. The proposed cache control hardware has an extremely small overhead and gives a small effect on the chip design.

Therefore, this chapter has established a cache control mechanism for CMP to achieve high-performance and low-power.

Chapter 5

Conclusions

In this dissertation, the efficient cache mechanism has been explored for the low-power and high-performance microprocessors. In recent microprocessors, cache memories are one of important elements to achieve high-performance processing with overcoming the memory wall problem. However, there are two open problems to realize an efficient cache mechanism for the low-power and high-performance microprocessors; the power consumption problem and the shared cache conflict problem.

Under this situation, this dissertation has proposed the cache control mechanisms for a low-power and high-performance cache memory based on the following three approaches.

- The cache requirement metric based on the cache access locality has been introduced to assess the minimum cache capacity with keeping the performance of application running.
- The cache power control mechanism based on the cache requirement metric has been introduced to reduce the power consumption with keeping the performance of cache memories.
- The cache partitioning mechanism based on the cache requirement has been introduced to avoid the unfair performance degradation of shared cache memories for CMP architecture.

In Chapter 2, the cache access behaviors have been quantitatively analyzed and the cache requirement metric based on the cache access locality has been introduced in order to assess a minimum cache capacity with keeping the performance of application running. To analyze the cache access behaviors, a stack distance profiling method has been introduced. The stack distance profiling measures data reusability of cache lines by monitoring an LRU stack. It is possible to assess the temporal access locality and the degree of the cache requirement by analyzing stack distance distributions (SDDs), which are derived from the stack distance profiling. Additionally, to analyze and quantify the characteristic of SDD, SDD has to be approximately modeled. This chapter has hypothesized that the SDDs obey Zipf's law, and then verify the hypothesis against real applications. The verification has indicated that almost all applications have sufficiently large correlation coefficients between SDDs and Zipf's distributions. When the SDDs obey Zipf's distributions, the parameter in Zipf's law indicates characteristics of the cache access locality. To handily obtain the cache access locality and to utilize it for the evaluation of the cache requirement, this dissertation has proposed and verified the cache requirement metric D, which uses the number of MRU and LRU entries accessed in a certain period of cache accesses. According to this approach, this chapter has established the cache requirement metric for the high-performance and low-power cache mechanism.

In Chapter 3, the low-power cache mechanism using the power-gating mechanism has been designed. The proposed cache mechanism defines the activated and inactivated cache area based on the cache requirement. The mechanism dynamically monitors the cache requirement, and then it appropriately controls the activated cache area using the assessment of the local and global behavior. To realize appropriate cache control, the assessment of the cache requirement uses the local and global information of the cache requirement. The local assessment is carried out using the statistics of LRU states of accessed entries. Besides, the global assessment using the n-bit state machine is considered to avoid aggressive reactions to the quick change of the cache requirement in a short period. When the estimation is up-sizing, the power control hardware selects one from inactivated ways and activates it. When the estimation is down-sizing, the hardware selects one from activated ways and inactivates it after write back cached data to the main memory.

Experimental results have indicated that the proposed cache mechanism achieves the stable cache control to find an appropriate trade-off between activated cache area and its achieved performance. The local assessment of the cache requirement with two thresholds has achieved performance-oriented control with more activated ways to keep the higher performance, and lower-associativity-oriented control at the expense of CPI. In the global assessment, an n-bit asymmetric state machine works well to appropriately control the number of activated ways in the case of the benchmarks with highly-irregular access behaviors. Accordingly, this chapter has established the cache power control methodology to supply power to cache area depending on the cache requirement.

In Chapter 4, the extended low-power and high-performance cache mechanism has been designed for the shared cache for a two-core CMP. To effectively execute multiple workloads in the case where multiple cores executing independent programs share a cache, the shared cache requires a mechanism to avoid performance degradation by shared cache conflicts. The proposed shared cache mechanism consists of the way-allocation function for cache partitioning and the power control function. The way-allocation function of the proposed shared cache mechanism defines the allocated cache area by making comparison between the cache requirements of applications simultaneously executed in each core. The way-adaptable cache mechanism is applied to the power control function to conserve the power consumption by inactivating these less needed ways. The function appropriately controls the number of activated ways in the allocated cache area by the way-allocation function. The proposed shared cache mechanism searches most appropriate allocated and activated ways by iterating these two processes.

