論文 / 著書情報
Article / Book Information

題目(和文)	メニーコアの協調キャッシュに関する研究
Title(English)	Research on many-core cooperative caching
著者(和文)	藤枝直輝
Author(English)	Naoki Fujieda
出典(和文)	学位:博士(工学), 学位授与機関:東京工業大学, 報告番号:甲第9190号, 授与年月日:2013年3月26日, 学位の種別:課程博士, 審査員:吉瀬 謙二
Citation(English)	Degree:Doctor (Engineering), Conferring organization: Tokyo Institute of Technology, Report number:甲第9190号, Conferred date:2013/3/26, Degree Type:Course doctor, Examiner:
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

TOKYO TECH Pursuing Excellence
TOKYO INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Research on Many-core Cooperative Caching

（メニーコアの協調キャッシュに関する研究）

A dissertation submitted in partial fulfillment of
the requirements for the degree of Doctor of Engineering

Naoki Fujieda

March 2013

Graduate School of Information Science and Engineering
Department of Computer Science
Tokyo Institute of Technology

Abstract

The trend of progress in processor architectures has changed to increasing the number of processor cores in a single chip. The past fashion was improving the working frequency and/or the average number of instruction per cycle (IPC) they can execute, that is, the single-thread performance. However, this strategy does not match with the hardware cost and the power consumption any more. Multicore processors are now the majority of processors even for cell phones and handheld game consoles. If this trend continues, the era of many-core processors that have tens or hundreds of cores in a chip will come in the near future.

In the many-core era, there are increasing demands for efficient memory systems to realize high-performance processors. They usually have multiple levels of cache memories and each level has different function. The primary or L1 caches are the fastest and the smallest caches and thus supply the most common data as quickly as possible. On the other hand, the last level caches (LLCs) are the largest caches and expected to provide as many data from there, rather than from the main memory, as they can. Furthermore, they are not so sensitive to latency as the higher caches. Therefore, they can be complex and smart to meet the demand.

Recently, for the LLCs, the cooperative caching (CC) has been proposed. In the CC, each core keeps accessed data mainly in its own cache like private caches. The difference is that the CC allows cores transferring (or spilling) a part of their evicted cache lines to other cores. As a result, cores that are short of cache space can dynamically borrow some cache lines from other cores leaving most of their caches unused. However, to maximize the advantage of the CC, optimized methods of spilling are essential. Hence, there are a number of previous studies on them.

In this thesis, I propose a brand-new approach named ASCEND (Adaptive Spill Control with extra ENtries of Directory) for optimizing the CC. Not to mention high efficiency, high scalability — capability to apply the method to tens of cores — is also required for many-core processors. There were no past researches to achieve both of them; ASCEND is the first one.

The target architecture of my approach is the Distributed Cooperative Caching (DCC), a scalable variation of the CC. It has distributed organizations called Distributed Coherence Engines (DCEs) to keep shared cache lines coherent. An important discovery in my proposal is that they need some extra entries for preventing performance degradation. I find that the currently unused hardware that was

invalidated is useful for analyzing characteristics of cores in detail, for it has the information of the cache lines that were recently removed from the chip. ASCEND extracts them by two kinds of units named Spiller Selectors and Receiver Selectors and control the spilling adaptively.

The contributions of this thesis are threefold. One of the secondary contributions is the classification of various recent studies to improve cache performance. I divide them into some categories by their targets or features. Since there are many studies on this field, it may be hard to hit upon a new idea when we persist on only one category. Instead, mixing knowledge of more than one category can become a hint of it. I believe my classification provides a wide view of cache optimization schemes and a clue of a novel approach for them.

The other secondary contribution is implementation of a useful infrastructure for research on many-core processors with shared memory. To evaluate the methods in the many-core environment, I am developing a simulator of many-core processors with the CCs named SimMccc. It utilizes SimMips, a simple and practical MIPS system simulator, and inherits the characteristics from this. Hence, it can be utilized as a great infrastructure of the CCs.

The most important contribution of this thesis is to show an efficient and scalable method for the CCs to spill effectively, through the proposal and the evaluation of ASCEND. It is expected to play an important role in the future many-core era. I evaluate various methods in both multicore and many-core environments. The results showed that my method was more efficient than an existing efficient but non-scalable method in the multicore environment. They also showed that it outperformed a scalable but not-so-efficient method in the many-core environment. That is to say, the evaluation of two confirmed that it is an efficient and scalable method.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The rapid improvement in processor performance from the late 1980s to 2003 mainly came from the rise in operating frequency. Figure 1.1 shows the improvement in processor performance and its details. In addition to frequency, various architectural techniques improved the performance per clock cycle. These techniques include pipelining, branch prediction, and the theme of this thesis — efficient use of cache memories.

The trend of progress in processor technology has changed to increasing the number of processor cores in a single chip. In 2003, the rapid rise in frequency ended due to increasing power consumption and wire delay. Since the late 2000s, multicore processors that integrate multiple cores in a single chip have been widely used. If the trend continues, the era of many-core processors that have tens or hundreds of cores in a chip will come in the near future.

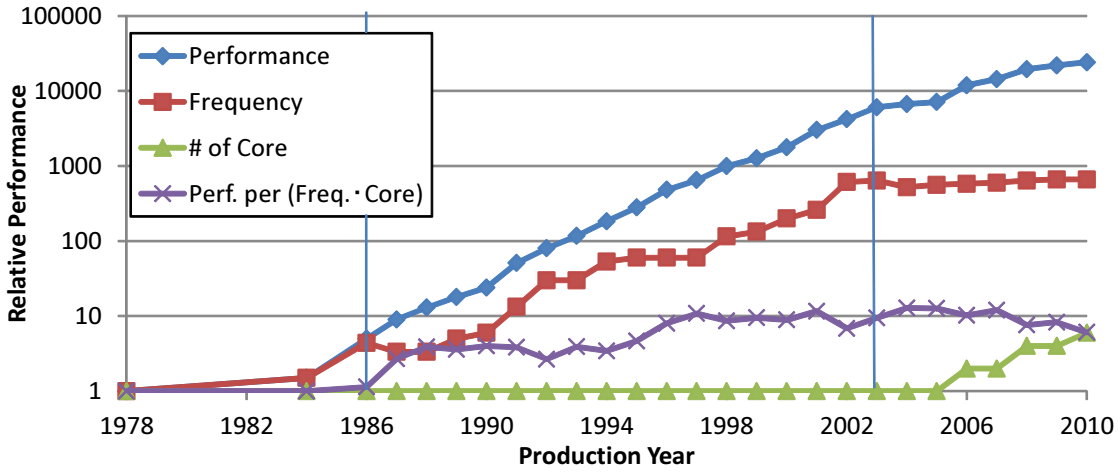


Figure 1.1 The change of processor improvement and its details. All values are relative to VAX-11/780 in 1978. Each point represents the performance of a specific processor that was shipped in the corresponding year. The data are quoted from [1]

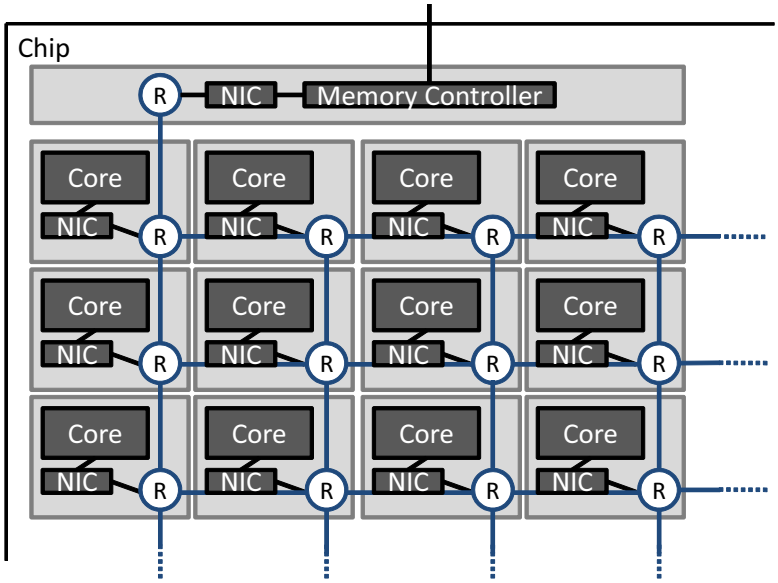


Figure 1.2 A conceptual model of many-core processors.

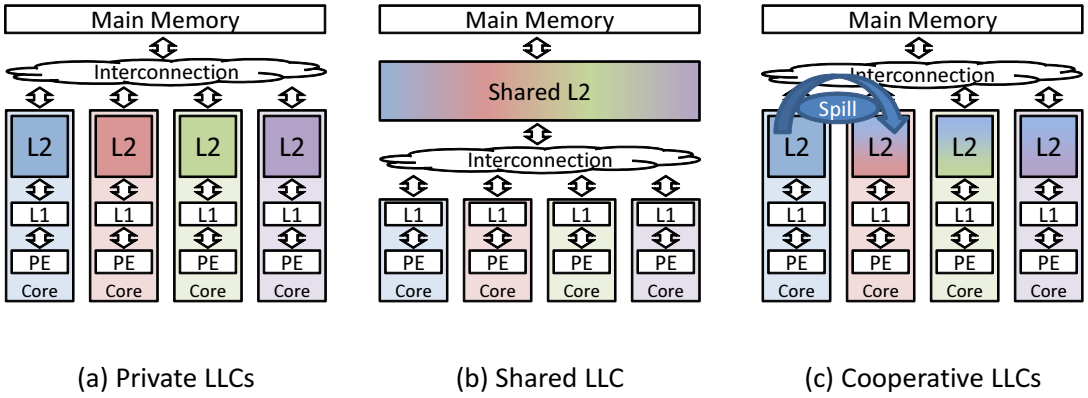
Figure 1.2 shows a typical conceptual model of many-core chip. In many-core, tens or hundreds of processor cores are connected in a two-dimensional mesh in a single chip. Each core is connected to a router (R in the figure) via a network interface controller (NIC). The memory controller is also connected to its corresponding router. My target many-core architecture assumes that each core has multiple levels of caches.

The problem with such progress in processor at all times is the increase of the gap of improvement between processors and main memories, or so-called memory wall. It turns memory bandwidth into more precious resources.

As a result, it is inevitable for high-performance processors to supply as many data from on-chip cache memories to save precious memory bandwidth. The best way to achieve it is to improve the performance of last level caches (LLCs) that are the largest caches and the closest to the main memory. There are a number of approaches for improving LLCs [2] [3] [4].

In multicore and many-core, an important design point is the organization of LLCs. Figure 1.3 shows three kinds of LLCs that have different organizations. The traditional cache organizations are private caches shown as (a) and a shared cache shown as (b). To exploit the advantage of both the organizations, cooperative caches [5] [6] shown as (c) are proposed. I explain their respective features below.

(a) in the figure shows the private LLCs. Each core stores data in its own small cache. Since the region to which a core refers is small, they keep the latency low. However, it is not flexible about capacity. That is, even if a core that is short of cache space is running with another core leaving most



(a) Private LLCs (b) Shared LLC (c) Cooperative LLCs

Figure 1.3 Cooperative LLCs that try to take advantage of both private LLCs and shared LLC.

of its caches unused, the former core cannot use the unused cache region of the latter core.

(b) in the figure represents the shared LLC. All the cores access a single, large, and shared cache (shown in Shared L2). Though it is highly flexible about capacity, the average latency is longer than private caches, for all the cores must refer to the entire cache that is bigger and slower than private caches. To make matters worse, a harmful application may occupy most of the cache and contaminate the cache space for the other cores.

I now explain the cooperative caches shown as (c). The recently studied cooperative caches look like private caches, but they also permit the cores to store their own data to other cores. It is realized by allowing a part of cache lines evicted from one core forwarding (or *spilling*) to another core. Spilling overcomes the weakness of private caches and improves the overall performance by letting cores that require a large amount of frequently reused data borrow some cache lines from unused cache regions of other cores. Moreover, thanks to temporal locality, most cache references are done in local and thus the average latency is still small. However, adaptive spilling is essential for taking advantage of cooperative caches. In recent years, many of such spilling techniques [7] [8] [9] [10] have been proposed.

In this thesis, I propose a highly efficient and scalable approach named **ASCEND (Adaptive Spill Control with extra ENtries of Directory)** to control spilling. In addition to high efficiency, high scalability, or capability to apply the method to tens of cores, is also required for the future many-core era. Achieving both the properties is challenging, and ASCEND is the first method to realize it.

While existent methods use one kind of units to analyze the characteristics of applications, the method examines them in detail using two kinds of units with different criteria. Spiller selectors decide which cores should spill their evicted lines. Receiver selectors choose the destination of the spilled lines. These units analyze the characteristics precisely and forward lines adaptively. In addition, they are distributed in the chip and each unit makes a decision independently. It prevents concentration of

load and thus achieves high scalability.

I verify the effectiveness of ASCEND through evaluation using cycle-level simulation. In the evaluation, cores execute multiple applications that are different in demands on cache capacity at the same time. My approach detects the difference correctly and improves the performance by spilling cache lines from cores which want to borrow lines to those which have unused regions. Through the verification, I clarify a spilling method that achieves both high efficiency and scalability.

1.2 Contribution

The contributions of this thesis are as follows:

1. to classify recent methods for cache optimization and show a direction to researches on this area;
2. to show the implementation of a useful simulation infrastructure for research on many-core processor with shared memory; and
3. to show a highly-efficient and scalable spilling to make use of the scheme of cooperative caches.

I refer to respective contributions below.

As one of the secondary contribution, I explain the aims and the features of a number of recent researches for improving caches and classify them. They include various topics and are not limited to cooperative caches and their optimization. Therefore, it is useful for researchers to get a wide perspective as an introduction to cache optimization. I also show a future direction of research on this area based on my classification.

The other secondary contribution is implementation of an infrastructure for research on many-core processors with shared memory. I make use of **SimMips** [11], a MIPS system simulator that we have developed, as a part of the infrastructure. Moreover, I am developing a new SimMips-based simulator named SimMccc (Simulator of Many-Core Cooperative Caching) [*1] to evaluate the target many-core. It can be utilized as a great infrastructure of cooperative caches.

The primary contribution is to show an efficient and scalable method for the cooperative caches through the proposal and evaluation of ASCEND. Some existing methods are not so efficient due to their insufficient analysis of demands on capacity. Others are not scalable with frequent broadcasts for the analysis. My proposal is distinctively different from them. It achieves almost the same efficiency as an existing high-performance method, without losing scalability. It is expected to play an important role in the future many-core era.

[*1] It is pronounced as *Sim-M-C-Three*.

1.3 Outline of this thesis

The outline of the subsequent chapters is as follows.

In Chapter 2, I explain various existing methods for efficient use of caches. As the trend of processors changed from single-core to multicore, optimization techniques began to need to be aware of multiple threads running in parallel. Moreover, scalability is becoming an important problem in anticipation of the many-core era. From these circumstances, I explain the methods for single-core, multicore, and many-core in order. Afterward, I refer to the target architecture, the Distributed Cooperative Caching (DCC), and its optimizations.

In Chapter 3, I propose a useful simulation infrastructure that centers on SimMips. To verify both efficiency and scalability, I define two respective simulation environments and describe how to simulate them with the proposed infrastructure. By comparison with other infrastructures, I clarify the advantage of ours. Afterward, I quantitatively show the problem of the existing optimizations for the DCC.

In Chapter 4, I propose ASCEND as an efficient and scalable optimization for the DCC. I first describe an important characteristic of the DCC, or that it needs some extra hardware being reserved. I then explain how my approach utilizes it by taking some examples. Afterward, I show the architectural organization of the DCC with ASCEND, two different ways to describe the additional units or the Spiller Selectors and the Receiver Selectors, and an optional extension named Weak Receiving.

In Chapter 5, I show the results of the evaluation of ASCEND and have some discussions from them. I also calculate the amount of the additional hardware after showing the results of the simulation.

I conclude this thesis in Chapter 6.

Chapter 2

Background

In this chapter, I mention various recent prior researches to make caches efficient. In the actual cache organization, it is natural to use a combination of methods across multiple categories rather than to rely on a single one. In addition, because some categories have been fairly researched, it may be difficult to propose a new idea focusing only on a single category. Instead, mixing up knowledge of other categories can be a hint of making cache management more efficient. For these reasons, I categorize related methods by their targets and features.

The primary classification in this chapter is based on the points that the methods focus on. They can be divided into three: characteristics of single-thread execution, interaction among multiple cores, or scalability for applying the methods to many-core. Each category has some subcategories of different features.

In Section 2.1, I refer to methods that exploit characteristics of single-thread execution. They consist of dead block prediction, replacement algorithm, and improving the efficiency of associativity.

In Section 2.2, I mention multicore-aware methods that consider the interaction among multiple cores that executes various applications. They are composed of thread-aware replacement algorithm and cache partitioning.

In Section 2.3, I state endeavors for future many-core processors. They need scalability or properties that their performance is not limited by the increase of the core count. They include Non-uniform Cache Architecture (NUCA) and improving directory-based coherence managements.

The Distributed Cooperative Caching (DCC) is a variation of cooperative caches that my approach is based on. Since it tries to improving scalability by distributing its organization over a chip, it can be included in the third category. However, its features and behavior is important for discussions in following chapters. Therefore, I devote Section 2.4 to cooperative caches and the DCC.

My approach ASCEND is an original method to control spilling in cooperative caches. Methods that focus on the difference in demands on capacity between cores and that manage spilling are considered as a form of cache partitioning, which is included in the second category. However, since they strongly depend on the scheme of cooperative caches and I have to discuss the features and the problem of

existing methods for spilling, I mention them later in Section 2.4.

2.1 Cache Optimization mainly for Single Thread

2.1.1 Dead Block Prediction

Lai et al. [12] first introduced the concept of dead block prediction. In the context of dead block prediction, if a cache line will be accessed again before it is evicted, then it is considered as *live*. In contrast, if it is no longer reused, it is considered as *dead*.

If we identify dead lines before the eviction, we benefit from them by invalidating, turning off, or replacing them, for they are no longer needed to be kept. To get this advantage, dead block predictors predict whether a line is live or dead, based on the information gathered on past access.

What kind of information predictors gather and how it is utilized vary greatly with methods. They can be divided into trace-based and time-based. I explain below how each kind of predictors make use of past information.

Trace-based Prediction

Trace-based predictors use traces of instructions, that is, the program counter (PC) of the instruction accessing a line or a set of PCs. When multiple lines are accessed by the same sequence of instructions, whether they are reused or not usually match each other. Many predictors aiming for performance adopt this strategy.

The first approach by Lai et al. [12] can be used for computer systems with symmetric multiprocessing (SMP). They added a trace-based predictor to a method, called Self-Invalidation [13], which a processor invalidated or wrote back cache lines voluntarily and earlier. Since wrong invalidation of lines that would be reused harmed the performance, they predicted dead lines and invalidate them in advance. Thus, they succeeded in reducing the latency on insertion without harmful effects.

A later proposal by Lai et al. [14] added the idea of dead block prediction to a prefetcher. Their predictor keeps not only traces when lines become dead but also the address of the lines that will be requested just before the dead lines are evicted. It enables the prefetcher to decide the prefetching address and its destination at the same time.

In contrast to these two methods, Virtual Victim Cache [15] does not consider the lines being expected to be dead as the immediate targets of invalidation. Instead, they are just marked as dead. When an inserted line causes an eviction afterwards, the cache searches for a line that is invalid or marked as dead in the secondary set of the evicted line. If such a line is found, the cache uses it as a shelter of the evicted line.

Bypassing the LLC or placing inserted lines only in the higher levels of caches is also possible. Sampling Dead Block Prediction (SDBP) [2] is a method that applies bypassing. In the SDBP, if the

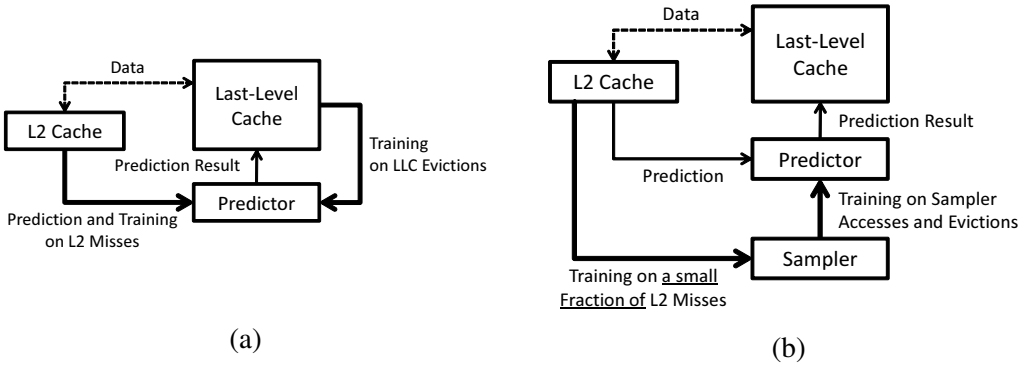


Figure 2.1 Differences between trace-based dead block predictor with and without sampler

LLC does not have a spare line and a newly inserted line is predicted dead, it is placed only in the higher caches. This bypassing exploits the fact that the lines whose temporal locality can completely extracted by the higher caches have no necessity to be kept in the LLC.

A distinct organization of this method is a small tag array as a sampler dedicated to dead block prediction. Not the main cache but the sampler is used for prediction. It greatly reduces the additional hardware and the power consumption. I show the difference between a conventional dead block prediction and the SDBP in Figure 2.1.

In a typical dead block predictor shown in Figure 2.1 (a), if a line misses in the L2 cache, the address of the line and the miss-causing PC are given to the predictor. The predictor sends the outcome based on the past information to the LLC. The miss-causing PCs are also sent to the LLC and stored in their tag array. When the missed line hits in the LLC, the predictor learns that the PC stored in the corresponding tag is likely to produce live lines, and then it updates the stored PC. Conversely, when a line is evicted from the LLC, the predictor learns that the stored PC is likely to cause dead lines.

In the SDBP shown in Figure 2.1 (b), on the other hand, the predictor traces the PCs of a small fraction of L2 misses using the sampler and updates the predictor table with the result. The reference to the table is done on every L2 miss. In addition, the tag array in the sampler can be different in the number of sets, the number of ways, replacement algorithm, etc. from that in the main cache. These features contribute to improving the accuracy of prediction.

Time-based Prediction

Time-based predictors utilize the elapsed time from the last access of each cache line. The predictors decide a threshold based on the past interval between accesses. If the elapsed time reaches the threshold, the line becomes predicted as dead.

Figure 2.2 shows an organization of a typical time-based predictor. Other than the predictor itself, following per-line fields of additional metadata in the LLC are essential for time-based prediction:

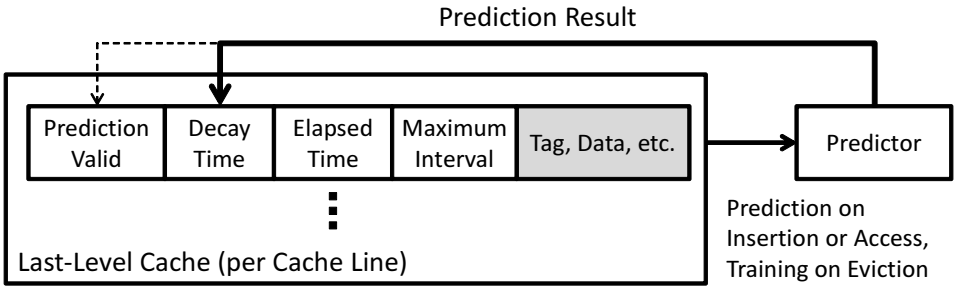


Figure 2.2 Organization of a typical time-based dead block predictor.

- **Prediction Valid** that shows whether lines are treated differently by the prediction,
- **Decay Time** or the threshold of the elapsed time that is provided by the predictor,
- **Elapsed Time** that indicates the time from the last access, and
- **Maximum Interval** that shows the longest interval between accesses since the insertion of the line.

Other metadata are also added depending on the method.

The predictor expects the decay time (and whether the prediction is valid in some methods) and stores it into the cache line. The elapsed time is set to zero every time the line is accessed and incremented in method-specific conditions. If the prediction is valid and the elapsed time exceeds the decay time, the line is considered as dead. Besides, on every cache hit, the maximum interval is compared with the elapsed time and updated if needed. This field is sent to the predictor on eviction and used for training.

I explain methods using time-based predictors. IATAC [16] uses one to find which lines should be turned off to reduce leakage current. The predictor of IATAC uses the number of cycle as elapsed time and the prediction is based on the number of hits of the line since the insertion. Thanks to dynamic modification of the threshold, it shows high accuracy and thus succeeds in reducing leakage by turning off a large part of caches with a small penalty.

Counter-based predictors, such as one that was proposed by Kharbutli and Solihin [17], are sometimes classified differently from time-based predictors. However, I consider that these two kinds of predictors are essentially identical. Their predictor uses the number of scans through the set that a line belongs to as elapsed time. The prediction on insertion is based on the address of the line and the accessing PC. The result is valid only when the past two maximum intervals match. As done in many trace-based predictors, lines being predicted dead become the prior targets of replacement.

All the dead block predictors I have mentioned above use either one reference to the cache or passage of specific time from it as a trigger of prediction. However, more than one reference can be treated as a

single trigger. One of such ideas is called Cache Burst [18]. It treats a series of access to MRU (Most Recently Used) line, or a burst, as a trigger of prediction. This modification enables a prefetcher in the L1 cache to be precise and power-efficient. Unfortunately, it has less advantage in the L2 or lower caches because such extreme locality is filtered out in the L1 cache.

2.1.2 Replacement Algorithm

When there are no invalid lines in the same set as an inserted line, which line the cache selects for a victim of replacement has a significant influence on the cache performance. Figure 2.3 shows the behavior of some replacement algorithm that I mention in this section. In the figure, time-varying behavior of a 3-way cache (expressed in three blocks) given an access pattern of repetition of `A`, `B`, `A`, `B`, `C`, and `D` is shown. If the cache hits, the corresponding way is shown in red.

An LRU (Least Recently Used) replacement algorithm of choosing the line whose *last* access is the farthest from now and its approximation (called pseudo-LRU) are often used, for they are easy to implement and their average performance is not bad. However, the LRU performs poorly with some access patterns. One of the typical worst cases is repeatedly accessing a region slightly larger than the cache size. In this case, the victim lines are always what the processor will use next. As a result, the cache never gets hits.

In the leftmost column of Figure 2.3, the LRU is expressed as an algorithm placing an inserted line in the leftmost or most prioritized block. When `D` is inserted in the fourth row, the least prioritized line, `A`, is evicted, though it will be reused next. In similar, when `A` is inserted again in the fifth row, `B`, which is supposed to be reused in the sixth row, is removed. As a result, the LRU cache get only two hits per iteration (from sixth to eighth rows). In this section, I mention some methods to solve this problem and to aim for higher hit rates.

Belady found out the theoretically optimal replacement [19]. It is often called Belady's OPT algorithm. If the cache knew the complete sequence of future cache access, the optimal target for replacement would be the least imminent line or the line whose *next* access would be the farthest from now. The second column from the left in Figure 2.3 shows the optimal replacement. When `D` is inserted in the fourth row, for `A`, `B`, and `C` are reused earlier than `D`, `D` itself is the least imminent. Therefore, `D` is not stored in the cache. Consequently, the other lines receive hits and thus the hit count per iteration is five.

Unfortunately, this algorithm uses future information and thus is practically impossible to implement, even though we can get its outcomes a posteriori. For this reason, a number of near-optimal, implementable algorithms have been proposed in the recent years. Some of them try to emulate the optimal replacement, and others exploit the characteristics of processors or access patterns or both.

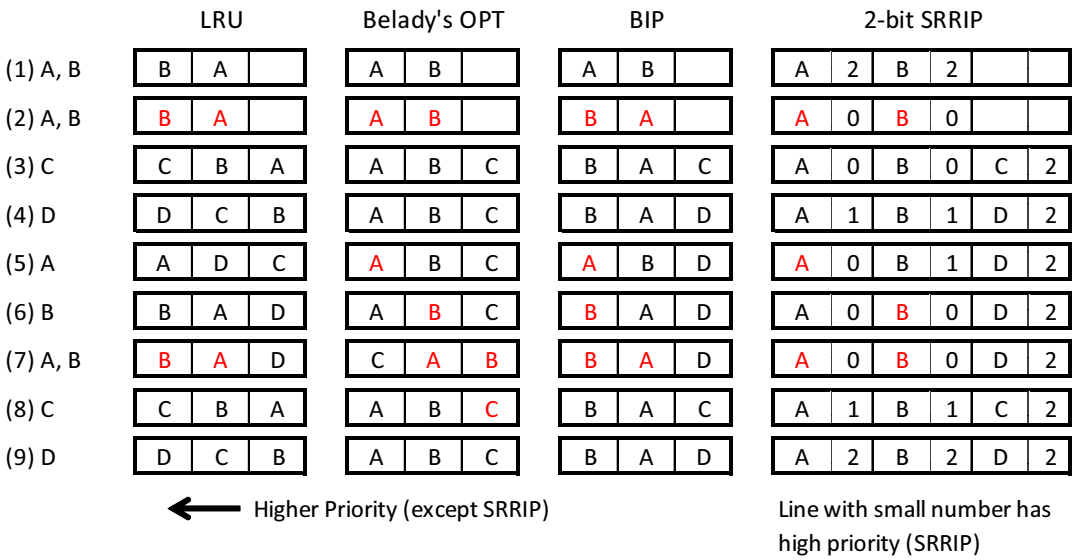


Figure 2.3 Examples of some replacement algorithms.

Emulating the optimal replacement

Shepherd Cache [20] tries to emulate Belady's OPT algorithm. It uses a FIFO (First-in First-out) cache called a Shepherd Cache along with the main cache. The Shepherd Cache temporarily suspends replacement of the main cache while inserted data are kept there. This allows the main cache artificially reproducing the ideal algorithm.

Keramidas et al. [21] proposed another approach to find the least imminent line. They got what time the block would be accessed next through prediction. Although I categorize it into a replacement algorithm, the idea is similar to trace-based dead block prediction.

Focusing on the characteristics of processors and/or cache access

MLP-aware Cache Replacement [22] notes that the effect of cache misses is not uniform in out-of-order processors because of memory-level parallelism (MLP). It prioritizes lines that have low MLP or have long latency on miss.

Bimodal Insertion Policy (BIP) [23] places most of the newly inserted lines on the LRU position, while a fraction of them are placed on the MRU position. If a line receives a hit, it is moved to the MRU. It is effective with the worst case of the LRU that I have mentioned above.

In the third column from left of Figure 2.3, all the inserted lines are placed on the LRU (the rightmost block) in BIP for ease of explanation. In the example, frequently used `A` and `B` tend to have higher priorities, but `C` and `D` do not. Since `A` and `B` hit twice, the number of hits per iteration is four. Practically, it sometimes places `C` and `D` on the MRU and thus the average number of hits is slightly smaller than four.

In both algorithms, they greatly outperform the LRU if access patterns match the method. However, if not, they can be less effective than the LRU. To deal with the shortcomings, some methods including them make use of tournaments with a small part of sets, called Set-Dueling. Set-Dueling fixes the replacement policy of tens of sets to the LRU and that of other tens of sets to the proposed scheme. The other sets follow one of the policies that cause fewer misses in the sampling sets. The methods with the Set-Dueling are named Sampling Based Adaptive Replacement (SBAR) for the MLP-aware Cache Replacement and Dynamic Insertion Policy (DIP) for the BIP.

To achieve higher efficiency with smaller hardware than the LRU, Re-Reference Interval Prediction (RRIP) [3] was proposed. It assumes a pattern of repeatedly accessing a small region while scanning a large region as a weakness of the LRU. It prioritizes frequently reused lines and slows the decline of priority to prevent the scan of the large region from evicting the small region. As a result, the cache gets a large number of hits despite existence of scans.

RRIP needs a priority counter of some bits called Re-Reference Prediction Value (RRPV) per cache line. Lines with small number of the counter have high priority and likely to be kept. Following explanation assumes 2-bit counters; the value of the counter with the lowest priority is 3. In Static RRIP (SRRIP), newly inserted blocks have the priorities of the second lowest or 2. They do not become the highest or 0 until the corresponding blocks are reused. Candidates for replacement are the lines with the priorities of 3. If there are no such lines, all the counters in the set are incremented and scanned again.

The rightmost column of Figure 2.3 shows an example of SRRIP. Numbers in the right of columns stand for the priorities. When `A` and `B` are reused in the second row, their priorities become 0. In contrast, `C` has the priority of 2, for it is not reused. Afterwards, on searching for a line to be replaced, it looks for a line with the lowest priority once. Since no such lines exist, all the counters are incremented. Now that the priority of `C` becomes the lowest, `C` is selected as the victim. As a result, `A` and `B` hits twice and the cache gets four hits per iteration. In addition, although the priorities of `A` and `B` are incremented twice in the second iteration (on the insertions of `C` and `D`), they are never evicted.

SRRIP has the same problem as the LRU on some access patterns. To cope with it, Bimodal RRIP (BRRIP), where a fraction of inserted lines are given the different priority, and Dynamic RRIP (DRRIP), which compares SRRIP with BRRIP using Set-Dueling, are proposed. They correspond to the BIP and the DIP for the LRU.

SHiP [24] is based on the SRRIP, reflecting the results of dead block prediction. When inserted lines are predicted dead, they are given the lowest priority and soon will be removed.

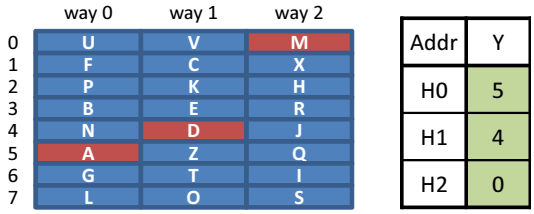


Figure 2.4 An example of a 3-way Skewed-associative Cache.

2.1.3 Improving Associativity

An ideal full-associative cache, where a cache line can be placed in any block, is able to replace the least prioritized line in the cache at all times. However, common set-associative caches limit the placement of a line to the set (several blocks) according to its address It causes the inequity of usage and affects the efficiency. If frequently used sets and rarely used ones are mixed, fresh lines (likely to be reused) can be soon removed from the cache in one set, while stale lines (unlikely to be reused) stay there in another set.

An obvious way to mitigate the imbalance is to increase the number of ways. However, it has limitations of the rise of the latency and the power consumption, due to the increase of the number of lines accessed at once.

The other, more preferable way is to select the least prioritized line close to globally with keeping the number of ways small. I explain such designs to “improve associativity” [25].

One of the solutions to the inequity is to use a hash function that is randomized well. Considering a tradeoff between the randomness and the latency, it requires both hardware simplicity and enough randomness. It is possible design point to use bitwise XOR of multiple parts of the address rather than its single part. In fact, some commercial processors can use this kind of hash function [26].

Multiple hash functions further improve the inequity. Column-associative Cache [27] is based on a direct-mapped cache, where a cache line has only one candidate for placement, but each address has a primary set and a secondary set calculated by different hash functions. The cache is checked twice if a line is missed in its primary set.

Skewed-associative Cache [28] was proposed almost at the same time as the Column-associative Cache. Each way has a different hash function, and a line has different placement of sets in each way. Figure 2.4 shows a 3-way Skewed-associative Cache which consists of eight sets. The left of the figure stands for the tag array. The characters shown in the array are assumption of the current placement. Each letter represents a different line. The right of the figure shows output of the hash functions. When we insert `Y` to this, the cache first calculates the set indexes in ways with their respective hash functions.

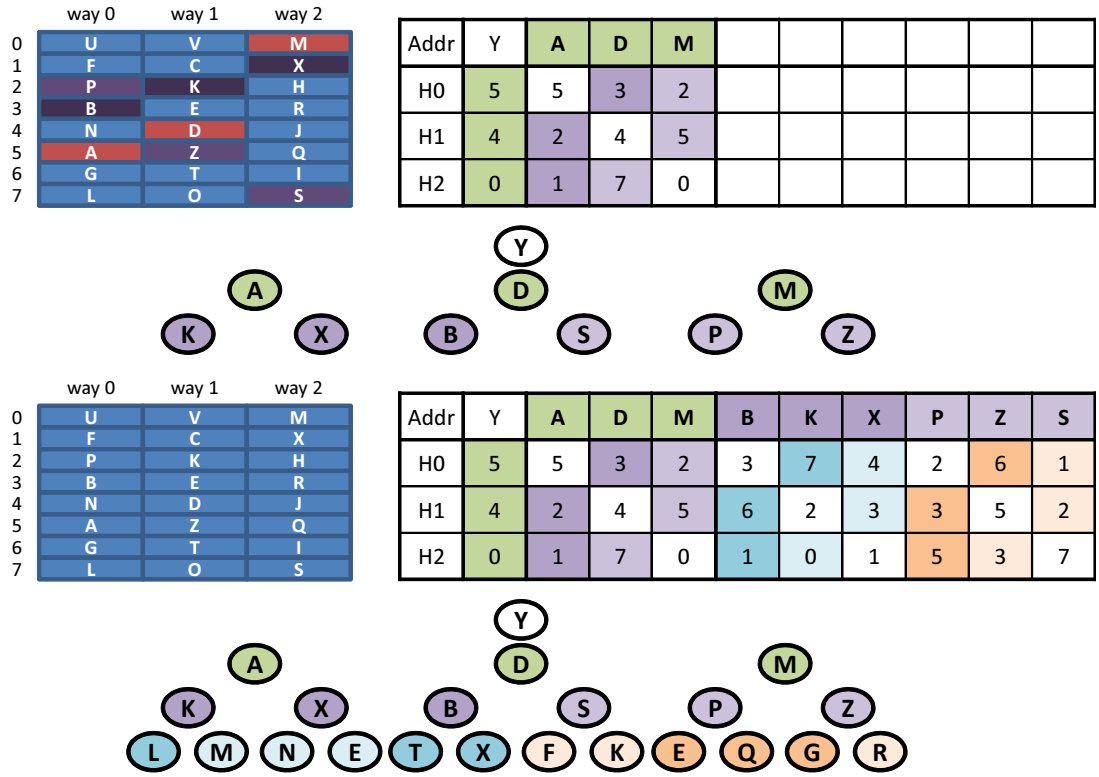


Figure 2.5 An example of a quest for candidates for eviction in the ZCache.

In the figure, hash functions H0, H1, and H2 correspond to way 0, way 1, and way 2, respectively. As the hash values to `Y` are 5 for H0, 4 for H1, and 0 for H2, the lines being accessed and becoming the candidates for replacement are set 5 in way 0, set 4 in way 1, and set 0 in way 2; that is, `A`, `D`, and `M`.

ZCache [25] extended this idea further. It determines the locations of blocks as the same way as the Skewed-associative Cache, it adds the candidates for eviction and thus improves the associativity.

In the Skewed-associative Cache, each line in the candidates for placement of an inserted block has different locations in the other sets than its current one. ZCache allows lines already stored being relocated to the other set. This enables the lines in the candidates for relocation to be added to the candidates for eviction.

Figure 2.5 shows how ZCache adds the candidates for eviction. The contents of the tag array and the inserted line `Y` are the same as Figure 2.4. First, it calculates the candidates the same way as the Skewed-associative Cache. As I have mentioned, they are `A`, `D`, and `M`. It then calculates their hash value to find their alternative locations. For example, when the hash values to `A` are 2 for H1 and 1 for H2, it may be relocated to either set 2 in way 1 or set 1 in way 2. This relocation leads to the eviction of `K` or `X`. Conversely, `K` and `X` are also considered as the candidates for eviction. In similar, `B`, `S`; `P`, and `Z` are added to the candidates.

Furthermore, by allowing multiple relocations and searching for candidates repeatedly, ZCache increases the number of candidates greatly. It also calculates the indexes of the additional candidates and adds the corresponding lines to the candidates for eviction, as shown in the lower of Figure 2.5. If a 3-way ZCache permits relocations up to twice at a time, the maximum number of candidates is 21. It selects an evicting line from many candidates with a global replacement policy. Afterwards, it replaces and relocates the lines as needed. Assume the evicted line is `N`, the third from the left in the bottom of the candidates represented in the figure. `X` is moved where `N` was; `A` is located where `X` was; and then `Y` is inserted where `A` was.

Although ZCache greatly improves the associativity with almost the same amount of hardware as set-associative caches, it needs frequent access to the tag array and thus the controller becomes much complex.

As another solution to the inequity, V-Way Cache [29] uses the dynamic cross-reference between tags and data. While tags and data have predefined one-to-one mapping in usual caches, V-Way Cache gives variable mapping with additional per-line storages keeping cross-reference.

The point is that the number of elements in tag array and data array are not necessarily the same. Since an invalid tag does not need to reference data, the number of tags can be bigger than the number of lines in the data array. Even if doing so, the amount of additional hardware is relatively small, for the hardware cost per entry in the tag array is smaller than that in the data array. The idea of having some additional tags is similar to the DCC, the target architecture.

Thanks to extra tags, the most common situation on insertion is that an invalid tag is found but the data array is full. In this case, V-Way Cache can use a global replacement policy (which is different from what I have mentioned in ZCache) to select a line to be evicted. After it decides the evicting line, it invalidates the corresponding tag with reference to the tag array, and then stores the inserted line where the evicted line was. As a result, it benefits from the global replacement, keeping the hit latency low.

2.2 Cache Optimization focusing on Multithread

2.2.1 Thread-aware Replacement Algorithm

The DIP that I have mentioned in Section 2.1.2 cannot apply efficiently to multiple cores or threads as it is. Applying either the LRU or the BIP to all the core implies that the performance of some cores sacrifices for that of the other. On the other hand, permitting each core to choose one of them freely means that the number of candidates for globally optimal policy increases to a power of 2. This makes it impractical or even impossible to sample all the candidates.

Thread Aware DIP (TADIP) [30] presents the multiple ways to find a near-optimal policy by sam-

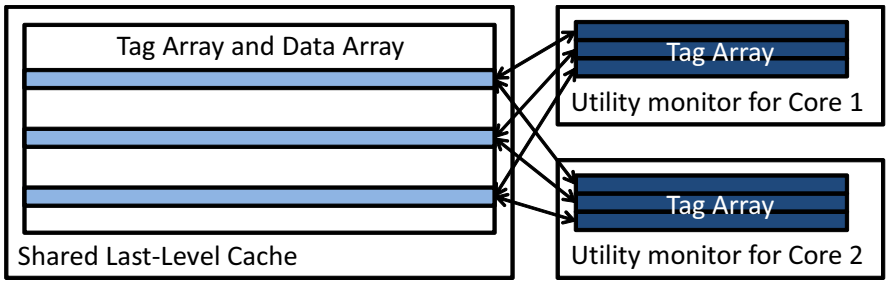


Figure 2.6 An organization of the utility-based cache partitioning.

pling some of them. It reduces the number of sets needed for sampling to a proportional to the number of cores.

Meanwhile, though most replacement algorithms only focus on the priorities of inserted lines, Promotion/Insertion Pseudo Partitioning (PIPP) [31] also considers those of reused lines. It promotes reused blocks a little, rather than give them the highest priority. It also gets the effect of cache partitioning, which I mention next, with different priorities on insertion by the cores.

2.2.2 Cache Partitioning

Cache partitioning, or allocating shared resource of cache capacity among cores, can be considered as a form of thread-aware replacement algorithms. However, this topic includes many related researches, and so I consider it as a different category.

The points of design of cache partitioning are threefold: an estimation of cache performance, a decision of requirements to be met, and a way to realize allocation.

How do they estimate the cache performance?

Dynamic modification to allocations is essential for efficient cache partitioning. The first step for achieving it is to estimate cache performance by the size.

Suh et al. [32] [33] conducted the first such attempts. They estimated the performance by the size with way-divided counters that are incremented on a hit in the corresponding way. A shortcoming of their method is its low precision because using the shared main cache leads to interference of the estimation of each other.

To overcome the flaw of their method, have per-core, independent tag arrays dedicated for estimation can be used. Nevertheless, arrays to the whole cache is impractical in terms of hardware cost. To reduce the cost, Utility-based Cache Partitioning (UCP) [4] and Cache-Partitioning Aware Replacement Policy (CPARP) [34] were proposed. They limit the region of the cache that requires access to the additional tag arrays in different ways.

Figure 2.6 and Figure 2.7 show organizations of the UCP and the CPARP, respectively. Tinted parts

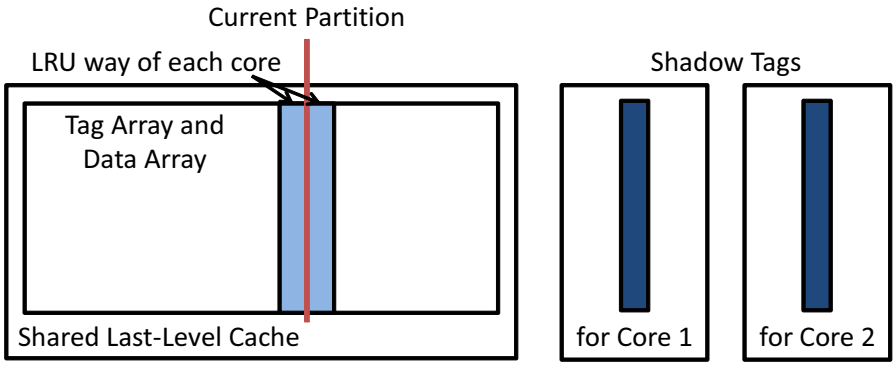


Figure 2.7 An organization of the cache-partitioning aware replacement policy.

represent the portions of the cache used for estimation. Dark-colored parts stand for the additional tag arrays.

The UCP, shown in Figure 2.6, adds tag arrays to a small part of the sets in the cache. The arrays are called Utility Monitors. It reduces the cost of the conflict-free estimation. However, it requires either large amount of hardware or a help by software to decide partitions from them.

On the other hand, the CPARP, shown in Figure 2.7, limits the ways rather than the sets. In other words, it only estimates the performance of the caches when it increases or decreases the current allocations by one way. To estimate the performance on increased allocation, it makes additional tags called Shadow Tags keep the tag of the last evicted line in each set and counts the number of hits in the Shadow Tags. To estimate the performance on decreased allocation, it counts the number of hits in the LRU way of the current allocation. Like the UCP, the CPARP also reduce the hardware cost. Moreover, it is easier to decide new allocations than the UCP. Nevertheless, it gives the wrong estimations to applications that have non-convex relations between the capacity and the performance.

The use of such estimations is not limited to cache partitioning. CRUISE [35] is one of its examples. It reflects the results of the estimation into thread scheduling. It divides the threads into some categories by the estimation. If a pair of threads that harm each other is found, it prevents the threads from being allocated at the same time to cores sharing cache.

What is the goal of the partitioning?

Once the cache performance is estimated, what the partitioning methods should consider next is to decide which kind of requirements they aim for. The requirements include performance, fairness, quality of service (QoS), and combinations of them.

Methods for performance try to maximize the total throughput, or typically, to minimize the sum of the number of cache misses. A common barometer of improvement is the sum of the relative IPC (Instruction Per Cycle). They are the mainstream of related work; all the three methods mentioned

above aim for performance.

Methods for fairness want the threads improved equally. Therefore, a typical barometer is the equality of the relative reduction of the number of cache misses. Fair Cache Sharing [36] is one of them. It measures the barometer of performance of each core in a fixed period. If the difference of the barometers between the highest and the lowest exceeds a threshold, a move of partition between them occurs. It presents some different formulas as the barometers.

Some methods for QoS emphasize the elimination of cores that are heavily harmed. Hence, a common standard is the minimum of relative IPC. Some performance-oriented methods also present their QoS-aware variations where cache misses by cores whose performance declines are penalized.

Cooperative Cache Partitioning [37] allows sacrificing temporal performance for average performance. When there are several threads that benefit much from larger caches, it dares to bias the allocations temporarily and rotate them periodically. Although it improves both performance and QoS on average, it requires OS-based complex control.

The other methods for QoS look at its another aspect, that is, their goal is to make the behavior of the caches close to that according to predefined priorities. Iyer proposed a framework named CQoS [38] for caches considering this kind of QoS. It modifies the allocations of the cache according to per-core priorities. He confirmed that his framework achieve the different behavior by priorities in some scenarios.

Virtual Private Caches [39] present a way to control allocations of not only the capacity but also the bandwidth of the cache according to priorities, after a method to guarantee QoS in a point of network bandwidth.

Incidentally, an OS-based method that was proposed by Lin et al. [40] is applicable to any of these purposes. It uses different algorithms and performance barometers by purpose. Hsu et al. [41] minutely analyzed interaction of the purposes, and relation between the purposes and performance standards.

How do you enforce the assignment?

When the allocation of capacity of each core is determined, the last consideration is how to realize it. The most common enforcement is to limit the number of ways where cores can place their line [42] [43]. It is often called way-partitioning.

Rafique [44] proposed a method to limit the number of lines rather than ways. It counts the current number of lines that belong to each core. When it exceeds the limit, lines in that core are replaced in preference.

Liu et al. [45] proposed a partitioning in bank granularity. They assumed caches that were divided into some banks. They limited banks that first responded to requests of each core so that it balanced performance with power consumption.

The OS-based method by Lin et al. [40], which I have mentioned above, adapts the idea of Page Coloring [46] that modifies mapping of logical address to physical address.

Vantage [47] is based on ZCache [25], which I have mentioned in Section 2.1.3. It divides the whole cache into managed and unmanaged regions. Evicting lines are usually chosen from unmanaged lines. It modifies the per-core threshold for demoting lines from the managed region to the unmanaged region in order to enforce the allocations.

The last kind of enforcements I mention is pseudo enforcements. They do not require precise partitions; they only need some criteria of allocations. Iyer's CQoS [38] presents one of them with probabilities of storing lines in addition to ordinary way-partitioning. PIPP [31] can be categorized here in the point of changing the priorities of inserted lines.

Spilling managements of cooperative caches, which my approach belongs to and I introduce later in this chapter, are also included in pseudo enforcements, though some of them combine way-partitioning.

2.3 Toward Many-core Caching

2.3.1 Non-uniform Cache Architecture (NUCA)

As caches get larger, increase of latency of a single large cache became a problem due to some limitations like wire delay. Instead, to divide it into multiple tiled banks has an advantage in the aspect of average latency, though latency of each access becomes different. These organizations are called Non-uniform Cache Architectures (NUCAs) [48] [49].

In NUCAs, dynamic migrations, which are utilized in D-NUCA [48], NuRAPID [49], etc., further reduces the average latency. It means that frequently accessed lines are relocated to banks that are close to the core, that is, has lower latencies.

Although the target of the first proposal of NUCA is large caches in single-core processors, they suit multicore or many-core processors with a large number of cores very well, because banked organizations prevent the concentration of loads. Therefore, most of the researches in this category adapt a NUCA to multicore and many-core.

CMP-DNUCA [50] and NUCA Substrate [51] investigate dynamic migrations like D-NUCA in the different organizations of cores and banked cache. Figure 2.8 shows their respective organizations. Each Colored figure represents a bank or a group of banks in the cache.

In single-core processors, the only direction where frequently accessed lines are pulled is where the core is. In contrast, frequently shared lines can be pulled from multiple directions in multicore processors. As a result, such lines are placed in near the mean of locations of cores or close to the core that accesses the lines the most frequently. However, such frequent migrations are wasteful and may leads to inefficient placements.

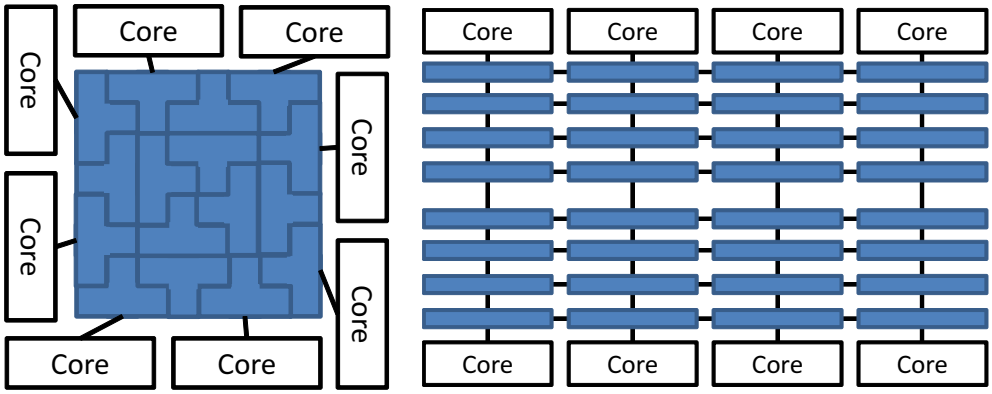


Figure 2.8 Different NUCA organizations in multicore processors.

Migration-based NUCA, or M-NUCA [52], avoid the problem with frequent migrations. It first detects frequently shared lines, and then counts the number of access from each core to them. Actual migration is not occurred until the cache gathers sufficient information. Thus, it gets placements that are more efficient, in addition to reducing the number of migration.

Victim Replication [53] reduces the average latency not by dynamic migrations but by dynamic replications. When a line is evicted from an L1 cache and a proper placement is found in its local bank of the shared L2 cache, a replica of the evicted line (note that the original have been stored in a remote bank) is placed there. Replicas behave just like Victim Cache [54] of the L1. Since the other cores believe that the L1 in that core still holds the evicted line, the other cores send a request of invalidation or write-back in the usual way. Hence, it keeps coherency in the same way as conventional shared caches, except a part of invalidation or write-back requires a search in the local bank of the L2.

CMP-NuRAPID [55] combines migration and replication. Just like V-Way Cache [29] in Section 2.1.3, it has dynamic cross-reference between tag and data arrays. The tag arrays are private while the data arrays are banked and shared. The tag arrays have many entries for a situation a few cores occupy most of the cache. Proper managements of cross-reference enable migrations and replications. However, they make the coherency control much complex and difficult to implement. In addition, the maximum capacity available to each core is limited by the number of entries in the tag array.

Reactive NUCA [56] applies different strategies of locations by access tendencies with a help of software. It classifies data into instruction, private data, and shared data.

The key ideas of multicore or many-core NUCAs, that is, distributing loads with banked organization and improving the average latency by frequently used data close to the core are similar to those of ccNUMA (Cache Coherent Non-Uniform Memory Architecture) [57] and COMA (Cache-Only Memory Architecture) [58], which used to be studied actively in parallel computers with distributed shared

memory. They allow non-uniform latencies between the memory in a local node and those in the remote nodes. Since each node accesses to its local memory independently, the whole memories in the system are considered to be divided into many banks. Besides, in ccNUMA, a part of local memory is used as a cache to remote memory and thus it improves the average latency. COMA extends this idea. It does not specify the home node of a line: the entire local memory is treated as a cache to remote memory (even if there is only one node in the system that keeps a line). This reduces the number of replicas and thus improves the efficiency.

The difference between NUCAs and ccNUMA or COMA is the target of the ideas: ccNUMA and COMA apply them to main memories of multiprocessors; NUCAs apply them to on-chip caches of multicore or many-core processors. It connects to an important architectural choice. NUCAs hold caches to the main memory that in the slow off-chip location and on-chip caches are more costly resources than node memories. For these reasons, NUCAs have more serious necessity of efficient use of capacity than ccNUMA and COMA. Moreover, in multicore or many-core NUCAs, cores and/or banks of caches are connected via quite fast on-chip communications. Hence, they have a large room for improving the efficiency in capacity and reducing the latency with on-chip communications, like the dynamic migration techniques as I have mentioned.

2.3.2 Scalable Coherence Control

When caches are shared among multiple cores, cache coherency is an important problem. Once a core writes something to a memory address, all the cores must read the written value from that address. In addition, when multiple cores write to the same address, the order of access observed from each core must be identical.

The two major ways to guarantee coherence are snoop-based and directory-based. In a snoop-based coherence scheme, all the caches watch for write operations broadcasted to a bus. If a cache finds a write operation whose address is the same as one of its own lines, it invalidates the stored line because it no longer keeps the latest data. In a directory-based scheme, an additional organization called a directory dedicates itself to coherence. All the cores beg permission from the directory before newly accessing the cache. The directory sends requests of invalidation or write-back to the corresponding cores if needed. After that, it gives permission to the requesting core.

While snoop-based schemes are preferred by processors with a small number of cores, directory-based schemes are favored by those with a large number of cores. Although snooping is good at latency, frequent broadcasts limit the scalability. In consequence, they suit a small core count. On the other hand, directories hardly harm the scalability but make the best-case latency longer, for all the requests must be done through the directories.

As cooperative caches seek the advantages of both private and shared caches, Token Coherence [59]

tried to achieve the advantages of both snoop-based and directory-based schemes. There coherency is controlled by a number of tokens managed by address. Permissions of read or write operations are given when the core has at least one token or all of the tokens, respectively. It is applicable to some interconnections where snooping is impossible or impractical. However, its properties are means between snoop-based and directory-based with the same interconnections, that is, it is not so good at either latency or scalability.

Most directory-based coherence schemes for multicore and many-core use directory caches, which store the directories corresponding only to the cached region in the memory. Some recent methods for directory caches try to reduce their hardware amount in anticipation of the future many-core era.

SPACE [60] regards it as a problem that each entry in conventional directory caches has a vector of the same bits as the number of cores, which represents the cores sharing the corresponding line. Since the number of typical sharing patterns is much smaller than that of possible patterns, it compresses the vector to reduce the length of the entry.

WAYPOINT [61] reduces the performance loss when directory caches are filled. To balance hardware cost with performance overhead, it enables information of directory caches to be stored to the regular caches too. My approach also has interest in conflicts in the directory caches. The difference of strategies between WAYPOINT and mine is described in Chapter 4 in detail.

Cuesta et al. [62] proposed an OS-based method that achieved both the reduced number of entries of directory caches and the improved performance by limiting the region managed by the directory caches. TLB (Translation Lookaside Buffer) keeps which pages have been shared. If the page that a requested line belongs to has been not shared, the cache can directly send a request of the line to the main memory. However, when such a page starts being shared, all the cached lines in the page must be managed by the directory caches through gathering information about them from the whole cache, or simply be discarded from the cache. After that, the lines start to be under the management of the directory caches as usual. This method works well especially if the private memory region of cores are divided in a unit larger than a page. Nevertheless, it does not suit the methods with replications or migrations, which I have mentioned in Section 2.3.1.

2.4 Cooperative Caches and their Optimization

2.4.1 Overview of Cooperative Caches

NUCAs for multicore or many-core processors, which I have explained in Section 2.3.1, are regarded as methods that enable shared caches to benefit from an advantage of low latency of private caches. In contrast, cooperative caches add an advantage of shared caches or flexibility on capacity to private caches.

Chang et al. proposed the Cooperative Caching [5], the first organization of cooperative caches. The term Cooperative Caching itself is invented in a context of caching of file systems [63], rather than that of processors. In this context, when a piece of cached information is about to be removed, the node transfers it to another node up to fixed times if no nodes have its replica. This enables file systems to store much unique data and to serve frequently used data quickly to the users. Chang et al. extend this idea to caching of processors. Namely, when a cache line is about to be evicted from the chip, the cache allows it being migrated (or *spilled*) to another cache if no other caches have its duplicate.

The Cooperative Caching has a single centralized organization called Coherence Engine, which stores duplicate tags of all the cache line in the chip. However, concentration of access to the centralized unit limits the scalability.

The Distributed Cooperative Caching (DCC) [6] divides the unit into a number of Distributed Coherence Engines (DCEs) and distributes them over the chip. Each DCE is partial directory cache, that is, it manages coherency of an interleaved part in the address space.

In the rest of this thesis, I often call a directory cache just 'a directory' to avoid the confusion with the regular caches. In addition, I use a term 'a line' or 'a cache line' for a block in the regular caches; I use 'an entry' or 'a directory entry' for an element in the directory caches (or the DCEs).

Although the CC and the DCC suppose directory-based coherence, the idea of spilling is also applicable to snooping, which is usually used for a small number of cores. Therefore, some methods to optimize spilling, which I mention in the next section, depend on snooping.

2.4.2 Distributed Cooperative Caching

Figure 2.9 (a) shows the organization of the DCC, where L2 caches are the last level. Each core has a PE (Processing Element), L1 caches, and an L2 cache. DCEs are distributed over the chip. Although the number of cores and that of DCEs are not necessarily the same, both of them are 4 in Figure 2.9 for ease of the figure. The cores, the DCEs, and a main memory are connected via some interconnects.

I explain typical behavior of the DCC with from (b) to (e) in Figure 2.9.

First of all, I assume a situation that both an empty line in a cache and an empty entry in a DCE are found. Example (b) shows the behavior when a core misses in its own L2 and no other cores have the requested line. The core sends a request to one of the DCEs specified by the line address. The requested DCE searches for a valid entry corresponding to the line. However, it does not have such an entry. Therefore, it writes the tag and some supplementary information to an empty entry while it forwards the request to the main memory. The main memory supplies the required data to the core.

On the other hand, if the requested line hits in one of the remote core, that is, it has already been kept by another core (other cores), the behavior is like Example (c). The requested DCE checks its directory cache in the same way as Example (b). Since a valid entry is found this time, the request is forwarded

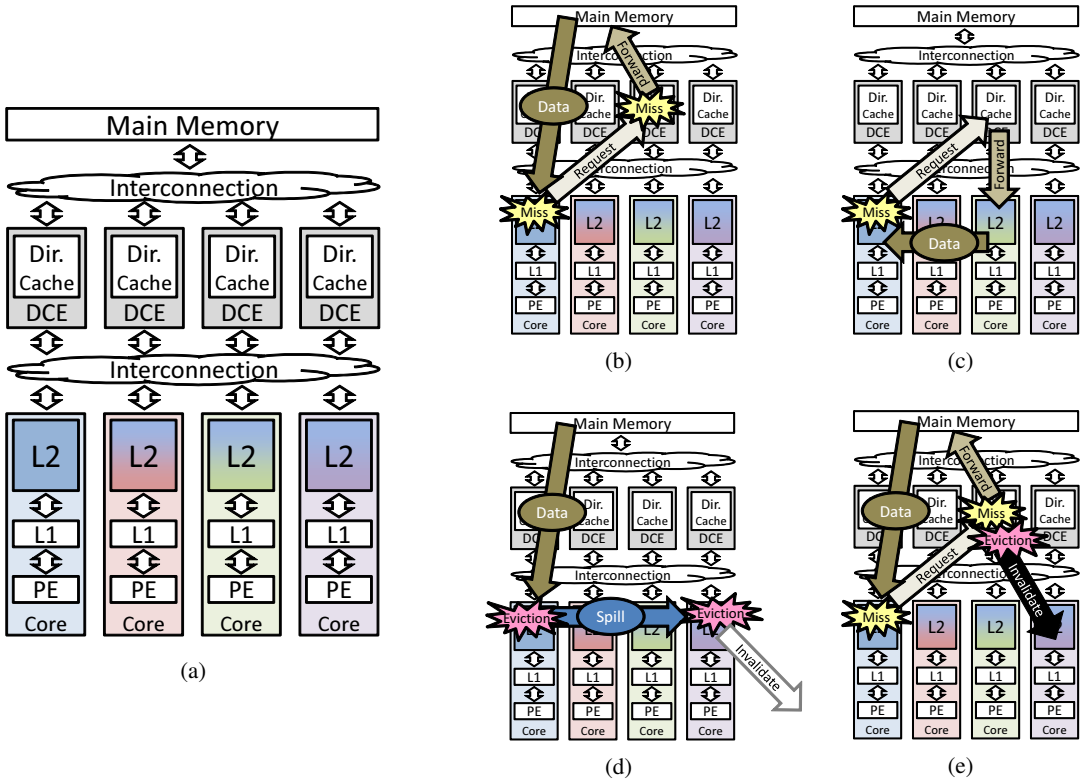


Figure 2.9 The organization and working examples of the Distributed Cooperative Caching.

to one of the sharer rather than the main memory. It modifies the entry so that the requester is added to the sharer. When the core receives the forwarded request, it transfers its own line to the requesting core via cache-to-cache interconnection. In this case, the cooperative caches avoid the access to the main memory, and thus reduce the latency.