The evaluation results have shown that the proposed shared cache mechanism is comparable to the utility-based scheme with more effective power saving. The results of all functions have indicated that the proposed shared mechanism achieves the performance improvement, and moreover it reduces power consumption with keeping the performance by using the power control mechanism. The power control policy of the mechanism can be adjusted from a performance-oriented configuration to an energy-oriented one. The way-allocation cache with a performance-oriented parameter setting has reduced an energy consumption by 20%, while keeping the performance in comparison with a conventional one. On the other hand, the cache with an energy-oriented parameter setting can reduce 55% of energy consumption with a performance degradation of 13%. Moreover, the proposed control mechanism has been designed to evaluate hardware overheads of the cache control circuit. The evaluation of hardware overheads has shown that the cache control hardware has an extremely small overhead and a small effect on the chip design. These results have indicated that the proposed shared cache mechanism is a promising method for the high-performance and low-power cache design. Therefore, this chapter has established a cache control mechanism for CMP to achieve high-performance and low-power.

From above approaches, this dissertation proves that the proposed cache mechanism realizes the low-power and high-performance processors. This innovative progress presented in this dissertation conduces to the dawn of new cache design for the low-power and many-core processors.

Bibliography

[1] Fred J. Pollack. New microarchitecture challenges in the coming generations of cmos process technologies. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, page 2, 1999.

[2] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[3] Cameron McNairy and Rohit Bhatia. Montecito: A dual-core, dual-thread itanium processor. *IEEE Micro*, 25(2):10–20, 2005.

[4] Samuel Naffziger, Blaine Stackhouse, Tom Grutkowski, Doug Josephson, Jayen Desai, Elad Alon, and Mark Horowitz. The implementation of a 2-core, multi-threaded itanium family processor. *IEEE Journal of Solid-state circuits*, 41(1):197–209, January 2006.

[5] Adrian Carbine and Derek Feltham. Pentium pro processor design for test and debug. *IEEE Design & Test of Computer*, 15(3):77–82, 1998.

[6] S. Rusu, S. Tam, H. Muljono, D. Ayers, and J. Chang. A dual-core multi-threaded Xeon®processor with 16mb l3 cache. In *ISSCC: Technical Paper of IEEE International Solid-State Circuits Conference*, pages 315–325, February 2006.

[7] Stefan Rusu, Simon Tam, Harry Muljono, David Ayers, Jonathan Chang, Brian Cherkauer, Jason Stinson, John Benoit, Raj Varada, Justin Leung, Rahul D. Limaye,

and Sujal Vora. A 65-nm dual-core multithreaded Xeon processor with 16-mb l3 cache. *IEEE Journal of Solid-state circuits*, 42(1):17–25, January 2007.

[8] 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.

[9] Anantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen. Low-power cmos digital design. *IEEE Journal of Solid-state circuits*, 27(4):473–484, April 1992.

[10] Harry J. M. Veendrick. Short-circuit dissipation of static cmos circuitry and its impact on the design of buffer circuits. *IEEE Journal of Solid-state circuits*, 19(4):468–473, August 1984.

[11] J. Abam Butts and Gurindar S. Sohi. A static power model for architects. In *MICRO 33: Proceedings of the 33th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 191–201, December 2000.

[12] Nam Sung Kim, Todd Austin, David Blaauw, Trevor Mudge, Krisztián Flautner, Jie S. Hu, Mary Jane Irwin, Mahmut Kandemir, and Vijaykrishnan Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, 2003.

[13] International technology roadmap for semiconductors. http://public.itrs.net.

[14] Yan Meng, Timothy Sherwood, and Ryan Kastner. Exploring the limits of leakage power reduction in caches. *ACM Transactions on Architecture and Code Optimization*, 2(3):221–246, 2005.

[15] Yan Meng, Timothy Sherwood, and Ryan Kastner. On the limits of leakage power reduction in caches. In *the Proceedings of International Symposium on High-Performance Computer Architecture (HPCA-11)*, February 2005.

[16] M. R. Stan and K. Skadron. Power-aware computing. *Computer*, 36(12):35–38, 2003.

[17] 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), February 2004.