Secondly, I assume that there is an empty directory entry but no empty cache lines are found. Example (d) shows the behavior in such a situation. To store the requested line, one of the lines in the same set as the requested line is chosen and evicted. The most important feature of the cooperative caches is to allow the evicted line spilled to another core. When doing so, the core forwards the line to one of the other cores of its choice and notifies the occurrence of spilling to the corresponding DCE . The DCE properly modifies the sharer stored in the entry.

When the core receives a spilled line, if it has no empty lines in the set where the spilled line is going to be kept, one of the lines is also evicted. To avoid ripple effects, the hereby evicted line is prohibited from spilling: the line is just evicted from the chip.

Moreover, the DCC limits the number of continuous spilling of a single line to up to one, that is, when a spilled line is evicted from a cache without being reused, it is removed from the chip. To see whether lines are spilled from the other cores or not, each line has an additional bit. This bit is set when

the corresponding line is spilled and cleared when it is accessed by cores. If the bit is set on eviction, the line will not be spilled. In addition to already spilled lines, lines that are currently shared are also refused to be spilled, because the number of unique lines in the chip is not declined though they are just discarded. In this thesis, I define lines that are not either spilled after their last access or shared at present as *movable* lines.

Lastly, I explain a situation that no empty directory entries are found using Example (e). Like Example (b), the requested DCE searches for the entry corresponding to the required line. Unfortunately, it does not find either the corresponding entry or an empty entry for new directory. In this case, one of the entries is selected for a replacement and is invalidated. It results in sending requests of invalidation or write-back to all the sharers of the corresponding line. The new directory is stored in the invalidated entry.

2.4.3 Optimization of Spilling

A typical case where cooperative caches benefit is that cores that require a large cache borrow some cache lines from unused cache regions of other cores. Since most cache references are done in local, the average latency is small. Moreover, they improve the performance of cores that needs additional capacity without degrading the other cores.

On the other hand, bad situations for the cooperative caches also exist. One of them is that cores that hardly have unused regions and thus are sensitive to pollution are running with harmful cores that spill many lines. In this case, the improvement of the latter cores can be offset by the loss of the former cores.

Therefore, it is desirable to spill lines preferentially to cores with unused cache regions, instead of cores sensitive to pollution. Alternatively, when there are no cores with most of their caches unused, it might be wise to prohibit all the cores from spilling. Hence, an important way to improve the cooperative caches is to estimate the demand on capacity of each core precisely in order to spill lines properly. I mention such methods in this section.

Adaptive Selective Replication [7] has an organization to decide whether the spilling is allowed or not with a probability. The probability of each core is adaptively modified with some performance counters. However, it does not consider the destination of spilled lines, and thus has a limited effect.

Elastic Cooperative Caching (ElasticCC) [9] extends a method of cache partitioning, which is called ASP-NUCA [64], to the DCC. Figure 2.10 shows its organization and virtual hierarchy. Each last level cache is virtually partitioned to private region and shared region: The private region only stores cache lines accessed by its own core; the shared region accepts the spilled lines. In other words, each private region is considered as virtual private L2 cache and the whole of shared regions are regarded as virtual shared L3 cache. The virtual partitions are modified periodically. For load balancing, the destinations

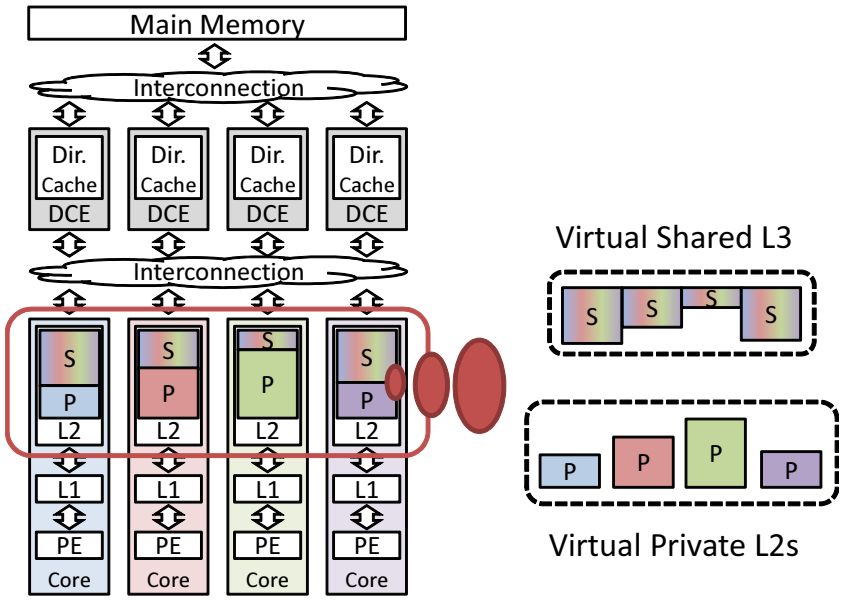


Figure 2.10 Organization and virtual cache hierarchy of the ElasticCC.

of spilled lines are selected in proportion to the size of the shared regions. Since this method rarely requires broadcasts for spilling management, it has high scalability. Nevertheless, it has two kinds of shortcomings. One is limited independency of performance. This is because it requires at least one way per core for the shared region. The other is that it does not suit some replacement algorithms, for it is based on way-partitioning.

Focusing on high scalability of the ElasticCC, we previously proposed HFC (Hit Frequency Counters) Partitioning [65] based on the ElasticCC as a spilling method appropriate to many-core. It tries to improve the precision of repartitioning.

Adaptive Variable-Granularity Cooperative Caching (AVGCC) [10] focuses on a situation that the demands on capacity vary with sets. While most methods manage spilling in core granularity, the AVGCC does it in granularity of a set or a group of sets. This finer granularity improves efficiency in some applications. However, the other applications do not benefit from this method. When it is used with hash functions that I have mentioned in Section 2.1.3, the number of applications favoring the AVGCC may even get small. Moreover, too fine granularity can harm the precision of the performance estimation.

Dynamic Spill-Receive (DSR) [8] has particularly high efficiency among such spilling managements. It categorizes the cores into spillers that can forward their evicted lines and receivers that accept the spilled lines.

Figure 2.11 shows how the DSR classifies cores. It devotes around 32 sets per core (only two sets

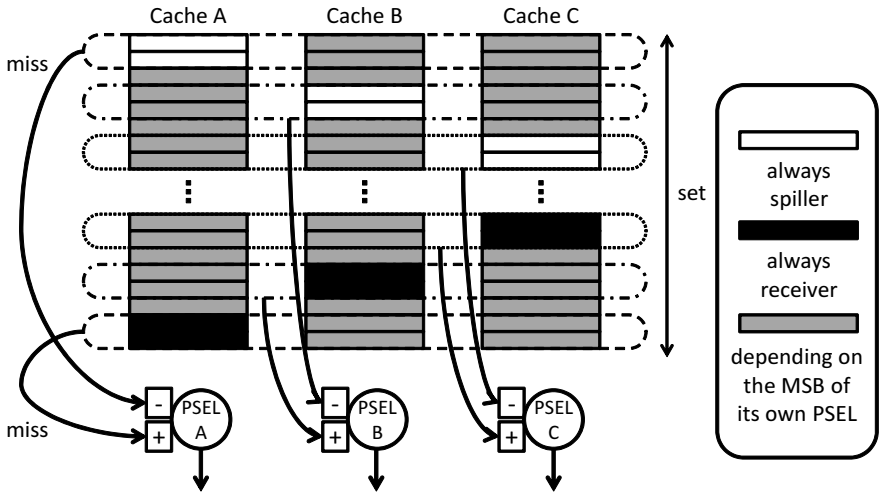


Figure 2.11 Samplings of sets in the DSR.

in them are shown in the figure for ease) to sampling sets where the evicted lines are always permitted to spill. In similar, it devotes the same number of other sets to another group of sampling sets where the corresponding core is always a receiver. Afterwards, it counts the number of misses in each group of the sampling sets in each core with per-core counters called PSELs. A miss in 'always spiller' sets decrements the corresponding counter; a miss in 'always receiver' sets increments it. The majority of the remaining sets follow the most significant bit (MSB) of the counter. The MSB of 0 means that the number of misses on being receiver is less than that on being spiller, therefore they choose to be receiver. In contrast, they choose to be spiller when the MSB is 1. This sampling-based prediction shows high precision and makes the spilling management efficient in the most cases.

Nevertheless, the DSR have a shortcoming of lack of scalability. Since it must share information of all the cache misses among cores, it requires a snooping-based coherency scheme or frequent broadcasts. Besides, while they need about 64 sets per core for the sampling for precise estimation, the caches will not have such a large number of sets if the number of cores increases. This results in the decline in the number of sampling sets per core, that is, the prediction accuracy. Therefore, the lack of scalability of DSR spoils the efficiency in many-core.

The conclusion of this section is that, as far as I know, there are no existing methods to manage spilling that achieve both high efficiency and scalability. I propose the first method to meet both of them in Chapter 4.

2.5 Summary

In this chapter, I explained many related researches in some categories. I first divided them into three classifications: mainly for a single core, considering multicore, and scalability-oriented for many-core. The first classification consists of dead block prediction, replacement algorithm, and improving the efficiency of associativity. The second is composed of thread-aware replacement algorithm and cache partitioning. The last includes improvement of NUCA and coherence managements.

Based on these backgrounds, I referred to the DCC, a scalable approach to add an advantage of shared caches to private caches, as the target architecture of my proposal. It manages coherence separately with directory caches called DCEs. A core that misses in its portion of the LLC sends a request to one of the DCEs. If there is a hit in the DCE, one of the sharers of the line rather than the main memory supplies the data.

To benefit from flexibility on capacity, the DCC allows some evicted lines being spilled to other cores. However, some combinations of application harm the efficiency of spilling. To avoid this requires a proper method to manage the spilling. There are a number of such previous methods. Unfortunately, I found that there were no methods that achieved both efficiency and scalability. This is what I am going to solve.

Chapter 3

Proposal of Useful Simulation Infrastructure

In this chapter, I propose a useful simulation infrastructure to evaluate methods for multicore and many-core processors with shared memory. To confirm both efficiency and scalability, I evaluate them in two different environments. One models modern multicore processors. The other is under assumption of future many-core processors. I use respective in-house simulators of them that center a MIPS simulator named **SimMips**[11], which we have developed.

In the multicore environment, while I use a trace-based simulator written in Ruby to get the evaluation results for the simplicity of description, SimMips is utilized as a useful and powerful tool to obtain traces. Though outputting the traces requires some modification to the simulator, the design of SimMips makes the modification easy.

For the many-core environment, I am developing an execution-based, cycle-accurate simulator named SimMccc (Simulator of Many-Core Cooperative Caching), where a part of SimMips embedded as a component. It has practical simulation speed even though it simulates caches, directories, and a network in detail.

Later in this chapter, I mention the evaluation of the existing methods to optimize spilling in cooperative caches to show the usability of the infrastructure. As I have explained in the previous chapter, none of the existing methods have both efficiency and scalability. I quantitatively confirm the fact through the evaluation with the proposed infrastructure.

In Section 3.1, I define the organization of the environments. I mention the simulation infrastructures based on SimMips in Section 3.2 and the comparison with other infrastructures in Section 3.3. I then explain the details of the multicore and many-core environments in Section 3.4 and Section 3.5, respectively. Both consist of the parameters, the selection of benchmark applications, and the construction of workloads. In Section 3.6, I show the evaluation results of the existing cache architectures, including cooperative caches with some spilling optimizations.

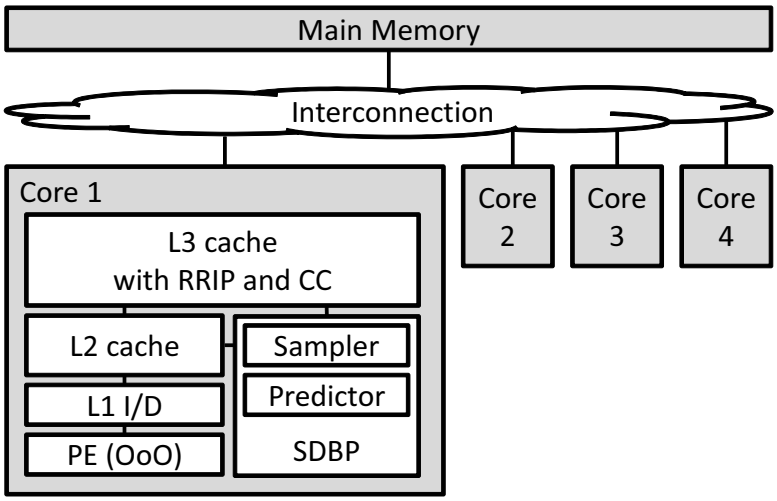


Figure 3.1 The architectural model for the multicore environment.

3.1 Target Architectures

3.1.1 Modeling Modern Multicore

Figure 3.1 shows the architectural model of the target multicore processor. The Processing Element (PE) supports high-performance out-of-order (OoO) execution. It has a memory hierarchy that consists of L1 instruction caches, L1 data caches, L2 private caches and L3 cooperative caches, and main memory. In the L3, some evicted lines are spilled to other cores. This model does not specify coherency scheme or interconnection.

The L3 adapts the Sampling Dead Block Prediction (SDBP) [2] as a dead block prediction method and the Re-Reference Interval Prediction (RRIP) [3] as a replacement algorithm. They urge hardly reused lines to be removed from the chip. According to the results of preliminary evaluation for the selection of architectures, although applying either of them improves efficiency to some extent, using them together has further effect. Therefore, the target architecture adapts both of them.

On adapting the SDBP, I prefer that each core has a sampler and a predictor, which is independent from the other cores, than all the cores share the units like the original proposal [2]. While the target of the original is a shared cache, cooperative caches are more close to private caches than shared caches. Hence, it is natural to have private units as well as caches.

Cache lines predicted dead by the SDBP on insertion do not bypass the L3, but are given the lowest priority in the same way as the SHiP [24]. They are also excluded from targets of spilling. The prediction tables of the SDBP are composed of 3-bit saturating counters, which are also inspired by the SHiP, rather than 2-bit skewed counters [2].

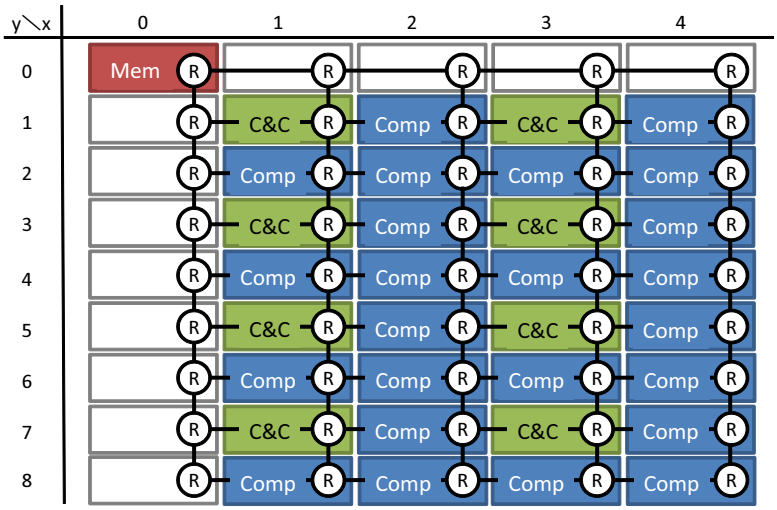


Figure 3.2 The architectural model for the many-core environment.

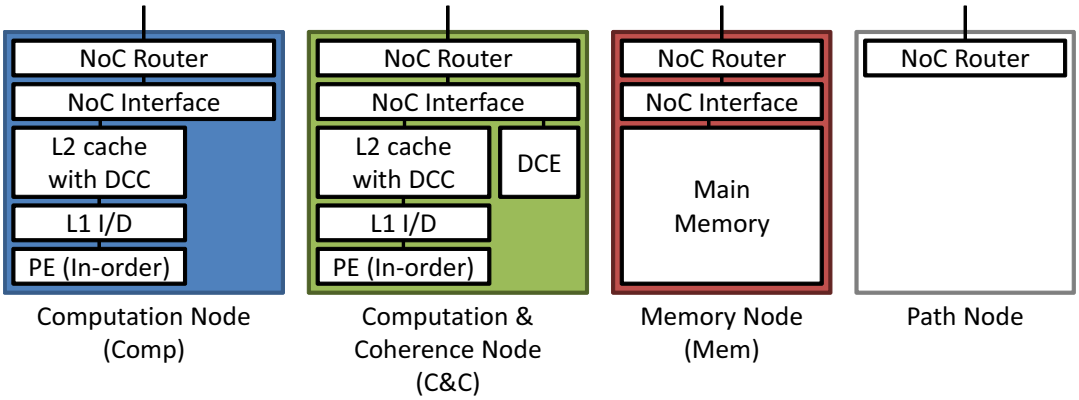


Figure 3.3 Four types of node on the target many-core.

3.1.2 Modeling Future Many-core

Figure 3.2 shows the model of the target architecture in the many-core environment. It consists of some kinds of nodes. The most important feature of this environment is a network-on-chip (NoC) [66] as an interconnection among nodes. Each node has an on-chip router (R in the figure), which is connected to up to four adjacent routers. They enable communication between any pair of nodes by transferring packets sent from elements of nodes. In the target architecture, 45 nodes of 5 columns and 9 rows constitute a single chip.

The target architecture classifies nodes into four types shown in Figure 3.3. A blue node is a computation node (Comp). It includes an in-order Processing Element (PE), an L1 instruction cache, an L1 data cache, an L2 cooperative cache, and an NoC interface for sending/receiving packets to/from

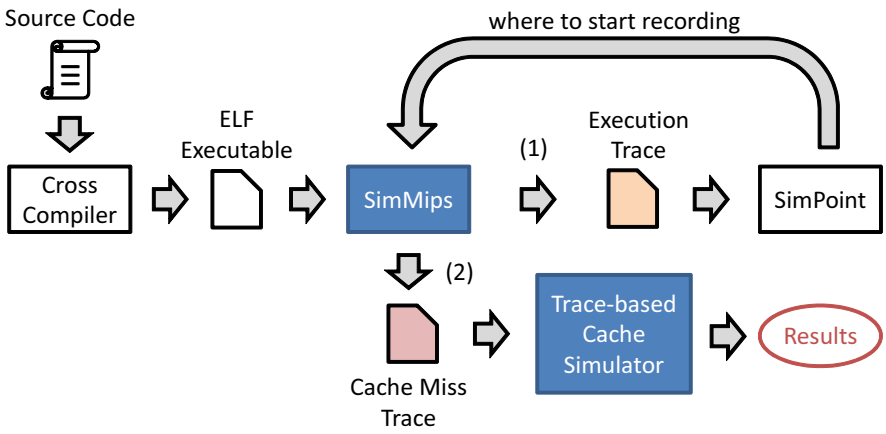


Figure 3.4 Flow of the evaluation in the multicore environment.

the NoC. A green node is a computation and coherence node (C&C) that has a DCE or a directory for maintaining coherence in addition to elements of the computation node. A red node is a memory node (Mem), including a (controller of) main memory and an NoC interface. A white path node only has an NoC router. It is used for relaying packets generated in the other types of nodes.

Tiling 24 computation nodes, 8 C&C nodes, 1 memory node, and 12 path nodes as shown in Figure 3.2 constitutes a single many-core processor with 32 (24 + 8) cores and 8 directories. The ratio between the numbers of computation nodes and C&C nodes is selected in consideration of load balancing.

In this environment, I do not apply the SDBP and the RRIP. Therefore, the caches treat all the lines as live and decide lines to be replaced under the LRU.

3.2 Simulation Infrastructure

3.2.1 SimMips: MIPS System Simulator

The key element of the simulation infrastructure is a MIPS system simulator called SimMips [11], which we have developed and released as free software. It is written in plain c++ code and its structure is easily modified and extended. Moreover, it has a practical simulation speed of about 10 million instructions per second. These features make prototyping of a new architecture easy and thus increase the availability in research.

SimMips offers two modes, OS-Mode and App-Mode. The App-Mode stands for the application mode where statically linked user program runs. The OS-Mode read an OS kernel as an executable file. I use the App-Mode here, that is, system calls are emulated by the simulator and the mapping between logical and physical addresses is fixed.

Figure 3.4 depicts the flow of the evaluation in the multicore environment using SimMips. Blue

box represents a tool that I developed or modified for my infrastructure. Since detailed simulation is much slower than real machines, it is impractical to simulate the whole benchmark. Instead, we usually simulate a part of the execution that represents the benchmark. We can use SimPoint [67] to detect the most typical part. I utilize it to decide where to start detailed simulation.

I give SimMips read an ELF executable file and obtain an execution trace that includes the location and number of branch instructions executed for each period of cycles. When we give this trace to SimPoint, it clusters periods similar to each other and outputs in which period the most typical part of the benchmark is executed.

The main evaluation to get the evaluation result is done by a trace-based cache simulator that reads traces of cache misses. SimMips is used again to obtain them. This time it simulates the benchmark as usual before the execution reaches the starting point obtained by SimPoint. After that, it switches to the detailed simulation and begin to output the target address, the location, etc. of each instruction that missed the L1 cache.

I extend SimMips for obtaining traces I have explained above. The extensions include the implementation of emulation of some system calls, adding L1 caches and write buffer, and the logging functions of branch instructions and cache misses.

SimMips is a functional-level simulator: adopting a model that a processor executes a single instruction in a cycle. In the design of SimMips, execution of an instruction is divided into eight methods with consideration of typical pipeline stages of processors (instruction fetch, decode, etc.) and done by calling them in order. This design makes it easy to make a prospect of additional features.

Take as an example the addition of L1 caches and write buffer. Figure 3.5 represents a part of the description in the modified version of SimMips. I show the `Mips::drive` method that corresponds to the execution of an instruction by a MIPS processor core. In the original SimMips, as I have mentioned, calling the corresponding methods to pipeline stages in order, such as `fetch()` and `decode()`, forms the execution of a single instruction.

After modification, it checks if the core can continue executing the instruction with methods such as `fetch_ok()` and `decode_ok()`. In these methods, the core sees if the fetched instruction is kept in the instruction cache, the instruction requires memory access, and so on. If the core cannot continue running, it memorizes from which stage it resumes and then goes into a stall. From the next cycle, as described in the block from the 4th through the 13th line, it checks whether it comes to be able to resume, and if so, it resume the execution from the memorized stage.

After adding L1 caches and write buffer, the main reasons of the stall is as follows:

- the core misses the instruction cache and fails to supply the instruction to the decode stage (line 4 through 6, 18 and 19),

```
 1 void Mips::drive()
 2 {
 3   wb.drive();
 4   if (state == CPU_DECODE)
 5     if (! cache->read(L1Cache::IC, ca))
 6       return;
 7   if (state == CPU_EXECUTE && wb.running())
 8     return;
 9   if (state == CPU_MEMSEND && !wb.writable(inst->paddr))
10     return;
11   if (state == CPU_MEMRECV)
12     if (! cache->read(L1Cache::DC, ca))
13       return;
14
15   switch (state) { // NOTE: This is fall through
16   case CPU_FETCH:
17     fetch();
18     if (!decode_ok())
19       break;
20   case CPU_DECODE:
21     decode();
22     regfetch();
23     if (!execute_ok())
24       break;
25   case CPU_EXECUTE:
26     execute();
27     ...
```

Figure 3.5 A part of the `Mips::drive` method that executes an instruction if possible.

- it fails to write at the memory store instructions due to the full write buffer (line 9 and 10), and
- it misses the data cache at the memory load instructions (line 11 through 13).

The corresponding methods to these stages are `decode()`, `memsend()`, and `memreceive()`, respectively. Therefore, I put the additional methods right before the corresponding methods.

Incidentally, another check is added before the execute stage (`execute()`) in order to guarantee that the write buffer stores all the data to the cache before emulating a system call.

Such ease of prospects with the design and description of SimMips is important in shortening the time required for expansion and making us concentrate on the essential part of the research.

3.2.2 SimMccc: Simulator of Many-core Cooperative Caching

SimMc [68] is a simulator for many-core processors where SimMips is embedded as a computation core. It simulates M-Core Architecture [68] or a many-core architecture with distributed memory:

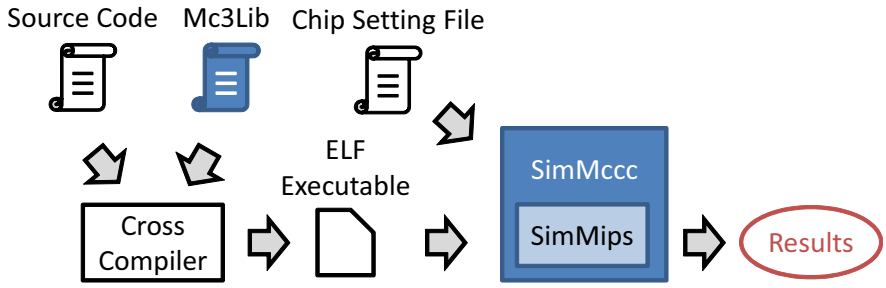


Figure 3.6 Flow of the evaluation in the many-core environment.

each node has an independent node memory and exchanges data with other node explicitly using direct memory access (DMA).

I am developing SimMccc that simulates many-core processors with shared memory by extending SimMc. In the many-core environment, the time for inter-node communication via an NoC heavily contributes the latencies of remote cache hits or cache misses. Thanks to SimMc's detailed simulation of an NoC, SimMccc precisely measures the latencies. This is important for the validity of the evaluation. The detailed NoC simulation also enables to do interesting experiments like that I will show in Section 5.4.2.

Figure 3.6 shows the flow of the evaluation with SimMccc. Applications for the target many-core processors are written in C/C++. Their ELF executables are obtained by a cross compiler with a library called Mc3Lib that provides atomic memory operations, synchronization, timers, and so on. In addition to one or more ELF executable files, SimMccc requires a chip setting file that specifies the configuration of simulated many-core, such as the placement of nodes and applications, the sizes of caches, etc. The evaluation results are obtained through the simulation.

The embedded component of SimMips (shown as the tinted box in Figure 3.6) is responsible for the execution of MIPS instructions. It executes instructions only when the required data are accessible in the local primary caches. When it misses, the memory hierarchy presented by SimMccc, which is composed of the caches, the directories, and the main memory, eventually supplies the data.

Communication among nodes in the SimMccc is formed by multiple layers as shown in Figure 3.7. Units in different nodes on the same layer communicate via paths below that layer. Interfaces between layers are appropriately designed so that the units can communicate without regard to the detail of the lower layers. This is important in point of ease of description and improvement, because this makes it easy to reuse the codes and isolate the problem.

I explain each layer in order. The lowest layer is a network-on-chip layer (NoC Layer). NoC routers and NoC interface (called INCC or Inter-Node Communication Controller in SimMc and SimMccc) communicate each other via this layer. Their message is called a packet that consists of one or more

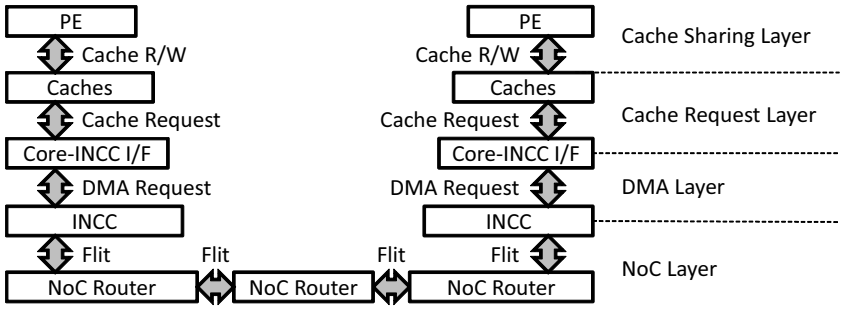


Figure 3.7 Multiple layers of communication in SimMccc.

flits (flow control digits). NoC routers see the destination of each packet and select the proper route.

The second layer is a direct memory access layer (DMA Layer). NoC interfaces accept a DMA command as an external input. It includes a destination node, source and destination addresses, and so on. NoC interfaces generate one or more packets from the information in the command and, if needed, the data being read from the source address. Sending these packets to the NoC layer enables DMA communication.

Original SimMc only offers these two layers: data sharing among the cores is realized by their direct issuance of DMA commands. SimMccc offers additional two layers above them.

The first additional layer or the third layer is a cache request layer. A cache request consists of a request type, a destination unit, a target address, and (on some types) data or additional information. Caches, directories, and a main memory controller send cache requests to each other in order to realize cache sharing. For example, an L1 cache miss on read generates a read cache request for the L2 cache. As another example, when a directory receives a write request to a shared line, it sends requests of cache-to-cache transfer or invalidation to the caches sharing the line. If a unit on this layer sends a cache request to another unit in other node, the request needs to be passed via the lower layers. In SimMccc, this is enabled by an interface unit between cache request layer and DMA layer, called Core/INCC interface (Core/INCC I/F). It interconverts between a cache request and a DMA command.

The other additional layer or the highest layer is a cache sharing layer. Cores use this layer rather than DMA layer in SimMccc. L1 caches offer memory access as an interface to cores. If an L1 cache misses, related units eventually provide the appropriate line in the L1 and then it is accessible from the core.

An interface for atomic memory operations are also prepared for this layer. Most RISC processors realize an atomic operation with a pair of special memory instructions called load linked and store conditional [1]. In my implementation, the read access of load-linked returns the same result as normal read access; however, it requires not permission for read but permission for write. the write access of store-conditional checks if the cache still keeps the permission that load-linked access requests. The

access succeeds if the permission remains; if the permission has been lost, the cache notifies failure of the access to the core immediately without propagating the cache miss.

Cores realize atomic operations by executing them consecutively. Synchronization such as lock and barrier is provided by library functions of Mc3Lib. Application programmers can describe synchronization easily by just calling these functions.

I mention cache sharing in SimMccc more in detail. The key to correct execution in processors with shared memory is an appropriate design of a coherence protocol. If the protocol has a fault, the caches may give wrong results to the cores or, even worse, they may even cause deadlock. Therefore, the protocol needs to be considered and tested well.

Since caches and directories receive a cache request as an input as I have mentioned, the problem of coherence protocols is regarded as the design of tables that give the operations and the next state of the line/entry according to the request type and the current state of the line/entry. Examples of tables with the simple MSI protocol are shown in [1]. However, the protocol that I consider here is much more complex for the following reasons:

- L1 and L2 caches are treated as separate units. For example, when an L2 invalidates a line that exists in the L1, the L2 needs to wait for invalidation or write-back of the L1.
- Instead of the MSI protocol, the MOESI protocol is used. Thanks to additional states of the MOESI, the caches distinguish *movable* lines in the DCC.
- Some evicted lines from one cache are spilled to another cache. An important feature of the DCC complicates the protocol.
- In addition to the caches, the directories invalidate cache lines due to conflicts of directory entries. It is also a feature of the DCC. In particular, when spilling from a cache and invalidation from a directory occur almost at the same time, the protocol requires additional transient states. As a result, the protocol gets much more complex.

To mitigate the complexity of the protocol, I make the following assumptions:

- The protocol forbids instruction and data caches in the same core to keep a line at the same time. This requires only a few modifications on the linker script so that a line cannot contain both instruction and data regions.
- Point-to-point order of requests is preserved: cache requests from unit A to unit B must arrive in order. The directory-based protocol shown in [1] is also having this assumption. It constrains the routing from a sender to a receiver. This constraint is general in NoCs of many-core processors.
- Replaced lines are kept in small buffer until they are completely invalidated in the unit. This enables invalidation (or write-back) and insertion to proceed simultaneously. As a result, situations

Table 3.1 Transition of L1 caches.

Request	MCR_READ		MCR_READP		MCR_WRITE	
State	Read		Read(Linked)		Write	
M	Readable		Readable		Writable	
S	Readable		Readable		WRITE to L2	IM
I	READ to L2	IS	WRITE to L2	ISM	WRITE to L2	IM
IM						
ISM						
IS						
IMI						
ISI						
Request			MCR_DATA		MCR_INVALID	
State	Replace		Data		Invalidate	
M	PUTM to L2	I			PUTM to L2 ACK to L2	I
S	INV_L1 to L2	I			ACK to L2	I
I					ACK to L2	
IM			Fill and Write	M	ACK to L2	IMI
ISM			Fill and Read	M	ACK to L2	ISI
IS			Fill and Read	S	ACK to L2	ISI
IMI			Fill and Write PUTM to L2	I	ACK to L2	
ISI			Fill and Read	I	ACK to L2	

of suspended transaction are reduced and thus the protocol is simplified.

Despite such simplifications, the actual protocol has considerable complexity. Table 3.1, Table 3.9, and Table 3.10 show the transition of L1 caches, L2 caches, and directories, respectively. Note that Table 3.9 is divided into multiple pages. For the convenience of the layout, Table 3.9 and Table 3.10 are at the end of the chapter.

Each column represents a type of cache request. Each row stands for the state of the corresponding line/entry. Each cell is divided into right and left.

The left of a cell represents the operations that the unit should perform. A Cell with a light gray background means that the cache request is suspended until the transient state is dissolved by other requests. A Cell with a dark gray background stands for an unreachable situation under normal circumstances; it is reported as an error. Red strings in a column or a cell in Table 3.9 means the request should be sent from the L1 caches. Blue strings stands for requests from the other units. For simplicity of notation, sending request *A* to unit *B* is expressed as "*A* to *B*" and the requester core is abbreviated to *Reqer*. In addition, a table to memorize the cores that have sent requests is described as RT (Requester Table). This table is used when the unit must respond to the requests later (e.g. data is not ready).

The right of a cell represents the next state of the line/entry. If more than one state is written, the next state depends on the outcome of the conditions written in the left of a cell.

For instance, when a cache line in an L2 cache is under the M (Modified) state, and another core sends a GetS or a read request to the line to the cache through a directory, the corresponding cell has

Table 3.2 Source code organization of SimMccc Version 0.9.8.

Name	# of lines	feature
(top directory)	22	main function
env	1,699	simulation environment
core	*6,051*	processor core and system call emulation
cache	2,473	cache controller
dir	947	DCE (directory cache)
memory	1,303	memory controller and Core-INCC I/F
network	*1,434*	NoC router and INCC
partition	243	cache repartitioning of ElasticCC
spill	790	spilling management
common	842	utilities, constants, etc.
(Total)	15,804	

the following expression:

if L1_valid:	
INV to L1	MO
Add Reqer to RT	/
otherwise:	O
DATA to Reqer	

In this case, the request is handled differently depending on whether the data remains in the L1 caches. If so, since one of the L1 caches only may hold the latest data, the L2 cannot send the data immediately. Instead, the L2 sends an invalidation (INV) request to the L1s [*1] in order to have them write back the latest data if needed. In addition, the L2 records the sender to the RT in the line and sets the state of the line to MO (Modified to Owned). The reply is sent after receiving Acks (acknowledgements) from L1 caches (See the transition of the LastAck request to the MO state).

If the data do not remain in L1 caches, the L2 cache can reply to the requester immediately. The state of the line becomes O (Owned).

I mention the source code and the execution speed of SimMccc. Table 3.2 summarizes the organization of source code of the latest SimMccc. The leftmost column represents the name of subdirectories. The source code is divided into multiple subdirectories by purpose.

The total number of lines of the source code is more than 15,000. However, most files in core directory are taken from the modified version of SimMips and all the descriptions in network directory come from SimMc. When excluding them and common descriptions of processor simulators such as a program loader, newly implemented part is about 7,500 lines.

Since SimMips itself executes 10M instructions per second. Assuming no overhead, SimMccc would

[*1] Since a request of invalidation always requires an Ack from both L1 caches, incrementing AckCount or the number of Acks to be sent by two is included in the process of sending INV to the L1s. Such implicit operations are omitted for simplicity of the tables.

simulate about 300k cycles of a 32-core processor per second. However, the actual speed is limited for the simulation of caches, directories, and an NoC. In particular, detailed NoC simulation takes much time. The simulation speed depends on the network usage i.e. the frequency of misses in the LLCs.

When I simulated some 32-core workloads on a computer where SimMips ran at 10M instructions per second, the actual simulation speed varied from 5k to 15k cycles per second. The result means that its slowdown is 20-60 times more than that proportional to the number of cores. Nevertheless, it also means SimMccc can complete 100M cycles of simulation within 6 hours. I think that if the time an execution takes is less than 8 hours, that is, if a simulation that was started late at night is finished by the next morning, the slowness of simulators is acceptable. Therefore, this simulation speed is acceptable.

3.3 Comparison with other infrastructures

As I have mentioned above, my infrastructure that centers SimMips offers a trace-based simulator for the multicore environment and an execution-based simulator (SimMccc) for the many-core environment.

Which type of simulation should we use depends on target architectures. Trace-based simulation is simple and fast because the computation unrelated to what to simulate is reduced. However, it is difficult to deal with multithreaded applications, and what is worse, traces may require huge amount of disk space. On the other hand, since execution-based simulation generates necessary information on the fly, it can simulate multithreaded applications precisely. Nevertheless, it sometimes gets too complex and slow to use the simulator as a practical infrastructure.

My infrastructure offers both types of simulation. Moreover, the simulators are carefully implemented so that the disadvantages of simulation may not lose their practicalities.

Of course, there are a number of infrastructures to evaluate memory systems for multicore and many-core processors. Though most of them are in-house and closed (CMP$im [69] for example), several infrastructures are available for research.

In particular, GEMS [70] is sometimes utilized by multicore and many-core cache researchers. The most important element of GEMS is a memory system simulator called Ruby, which is driven by an execution-based functional simulator. As a driver, a full-system functional simulator Simics [71] had been used for long; however, recently the M5 simulator [72], an open-source full-system simulator, has merged with GEMS. The new simulator that combined the M5 and GEMS is called gem5 [73].

Both SimMccc and gem5 have features of detailed memory system simulation. Though gem5 has a lot of functions that SimMccc does not have, only SimMccc supports cooperative caches. As I have mentioned in the previous section, the design of a coherence protocol for cooperative caches has some difficulties and takes much time. Therefore, my ready-to-use cooperative cache simulation will be useful for research on caches of future multicore and many-core processors.

Table 3.3 Architectural parameters of the multicore environment.

Element	Parameter	Value
Processor	# of Cores	4
	Model	4-issue OoO, 128-entry ROB
	ISA	MIPS32
Memory	Line Size	64B
	L1 I/D	32KB, 4-way, 1 cycle
	L2 private	256KB, 8-way, LRU, 10 cycles
	L3 (if private)	2MB, 16-way, 3-bit SRRIP,
		30 cycles (local), 50 cycles (remote)
	L3 (if shared)	8MB, 16-way, 40 cycles
	Main Memory	250 cycles
DBP	Predictor	SDBP, 8K 3-bit counters
	Sampler	Samples 32 out of 4096 sets,
		12-way, LRU
ASCEND	Additional Tags	65536-entry, 8-way, FIFO

SimMc [68], our existing simulator that SimMccc is based on is also one of open infrastructures. There are several components common to SimMc and SimMccc such as processor cores (i.e. SimMips) and on-chip routers. The main difference between them is the memory model. While SimMc expects distributed memory, SimMccc relies on shared memory where all the cores share a single address space and cores implicitly exchange data by accessing the same address. Choosing either distributed or shared memory is one of important design points for many-core processors; however, I do not discuss it here.

3.4 Experimental methodology for Multicore Environment

I mention the detail of the experiment with the multicore environment whose architectural model was defined in Section 3.1.1. I use the trace-based simulator written in Ruby. It is given traces where the type and the number of executed instructions of all the L1 misses, and the memory address and the instruction PC of all the L2 misses is recorded. The trace files are obtained with an extended version of SimMips.

Table 3.3 shows the architectural parameter. All the latencies shown in the table are expressed as round-trip time: instructions that miss the L1 caches are available after the passage of one of the time shown in the table. The parameters of caches are determined by reference to the simulator used in the Cache Replacement Championship [74] that is held in 2010. The parameters of dead block predictors (DBP) are set in consideration of the existence of spilling. Although the number of sets in the sampler before sampling is different from that of the L3 cache (2048 for private or 8192 for shared), it is not problematic with the SDBP.

It simulates 4-core multiprogrammed workloads using traces of 1G instructions of 12 SPEC

Table 3.4 The list of applications and their classes.

Class	Application	Skip	BaseIPC	BaseMPKI
A	445.gobmk	4.5B	3.112	0.067
	458.sjeng	2.5B	3.358	0.291
	433.milc	6.0B	0.818	15.024
B	435.gromacs	2.5B	2.994	0.593
	447.dealII	2.0B	3.817	0.088
	482.sphinx3	6.0B	1.146	6.136
C	401.bzip2	2.0B	3.004	0.714
	450.soplex	7.0B	2.011	5.052
	464.h264ref	25.0B	3.410	0.358
D	436.cactusADM	2.0B	2.953	2.096
	471.omnetpp	32.0B	1.387	7.340
	473.astar	13.5B	1.702	3.084

CPU2006 benchmarks shown in Table 3.4. The benchmarks are selected in consideration of the characteristics of memory access and classified into 4 groups. The selection and classification of the benchmarks are explained later. The column *skip* means how many instructions from the beginning are skipped. The column *BaseIPC* and *BaseMPKI* represent the baseline IPC and MPKI (Miss per Kilo Instruction) obtained with a 2MB private cache.

I determine the number of skipped instructions with SimPoint. I obtain the execution traces of 50G instructions from the beginning and give them to SimPoint. As the most typical part of execution of each application is identified, I decide the number of skipped instructions in order that the detailed simulation includes it.

The simulation continues until all the cores complete executing 500M instructions and the IPC (instructions per cycle) of each core in the first 500M instructions is recorded. If a core runs out the whole trace, then it reads from the beginning of the trace again. I calculate the performance and the QoS by comparing the obtained IPCs with the baseline IPCs. I use the average and the minimum of relative IPCs as standards of performance and QoS, respectively.

I evaluate the following 6 settings in this chapter:

Private has private caches that are used as the baseline. It is the same as the DCC without spilling.

Shared has shared cache. Its latency is a little longer than that of local hit in the private caches.

DCC means the Distributed Cooperative Caching [6]. Since there is no spilling management, it spills all the evicted lines that are *movable*. Remote hits have longer latency than local hits.

ElasticCC stands for the Elastic Cooperative Caching [9]. It used not SRRIP but the LRU as a replacement algorithm, because way-partitioning and its repartitioning method strongly depend on the LRU.

Table 3.5 The classification of applications for the multicore workload.

		Spilling Necessity	
		-	+
Pollution	-	A	D
Sensitivity	+	B	C

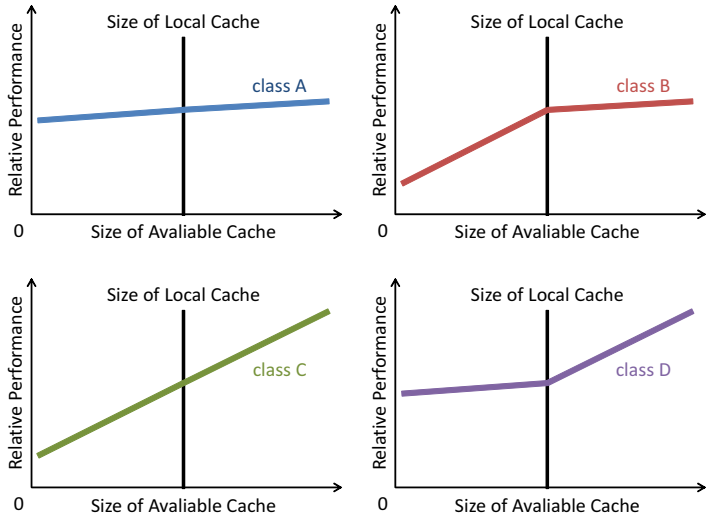


Figure 3.8 Typical relationship of each class between available cache size and performance.

DSR stands for the Dynamic Spill-Receive [8]. Although it is not so scalable, it performs well in the fewer number of cores.

DSR-QoS represents QoS-aware DSR. A variation of the DSR that penalize slowing cores.

For the selection of benchmarks, I define two axes that represent the characteristics of capacity. One axis is spilling necessity, the degree of improvement of a core with spilling. The other axis is pollution sensitivity, the degree of degradation of a core with receiving.

The benchmarks are divided by these two axes into 4 classes as shown in Table 3.5. I select the applications from all the SPEC CPU2006 benchmarks so that each class can contain three applications.

To measure the indexes of the axes, I execute an application alone in almost the same environment as the multicore environment. When I survey indexes of spilling necessity, the size of the local cache is fixed to 2MB and the size of the available remote cache is varied from 0B to 2MB. When I survey indexes of pollution sensitivity, the local cache accepts dummy lines in a fixed rate in order to emulate pollution and the remote cache is unavailable (i.e. spilling is prohibited). Except no pollution, the rate varies from 0.1 lines to 10 lines per 1,000 instructions, increasing exponentially. I measure the relative IPCs of applications with different sizes of the remote caches or different rates of pollution.

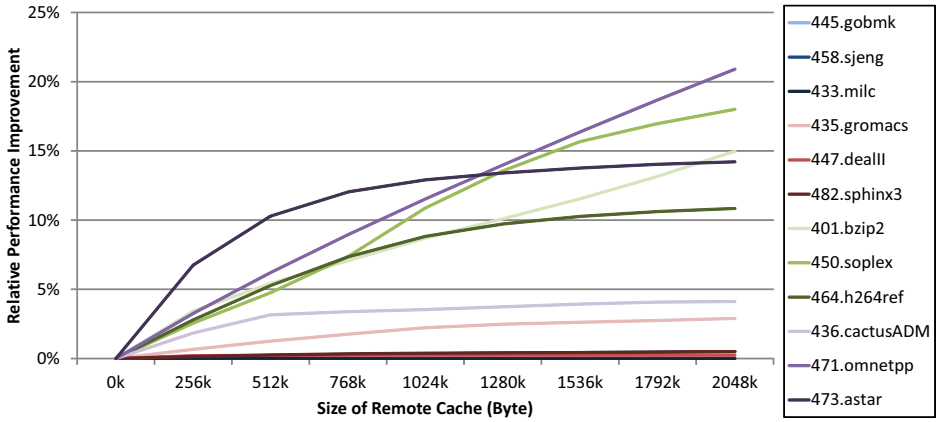


Figure 3.9 The barometer of spill necessity of the applications.

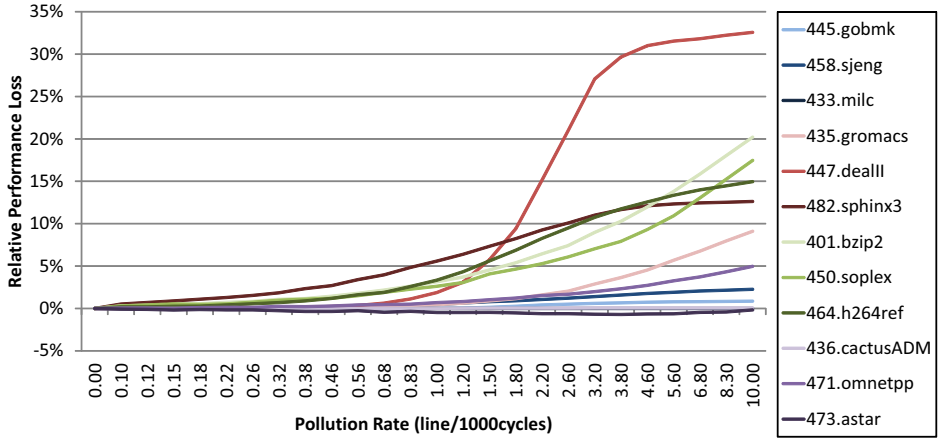


Figure 3.10 The barometer of pollution sensitivity of the applications.

Figure 3.9 and Figure 3.10 show the results of measurement for the indexes of spill necessity and pollution sensitivity, respectively. The x-axis is the size of the remote caches (Figure 3.9) or the pollution rate (Figure 3.10). The y-axis is the improvement or the degradation of the performance. Lines drawn in upper part of the graph mean high indexes of the corresponding axis. Each line represents the separated application and is colored according to its class. I assign blue, red, green, and purple to Class A, B, C and D, respectively. Note that green and purple lines (Class C and D) are drawn in upper part of Figure 3.9 and red and green lines (Class B and C) are shown in upper of Figure 3.10.

4-core multiprogrammed workloads are formed by combining 4 different applications from them. When we choose 4 applications from 12, the number of combination is as many as $\binom{12}{4} = 495$. It is unrealistic to simulate them all. Instead, I simulate 48 out of 495 workloads selected in the following way and calculate the average performance and QoS with weighted arithmetic mean.

Table 3.6 The combinational patterns of classes for workload selection.

Pattern	◯ ◯ ◯ △	◯ ◯ △ △	◯ ◯ △ □	◯ △ □ ×	Total
Combination of Classes	12	6	12	1	31
Selected Workloads	12	6	24	6	48
Total Workloads	36	54	324	81	495
Weight of Pattern	3	9	13.5	13.5	

First, suppose we choose 4 applications from 4 classes, allowing duplications of up to three. The number of multicombination is $\binom{4+4-1}{4} - 4 = 35 - 4 = 31$. Note that we exclude combinations that are composed of a single class, because each class has only three benchmarks. Each combination belongs to one of the patterns shown in Table 3.6. The number of workloads obtained from a combination of classes in each pattern is 3, 9, 27, or 81. For example, a combination of Class A, A, A, and B belongs to the pattern ◯ ◯ ◯△. The corresponding workloads to this combination are formed by all the three applications in class A and one of the applications in class B. Hence, the number of the corresponding workloads is 3.

For each combination of classes, I choose 1, 2, or 6 (according to the pattern) of the corresponding workloads randomly; but the number of times that each application is selected must be the same. As a result, I obtain 48 out of 495 workloads. The weights for weighted mean are obtained by dividing the number of the corresponding workloads to the combination of classes by the number of selected workloads from them. For example, when the A-A-A-B workload I have shown above is the only workload chosen from the corresponding workloads, the weight of this workload is $3/1 = 3$.

3.5 Experimental methodology for Many-core Environment

I next refer to the detail of the experiment with the many-core environment defined in Section 3.1.2. I use SimMccc for the evaluation.

Table 3.7 is the list of major architectural parameter. To determine the set in caches and directories, tabulation hash functions [75] are used. They are realized with an around-10-input XOR gate per bit. They are used to avoid extremely unfair usage of sets that may occur when we run replicas of the same application.

Depending on the type of the destination unit, different virtual channels of the NoC are used. This usage of virtual channels improves the efficiency of the network while keeping the requirement of the point-to-point ordering.

Transferring a flit to one of the adjacent routers usually takes 3 cycles; however, it may take more

Table 3.7 Evaluation parameters for the many-core environment.

Element	Parameter	Value
Processor	# of Cores	32
	Model	In-order, IPC = 1 if L1 hits
	ISA	MIPS32
Memory	Line Size	64B
	L1 I/D	8KB, 2-way, 1 cycle
	L2 private	256KB, 8-way, LRU
		10 cycles, inclusive
	DCE	8 DCEs, 24576 entries/DCE, 12-way
	Main Memory	250 cycles
Network	Topology	2D mesh
	Router	5-in 5-out, 2 VCs,
		X-Y dimentional order routing
	Latency	3 cycles/hop
	Bandwidth	16B/cycle

when the network is crowded. An inter-node request is not accepted until all of the flits in the corresponding packet are arrived at the destination node. When the memory node is asked for data, the requested data become ready for being sent to the network 250 cycles after the acceptance of the request.

In the evaluation, I run four 8-thread applications with 32 cores. The simulation continues until all the cores complete predefined tasks and the number of cycles of each application elapsed for the task. I exclude the time for preparing data in the memory and warming-up of placing a part of them to the caches. After all the cores have finished the warming-up, they start executing their task together. By comparing the reciprocal of the number of cycles (i.e. throughput) obtained [*2] with the baseline, I calculate the performance and QoS.

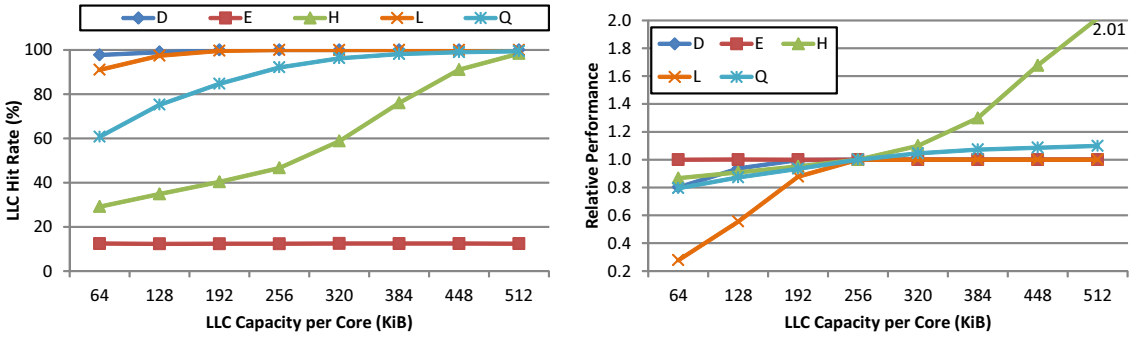
I evaluate 4 settings here: Private, DCC, ElasticCC, and DSR. Private, DCC, and ElasticCC are similar to those in the multicore environment. The DSR is difficult to be realized effectively in many-core processors due to the requirement of sharing the information of all the LLC misses among the cores. In the evaluation, I assume the existence of dedicated communication channel where the cores can broadcast the information of misses without latency. Therefore, it shows the theoretical maximum performance with the DSR.

Table 3.8 shows the five benchmarks used in this evaluation. They all are 8-thread parallel applications. The characteristics of caches in the benchmarks are shown in Figure 3.11. The x-axis is the size of L2 private caches per core. The y-axis is the hit rate of the L2 caches (the left graph) or the

[*2] It corresponds to the IPC in the multicore environment. In parallel applications, since busy loops for synchronization may occur, IPCs are not appropriate as standards of performance evaluation.

Table 3.8 Benchmarks to evaluate many-core environment.

Initial	Name	Description	Parameter	Paralellization
D	dijkstra	shortest path finder	128 nodes	task
E	equation	equation solver kernel [76]	1,024 elements square	data
H	himeno	Himeno benchmark [77]	XS size	data
L	lu	LU decomposition	448 elements square	data
Q	qsort	quick sort	448ki elements	data



L2 hit rate Performance

Figure 3.11 Effects of cache size on benchmarks for evaluation.

performance relative to with 256KB per core caches (the right graph).

I summarize the characteristic of each application.

dijkstra Since the size of the working set is much smaller than that of L2s, the cores can easily accept the spilled lines from others.

equation For the working set is much larger than the size of L2s, hit rate and performance seldom improve with larger caches. If the cores spill evicted lines to others, they may cause terrible performance loss of others. Hence, they should be prohibited from spilling.

himeno The working set is larger than the size of L2s and it benefits very much from additional cache size. The cores running it should spill their evicted lines.

lu Since the size of the working set is almost the same as that of L2s, the performance drops with smaller caches. It is desirable to prohibit or restrict their acceptance of spilled lines. However, because the working set gets small as execution proceeds, the chance of accepting lines without performance degradation may exist.

qsort Although the working set is as large as L2s like lu, the rate of performance drop is smaller than lu. In addition, the variation of access pattern by core and time is relatively large. The performance can be improved with applying different policies to the cores running it.

In the many-core evaluation, a workload consists of 4 applications where overlapping are allowed.

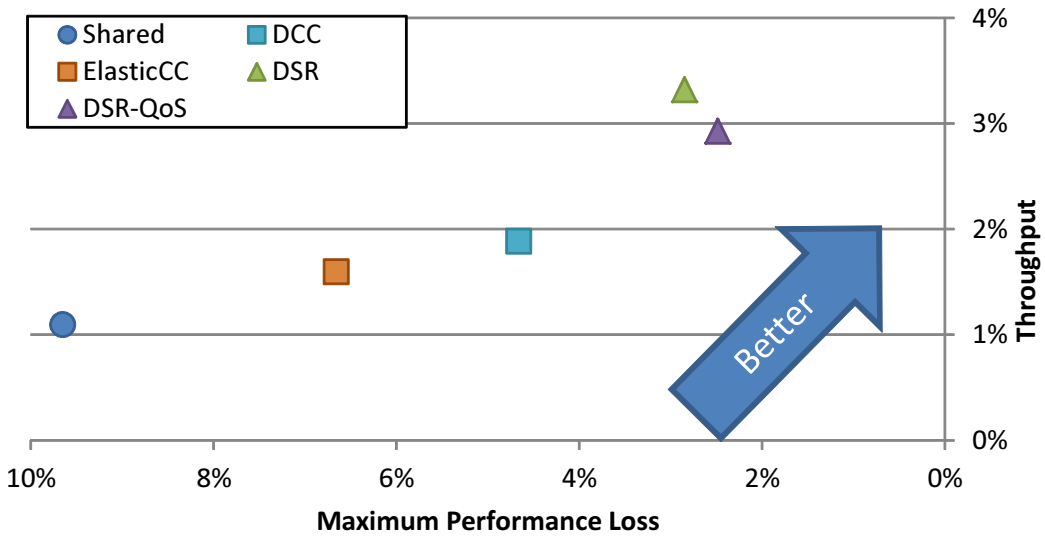


Figure 3.12 Performance and QoS of various methods in the multicore environment.

When we choose 4 applications from 5 without regard to order, allowing replications, the number of combination is $\binom{5+4-1}{4} = 70$. This time I simulate all the 70 workloads.

The name of workload is made up from the initials of selected applications. However, multiple candidates of names with different order are possible. For example, LLLH, LLHL, LHLL, and HLLL consist of the same combination of applications: himeno and three replicas of lu. I select one of the candidates as the name of workload with the following rules: the names of applications with more replicas precede; the applications with the same numbers of replicas are arranged in alphabetical order. For instance, the workload that consists of himeno, qsort, and two replicas of lu is named LLHQ.

The placement of applications in the chip is decided based on the name of the workload. I divide 32 cores by x-axis and y-axis into 4 groups: upper-left 8 cores, upper-right 8 cores, lower-left 8 cores, and lower-right 8 cores. I then assign the applications corresponding to the first, second, third, and last characters to the groups, respectively. Taking the example of LLHQ, I assign lu to the upper-left group, another lu to the upper-right, himeno to the lower-left, and qsort to the lower-right.

3.6 Evaluation Results for Existing Cache Architectures

Figure 3.12 shows the evaluation results for the multicore environment. The y-axis is the weighted mean of the arithmetic mean of relative IPCs. The x-axis is the weighted mean of losses in the relative IPC of the most slowed application. These are standards of performance and QoS, respectively. A method whose average relative IPC is higher is better. If the average relative IPCs are almost the same, a method whose maximum IPC loss is lower is better. In short, the more close to the upper right corner the point is, the better method is.

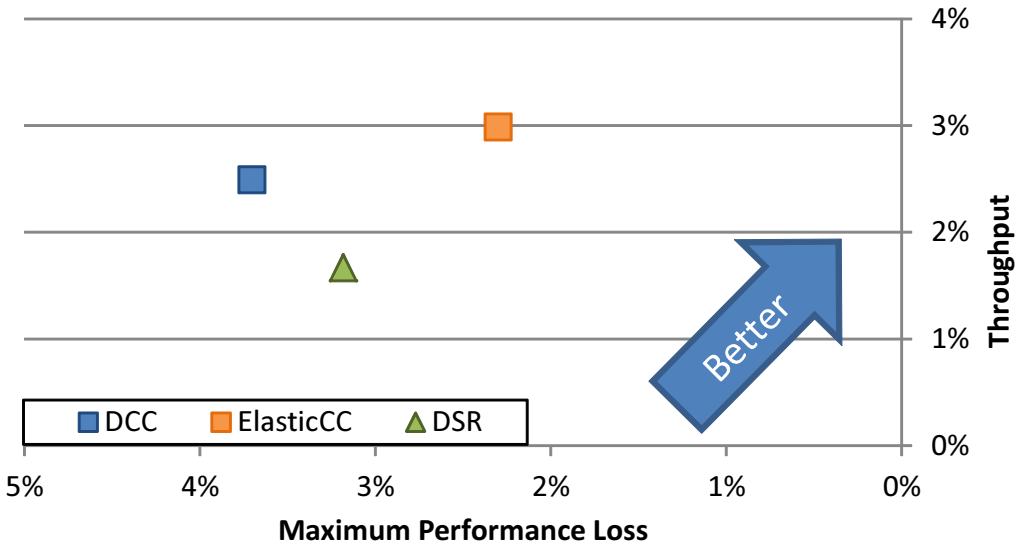


Figure 3.13 Performance and QoS of various methods in the many-core environment.

In this evaluation, Shared had better average relative IPC than the baseline (Private); however, its maximum IPC loss was as much as 10% because of some applications that were harmed seriously.

DCC performed better than Shared in both performance and QoS, though it did not limit spilling. The average relative IPC was 1.9% and the maximum IPC loss was 4.7%, on average. This result implies that the DCC alone achieves the advantage of both private and shared caches such as low latency, independence of performance, and flexibility on capacity to some extent. Nevertheless, the achievement is still limited.

ElasticCC performed worse than DCC in both despite its intention of high efficiency. The IPC improved by an average of 1.6% and the average of the maximum IPC loss was 6.7%. The degradation of the relative IPC may be due to the replacement algorithm. According to an advance evaluation with a single core, the average performance degradation with the LRU is about 0.5%. However, the maximum IPC loss became much worse. The possible reason is that ElasticCC requires virtual shared region of at least one way for each core or it expands the shared regions more than necessary.

DSR limits spilling by dividing the cores into spiller cores that only spill their evicted lines and receiver cores that only accept the lines from the spiller cores. This limitation worked well in the evaluation. The improvement of the IPC was 3.3% and the maximum IPC loss was 2.9%, on average, which were much better than DCC.

For DSR-QoS, a variation of DSR for QoS improvement, improved the QoS by the sacrifice of the performance. The IPC improvement was 2.9% and the maximum IPC loss was 2.5%, on average.

Figure 3.13 shows the evaluation results for the many-core environment. The way to read the graph

is the same as Figure 3.12: the more close to the upper right corner, the better.

Like the result in the multicore environment, DCC improved the performance but degrades the QoS. The improvement of throughput of DCC over Private was 2.5% and the maximum throughput loss was 3.7%, on average. ElasticCC this time outperformed DCC in both performance and QoS. The average throughput improvement was 3.0% and the average of the maximum throughput loss was 2.3%.

However, DSR had less performance than DCC despite the assumption of the ideal communication channel for sharing the information of misses, though its QoS was better than DCC. The throughput improvement was 1.7% and the maximum throughput loss was 3.2%, on average. In the many-core environment, each cache has just 512 sets and thus the number of sets available for sampling to each core is only 16 (8 for spilling, 8 for receiving). This reduces the precision of performance estimation. What is worse, since parallel applications often requires synchronization, if any of cores is given wrong prediction, its negative effect may spread to the whole application. Such circumstances that are specific to many-core work against DSR.

Through these evaluation results, I quantitatively confirmed that the existing methods for the optimized spilling do not meet both efficiency and scalability. DSR performed well in the multicore environment; however, it had poor performance in the many-core environment where scalability is important. Conversely, ElasticCC worked well in the many-core but did not in the multicore. I explain and confirm the efficiency and the scalability of my proposal in the following chapters.