[18] Ticky Thakkar. Mobile internet devices enabling the best internet experience in your pocket. In *COOLChips XI: Proceedings of the IEEE Symposium on Low-Power and High-Speed Chips*, pages 329–339, April 2008.

[19] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[20] Srilatha Manne, Artur Klauser, and Dirk Grunwald. Pipeline gating: speculation control for energy reduction. *ACM SIGARCH Computer Architecture News*, 26(3):132–141, 1998.

[21] N.P. Jouppi, P. Boyle, J. Dion, M.J. Doherty, A. Eustace, R. Haddad, R. Mayo, S. Menon, L. Monier, D. Stark, S. Turrini, and L. Yang. A 300 mhz 115 w 32 b bipolar ecl microprocessor with on-chip caches. In *ISSCC: Technical Paper of IEEE International Solid-State Circuits Conference*, pages 84–85, February 1993.

[22] N.P. Jouppi, P. Boyle, J. Dion, M.J. Doherty, A. Eustace, R.W. Haddad, R. Mayo, S. Menon, L.M. Monier, D. Stark, S. Turrini, J.L. Yang, R. Hamburgen, J.S. Fitch, and R. Kao. A 300-mhz 115-w 32-b bipolar ecl microprocessor. *IEEE Journal of Solid-state circuits*, 28(11):1152–1166, November 1993.

[23] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. *ACM SIGOPS Operating Systems Review*, 30(5):2–11, 1996.

[24] David W. Wall. Limits of instruction-level parallelism. *ACM SIGPLAN Notices*, 26(4):176–188, 1991.

[25] Randal Goodall, Denis Fandel, Alan Allan, Paul Landler, and Howard R. Huff. Long-term productivity mechanisms of the semiconductor industry. In *Proceedings of the American Electorochemical Society Semiconductor Silicon*, pages 125–143, 2002.

[26] Cyrus Bazeghi, Francisco J. Mesa-Martinez, and Jose Renau. ucomplexity: Estimating processor design effort. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 209–218, 2005.

[27] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The m-machine multicomputer. In *MICRO 28: Proceedings of the 28th annual international symposium on Microarchitecture*, pages 146–156, 1995.

[28] B. A. Nayfeh and K. Olukotun. A single-chip multiprocessor. *Computer*, 30(9):79–85, Sept. 1997.

[29] Advanced Micro Devices, Inc. http://www.amd.com/us-en/.

[30] Ron Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, March 2004.

[31] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: a 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, March-April 2005.

[32] B. A. Nayfeh and K. Olukotun. Exploring the design space for a shared-cache multiprocessor. In *ISCA '94: Proceedings of the 21st annual international symposium on Computer architecture*, pages 166–175, 1994.

[33] George K. Zipf. *Human Behavior and the Principle of Least Effort.* Addison-Wesley, 1949.

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

[35] Mark D. Hill and Alan J. Smith. Evaluating associativity in cpu caches. *IEEE Transactions on Computers*, 38(12):1612–1630, 1989.

[36] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 340–351, 2005.

[37] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: evidence and implications. *INFOCOM '99: Proceedings of the Eighteenth Annual Joint Conference*, 1:126–134, Mar 1999.

[38] Carlos Cunha, Azer Bestavros, and Mark Crovella. Characteristics of World Wide Web Client-based Traces. Technical Report BUCS-TR-1995-010, Boston University, CS Dept, Boston, MA 02215, April 1995.

[39] Matei Ripeanu. Note on zipf distribution in top500 supercomputers list. Technical report, IEEE Distributed Systems Online, October 2006.

[40] Dror G. Feitelson. On the interpretation of top500 data. *International Journal of High Performance Computing Applications*, 13(2):146–153, 1999.

[41] TOP500 Supercomputer Sites. http://www.top500.org/.

[42] B. Gutenberg and C. F. Richter. Frequency and energy of earthquakes. *Seismicity of the Earth and Associated Phenomena*, pages 17–19, 1954.

[43] A. Bardine, P. Foglia, G. Gabrielli, C. A. Prete, and P. Stenström. Improving power efficiency of d-nuca caches. *ACM SIGARCH Computer Architecture News*, 35(4):53–58, Sept. 2007.