3.7 Summary

In this chapter, I mentioned the two different simulation environments and proposed the simulation infrastructure for them to evaluate methods. The multicore environment corresponds to the modern multicore processors with powerful cores and a few LLCs of 2MB per core; the many-core environment corresponds to the future many-core processors with simple cores and many LLCs of 256KB per core. For each environment, I gave tens of workloads that have various sharing patterns to the LLCs.

I described a useful simulation infrastructure that centers SimMips in detail. The flow of simulating the multicore environment requires two kinds of traces. I obtain them with a modified version of SimMips. I developed a many-core processor simulator SimMccc, which utilize SimMips as processor cores and SimMc as network-on-chip, to evaluate the many-core environment. In addition to two communication layers of NoC and DMA that SimMc offers, it has two more layers of cache request and cache sharing. They improve the simplicity and reusability of the description. Besides, I referred to the design of coherence protocol in consideration of the features of the DCC, that is, spilling evicted cache lines and conflict of directory entries.

In the last of this chapter, I evaluated some existing cache architectures with the proposed infrastructure. The results showed that the existing spilling optimizations, DSR and ElasticCC, did not achieve

both efficiency and scalability.

Table 3.9 Transition of L2 caches.

Request	MCR_READ		MCR_WRITE						MCR_INV_DIR	
State	Read		Write		Replace(Read/Write)		Replace (PutP)		Invalidate (from Dir)	
M	DATA to L1		DATA to L1		if L1_valid: INV to L1 otherwise: PUTP to Dir	MP / SI	if L1_valid: INV to L1 otherwise: PUTM to Dir	MSI / SI	if L1_valid: INV to L1 otherwise: PUTM to Dir	MSI / SI
O	DATA to L1		GETM to Dir clear L1_valid	OM	(if L1_valid: INV to L1) PUTP to Dir	SI	(if L1_valid: INV to L1) PUTM to Dir	SI	(if L1_valid: INV to L1) PUTM to Dir	SI
E	DATA to L1		GETM to Dir clear L1_valid	SM	(if L1_valid: INV to L1) PUTP to Dir	SI	(if L1_valid: INV to L1) INV to Dir	SI	(if L1_valid: INV to L1) INV to Dir	SI
S	DATA to L1		GETM to Dir clear L1_valid	SM	(if L1_valid: INV to L1) PUTP to Dir	SI	(if L1_valid: INV to L1) INV to Dir	SI	(if L1_valid: INV to L1) INV to Dir	SI
I	GETS to Dir	IE	GETM to Dir	IM	-					
P	DATA to L1 HITP to Dir	E	GETM to Dir	SM	INV to Dir	SI	INV to Dir	SI	INV to Dir	SI
MO								MSI		MSI
MSI										
MI										
MP								MSI		MSI
OM								OMI	PUTM to Dir	
OMI									PUTM to Dir	
SM								SMI	INV to Dir	
SMI									INV to Dir	
SI										
IM								IMIr		IMIr
IE								ISIr		ISIr
IS								ISIr		ISIr
IMO								IMIr		IMIr
IMI								IMIr		IMIr
IMIr										
ISI								ISIr		ISIr
ISIr										
PE								PSI	INV to Dir	
PS								PSI	INV to Dir	
PSI									INV to Dir	

Table. 3.9 Transition of L2 caches (Continued).

Request	MCR_PUTM		MCR_INV_L1		MCR_INVALID		MCR_GETS		MCR_GETM	
State	PutM		Invalidate		Invalidate(other core)		GetS		GetM	
M	Fill clear L1_valid		clear L1_valid				if L1_valid: INV to L1 Add Reqer to RT otherwise: DATA to Reqer	MO / O	if L1_valid: INV to L1 Add Reqer to RT otherwise: DATA to Reqer	MI / I
O			clear L1_valid				DATA to Reqer	O	DATA to Reqer (if L1_valid: INV to L1)	SI/I
E			clear L1_valid				DATA to Reqer	S	DATA to Reqer (if L1_valid: INV to L1)	SI/I
S			clear L1_valid		ACK to Reqer if L1_valid: INV to L1	SI/I	DATA to Reqer		DATA to Reqer (if L1_valid: INV to L1)	SI/I
I					ACK to Reqer					
P							DATA to Reqer	S	DATA to Reqer (if L1_valid: INV to L1)	SI/I
MO	Fill clear L1_valid		-				Add Reqer to RT		Add Reqer to RT	MI
MSI	Fill		-				Add Reqer to RT		Add Reqer to RT	MI
MI	Fill		-				Add Reqer to RT		Add Reqer to RT	
MP	Fill		-				Add Reqer to RT	MSI	Add Reqer to RT	MI
OM			-				DATA to Reqer		If Reqer is self: Add AckCount otherwise: Add Reqer to RT	IM
OMI			-				DATA to Reqer		If Reqer is self: Add AckCount otherwise: Add Reqer to RT	IMIr
SM			-		ACK to Reqer※1	IM	DATA to Reqer		If Reqer is self: Add AckCount otherwise: Add Reqer to RT	IM
SMI			-		ACK to Reqer※1	IMIr	DATA to Reqer		If Reqer is self: Add AckCount otherwise: Add Reqer to RT	IMIr
SI			-		ACK to Reqer※1		DATA to Reqer		DATA to Reqer	
IM					ACK to Reqer※1		Add Reqer to RT	IMO	Add Reqer to RT	IMI
IE					ACK to Reqer	ISI	Add Reqer to RT	IS	Add Reqer to RT	ISI
IS					ACK to Reqer	ISI	Add Reqer to RT		Add Reqer to RT	ISI
IMO					ACK to Reqer※1	IMI	Add Reqer to RT		Add Reqer to RT	IMI
IMI					ACK to Reqer※1					
IMIr			-		ACK to Reqer※1		Add Reqer to RT		Add Reqer to RT	
ISI					ACK to Reqer※1		Add Reqer to RT			
ISIr					ACK to Reqer※1		Add Reqer to RT			
PE							if Reqer is self: DATA to L1 otherwise: DATA to Reqer	E / PS	DATA to Reqer	IE
PS					ACK to Reqer※1	IS	if Reqer is self: DATA to L1 otherwise: DATA to Reqer	S / PS	DATA to Reqer	IS
PSI					ACK to Reqer※1	ISIr	if Reqer is self: DATA to L1 INV to L1 INV to Dir otherwise: DATA to Reqer	SI / PSI	DATA to Reqer	ISI

※1: Dec AckCount if Reqer is self

Table. 3.9 Transition of L2 caches (Continued).

Request	MCR_DATA		MCR_ACK		MCR_LASTACK		MCR_PUTP	
State	Data		Ack(fromL1/Dir)		LastAck (if AckCount==0)		PutP	
M								
O								
E								
S								
I							Fill	P
P								
MO			Dec AckCount		DATA to Reqers	O		
MSI			Dec AckCount		PUTM to Dir DATA to Reqers	SI		
MI			Dec AckCount		DATA to Reqers	I		
MP			Dec AckCount		PUTP to Dir DATA to Reqers	SI		
OM	Fill Add AckCount		Dec AckCount			IM		
OMI	Fill Add AckCount		Dec AckCount			IMI		
SM	Fill Add AckCount		Dec AckCount			IM		
SMI	Fill Add AckCount		Dec AckCount			IMI		
SI			Dec AckCount			I		
IM	Fill Add AckCount		Dec AckCount		DATA to L1	M	Fill	SM
IE	Fill DATA to L1 DATA to Reqers	E/S ※2					Fill	PE
IS	Fill DATA to L1	S						
IMO	Fill Add AckCount		Dec AckCount		DATA to L1 INV to L1	MO		
IMI	Fill Add AckCount		Dec AckCount		DATA to L1 INV to L1	MI		
IMIr	Fill Add AckCount		Dec AckCount		DATA to L1 INV to L1	MSI		
ISI	Fill DATA to L1 INV to L1 DATA to Reqers	SI						
ISIr	Fill DATA to L1 INV to L1 DATA to Reqers INV to Dir	SI						
PE			Dec AckCount			IE		
PS			Dec AckCount			IS		
PSI			Dec AckCount			ISI		

※2: E if Sender is Main Memory

Table 3.10 Transition of directory caches.

Request / State	MCR_GETS GetS		MCR_GETM GetM		Replace		MCR_PUTM PutM ※1	
M	GETS to Owner Add Reqer to Sharer	O	GETM to Owner Set Reqer as Owner	M	INV_DIR to Owner	xI	ACK to Reqer PUTM to Memory Clear Owner Set Reqer as LastHolder	I

O/Os	GETS to Owner
	Add Reqer to Sharer
		O	GETM to Owner
			INVALID to Sharer(s)
			Clear Sharer
			Set Reqer as Owner
				M	INV_DIR to Owner
					INV_DIR to Sharer(s)
						xI	ACK to Reqer
							PUTM to Memory
							Clear Owner
							if no Sharers:
							Set Reqer as LastHolder
							if 1 Sharer:
								I
								/
								Es
								/
								S

E/Es	GETS to Sharer Add Reqer to Sharer	S	GETM to Sharer Clear Sharer Set Reqer as Owner	M	INV_DIR to Sharer	xI		
S	GETS to one of Sharers Add Reqer to Sharer	S	GETM to one of Sharers INVALID to other Sharers Clear Sharer Set Reqer as Owner	M	INV_DIR to Sharers	xI		
I	GETS to Memory Add Reqer to Sharer	E	GETM to Memory Set Reqer as Owner	M				
P	GETS to Sharer Clear Spiller if Reqer != Sharer: Add Reqer to Sharer	S / E	GETM to Sharer Clear Sharer and Spiller Set Reqer as Owner	M	INV_DIR to Sharer Clear Spiller	xI		
xI							ACK to Reqer PUTM to Memory Clear Owner	xI/I

Request / State	MCR_INVALID Invalidate		MCR_PUTP PutP ※3		MCR_HITP Hit on Spilled Line	
M	ACK to Reqer※2		ACK to Reqer PUTP to Destination PUTM to Memory Set Reqer as Spiller Set Destination as Sharer	P	-	
O/Os	ACK to Reqer Erase Reqer from Sharer(s) If no Sharers:	Os / O	ACK to Reqer PUTP to Destination PUTM to Memory Set Reqer as Spiller Set Destination as Sharer	P	-	
E/Es	ACK to Reqer Clear Sharer Set Reqer as LastHolder	I	ACK to Reqer PUTP to Destination Set Reqer as Spiller Set Destination as Sharer	P	-	
S	ACK to Reqer Erase Reqer from Sharer(s) If 1 Sharer:	Es / S			-	
I						
P	ACK to Reqer Set Spiller as LastHolder Clear Sharer and Spiller	I			Clear Spiller	Es
xI	ACK to Reqer Erase Reqer from Sharer(s) If no Sharers:	xI / I	(spilling will not be allowed)		-	

※1 if Owner changed, ACK only

※2 Reqer should be former sharer. ※3 If spilling is not allowed,
treat it as PUTM (if Reqer == Owner), or INVALID (otherwise)

Chapter 4

Proposal of ASCEND Architecture for Cooperative Caching

In this chapter, I propose the concept and the architecture of **ASCEND (Adaptive Spill Control with extra ENtries of Directory)** as a highly-efficient and scalable method for cooperative caches.

I focus on the directory caches in the Distributed Cooperative Caching (DCC). As I am going to mention, they inevitably have a number of extra entries in the directories for avoiding the performance degradation. If they keep the information of the entries instead of discarding it, they can store the tags of lines that were recently removed from the chip. ASCEND makes use of them to estimate the demand on capacity of each core so that the cores can spill evicted lines properly.

ASCEND integrates two units, called Spiller Selector and Receiver Selector, to each directory cache. They detect the difference in demands on capacity of the cores through the two kinds of special references to the directories, or **I-Reference** and **Extruded I-Reference**, and then manage spilling evicted lines and receiving the spilled lines. To maximize the efficiency of the spilling mechanism of cooperative caches and be applicable to many-core processors, ASCEND is designed for both high efficiency and scalability. In addition, the selectors are designed to be small hardware resource enough for the validity of the architecture.

I describe two different methods of ASCEND architecture. One is an original method that was proposed in [78]; the other is a highly-precise method for better efficiency. I also propose a method named Weak Receiving as an option of ASCEND for achieving better efficiency. I explain its feature, the validity of the idea, and the incorporation into ASCEND later in this chapter.

In Section 4.1, I first verify the adequacy of giving a larger number of directory entries than the number of cache lines. I have two kinds of discussions about estimating the necessary number of entries: probabilistic analysis and simulation-based preliminary evaluation. I decide the appropriate number of entries from these results.

In Section 4.2, I mention treatment of invalidated entries and reference to them, which I call I-Reference and Extruded I-Reference. I take an example of how these two kinds of I-Reference improve

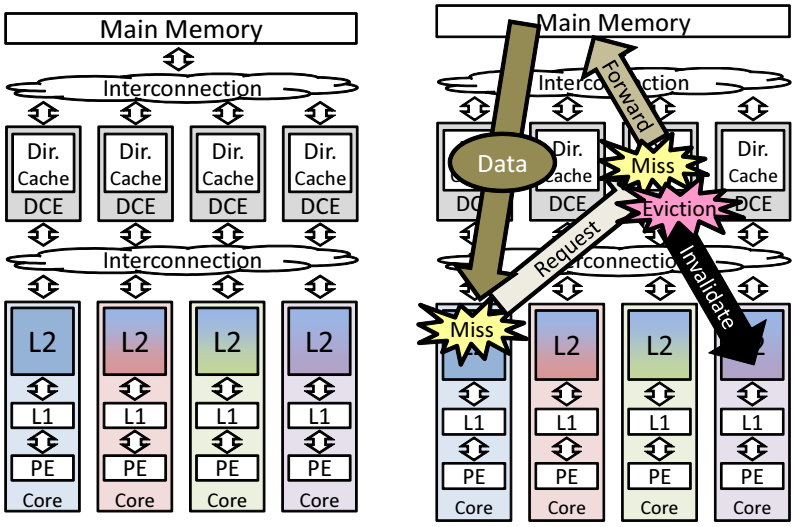


Figure 4.1 The review of the organization and the undesirable working example of the DCC.

the efficiency of caches later in the section.

In Section 4.3, I propose ASCEND and describe the architectural organization applying ASCEND. I then explain two different methods of ASCEND architecture, the original method [78] and the highly-precise method, in Section 4.4 and Section 4.5, respectively. Each explanation includes the policies of utilizing the special references, and the architectural models of Spiller and Receiver Selectors. I also explain the Weak Receiving, an option for the highly-precise method of ASCEND, later in Section 4.5.

I qualitatively compare ASCEND with other methods for optimizing spilling in Section 4.6.

4.1 Necessity of Extra Entries of Directory

4.1.1 Avoiding Replacement in directories

Although I explained the target architecture or the DCC in Chapter 2, first I review its organization and behavior that I focus on in this chapter.

The left of Figure 4.1 shows the organization of the DCC. The core consists of a PE (Processing Element), L1 caches, and an L2 cache. The DCEs (Distributed Coherence Engines) have directory caches. A valid directory entry is mapped to a cache line or lines being kept in the chip. These two kinds of units and the main memory are connected via some interconnects. Note that the number of cores and that of DCEs sometimes differ.

The right of the figure shows the working example that has shown in Figure 2.9 (e). When a core misses in its own L2, the core sends a request to one of the DCEs determined by the computation with the line address. The requested DCE looks for either a valid entry corresponding to the line or an

Table 4.1 The contrast between access frequency of caches and that of directories.

Cache Access Frequency	Type of Cache Accesses	Directory Access Frequency
Low	Any	Low
Moderate	Any	High
High	Shared Read/Write	High
High	Not Shared or Read only	**Low**

empty entry for making new directory to the requested line; regrettably, neither of them is found. In this undesirable case, one of the entries is selected for a victim and is invalidated. It leads to invalidation or written-back of all the sharers of the corresponding line, and may harm the cache performance.

If the invalidated lines are not frequently used, the effect of such undesired invalidation is small. Replacement algorithms for caches actually take advantage of this. However, the access frequency to directories and that to caches are not always similar. Therefore, replacement algorithms for directories can give wrong estimation about the access frequency of caches, that is, the entries corresponding to lines that are frequently accessed are also discarded.

I explain this problem by dividing the cores by frequency and type of its cache. Table 4.1 shows the relations between the frequency or type of cache access and the frequency of directory access. I only regard requests for read or write as *access* in the table for ease of explanation— requests for invalidation, write-back, spilling, etc. are not considered.

First of all, if a region is rarely accessed by the caches, cores make requests for it to the caches and the directory only few times before its removal. Therefore, the access frequency to the directory is low.

Secondly, if it has intermediate frequency of cache access, a part of cache access is filtered as local cache hit and thus some requests to the directory are omitted. However, it is sometimes evicted from its local cache and spilled to a remote cache (unless it is currently shared). When it is requested next, it hits in the remote cache. Such a hit produces a reuse of the directory entry. Hence, the frequency of directory access is high.

Thirdly, if it has high frequency of cache access and it is shared and written by cores, most requests to the directory are filtered out by the local caches. However, when one of the cores wants to write to it, the replicas in the other caches are invalidated or updated in order to keep coherency. This results in a recall of the directory entry. Therefore, this case also has high frequency of directory access.

Lastly, if it is repeatedly accessed by only one core or it is frequently read but not written from cores, a problem occurs. In this case, most requests to the directory are omitted like the third case. What is worse, directory access along with invalidation or update does not happen. As a result, the access frequency to the directory remains low and the directory cannot distinguish this case from the first case.

This indistinctiveness may cause replacement of directory entries corresponding to the repeatedly

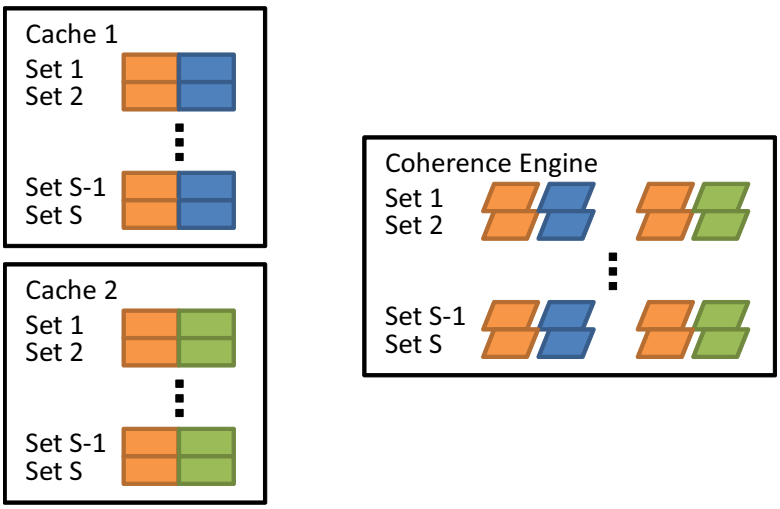


Figure 4.2 Cache lines and replicas of their tags in the centralized Coherence Engine.

accessed lines, that is, invalidation of frequently reused data. In consequence, the existence of the last case causes performance reduction of the DCC.

In the DCC, the numbers of entries and ways in the directories are independent from the number of lines and ways in the caches. Using this feature, we can apply one of the following two ways in order to avoid the competition in the directories that is connected to the undesirable invalidation.

The ideal way is to equalize the numbers of entries and lines, and to set the associativity of directories to the product of the associativity of caches and the number of cores. It means that the directories can store the replicas of all the tags in the caches, and thus it essentially eliminates the replacement in the directories. It is similar to the approach used in the Cooperative Caching [5]. Nevertheless, the huge associativity proportional to the number of cores lacks the feasibility.

As shown in Figure 4.2, the Cooperative Caching has the centralized Coherence Engine (CE) that stores the replicas of the tags. Though the number of sets in the CE is the same as that in each cache, the number of ways, or the number of blocks that the CE must access at once on finding lines, is proportional to the number of cores and thus it can be impractically large in many-core processors.

The practical way is to set the number of directory entries, the number of sets or the number of ways, or both, bigger than that of cache lines. The original DCC applies this: it sets the number of entries twice as many as the number of lines [6].

An important point with the latter way is to choose the proper number of directory entries. Since the reduction of performance loss is limited, we cannot benefit from too many entries. Furthermore, massive directories may make the performance worse due to long latency. Hence, it requires an appropriate tradeoff between the performance and the hardware cost.

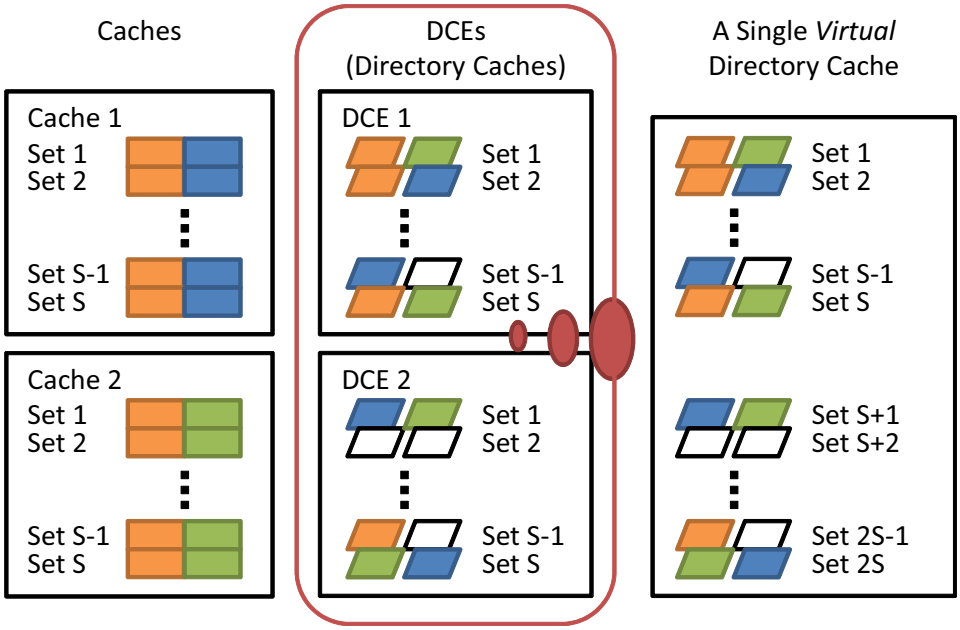


Figure 4.3 Cache lines in cores and directory entries in the DCEs. The whole DCEs are regarded as a single virtual directory cache.

4.1.2 Probabilistic Analysis

In this section, I estimate the sufficient number of directory entries using a probabilistic analysis.

Before the analysis, I mention the mapping from the directories to the caches. Figure 4.3 shows the placement of cache lines and directory entries when the numbers of lines and entries are equivalent. In the figure, I assume that both the number of the caches and the directories are 2 and they have S 2-way sets. Colored rectangles and parallelograms stand for valid cache lines and directory entries, respectively. Uncolored parallelograms mean invalid directory entries. Lines kept by only Cache 1 and their corresponding entries are shown in blue; private lines of Cache 2 and their equivalent entries are green; and lines and entries shared by both cores are colored orange. A valid directory entry is mapped to one or more identical (with the same address) cache line(s) in the chip.

It is notable here that the whole DCEs (directories) are considered as a single virtual directory caches. Each DCE manages the coherence of an address-interleaved part of the memory space. The same can be said for each set in DCEs. In consequence, when we assume a single large directory cache, each DCE is a part of it divided by a group of sets. Conversely, the whole DCEs are regarded as a large directory that has the same number of ways as each DCE and the number of sets equivalent to the product of the number of sets in each DCE and the number of DCEs.

From this virtual view, the number of sets in the virtual directory differs from that in each cache.

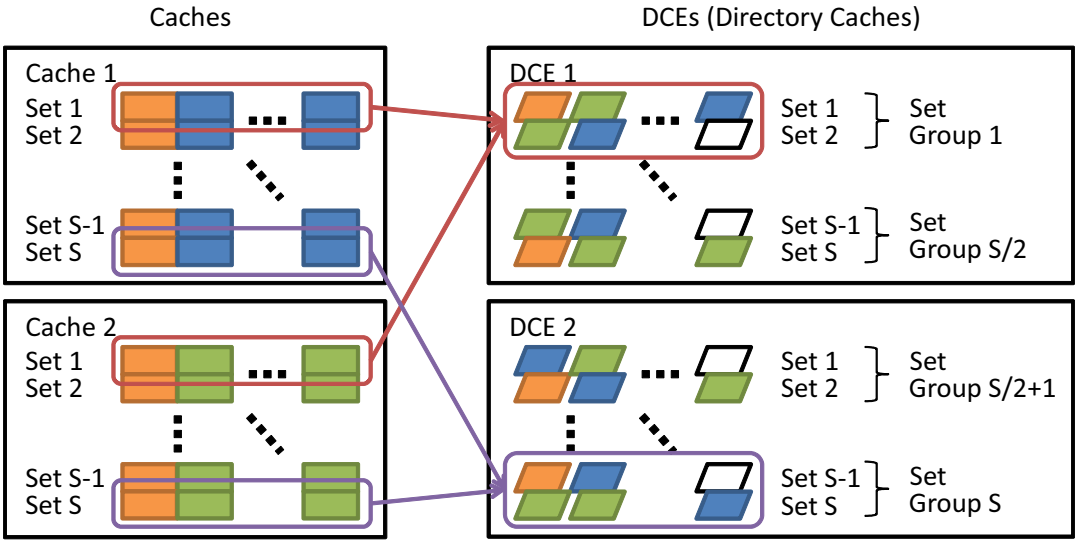


Figure 4.4 The first assumption of set mapping. There is one-to-one mapping between sets of the caches and set groups of the directory caches.

Therefore, even though a pair of lines are stored in the same sets of the caches, their corresponding entries can be located in different sets of the directories. Note that equalizing the numbers of sets in each cache and that in the virtual directory means that the number of ways in the directory becomes the product of the number of ways in each core and the number of cores; in other words, it just means the ideal way to eliminate the competition (see Figure 4.2).

This different mapping due to the difference in the numbers of sets is the fundamental problem. As a result, the numbers of valid directory entries in sets becomes unequal. Even if entries are assigned randomly enough (with a hash function as I have mentioned in Section 2.1.3, for example), we cannot remove this inequality. This is why the practical way to reduce the competition requires some extra directory entries.

I then estimate the number of directory entries sufficient to remove the most of competition. I make two different assumptions about the assignment of set in the directories. Under each assumption, I calculate the probability that a set in the directories can store all the entries that it should keep.

In the following discussions, N and S stand for the number of cores and that of sets in each cache, respectively. I assume that the total numbers of sets in caches and directories are the same, and so the ratio between the numbers of ways in caches and directories is the same as that between the number of lines and entries.

The first assumption is that if and only if a pair of cache lines belong to the same set in the caches, the corresponding entries belong to the same group of sets in the directories. In other words, I assume one-to-one mapping between a set in the caches and a set group in the directories. From the assumption

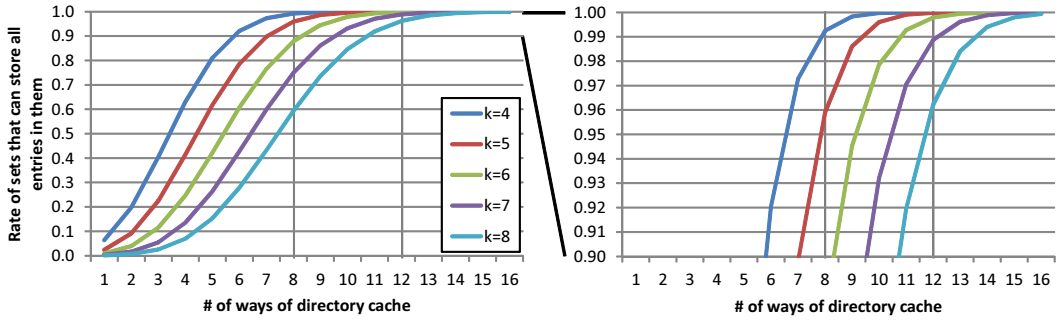


Figure 4.5 Directory cache size vs. probability of having sufficient entries (in the first assumption, N=4).

of the total number of sets, the number of sets in a set group is the same as the number of cores, or N.

Figure 4.4 depicts the relationship between a set in the caches and a set group in the directories. Two sets in the directories form a set group. The set groups are numbered serially from 1 to S. A directory entry corresponding to a cache line in Set k will be stored in one of the sets (specified by its address) in the Set Group k.

I calculate the probability that the number of valid directory entries in a set do not exceed W or the number of ways. Let kN be the total number of valid entries in a set group. Although the coefficient k varies according to the number of valid lines and the sharing degree in the caches, the maximum k is equivalent to W (when all the lines are valid and not shared). Since the probability that an entry belong to the specific set in the set group is $1/N$, the distribution of the number of entries that a set should keep follows a binominal distribution $B(kN, 1/N)$. Therefore, the probability that the number is not more than W is the cumulative distribution function of it, calculated as follows:

$$\sum_{w=0}^{W} \binom{kN}{w} \left(\frac{1}{N}\right)^{w} \left(1 - \frac{1}{N}\right)^{kN-w}$$

Figure 4.5 shows the relations between W (X-axis) and the probability (Y-axis) with some k. The number of cores, N, is set to 4 in the figure. The right graph is a partial enlargement of the left graph.

When there are a lot of invalid or shared lines, k becomes small. On $k = 4$, or there are half as many valid entries as total lines, even though the numbers of lines and entries are the same, over 99% of sets do not occur the competition. However, on $k = 8$ or the maximum k, the probability is dropped to about 60%. When W is increased to 12, that is, the number of directory entries is 1.5 times more than that of cache lines, about 96% of sets have the sufficient entries at last.

The second assumption eliminates the limitation of the first assumption. In other words, the assignments of sets in the caches and the directories are completely different. Figure 4.6 depicts the mapping between lines and entries in this assumption. The concept of set groups in the first assumption does not exist anymore: the indexes of the sets in the caches and the directories are totally unrelated.

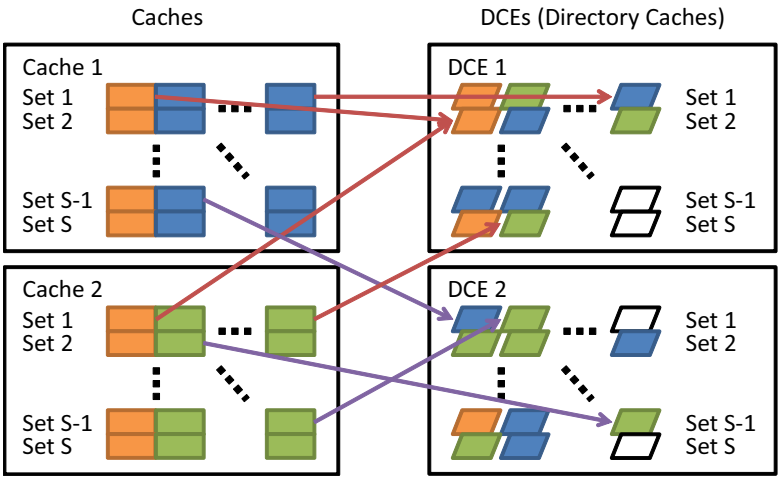


Figure 4.6 The second assumption of set mapping. The set indexes in the caches and the directory caches are completely different.

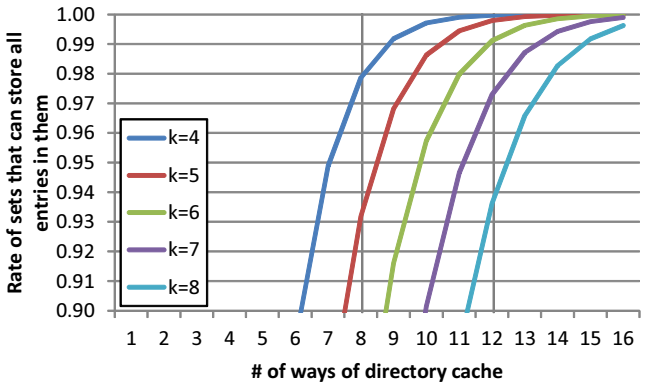
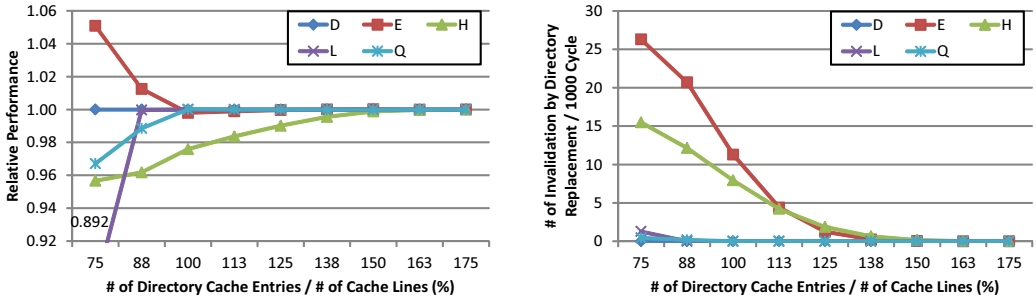


Figure 4.7 Directory cache size vs. probability of having sufficient entries (in the second assumption).

Like the first assumption, the total number of sets in the directories is set to SN, the product of the number of sets in each cache and the number of cores. Let kSN be the total number of valid entries in the directories. Since the probability that an entry belong to the specific set is $1/SN$, the distribution of the number of entries that a set should keep follows a binominal distribution $B(kSN, 1/SN)$. As SN is regarded as large enough, the distribution can be approximate by a Poisson distribution $Po(k)$. Therefore, the probability that the number is not more than W is calculated as follows:

$$\sum_{w=0}^{W} \frac{k^w e^{-k}}{w!}$$

Figure 4.7 shows the relations between W and the probability with some k in the second assumption. Note that this time I only show a partial enlargement. In comparison with Figure 4.5, every graph slightly shifts to the right, in other words, the number of sets with conflict in the directories is increased



(a) (b)

Figure 4.8 Directory cache size vs. performance and invalidation frequency.

a bit. This is caused by the increase of inequity. With 8-way caches and 12-way directories, the rate of sets with the sufficient number of entries is about 94% (recall that this was about 96% in the first assumption).

The probabilities are slightly improved when I assume the larger number of ways. For 16-way caches, 22-way directories (this means that the number of entries is 37.5% larger than that of lines) have almost the same probabilities as I have shown above.

Of course, how many lines are used and shared, and how many times of invalidation the competing sets can occur depend on the behavior of applications. Nevertheless, the probabilistic analysis I have shown in this section implies that the directories require 1.5 times as many entries as the lines.

4.1.3 Preliminary Experiment

As another way to estimate the sufficient number of directory entries, I make a preliminary evaluation with some applications. I evaluate the varied numbers of entries in the many-core environment that I have mentioned in Chapter 3. Since I run only one application at a time, the number of valid core is 8. I measure the number of invalidation and the performance (inverse of the execution time).

The results of the preliminary evaluation are shown in Figure 4.8. The x-axis is the ratio of the number of ways in the directories to that in the caches. The y-axis of graph (a) is the performance relative to having 175% of ways; the y-axis of graph (b) is the number of invalidation due to the replacement of directory entries per 1000 cycles. Each line stands for an individual application (See Section 5.4.2). As shown in graph (b), having 175% of ways almost eliminates the undesired invalidation. Therefore, the performance baseline in graph (a) is almost the same as having the infinite number of directories.

Sections of 75% and 88%, where the number of entries is smaller than that of lines, have some invalidation in all the applications, though the degrees are varied with application. The performance in some applications is seriously decreased. Though in sections from 100% to 138%, where the directories have the same or a bit more entries than the number of cache lines, the imbalance of sets causes the

invalidation in applications E and H. The performance is still decreased in application H. When having 150% of ways, the performance loss in application H becomes less than 0.2% at last. This result implies, just as the probabilistic analysis, that the directories require 150% as many entries as the lines.

4.2 New Concept of Utilizing Extra Directory Entries

4.2.1 Dealing with Invalidated Entries

The both results of estimations are that 1.5 times as many entries are required and sufficient to avoid the performance loss. I now consider how much additional hardware the extra entries require.

The size of a directory entry is much smaller than that of a cache line. Though it depends on the width of the physical address, the number of cores, etc., the typical size of a directory entry is from 50 to 100 bits. On the other hand, a cache line uses 512 or 1,024 bits just for the data. Therefore, a directory entry only consume less than a fifth or tenth of a cache lines. Although the directories have 1.5 times as many entries as the lines, the hardware amount of the extra entries is 10% at a maximum, or typically less than 5% of that of the caches; It is acceptable in most cases.

This is an explanation of why it is a reasonable choice that the number of entries is set to 1.5 times as many as that of lines.

However, if we make this choice, at least a third of entries (or half as many entries as the cache lines) remains unused. My approach to control the spilling, named **ASCEND (Adaptive Spill Control with extra ENtries of Directory)**, uses this unused region efficiently as a basic strategy.

When a cache line is invalidated and evicted from the chip, the corresponding directory entry is also invalidated. In general, the information in such an entry is just removed. In ASCEND, although the state of the entry becomes invalid as usual, its tag is kept along with the situation when it was invalidated. As a result, the tags of invalidated entries are regarded as an assembly of the tags corresponding to recently evicted cache lines.

If the directories detect and count hits in the invalidated entries, they can predict the benefit of spilling evicted lines and detect the negative impact of accepting the spilled lines. From this information, they can grant rights of spilling to cores that are considered to benefit from spilling their lines, and prohibit cores that are considered to be harmed by the spilled lines from receiving them. I think that they should improve the efficiency of caches and the performance of the processor. This is the basic policy of ASCEND.

The idea of utilizing commonly unused hardware is similar to the Cached Load/Store Queue [79]. Like the directory in the DCC, while a load/store queue sometimes requires many entries, most of them are usually invalidated. The Cached Load/Store Queue keeps the information in entries of the load/store queue invalidated due to the completion of data access. When the core requires the data stored in the

invalidated entries, not the cache but the load/store queue supplies the data to reduce the number of cache access. This method uses the kept information directly; however, ASCEND uses the invalidated tags indirectly — for the performance estimation on changing policies on spilling.

I now think back about the WAYPOINT [61], which I have mentioned in Chapter 2. The WAYPOINT avoids the undesired invalidation not by preparing extra directory entries but by borrowing the insufficient capacity from the main caches. This method also essentially eliminates the invalidation. Moreover, it enables the directories to have even the smaller number of entries than cache lines without undesired invalidation.

However, this method has capacity overhead because the insufficient entries are stored in the caches with some metadata such as pointers. Of course, it also has latency overhead because of the additional searches for the insufficient entries. Therefore, whether we adapt the architecture of the WAYPOINT or not, it is a consistent fact that we must apply the proper number of directory entries in consideration of overhead.

In addition, as I have mentioned, how many entries the directories require, or how many entries remain unused, relies on applications. Hence, even though the number of directory entries is reduced by the WAYPOINT, the benefit of ASCEND is unlikely to be eliminated. Rather, aggressively mixing both of them may become the best balance between the efficiency and the overhead.

I then explain how utilizing the extra entries change the treatment of the directories. When a directory with ASCEND receive a request of read or write, it searches for the corresponding entry not only from valid entries but also from invalidated entries in the set. If it is found, regardless of its state, it is selected. If not, it then looks for invalid entries and selects one of them in some way (e.g. first-in first-out). If neither the corresponding entry nor an invalidated entry is found, the undesired replacement of a directory entry occurs.

What ASCEND focuses on after one of the entries is selected is a situation that the corresponding entry is found but it is invalidated. In this thesis, I define such hits in the invalidated entries as **I-References** (after *invalid references*) of the requesting core. Cores that get the I-References frequently are likely to reuse the data removed from the chip right after their eviction. Hence, the frequency of the I-References can be used for the selection of cores that should be allowed spilling.

ASCEND also detects cores that will be harmed by spilled lines. If it does not consider them, the uneven improvement that some applications are boosted but other applications become heavily slow may occur. This behavior is usually regarded as unwanted.

To estimate the negative impact, I focus on evicted lines from the chip due directly to receiving spilled lines. Even though they are *movable* and they would be spilled to other cores on the regular evictions, they are prohibited from spilling in order to avoid the chain reaction. Therefore, it implies the negative impact on the performance that they are reused right after their evictions.

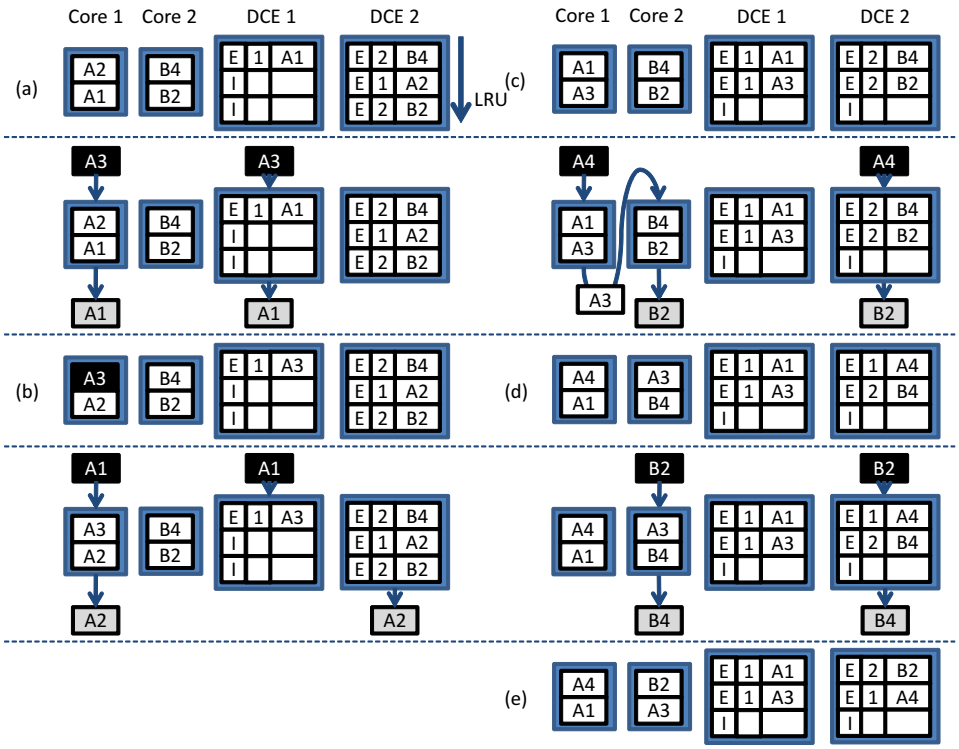


Figure 4.9 Transition of tags when invalidated tags are removed.

In ASCEND, if an evicted line due directly to accepting a spilled line is *movable* and is removed from the chip (i.e. no other cores have its replica), the directory keeps the evicting core on its invalidation. Afterwards, if such an entry is referenced again, it is detected as an **Extruded I-Reference** of the evicting core, in addition to an I-Reference of the requesting core. The number of the Extruded I-References can be used for the selection of cores that should not accept spilled lines.

4.2.2 Example of Two Kinds of I-Reference

Figure 4.9 and Figure 4.10 show an example of keeping the tag of invalidated entries and detecting the I-References and the Extruded I-References. For ease of explanation, each core has two lines and each directory has three entries. Each rows in the tables in the gray rectangles stands for an individual line or entry. Elements are arranged so that a recently used one comes to upper. A tag is expressed as a combination of an alphabet and an Arabic figure. Lines with an odd number in the tag are mapped to DCE 1; Lines with an even number are managed by DCE2. The leftmost field of the entry means the state of the directory: E (Exclusive) is the state that only one core has the corresponding line; I (Invalid) is the state that the entry is not in use.

The center field stands for the sharers of the corresponding line. Although it is normally expressed

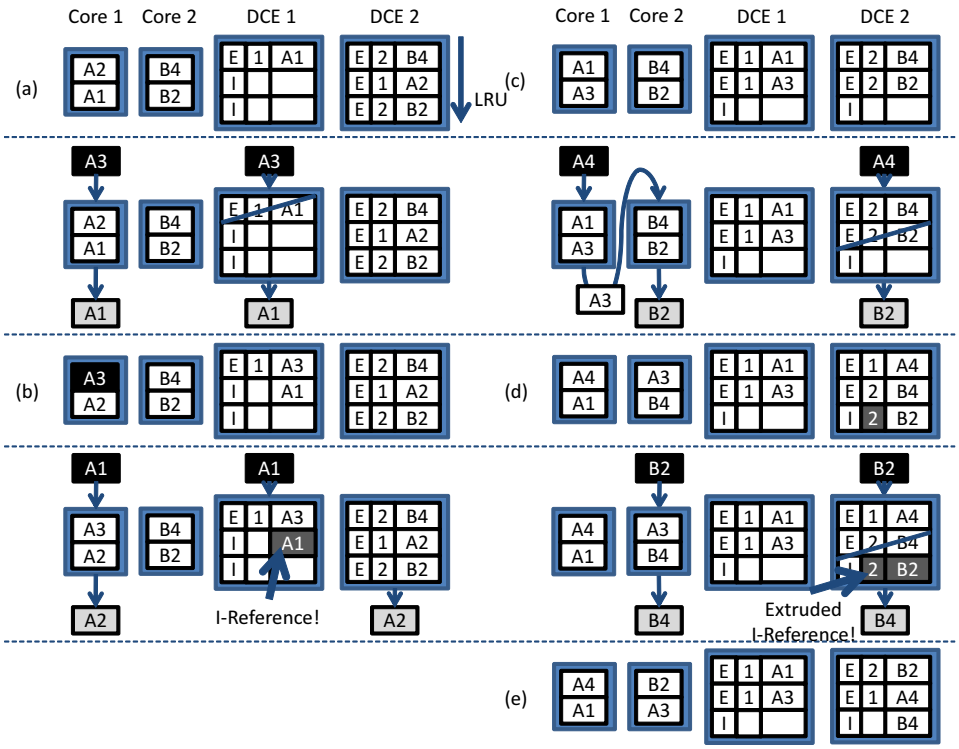


Figure 4.10 Transition of tags when invalidated tags are preserved.

as a bit vector, it is shown as the core ID, because the number of cores sharing a line is up to one in this example. The requesting line is represented by white letters on a black background. Some other fields that are unnecessary for the explanation, such as the state of the cache line, are omitted from the figure.

Let (a) the initial situation. Core 1 has A1 and A2; Core 2 has B2 and B4. A1 is managed by DCE 1 and A2, B2, and B4 are controlled by DCE 2. Both of cores are prohibited from spilling to the other core at this time.

Figure 4.9 depicts the behavior of the conventional DCC. It discards the invalidated tags in the DCEs. Assume that Core 1 requests A3 after the situation (a). A3 is read from the main memory and A1 is evicted from Core 1 and then removed from the chip. It results in the situation (b). When Core 1 then demands A1, A1 is inserted and A2 is invalidated, thereby the situation becomes (c). In consequence, when Core 1 requests A4 and somehow it is allowed spilling, A4 is inserted, A3 is evicted from Core 1 and spilled to Core 2, and then B2 is evicted from the chip. As a result, the present situation is as shown as (d) Lastly, when Core 2 requires B2, B2 is read from the main memory and B4 is removed. The final situation is shown as (e).

Meanwhile, the behavior of the DCC with ASCEND is shown in Figure 4.10. A1 is evicted from the chip between the situations (a) and (b). However, the invalidated tag now remains in DCE 1, that is, DCE 1 has the information about the removal of A1. When Core 1 then sends a request of A1 to DCE

1, the invalidated entry is selected there. This is an I-Reference. ASCEND detects it as an I-Reference of Core 1. If Core 1 frequently gets such I-References, it will be allowed spilling to the other core.

Afterwards, Core 2 accepts A3, a spilled line from Core 1 between the situations (c) and (d). B2 is then evicted and removed from the chip. Note that it is invalidated due directly to receiving a spilled line and there are no replicas in the chip. In ASCEND, the entry corresponding to such a line is marked and the ID of the evicting core is kept in the center field. When Core 2 then demands B2 from DCE 2, the invalidated and marked entry is selected there. This is an Extruded I-Reference. ASCEND detects it as an Extruded I-Reference, in addition to an I-Reference, of Core 2. If Core 2 has the large number of such Extruded I-References, it will be prohibited from receiving spilled lines.

In this example, the evicting core and the requesting core afterward is the same. If they are different, they are detected separately as an I-Reference of the requesting core and an Extruded I-Reference of the evicting core.

It is seen when comparing with Figure 4.9 that ASCEND only modifies the invalidated part of the directories and has no effects on the valid entries.

4.3 Architectural Organization of ASCEND

Figure 4.11 shows the architectural organization of ASCEND. It is based on the Distributed Cooperative Caching (DCC), a scalable variation of cooperative caches that I have explained in Chapter 2. The chip consists of cores, DCEs (Distributed Coherence Engines) or directories, a (controller of) main memory, and some interconnections among them. Each core has multiple levels of caches and its last level cache (L2 in the figure) can forward its evicted lines to the last level caches in the other core. It assumes the most likely model of many-core processors as an extension of the current trends in processor. Therefore, although it does not require specific interconnections, they are needed to be fast enough in order not to be performance bottleneck.

The proposal only modifies the directories: in addition to the array of directory cache and the coherence controller, it includes a Spiller Selector and a Receiver Controller in each of the directories as its controllers. The Spiller Selectors grant rights of spilling to cores predicted to benefit more from remote caches than the others, using the number of the I-References of each core. The Receiver Selectors restrict or prohibit the acceptance of spilled lines by cores predicted to be damaged by the reduction of local caches with the numbers of the Extruded I-References. Both of them are designed to be realized with addition of small hardware in order that the additional hardware cost may not negatively affect the validity of the proposal. I will evaluate their hardware amount quantitatively in Chapter 5.

I propose two different methods of ASCEND architecture in Section 4.4 and Section 4.5. While they differ in the detailed organizations of the Spiller Selectors and the Receiver Selectors, the concept and the basic organization are common between them.

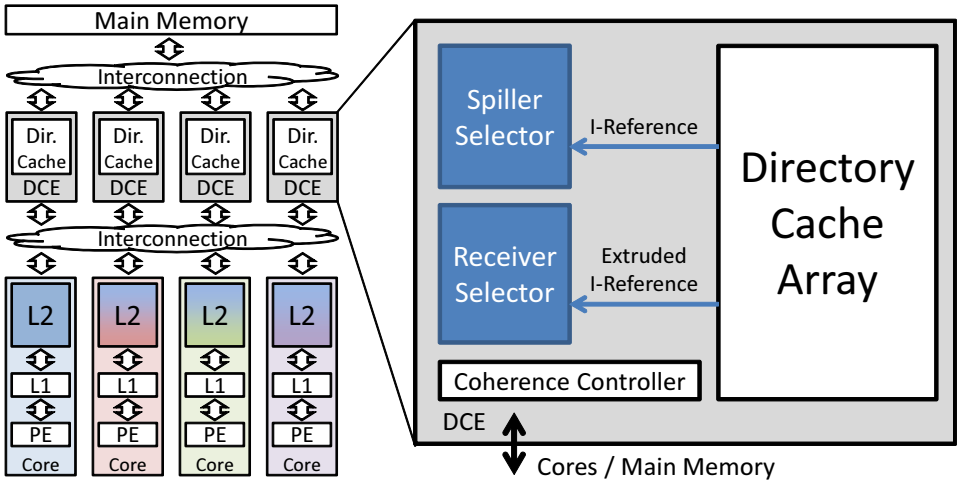


Figure 4.11 Organization of the DCC with ASCEND.

Note that the units in one directory work independently from those in the other directories. They only count the number of I-References and Extruded I-References detected in its own directory, and then generate the individual results from them. They do not either sum up the values of counters with other units or synchronize each other. Although this design reduces the precision of estimation due to the decline in the number of samples, it is important in terms of scalability.

However, when we assume too many directories, the decrease in the precision may be a problem. In this case, we have to increase the number of samples by some ways such as partial synchronization among a group of directories. I do not mention the detail of such methods.

In the DCC, if an evicted line is *movable*, the core determines the destination, forwards the line to it via cache-to-cache communication, and sends a notification of spilling to the directory. If not, the core sends a notification of invalidation or a request of write-back.

In ASCEND, unlike the DCC, the destination of spilled lines are determined by the Receiver Selectors in the directories. This changes a part of series of communications as follows. When a core evicts a line that may be *movable*, it sends a request of spilling along with the data of the line to the corresponding directory (just like a request of write-back in the DCC). The controllers of ASCEND decide whether the line is to be spilled or not. If the line is to be spilled, the directory forwards the line to the destination specified by the Receiver Selector. If not, the line is just invalidated or written back.

This modification replaces a notification of invalidating a line that is not *movable* with a request of spilling. In short, some messages without data become those with data. As a result, such communications have a bit longer latencies than the original. However, since they are not latency-critical, this has little effect on the performance.

In addition, coherence controllers in the directories are also modified in order to change the treatment

of invalidated lines for the detection of I-References and Extruded I-References. Although it makes the controllers slightly complex, it does not require additional capacity in the controllers and the directory arrays.

4.4 Original Method

4.4.1 Spiller Selector

ASCEND uses the number of I-References of each core to decide which cores should spill their lines. Cores with the large number of I-References are likely to use invalidated data right after they are removed, and thus suitable for allowing their spilling. In contrast, cores with the small number of I-References are considered as running applications whose working set is less than or quite more than the local cache size. Even if the directories allow them spilling their lines, they will not only fail to improve their performance, but they will have a negative impact on the other cores. Therefore, the directories should prohibit them from spilling.

Based on the observation above, I formulate the permission of spilling. The Spiller Selectors count the number of I-References occurred in their own directory. They test the counters in a period of fixed cycles. If the number is more than or equal to a certain threshold at this time, the corresponding core is permitted to spill for the next period.

I define the threshold as $(\sum_{i=1}^{N} I_i)/N \times k + 1$, where the numbers of I-References of Core $1, 2, ..., N$ are $I_1, I_2, ..., I_N$, respectively, and k is a constant. In short, it is the average number of I-References times k plus 1. k is set in order that N/k is an integer. From the threshold, the condition that Core a is permitted to spill represents $I_a \geq (\sum_{i=1}^{N} I_i)/N \times k + 1$. By transforming this, we get the following inequality:

$$(N/k - 1) \times I_a - \left((\sum_{i=1}^{N} I_i) - I_a\right) \geq N/k \tag{4.1}$$

Substituting $X = N/k - 1$ and $I_o = (\sum_{i=1}^{N} I_i) - I_a$ into Inequality (4.1), we obtain $X \times I_a - I_o \geq X + 1$. Since both sides are integers, it can be transformed into the following:

$$X \times I_a - I_o > X \tag{4.2}$$

Inequality (4.2) is also the condition of spilling of Core a. From Inequality (4.2), the decision can be done with I_a of the number of I-References in Core a, I_o of that in the other cores, and a constant X.

For instance, assume that we substitute $N = 8$ and $k = 4/5$, and that the total number of I-References is 1,000. In this case, the threshold is $1000/8 \times (4/5) + 1 = 101$. Since $X = N/k - 1 = 9$, Inequity (4.2) or the condition that Core a is allowed spilling becomes $9I_a - I_o > 9$. When we assign the threshold of 101 to I_a, for the left member becomes $9I_a - I_o = 9 \times 101 - (1000 - 101) = 10$, we can confirm the inequality is satisfied.

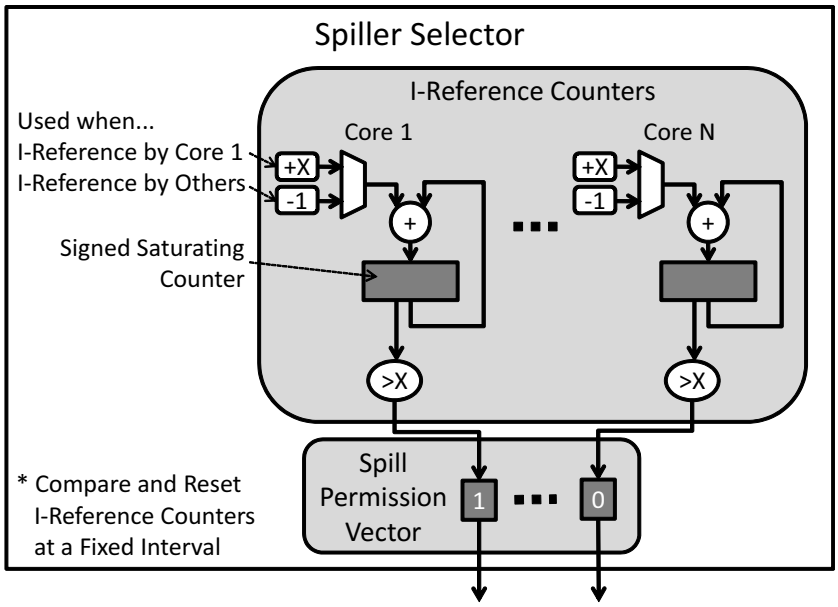


Figure 4.12 Detailed organization of Spiller Selector in the original method.

From the formulation above, I define the detail of Spiller Selectors. Figure 4.12 shows the organization of a Spiller Selector. It consists of the same number of I-Reference counters and the same bits of spill permission vector as the number of cores. The I-Reference counters represent the current likelihood that the corresponding cores benefit from spilling.

When an I-Reference occurs, the I-Reference counter of the causing core is incremented by the constant X; that of the other cores are decremented by 1. At the end of a period, the values of the counters are compared with X. If they are more than X, corresponding bits of the spill permission vector are set to 1.

When a directory receives a spilling request, it references the corresponding bit of the spill permission vector. If the bit is 1, it accepts the request and makes the Receiver Selector decide the destination. If the bit is 0, it declines the request. The request is treated just as a notification of invalidation or a request of write-back. In both cases, the directory updates the corresponding entry properly.

4.4.2 Receiver Selector

The selection rates of the destination core are determined mainly with the number of Extruded I-References. Cores with the large number of Extruded I-References are likely to be harmed by accepting spilled lines.

In the original method, the Receiver Selector checks whether a core is exposed to heavy performance degradation by the spilled lines or not. If the core is regarded to be harmed, it decreases the probability

that the core is selected as the destination of a spilled line. If not, it increases the probability. I define a threshold for the decision as a ratio of the number of Extruded I-References to the number of off-chip accesses that occur in the core.

I also consider the case that the negative effect of receiving lines is covered by the positive effect of spilling lines. To estimate the positive effect, the Receiver Selector also detects and counts cache hits on spilled lines. Fortunately, the directory caches do not require an additional state for the detection, because the states of the caches and the directories have already included the information whether the accessed line have been spilled or not after the last reference.

I formulate the decision of the performance degradation. Let E the number of Extruded I-References of a core, H the number of cache hits on spilled lines, and O the number of off-chip accesses that occur in the core. The selector considers the core to be harmed when the following inequality is met:

$$E - H \times l > O \times t \tag{4.3}$$

where l is a coefficient and t is the threshold.

By transforming this and substituting $X = l/t$ and $Y = 1/t$, we obtain the following inequality:

$$-X \times H - O + Y \times E > 0 \tag{4.4}$$

At this time, l and t are properly decided in order that both X and Y can be integers. From Inequality (4.4), we can make a decision about the performance degradation only with integer arithmetic using E, H, O, X, and Y.

I now define the detailed organization of Receiver Selectors from the formulation. Figure 4.13 depicts the detail of a Receiver Selector. It is composed of fairness counters, receive probability registers, a probability updater, and a weighted round-robin selector.

The fairness counter is decremented on a cache hit on a spilled line by X, decremented on an off-chip access by 1, and incremented on an Extruded I-Reference by Y. At the end of a period, the value of the counter is compared with 0. If it is more than 0, the selector increases the corresponding probability. If it is non-positive, the selector decreases the probability. The way to increase and decrease the probability is described later.

The receive probability register is assigned to each core. The probability updater has a register that keeps the probability that the selector does not spill the evicted line to any destination. In the original method, the sum of the values of the registers is constant: an increment of one register is accompanied by a decrement of another register.

Important policies on updating the registers are twofold. One is that a register with a large value should have a wide variation range, because the effect of variation is proportional to the current value. The other is that cores that are prohibited from spilling should be prioritized over other cores when both the types of cores ask for increase in their values.

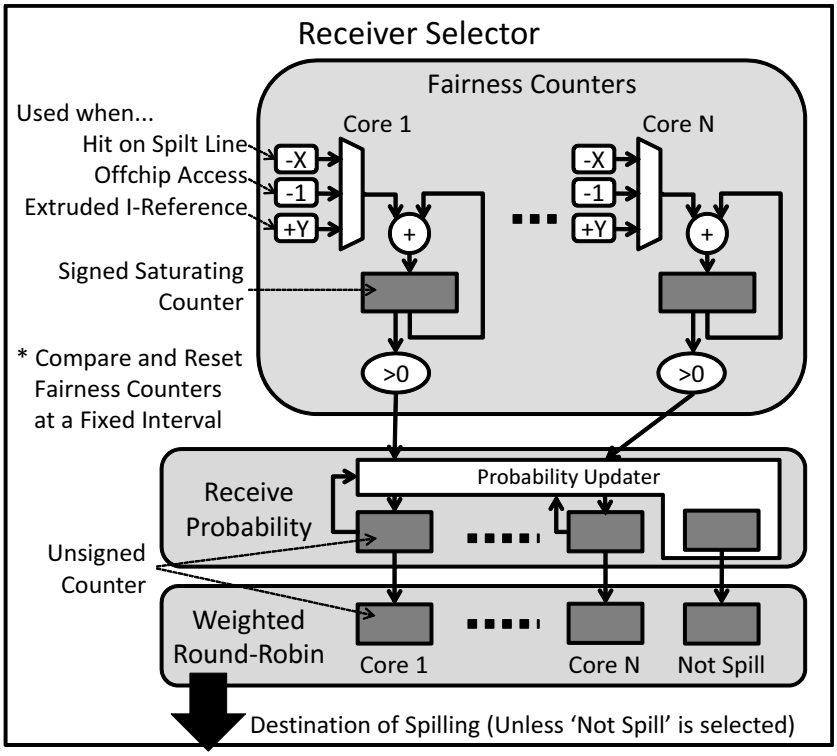


Figure 4.13 Detailed organization of Receiver Selector in the original method.

Considering the policies, I define the process of update as the following steps. The term *acquire n* means incrementing the register corresponding to the core by *n* and decrementing the *not spill* register by *n*. Conversely, the term *release n* stands for decrementing the corresponding register by *n* and incrementing the *not spill* register by *n*. If the value of the register to be decremented is less than *n*, the variation of the values is limited to it.

1. All the cores *release* the current value /8.
2. Cores with positive fairness counters *release* the current value /8 + 1.
3. Cores that have non-positive fairness counters and that are prohibited from spilling *acquire* the current value /4 + 1.
4. Cores that have non-positive fairness counters and that are allowed spilling *acquire* the current value /4 + 1.

When more than one core asks for *acquisition* in the third or fourth steps, the priority among them is randomly decided. The results of all the division are rounded down. In the initial state, all the registers corresponding to the cores are set to 0 and the *not spill* register is set to the maximum value for its bit width.