[44] N.L. Binkert, R.G. Dreslinski, L.R. Hsu, K.T. Lim, A.G. Saidi, and S.K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, July-Aug. 2006.

[45] The Standard Performance Evaluation Corporation. http://www.spec.org/.

[46] S.Rusu, H.Muljono, and B.Cherkauer. Itanium 2 Processor 6M: Higher Frequency and Larger L3 Cache. *IEEE Micro*, 24(2):10–18, 2004.

[47] Intel Corporation. Intel StrongARM SA-110 Microrprocessor . *http://www.intel.com/design/strong/sa_110doc.htm*, 1999.

[48] Kanad Ghose and Milind B. Kamble. Energy efficient cache organizations for superscalar processors. In *Proceedings of the Power-Driven Microarchitecture Workshop*, 1998.

[49] Johnson Kin, Munish Gupta, and William H. Mangione-Smith. The filter cache: an energy efficient memory structure. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 184–193, 1997.

[50] N.B.M. Hajj, C. Polyckronopoulos, and G. Stamoulist. Architectural and compiler support for energy reduction in the memory hierarchy of high performance microprocessors. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, pages 70–75, 1998.

[51] David H. Albonesi. Selective cache ways : On-demand cache resource allocation. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 248–259, 1999.

[52] Ashutosh S. Dhodapkar and James E. Smith. Managing multi-configuration hardware via dynamic working set analysis. *ACM SIGARCH Computer Architecture News*, 30(2):233–244, 2002.

[53] Steve Dropsho, Alper Buyuktosunoglu, Rajeev Balasubramonian, David H. Albonesi, Sandhya Dwarkadas, Greg Semeraro, Grigorios Magklis, and Michael L. Scott. Integrating adaptive on-chip storage structures for reduced dynamic power. In *PACT*

'02: Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques, pages 141–152, 2002.

[54] B. Calder, D. Grunwald, and J. Emer. Predictive sequential associative cache. In *HPCA '96: Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, pages 244–253, 1996.

[55] Koji Inoue, Tohru Ishihara, and Kazuaki Murakami. Way-predicting set-associative cache for high performance and low energy consumption. In *ISLPED '99: Proceedings of the 1999 international symposium on Low power electronics and design*, pages 273–275, 1999.

[56] Michael D. Powell, Amit Agarwal, T. N. Vijaykumar, Babak Falsafi, and Kaushik Roy. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 54–65, 2001.

[57] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 240–251, New York, NY, USA, 2001. ACM.

[58] Krisztián Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy caches: simple techniques for reducing leakage power. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 148–157, Washington, DC, USA, 2002. IEEE Computer Society.

[59] 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, 2001.

[60] 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 *ISLPED '00: the 2000 international symposium on Low power electronics and design*, pages 90–95. ACM, 2000.

[61] Se-Hyun Yang, Babak Falsafi, Michael D. Powell, Kaushik Roy, and T. N. Vijaykumar. An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance i-caches. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 147–158, 2001.

[62] John Hennessy and David Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, fourth edition, 2007.

[63] David A. Patterson and John L. Hennessy. *Computer organization & design: the hardware/software interface*. Morgan Kaufmann Publishers Inc., second edition, 1997.

[64] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2):59–67, 1996.

[65] Compaq. Alpha 21264 Microprocessor Hardware Reference Manual. *Technical report, Compaq Computer Corporation*, 1999.

[66] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335, 1997.

[67] N. Megiddo and D. S. Modha. A Simple Adaptive Cache Algorithm Outperforms LRU. *IBM Research Report*, 2003.

[68] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, 2004.

[69] David Tam, Reza Azimi, Livio Soares, and Michael Stumm. Managing shared l2 caches on multicore systems in software. In *WIOSCA: Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture*, 2007.

[70] Miquel Moreto Planas, Francisco Cazorla, Alex Ramirez, and Mateo Valer. Explaining dynamic cache partitioning speed ups. *IEEE Computer Architecture Letter*, 6(1), 2007.

[71] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *Journal of Supercomputing*, 28(1):7–26, 2004.

[72] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432, 2006.

[73] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 35(1):25–36, 2007.

[74] Steven J. E. Wilton and Norman P. Jouppi. Cacti: an enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, May 1996.