Figure 4.14 shows an example of the process of update. The number of cores is set to 4. All the

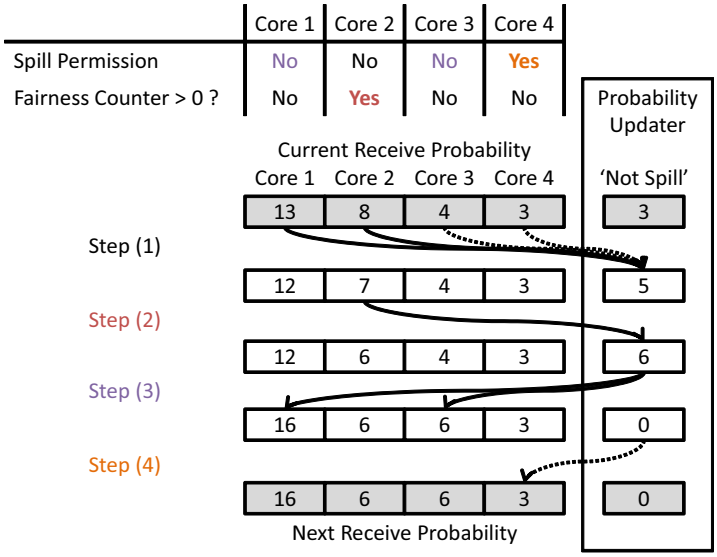


Figure 4.14 Example of update process of receive probability registers.

registers are 5-bit wide and thus the sum of the values is always 31. The respective values of the per-core registers before the update are 13, 8, 4, and 3. The value of the *not spill* register is 3. Core 4 is the only spiller in this period. At the end of the period, the value of fairness counter of Core 2 is positive and those of Core 1, 3, and 4 are negative.

In the first step, all the cores *release* 1, 1, 0, and 0, respectively. In the second step, Core 2, which has positive fairness counter, *releases* 1. In the third step, Core 1 and 3, which do not have permission to spill, *acquire* 4 and 2, respectively. In the last step, though Core 4 asks for acquisition, the corresponding register is not incremented, for the *not spill* register has already become 0. Consequently, the respective values of the per-core register after the update are 16, 6, 6, and 3. Note that Core 1, which had a large value before the update, has a wide range of variation and that the registers corresponding to Core 1 and 3, which are prohibited from spilling, are preferentially incremented when Core 1, 3, and 4 ask for acquisition. This example shows that the process of update meets both of the policies that I have mentioned.

Based on the updated probabilities, individual destinations of spilled lines are chosen by a selector of weighted round-robin [80]. In the weighted round-robin, the number of times each element is selected in a period is determined from the selection ratio assigned to it. For example, when elements A, B, and C has a selection ratio of 5:3:2, the selector sets a period to 5 + 3 + 2 = 10 rounds of selections and chooses elements so that A, B, and C can be selected 5 times, 3 times, and twice during the period, respectively.

The weighted round-robin selector has internal counters initialized with corresponding receive probabilities. When a line is allowed to be spilled by the Spiller Selector, which I have explained in the

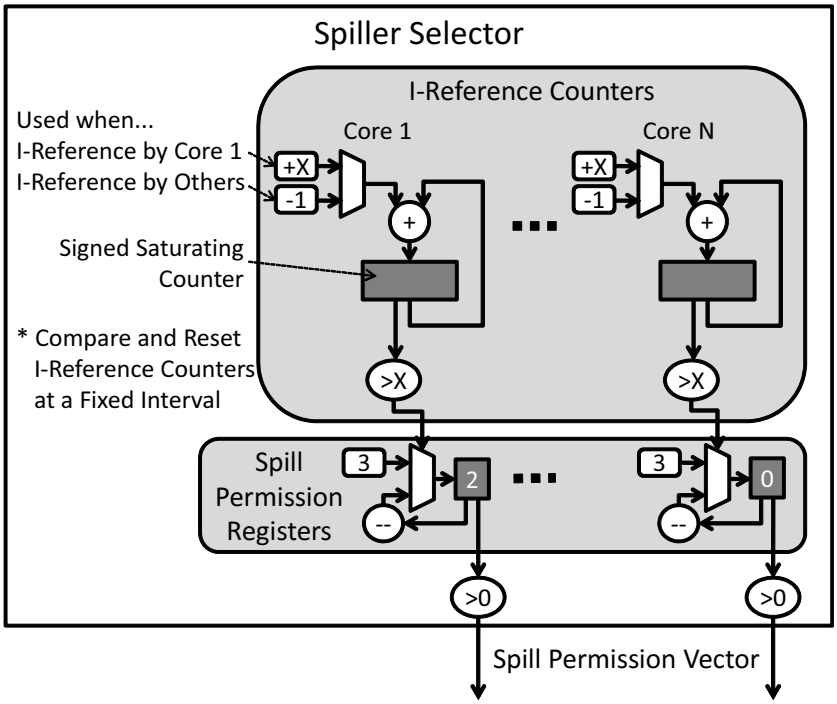


Figure 4.15 Detailed organization of Spiller Selector in the highly-precise method.

previous section, the selector chooses one of the counters that has the biggest number among them and the corresponding core is not the spiller itself. If it chooses a counter except *not spill*, it decides the corresponding core as the destination; if it chooses *not spill*, the evicted line is just removed from chip. Afterwards, the chosen counter is decremented. When all the internal counters become zero in this time, they are reset to the value of the receive probability registers.

4.5 Highly-Precise Method

4.5.1 Spiller Selector

The strategy to choose cores suitable for spilling is almost the same between the original method and the highly-precise method. The only difference between them in choosing spillers is how long the cores corresponding to the counters that exceed the threshold will be permitted to spilling lines.

Even though some cores frequently refer to recently evicted lines, the number of their detected I-References in a period may be accidentally less than the threshold, and as a result, they are sometimes prohibited from spilling. This loses their potential performance gain in some cases.

In order to avoid such opportunity loss, once the number of I-Reference exceeds the threshold, the corresponding core is permitted to spill for the next few periods, rather than for the next period. It requires small modification to the generation of the spill permission vector.

Figure 4.15 depicts the detailed organization of a Spiller Selector in the highly-precise method. Instead of the spill permission vector, it has spill permission register for each core. The register stands for the number of remaining periods to be allowed spilling. The spill permission vector is obtained by arranging the results of the test of whether the registers are non-zero.

The I-Reference counters are updated and compared with the threshold X in the same way as the original method. If they are more than X, corresponding spill permission registers are set to a predefined number (3 in the figure), If not, the registers are decremented (unless they are zero).

4.5.2 Receiver Selector

In the highly-precise method, the approach to estimate the negative impacts of spilled lines is widely different from the original method. I estimate the performance degradation from the number of Extruded I-References, and then restrict or prohibit the acceptance of spilled lines from the estimated degradation. The numbers of cache hits on spilled lines and that of off-chip accesses, which are used in the original method, is not considered at this time.

I mention the estimation of the performance degradation by receiving spilled lines. Let T the number of cycles in a period of update, E the number of Extruded I-References, k the capture rate of Extruded I-References or how much of the increase of misses are detected by Extruded I-References, and L the average number of stall cycles. The estimated degradation is as follows:

$$\begin{aligned} & 1 - \frac{\text{(cycles with pollution)}}{\text{(cycles without pollution)}} \\ = \quad & 1 - T/(T - \frac{EL}{k}) \\ = \quad & \frac{EL}{k}/(T - \frac{EL}{k}) \\ = \quad & E/(\frac{Tk}{L} - E) \end{aligned} \tag{4.5}$$

If E is small, the degradation can be approximated as $E/\frac{Tk}{L}$. Although k and L depend on architectures and applications, when we assign their typical values to them, we can now replace the denominator with a constant. Hence, E or the number of Extruded I-References itself can be used as an index of performance degradation.

In order to realize the restriction and the prohibition of receiving, I introduce T_p a primary threshold and T_s a secondary threshold. If the estimated degradation exceeds T_p, the core is prohibited from receiving spilled lines. If it is between T_p and T_s, the probability that the core is selected as the destination is gradually reduced in order that the directories can restrict receiving of the cores. Figure 4.16 depicts the relationship between the estimated degradation and the relative probability, where P_{max} is the default (maximum) probability.

I also propose a complementary barometer of relative probability of receiving: a ratio of inactive

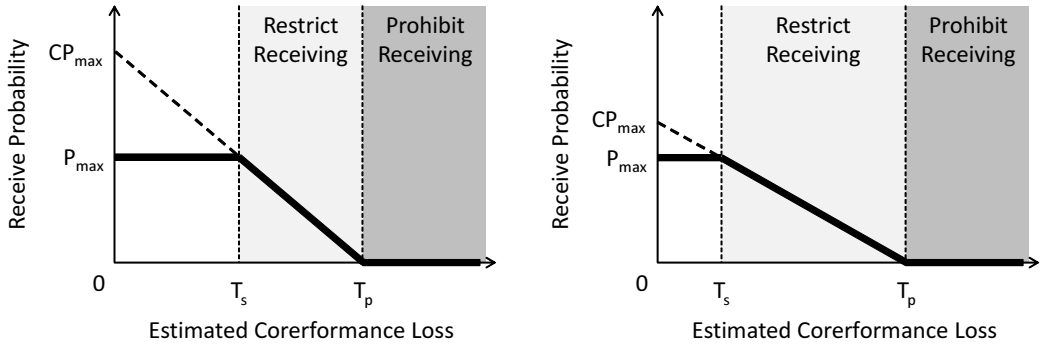


Figure 4.16 Restriction and prohibition of receiving evicted lines.

(not quite frequently reused) lines to those accessed by the core itself in the local cache of a core. If the ratio is small, that is, most of the cache is frequently reused, the potential of spilled lines to degrade the performance of that core is high. Therefore, the selector decreases T_s or the secondary threshold for that core so that its probability of being selected as the destination can be reduced easily. Graph (a) and (b) in Figure 4.16 shows the difference between the case with high inactivity and low inactivity, respectively. In fact, I modify not the secondary threshold but the slope of the graph (expressed as C in the figure), with consideration for the ease of implementation.

We can get the ratio in some replacement algorithms. For instance, when we use the RRIP[3], we can define an active cache line as a line with higher priority than on insertion. If we have the numbers of active lines and lines accessed by its own core, the ratio is calculated as follows:

$$1 - \frac{\text{the number of active lines}}{\text{the number of lines accessed by its own core}}$$

The numbers can be obtained with small counters.

Each core calculates the ratio with a precision of about 6 bits, and then multicast it to the directories before they update the probabilities. Although using multicasts is not always a good way in terms of scalability, since their frequency is very low, the effect on the performance and the scalability is considered small enough. Furthermore, if needed, we can start the communication sufficiently long before update and give it lower priority than usual in the interconnections in order to further reduce the effect. As a result, the effect can be negligibly small.

If we are not able to obtain the ratio, or the cost of multicasts is still critical, we can use the fixed number among all the cores instead. The ASCEND is still effective in this case. We can understand the demands of capacity more precisely with this barometer, though.

I now formulate the update of relative probabilities. Let E_a the number of Extruded I-References by Core a, T_p the primary threshold, I the ratio of inactive lines, and P_{max} the maximum probability. The

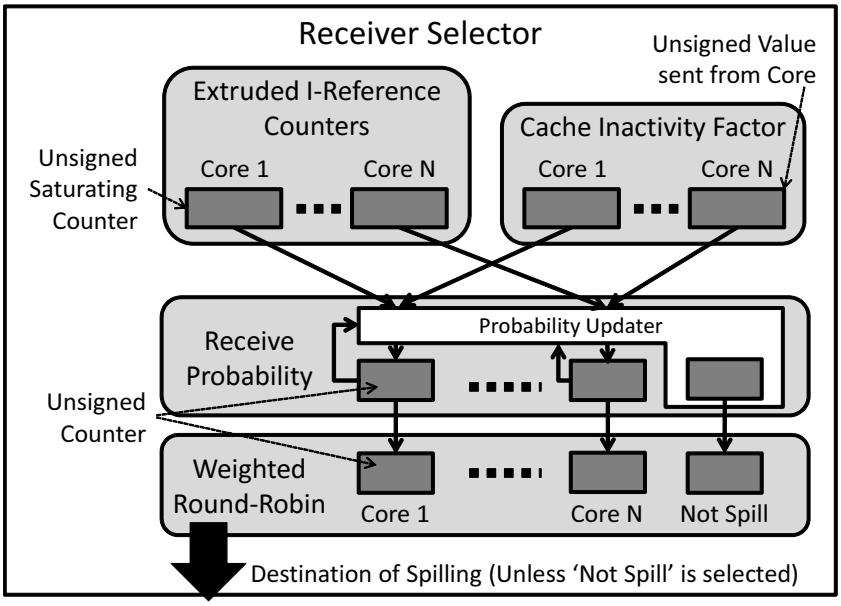


Figure 4.17 Detailed organization of Receiver Selector in the highly-precise method.

relative probability of Core a, or P_a, is calculated as follows:

$$P_a = \begin{cases} P_{max} & (E \leq (1 - 1/Ic)T_p) \\ Ic\frac{T_p - E}{T_p}P_{max} & ((1 - 1/Ic)T_p < E \leq T_p) \\ 0 & (E > T_p) \end{cases} \tag{4.6}$$

When low probabilities are set to all the cores, the practical possibility of receiving may become high. This may make the restriction of the acceptance meaningless. To avoid this situation, I add a choice that corresponds to not spilling the evicted line to any destination. The probability of selecting such a *not spill* option, say P_{no}, is defined as follows:

$$P_{no} = \begin{cases} 1 & (\max(P) = 0) \\ \frac{P_{max} - \max(P)}{\max(P)}\Sigma P & (0 < \max(P) < P_{max}) \\ 0 & (\max(P) = P_{max}) \end{cases} \tag{4.7}$$

where $\max(P)$ is the maximum of probabilities of the cores and ΣP is the sum of them.

Incidentally, P_{no} is less than NP_{max}. For a $\max(P)$, the maximum of ΣP is $N \times \max(p)$ (when $P_1 = P_2 = ... = P_n > 0$). Substitute this into Equation (4.7), we obtain the intended inequality: $P_{no} = N(P_{max} - \max(P)) < NP_{max}$.

I design the architectural model of Receiver Selector from the formulation. Figure 4.17 shows the organization of a Receiver Selector. It includes the same number of Extruded I-Reference counters and cache inactivity factors as the number of cores to calculate the probability. The Extruded I-Reference counters are unsigned saturating counters incremented on an Extruded I-Reference of the corresponding

cores. Cache inactivity factors are ratios of inactive lines to those accessed by the corresponding cores. They are interpreted as fixed-point numbers of about 6 bits.

The probability updater calculates the relative probability of each core with these two values and Equation (4.6) and stores the result to a register. The results are truncated and expressed as integers. Unlike the original method, the sum of the values of the registers is not fixed. Since $P_{no} < NP_{max}$, the register corresponding to P_{no} is $\lceil \log_2 N \rceil$ bits larger than the others, where $\lceil n \rceil$ represents the smallest integer that is more than or equal to n.

After calculating the relative probabilities, the way to choose individual destinations of spilled lines is the same as the original method (i.e. the weighted round-robin selector).

4.5.3 Weak Receiving — An Optional Extension of Receiver Selector

In this section, I propose an optional extension of ASCEND, named Weak Receiving, for achieving further efficiency. The bases of idea are some modern replacement algorithms — the RRIP[3] and the SHiP[24], which I have mentioned in Section 2.1.2.

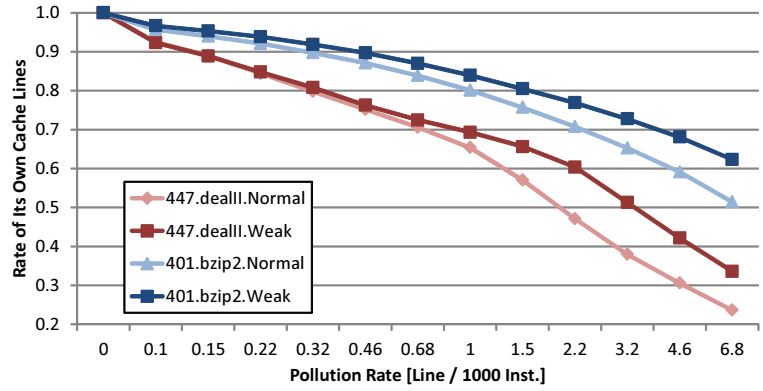
The SHiP applies a technique of dead block prediction to the RRIP. It predicts whether inserted lines are reused in the future. If the lines are predicted *dead* (or unlikely to be reused), it gives the lowest priority to them. Therefore, they soon become the targets of replacement.

Like the SHiP, we can differentiate between two separate kinds of lines by changing the priority on insertion in the RRIP. Applying this idea to the lines used by its own core and those spilled from other cores, the former lines can be kept in preference to the latter lines by just following the normal procedure of RRIP after the insertion. As a result, when the core receives evicted lines frequently, it can avoid its own cache lines being lost rapidly.

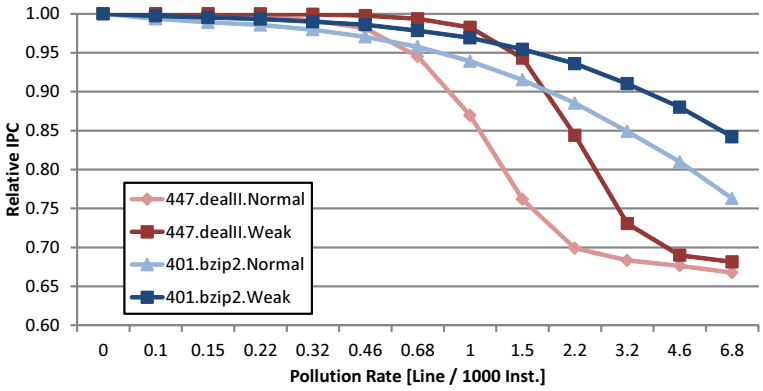
I call the technique that gives lower priority to spilled lines Weak Receiving. The target of the Weak Receiving is a little large RRIP that requires 3 or more bits for the priority of each line. In a 2-bit RRIP, the priority of the preferential lines is too close to that of reused lines. It may spoil the advantage of the RRIP because reused lines are unlikely to remain for long enough.

I consider applying this technique to a 3-bit RRIP, where the highest and lowest priorities are 0 and 7, respectively. In the conventional RRIP, the priority of an inserted line is always 6. In the RRIP with the Weak Receiving, the priority of an inserted line accessed by its own core is 5; that of a spilled line accepted from other core is 6.

Of course, giving the same priority can use the cache more efficiently in some cases. Therefore, it is desirable to prepare a way to use either spilling dynamically. For dynamic modification on the policy, I fix the priority of one kind of lines and change that of the other kind of lines depending on the situation. According to my preliminary evaluation, it is slightly better to fix the priority of spilled lines than to fix that of its own lines.



(a) Ratio of its own lines



(b) Relative performance

Figure 4.18 Pollution rate vs. percentage of cache lines that are kept local and relative performance.

I now mention the incorporation of the Weak Receiving into the main proposal. Since this technique focuses on the feature of the RRIP, it is applicable if we also adapt the RRIP as a replacement algorithm.

The point is what index is used for dynamic modification. A feasible index is the receive probability in the Receiver Selector. If it is less than a certain threshold, the Weak Receiving is considered the better choice in order to suppress negative impacts of receiving.

Nevertheless, the probabilities are decided independently in each Receiver Selector and the cores do not know them. We can use one of the following two ways to cope with this: one way is that each core asks all the directories for the corresponding probability and then sums them up; the other way is that directories send a flag whether using the Weak Receiving or not along with the forwarded line.

The former way might have a problem of scalability, just as the discussion on multicasts in the Receiver Selectors, however, it is usually negligible with a careful use. The latter way has a disadvantage that it is not applicable if we fix the priority of spilled lines. Therefore, I use the former way to switch two kinds of receiving.

To verify the validity of the Weak Receiving, I confirm that it actually reduces the effect of pollution

through a preliminary evaluation. I use the multicore environment that I have mentioned in Chapter 3. Since I run a single application, the number of cores is 1; instead, to emulate the pollution from other cores, I insert dummy lines in a predefined rate. The target applications are 401.bzip2 and 447.dealII that are especially sensitive to the pollution. I evaluate varied pollution rates, and measure the ratio of lines kept local and the reduction in relative performance.

I use a 3-bit RRIP as a replacement algorithm and the priority of its own lines on insertion is set to 5. I define Normal as the case the priority of spilled lines on insertion is also set to 5 and Weak as the case it is set to 6. Note that I fix the priority of its own lines in this case in order to equalize the performance without pollution.

The size of local cache is set to 2MB. The core is not allowed to spill but receives dummy lines in a fixed rate. The rates vary from 0.1 lines to 6.8 lines per 1,000 instructions, increasing exponentially. The other parameters are the same as in Chapter 5.

The results of the evaluation are shown in Figure 4.18. The x-axis represents the pollution rate (the number of inserted lines per 1,000 instructions), and the y-axis represents the ratio of keeping its own lines (the upper graph) or the relative IPC (the lower graph).

As the pollution rate increases, the number of lines available for its own core is decreased. When it comes to fail to keep the frequently reused lines, the relative IPC is dropped rapidly. In the rate where the performance drop begins, Weak is 1.5 to 2 times more than Normal. Moreover, focusing on the section that is before the drop begins, the ratios of keeping its own lines are almost the same. In other words, the chance for other cores to find the required lines here is seldom damaged.

From this evaluation, the Weak Receiving was confirmed as an efficient method to accepting more spilled lines without performance degradation.

4.6 Difference with other spilling optimizations

As I have shown qualitatively in Chapter 2 and quantitatively in Chapter 3, the problem of the existing methods to manage spilling is that they do not meet both high efficiency and high scalability. In this section, I describe how ASCEND solves the problem by showing the difference between it and the existing methods.

The most important difference with the other methods in terms of efficiency is that ASCEND uses two kinds of units that apply different criteria to analyze the demands of applications on cache capacity. The DSR (Dynamic Spill-Receive) [8] predicts which spilling or receiving is more suitable by comparing the numbers of cache misses in the sampling sets. Similarly, ElasticCC (Elastic Cooperative Caching) [9] compares the numbers of cache hits on the LRU sets of virtual private region and virtual shared region. Each core chooses either expanding its private region, expanding its shared region, or doing nothing, based on the result of the comparison. Both of them use a single criterion for estimating

the demands. ASCEND, on the other hand, utilizes the number of I-References for the estimation of performance gain and that of Extruded I-References for the detection of performance loss. These two axes of prediction are useful for precise analysis.

With respect to scalability, the main difference with the DSR, which has poor scalability, is the elimination of frequent broadcast. The DSR requires that all the core share information of cache misses. This design looks appropriate for multicore processors where bus-based interconnects and a snooping-based coherence are common. However, it is not feasible for many-core processors where a network-on-chip and a directory-based coherence are usually used, because it needs too much communication for sharing the information. In contrast, ASCEND only requires a minimal number of broadcasts by having selectors in one directory work independently from those in the others. This keeps ASCEND scalable and suitable for many-core processors.

4.7 Summary

In this chapter, I mentioned the concept and the architecture of the proposed ASCEND, including the way to utilize unused entries in the directories, the overall organization, the details of the components, and its optional extension. In explanation of each unit, I mentioned the principle and the formulation and then showed the detailed architectural model that followed the formulas and the decision about spilling and receiving with it.

I also proposed the Weak Receiving as an optional extension of ASCEND. It was based on a feature of a modern cache replacement algorithm. I referred to the idea, the application, and the validity of it.

Chapter 5

Evaluation of ASCEND

In this chapter, I mention the results of the evaluation of ASCEND that are proposed in the previous chapter, and some discussions based on the results. I first show the evaluation results for the multicore and many-core environments in Section 5.1 and Section 5.2, respectively. Secondly, I calculate the amount of the additional hardware in both the environments. Lastly, I have some discussions with the results and additional data.

5.1 Evaluation Results for Multicore Environment

In this section, I evaluate the proposal in the multicore environment. I add the following 4 settings to what I have evaluated in Chapter 3:

ASCEND-O stands for the original method of ASCEND that I have defined in Section 4.4.

ASCEND-H represents the highly-precise method of ASCEND that I have defined in Section 4.5.

ASCEND-H w/o WR means the highly-precise method of ASCEND without applying the Weak Receiving.

ASCEND-H fixed-CIF means the highly-precise method of ASCEND where the cache inactivity factors are fixed to 1.

I evaluate ASCEND-H w/o WR and ASCEND-H fixed-CIF for inspection of the change of efficiency by the Weak Receiving and the cache inefficiency factor, respectively.

In all of them, as the corresponding part to extra directory entries, I add a tag array with a half as many elements as the number of cache lines to the controllers of ASCEND. The tags of evicted lines that were marked as *live* by the dead block predictors are recorded in the array. The array is referenced on L3 miss. If the requested tag is found in the array, it is detected as an I-Reference (and an Extruded I-Reference).

I mention the parameters of ASCEND. For the Spiller Selectors, I set the period for update to 250k cycles, the parameter k to $4/5$, and the number of successive periods for which spilling is granted (in ASCEND-H) to 3. For the Receiver Selectors in the original method, I set the period to 250k cycles,

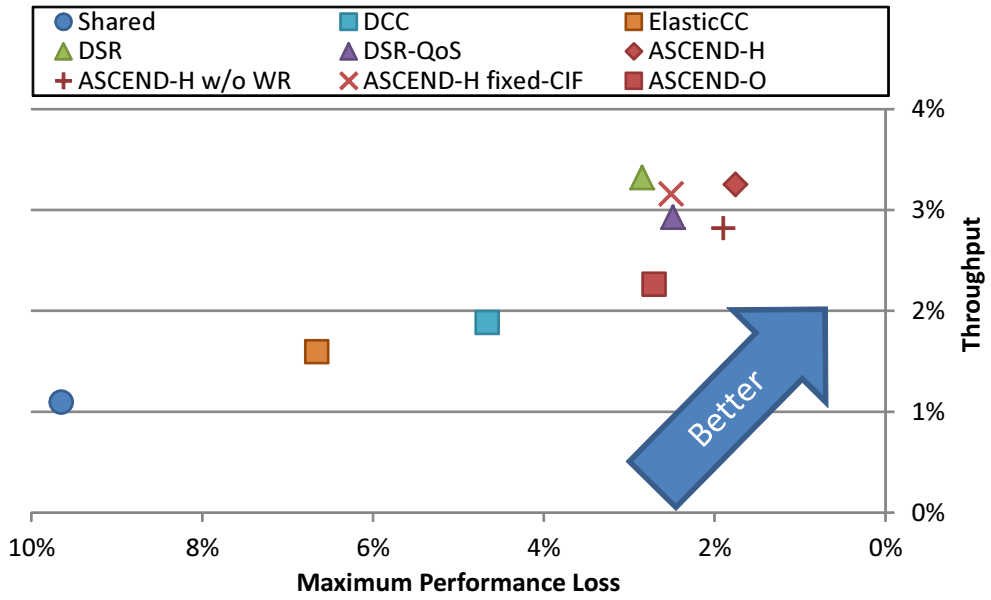


Figure 5.1 Comparison of performance and QoS of ASCEND with the other methods in the multicore environment.

the same as the Spiller Selectors, and the parameters l and t to 3/4 and 1/16, respectively. For the Receiver Selectors in the highly-precise method, I set the period to 1M cycles, the threshold T_p to 256, the maximum relative probability P_max to 15, and the coefficient c to 2.

These parameters take effects on the activeness of spilling, the activeness of receiving, and the hardware amount. For small k, the threshold for spilling permission gets small and thus the activeness of spilling is increased. Larger l makes the importance of hits on the spilled lines greater, and larger t makes the threshold for decreasing the probability to receive evicted lines larger. Both of them result in the increased activeness of receiving. Similarly, when T_p or c get large, the threshold for restriction or prohibition of receiving also gets large and thus the activeness of receiving is increased.

There is appropriate balance of the activeness. If they are too small, the controllers may miss the chance of improvement with spilling. On the other hand, if they are too large, some cores are harmed by spilled lines and thus the QoS is degraded. In addition, T_p also depends on the average number of stall cycles in the adopted architecture. I set the appropriate parameters above with preliminary evaluations.

For the length of the counters, I use 8-bit I-Reference counters, 10-bit fairness counters or Extruded I-Reference counters, and 6-bit cache inactivity factors. In the original method, all of the probability registers are set to 6-bit. In the highly-precise method, the probability registers for the cores are 4-bit and that for *Not Spill* is 6-bit. They are the minimal sufficient lengths to make almost the same decision as setting the lengths to infinite.

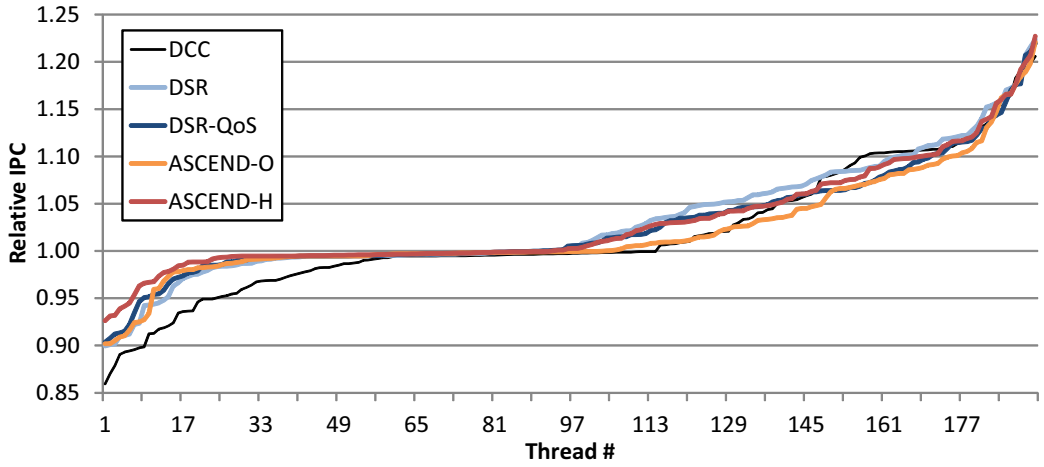


Figure 5.2 Relative IPC distribution in the multicore environment.

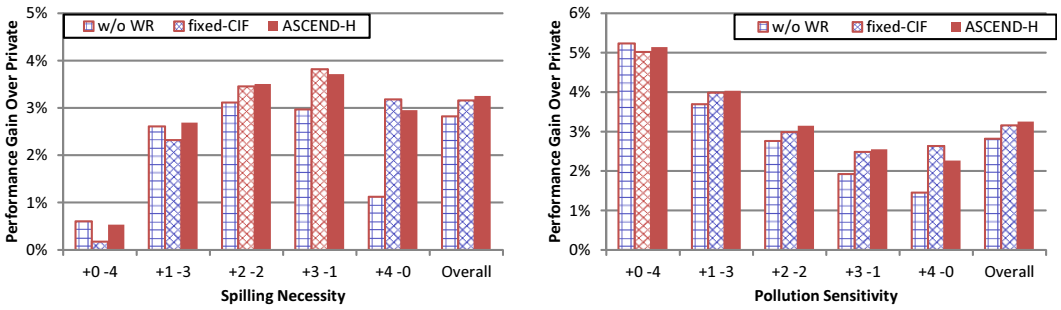
Figure 5.1 shows the result. ASCEND-O did not have better throughput than DSR, although it had almost the same maximum IPC loss as DSR. The relative IPC improvement was 2.3% and the maximum IPC loss was 2.7% on average. To the contrary, ASCEND-H had almost the same performance as DSR and better QoS than not only DSR but also DSR-QoS. The improvement of average relative IPC was 3.3% (looking more closely, DSR performed just 0.04% better than ASCEND-H), and the average of the maximum IPC loss was 1.8%. This result means that ASCEND-H has better efficiency than DSR.

ASCEND-H w/o WR, or not applying the Weak Receiving, mainly affected performance. The average improvement of relative IPC was 2.8% and the average of the maximum IPC loss was 1.9%. On the other hand, ASCEND-H fixed-CIF, or using the fixed cache inactivity factor, had an influence chiefly on QoS. The relative IPC gain was 3.2% and the maximum IPC loss was 2.5%, on average.

I now analyze the difference among DCC, DSR, DSR-QoS, ASCEND-O, and ASCEND-H. Figure 5.2 shows the graph plotting the distribution of relative IPCs of all the workloads that are sorted from lowest to highest. The x-axis represents individual applications. Since there are 48 workloads each of which consists of 4 applications, each line consists of 192 points. The y-axis is relative IPC.

DCC (with no limitation on spilling) has remarkable difference from DSR, DSR-QoS and ASCEND-H from 1st point through 48th and from 97th through 144th. This implies that unregulated spilling not only degrades the performance of cores that are sensitive to reduction in cache size, but also limits the improvement of cores that essentially benefit from spilling.

ASCEND-O did not perform well from 97th point through 144th, although it had similar performance to DSR and DSR-QoS on the others. It means that ASCEND-O failed to obtain the possible improvement of some applications. This seems to come from misprediction on their demands on cache capacity. However, judging from the performance of the slowed applications (in the left half of the



(a) (b)

Figure 5.3 Throughput variation with adopting options of ASCEND.

graph), the limitation of receiving evicted lines of them itself works well.

Comparing ASCEND-H with DSR and DSR-QoS, it outperformed them from 1st through 32nd, performed similarly to DSR-QoS from 33rd through 144th, and worked similarly to DSR from 145th through 192nd. The intension of suppressing the performance drop of sensitive cores by the sacrifice of the improvement of some cores is common to DSR-QoS and ASCEND-H, however, the effects differed. DSR-QoS penalizes misses from slowing cores. Nevertheless, it might change the penalties too late because the time required for precise decision was long. In contrast, ASCEND-H succeeded in estimating the negative impact precisely with shorter interval of time than DSR-QoS.

Lastly, I mention the effect of the Weak Receiving and the cache inactivity factor. Figure 5.3 shows the average relative IPC of ASCEND-H w/o WR, ASCEND-H fixed-CIF, and ASCEND-H with respect to each division of workloads. I divide the workloads by the number of applications that have high spilling necessity (in graph (a)) or high pollution sensitivity (in graph (b)). The x-axis represents each division. A bar with the label "+*A* -*B*" stands for the weighted mean of workloads that is made up from *A* applications with high indexes and *B* applications with low indexes. A bar with the label *Overall* means the weighted mean of all the workloads.

In both the graphs, if a workload included a large number of applications with high indexes, the difference of ASCEND-H w/o WR with ASCEND-H got higher. In particular, it had remarkable tendency for spilling necessity. When the workload includes a number of applications with high spilling necessity, the number of spilling becomes large. This can became easy for the Weak Receiving to exert its effect of accepting many lines without degradation.

Comparing ASCEND-H fixed-CIF with ASCEND-H in graph (a), when there were three or more spilling-necessary applications, the variable cache inactivity factors took a negative effect on performance. However, when up to one application had high spilling necessity, they took a positive effect. The use of the variable factors reduces the relative receive probability of some applications and, as a result, the chance of spilling. Even though it becomes some overhead in the former case, it avoids the

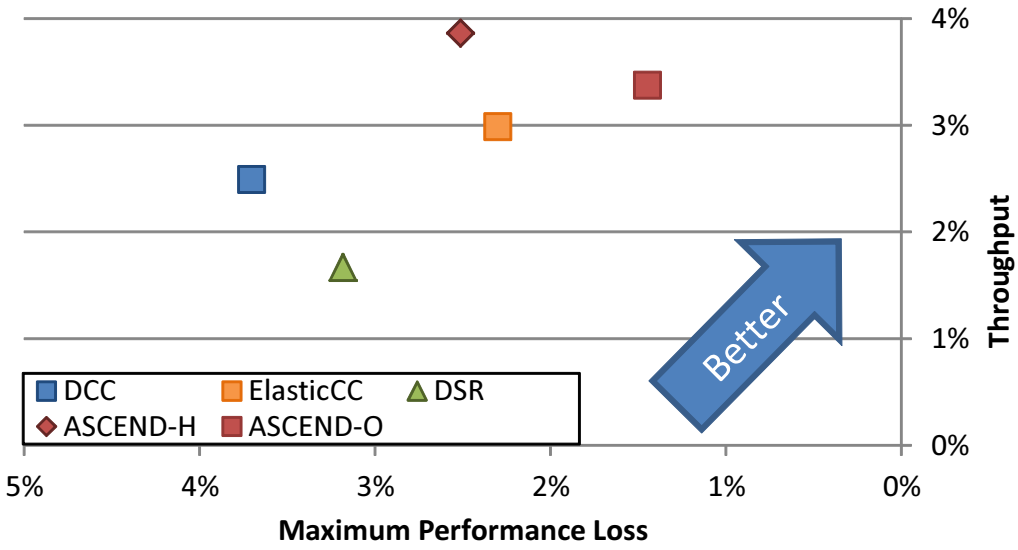


Figure 5.4 Comparison of performance and QoS of ASCEND with the other methods in the many-core environment.

serious performance loss of some cores in the latter case. Thus, it improves mostly the QoS.

5.2 Evaluation Results for Many-core Environment

I then evaluate ASCEND in the many-core environment. In addition to what I have evaluated in Chapter 3, I also evaluate ASCEND-O and ASCEND-H.

Unlike the multicore environment, ASCEND-H does not apply the Weak Receiving and the variable cache inactivity factors, because the base architecture does not adapt either dead block prediction or the RRIP.

Most of the parameters of ASCEND-O remain unchanged. However, the length of the probability registers is set to 8 bits in order to keep the average number of them almost the same as that in the multicore environment.

I use the same parameters of ASCEND-H as those shown in the previous section, except the threshold of the Receiver Selectors, or T_p, of 16. The reasons for the change are twofold: one is that the average number of detected Extruded I-Reference in a period is divided by the number of directories; the other is that the influence of a cache miss in out-of-order processors, where the latency of a miss is sometimes hidden, is smaller than that in out-of-order processors. Along with this, I modify the length of the Extruded I-Reference counters to 6 bits.

Figure 5.4 shows the results in the many-core environment. ASCEND-O showed a little better performance than ElasticCC and the best QoS among all the methods. The average throughput gain was 3.4% and the average of the maximum performance loss was 1.5%. ASCEND-H worked the best in

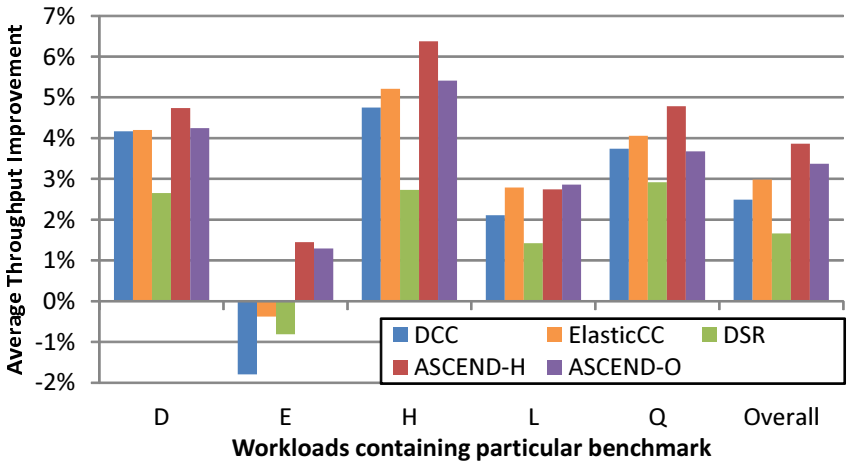


Figure 5.5 Performance in the many-core environment. Workloads are grouped by included application.

performance among these methods, though its QoS is a little lower than ElasticCC. The throughput improvement was an average of 3.9% and the maximum throughput loss was an average of 2.5%.

I now analyze the result in detail. Figure 5.5 shows the improvement of the average throughput with respect to each division of workloads. Each group of bars other than *Overall* represents the arithmetic mean of average throughput of workloads that includes the corresponding application to the label. The number of workloads including a specific application is 35 (out of 70). *Overall* means the average of all the 70 workloads.

ASCEND-H performed the best other than workloads including L (lu), where ASCEND-O showed the best performance. In particular, for workloads that comprise E (equation) that tends to pollute other caches, only both ASCEND showed the better performance than Private do.

The pollution of caches by the spilled lines that are unlikely to be reused might be removed with dead block prediction, just as I did in the multicore environment. However, the problem is that it is not sure whether it matches its hardware cost. Although the hardware amount of per-line traces is proportional to the cache size, the size of per-trace predictors (typically several kilobytes) does not depend on the cache size. As a result, the cost of predictors gets large for small caches. It discourages us from applying dead block prediction.

ASCEND prevents harmful cores from spilling their lines. According to Figure 5.5, this enables the caches to reduce the negative effect by unused spilled lines without dead block prediction, and contributes to the improvement of the other cores.

Based on the comparison between the two methods of ASCEND, workloads including an application that can benefit from spilling (H or Q) had a large difference in performance. Nevertheless, they performed almost the same in workloads containing an application that can negatively affect the others

(E) or that can be badly affected (L). These results imply that the original method is essentially more conservative than the highly-precise method.

5.3 Hardware Overhead

In this section, I evaluate the hardware overhead of ASCEND from its parameters in each environment.

First of all, I calculate the overhead of ASCEND-H in the multicore environment. The total amount of registers per core that the two selectors require is 34 bits. The "not spill" registers in the receiver selector demand 12 bits. Therefore, the total memories in the selectors is $34 \times 4 + 12 = 148$ bits. In addition, each cache requires two 16-bit counters (the number of lines can be expressed in 16 bits) to calculate the cache inactivity factor and a 6-bit counter for the Weak Receiving. After all, the total amount of required memory is $148 + (16 \times 2 + 6) \times 4 = 300$ bits.

Secondly, I compute the overhead of ASCEND-H in the many-core environment. Since the per-core registers in the selectors need 24 bits and the "not spill" registers require 16 bits, additional memory per directory is $24 \times 32 + 16 = 784$ bits. By multiplying it by the number of directories, the total amount of required memory becomes $784 \times 8 = 6,272$ bits.

In the same manner, the hardware overhead of ASCEND-O is calculated at $31 \times 4 + 12 = 136$ bits in the multicore and $(35 \times 32 + 16) \times 8 = 9,088$ bits in the many-core.

I also calculate the overhead of other methods. ElasticCC requires three kinds of hardware to each core: a register for the current number of virtual private ways, a counter for the decision of repartitioning, and a Spilled Block Allocator that decides the destination of spilled lines in proportion to the number of virtual shared ways. The register is 3-bit (for 8-way caches) or 4-bit (for 16-way). Though the length of the counter is not mentioned in [9], an 8-bit counter is sufficient as far as I know. The Spilled Block Allocator requires the number of ways minus one bit per core (note that at least one way is always virtually private). The sum of these is $(4 + 8 + 4 \times 15) \times 4 = 288$ bits in the multicore environment, or $(3 + 8 + 32 \times 7) \times 32 = 7,520$ bits in the many-core environment.

Finally, for DSR, the only additional hardware is a 10-bit counter (called PSEL) per core for judging strategy of the corresponding core. If all the cores can access a single group of common counters, the total overhead is 40 bits in the multicore environment or 320 bits in the many-core environment. However, it can be impractical to read the shared counters every time spilling is needed. Instead, if each core has a group of counters independently, and synchronizes them when needed, the overhead is multiplied by the number of cores. As a result, the required memory becomes 160 bits in the multicore environment or 10,240 bits in the many-core environment.

From these results, ASCEND has almost the same hardware overhead as other methods. Even in absolute terms, the amount of about 1KB is very small. Hence, the additional hardware amount is not likely to affect the validity of ASCEND negatively.

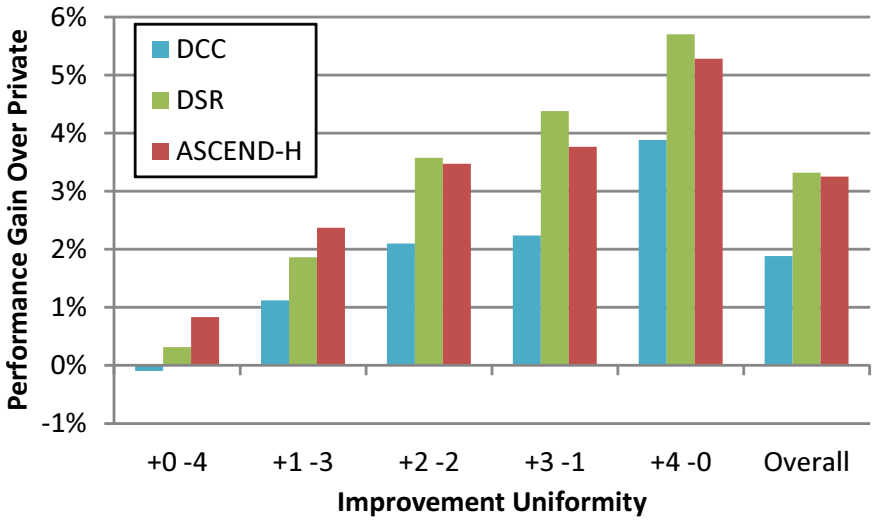


Figure 5.6 Throughput in the multicore environment. Workloads are grouped by improvement uniformity.

5.4 Discussion

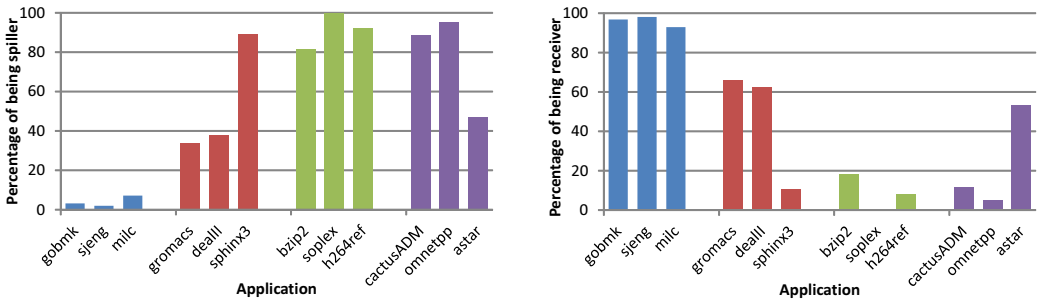
5.4.1 How did ASCEND classify the demands in capacity?

I discuss the difference between DSR and ASCEND-H from another perspective. In the benchmark selection, I have divided the applications by the two axes, that is, the spilling necessity and the pollution sensitivity. If the heights of these standards match, the tendency of performance loss according to the reduction of cache size and that of performance gain according to the additional cache also match. In short, such application has uniform improvement.

I define it as a new axis, which is named improvement uniformity, whether the heights of the two indexes match. Applications in Class A and C are considered to have high improvement uniformity; Class B and D are regarded to have low improvement uniformity (see also Figure 3.8). I divide the workloads by the number of applications with high improvement uniformity, and compare the average performance for each division.

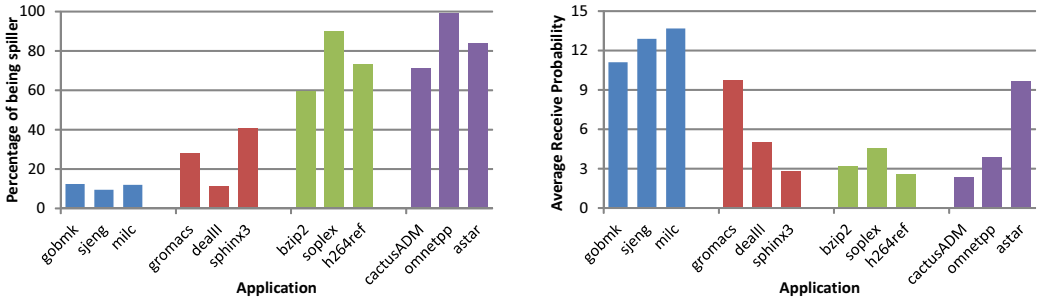
Figure 5.6 shows the average relative IPC of DCC, DSR and ASCEND-H with respect to each division of workloads. Just like Figure 5.3, the label "+*A* -*B*" means workloads that are composed of *A* applications with high uniformity and *B* applications with low uniformity.

According to the figure, while ASCEND-H worked worse than DSR in workloads including two or more applications with high uniformity, ASCEND-H outperformed DSR in those including zero or one application with high uniformity. For most applications that have uniform improvement, it is the most suitable strategy to only spill evicted lines (if performance will be improved) or to only receive



(a) (b)

Figure 5.7 The percentage of being spiller and that of being receiver in DSR.



(a) (b)

Figure 5.8 The percentage of being spiller and the average receive probability in ASCEND-H.

the spilled lines (if not). These applications suit DSR. On the other hand, some applications that have non-uniform improvement may require other strategies than those that DSR offers. ASCEND-H can benefit from such applications.

Figure 5.7 and 5.8 describe what strategy DSR and ASCEND-H actually employed on each application, respectively. The x-axis represents individual applications. Applications that belong to the same class are shown in bars colored the same color. The y-axis in Figure 5.7 (a) and Figure 5.8 (a) is the rate of time that the core is permitted to spill. The y-axis is Figure 5.7 (b) is the rate of time that the core is permitted to receive. The y-axis is Figure 5.8 (b) is the average of the relative probability that the core is chosen as the receiver. In short, a bar in graph (a) means aggressiveness of spilling and that in graph (b) stands for activeness of receiving. Note that the sum of (a) and (b) in Figure 5.7 always equals to 100%, because DSR necessarily permits either spilling or receiving.

For 9 applications other than *dealII*, *sphinx3*, and *astar*, the tendencies of the graphs are similar. However, for applications in Class A and C, DSR chose the strategy more extremely than ASCEND-H. This fluctuation of judgment in ASCEND-H is considered a reason why my method performs worse than theirs in some applications.

For *dealII* and *sphinx3*, ASCEND-H was reluctant to both spilling and receiving, that is, it predicted

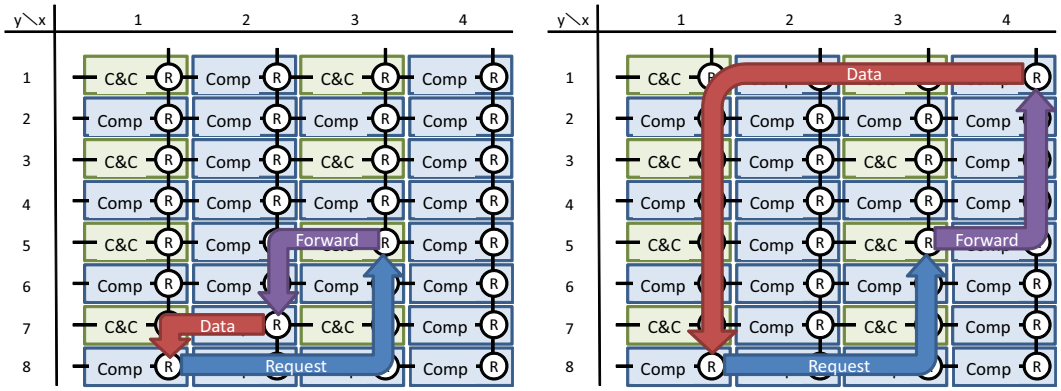


Figure 5.9 An example of the difference in latency by the distance between the requester and the holder.

that it was the best strategy to refrain from both of them. This implies that the division of DSR of spillers and receivers is not sufficient. One of the reasons why ASCEND-H outperformed DSR in some workloads is the proper treatment of such applications. The existence of applications like *dealII* and *sphinx3* and its importance have already been noted [10]. As a variation of DSR, DSR with three states (DSR-3S) that divided cores into spillers, receivers, and neutral was studied in [10].

In ASCEND-H, *astar* shows the different tendency from the others: it estimates that applying both spilling and receiving is the best strategy. This implies the potential for the fourth strategy. The other reason of the improvement of ASCEND-H is such a judgment on *astar*.

Nevertheless, the existence of applications such as *astar* is against intuition, If the core has borrowing region and lending region at the same time, it is natural to use the lending region by the core itself. I am not sure what behavior of *astar* causes such a judgment. If we analyze its behavior more closely, we may make a new discovery about the demands on capacity of caches and thus improve the efficiency of caches more.

5.4.2 Should we consider the distance between cores?

In the many-core environment, if a requested cache line is found in a remote cache, its latency varies according to positional relation among the requesting core, the corresponding directory, and the core holding the data. Figure 5.9 shows examples of such variation. In the explanation below, I express a core and a directory located at a coordinate (x, y) as Core (x, y) and Directory (x, y), respectively.

In both cases, Core (1, 8) sends a request to Directory (3, 5). I assume the requested line is found in a remote core that is different between the examples: Core (2, 7) in example (a) and Core (4, 1) in example (b).

A series of communication is shown by the arrows of Request, Forward, and Data. The total numbers

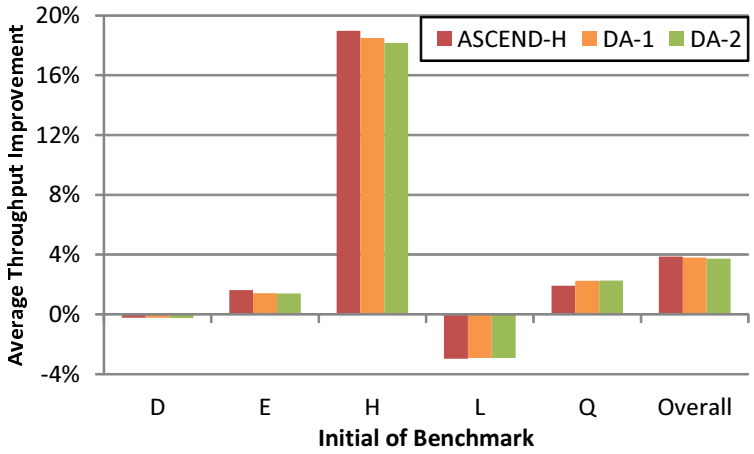


Figure 5.10 Throughput of distance-aware ASCEND.

of hops required are 10 in (a) and 20 in (b). When I assume communication latency of 3 cycles per hop, the difference in latency between the examples is as long as 30 cycles.

This difference in latency reminds us that it may be needed to consider the distance between cores on spilling. In fact, a previous study [81] tried to improve the performance and reduce the power consumption of network by limiting the destination of spilled line to the nearby cores.

For this reason, I now define a variation of ASCEND, named DA-ASCEND (Distance-Aware ASCEND), and explore it. Since just limiting the destination does not suit ASCEND, I realize distance-awareness by giving priority in the selection of the destination.

I modify the Weighted Round-robin in the Receiver Selector for DA-ASCEND. In the original ASCEND, if multiple candidates for the destination are found, that is, multiple counters have the same highest value, the destination is selected by a fixed priority. In DA-ASCEND, at this time, when a nearby core is included in the candidates, it is preferentially chosen as the destination. This prioritization is realized by an additional bit to the LSB of each counter. The bit is set when the corresponding core is near the requesting core. Since I can predefine whether a pair of cores are near, it requires a ROM of the number of bits up to the square of the number of cores.

I evaluate the performance of DA-ASCEND. The environment and parameters are the same as the many-core evaluation. In addition to the highly-precise method of ASCEND (ASCEND-H) that is used as a baseline, I measure the throughputs of two kinds of DA-ASCEND: DA-1 represents DA-ASCEND that regards the cores within 1 hop from, that is, the left, right, top and bottom of the requesting core as nearby cores; DA-2 stands for DA-ASCEND that considers the cores within 2 hops from the requesting cores as nearby cores. The numbers of cores that is near a specific core in DA-1 and DA-2 is 2-4 and 5-11, respectively.

Figure 5.10 shows the result of evaluation for DA-ASCEND. Each group of bars except *Overall* represents the average throughput improvement of the corresponding application (not workloads including the corresponding application) over Private. *Overall* means the average of all the 70 workloads. Not being shown in the figure, the probabilities that the selectors got different destination by prioritizing nearby cores in DA-1 and DA-2 were 20% and 31%, respectively.

In respect of the overall performance, both DA-1 and DA-2 performed slightly worse than the baseline. The improvements of average throughput of the baseline ASCEND, DA-1, and DA-2 over Private were 3.9%, 3.8%, and 3.7%, respectively.

In particular, the performance was decreased in H (himeno). When I analyzed the time waiting for the data for each core, it was decreased in the cores near the center; however, it was equal to or increased in the cores near the corner. If the nearby cores are restricted to receiving, spiller cores do not benefit from distance-awareness. This may become the performance bottleneck.

On the other hand, DA-ASCEND showed better performance in Q (qsort). Since Q has different access pattern by core and time, distance-aware receiving often occur. As a result, it become easy to benefit from distance-awareness.

What this result implies is that the merit of distance-awareness on spilling is small at present. Whether distance-awareness perform well or not depends on the distribution of the demands on capacity. That is to say, it is possible that placing cores that run the same application in a mass perform worse than daring to distribute them. However, this is a kind of problems of task placement and should be studied in another time.

5.5 Summary

I showed the evaluation result for confirming the validity of ASCEND in this chapter. Comparing with DSR and ElasticCC that has either performance or scalability, ASCEND showed higher efficiency of caches than both of them. It is the proof of the achievement by ASCEND of both performance and scalability. In addition, I calculated the hardware overhead and clarified its small amount of additional hardware.

Also in this chapter, I make some discussion based on the results. The especially interesting discovery is the existence of applications that are suitable for spilling and receiving at the same time. Analyzing the behavior of such applications in detail may become a help to further improve the efficiency of caches.

Chapter 6

Conclusions

6.1 Concluding Remarks

The importance of the efficient management of the last level caches (LLCs) is increased for the future many-core era. I focused on the Distributed Cooperative Caching (DCC), which could achieve low latency, independence of performance, flexibility on capacity, and high scalability. For efficient use of the DCC, optimizing one of its features, spilling, is required. However, there were no previous methods that achieved both high efficiency and high scalability.

The contributions that this thesis provided were threefold:

- to classify recent methods for cache optimization and show a direction to researches on this area;
- to show the implementation of a useful simulation infrastructure for research on many-core processor with shared memory; and
- to show a highly-efficient and scalable spilling to make use of the scheme of cooperative caches.

As one of the secondary contribution, I classified many recent researches for improving caches from its focus and intention. Some of them utilized the technique and the knowledge in not only their own category but also other categories. I believe that such mixture becomes hints of efficient cache management. I would be glad if my classification become a clue of a novel approach.

As the other secondary contribution, I showed the implementation of an infrastructure for research on many-core processors with shared memory. Its literal *core* is a MIPS system simulator SimMips. In this thesis, I showed that the simulators were described carefully for usability and it could evaluate a 32-core many-core in practical time. The developed infrastructure will be published as free software, after arrangement and review of the source code.

The primary contribution was to show methods for the cooperative caches that achieved both high efficiency and high scalability. The proposed architecture, ASCEND, was one of them. ASCEND utilize the extra entries in the directories, which is also a feature of the DCC, as hints for analyzing the demands on capacity of the caches. Its controllers are called the Spiller Selector and the Receiver

Selector. It achieves the efficiency by estimating the behavior of caches more precisely with these two controllers. In addition, distributing the units in the chip and analyzing independently with each unit improves the scalability. I evaluated the efficiency in 4-core and 32-core environments. In the 4-core, my method was more efficient than an existing efficient but non-scalable method. In the 32-core, it outperformed a scalable but not-so-efficient method. From these results, I confirmed that ASCEND was good at both performance and scalability.

6.2 Open Research Areas

There are several directions of future researches in this area. The following are some of them:

- to relieve the limitations,
- to analyze the fourth category of applications in more detail,
- to combine with future interconnections, and
- to examine the feasibility with hardware prototyping.

I refer to respective directions below.

First of all, we should relieve the limitations of ASCEND. As we have seen in the evaluation, although ASCEND shows high performance and QoS, it still has some limitations. A relief from them obviously remains as future work for more efficient and scalable methods.

One of the limitations is that the fluctuation of judgment may reduce the performance gain for applications that are obviously suitable for just spilling or just receiving. This caused the performance loss over the DSR in some combinations of applications in the multicore environment. We should analyze the reason and give more precise standards for judgment.

Another limitation is that we can apply some additional techniques to ASCEND only with a specific cache replacement algorithm. ASCEND has a mechanism of cache inactivity that enhance the restriction of receiving according to the degree of cache reuse. However, this mechanism assumes a specific replacement algorithm and not applicable to the LRU, the most typical algorithm. The same holds true for the Weak Receiving. These might affect some application negatively in the many-core environment. We can modify them to be applicable to various replacement algorithms.

Secondly, the fourth category of applications that are found by ASCEND, that is, applications where spilling their lines and receiving lines of others at the same time contribute to the performance are worth being analyzed. The characteristics of the first and the second categories are extracted by the DSR. The importance of the third category where neither spilling nor receiving is appropriate has already been pointed out. However, the fourth category that I revealed has not yet been inspected. Finding other applications in this category and considering what kind of demand on capacity generates it are remained as future work.

Thirdly, I used just common organization and parameters in the interconnection among the cores. However, for future network-on-chips, lots of methods and architectures have been proposed in recent years. For example, a *photonic* network-on-chip [82] is said to achieve much higher bandwidth and lower power consumption than conventional *electrical* NoC. Such alternative interconnections can even change the concept of scalability. Hence, we should keep an eye on the trend and consider methods of cache optimization with the future interconnections.

Lastly, I used software simulators for the evaluations in this thesis. Even though I designed the proposed architecture in consideration of the feasibility in hardware, and confirmed its behavior by a cycle-level software simulator, more precise verification in feasibility requires hardware prototyping. As a hardware infrastructure of many-core prototyping, we have proposed ScalableCore System [83], which has been corresponding to SimMc. If we extend ScalableCore so that cooperative caches can work on it, and verify the feasibility of ASCEND, its availability will be further increased.

Acknowledgements

I would like to express my deep gratitude to Associate Prof. Kenji Kise. He has been my supervisor and has supported me for 6 years from a bachelor student to a doctoral student in Tokyo Tech. His constant support, guidance, and encouragement has been essential for me to complete my thesis. In addition, he provided me with a comfortable research environment in Kise Laboratory. I also would like to thank all the members at Kise Laboratory, particularly Mr. Shimpei Sato and Mr. Shinya Takamaeda–Yamazaki, for active discussions and helpful advice.

I would like to sincerely thank Prof. Haruo Yokota, Associate Prof. Suguru Saito, Associate Prof. Takuo Watanabe, and Lecturer Haruhiko Kaneko for careful review and fruitful suggestion as members of the thesis committee.

A part of the development of SimMips, the core of the simulation infrastructures, was supported by Core Research for Evolutional Science and Technology (CREST), JST.

While I have studied as a master and doctoral student, I had worked as a president of MieruPC Inc., a venture company based on MieruPC or a research accomplished in Kise Laboratory, for two years. I would like to deeply thank all the people who gave me some helpful feedback about it, in particular Associate Prof. Hironori Nakajo at Tokyo University of Agriculture and Technology and Lecturer Ryotaro Kobayashi at Toyohashi University of Technology. I would like to thank Ms. Makiko Kawabata at Tokyo Electron Device Ltd. for support in sale and spread of MieruPC. I'm also thankful to Mr. Ryosuke Sasakawa for taking over the operation of the company.

In addition, I would like to deeply thank stage managers at Chor Kleines. Working there taught me not only how to manage concerts but also a way for self-management.

Finally, I am deeply grateful to my parents for a wide variety of support during the years of my studies.

Bibliography

[1] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach Fifth Edition*. Morgan Kaufmann, 2011.

[2] Samira Manabi Khan, Yingying Tian, and Daniel A. Jimenez. Sampling Dead Block Prediction for Last-Level Caches. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 175–186, 2010.

[3] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th annual international symposium on Computer architecture*, pp. 60–71, 2010.

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

[5] Jichuan Chang and Gurindar S. Sohi. Cooperative Caching for Chip Multiprocessors. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, pp. 264–276, 2006.

[6] Enric Herrero, José González, and Ramon Canal. Distributed cooperative caching. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pp. 134–143, 2008.

[7] Bradford M. Beckmann, Michael R. Marty, and David A. Wood. ASR: Adaptive Selective Replication for CMP Caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 443–454, 2006.

[8] M.K. Qureshi. Adaptive Spill-Receive for robust high-performance caching in CMPs. In *Proceedings of the 15th IEEE International Symposium on High Performance Computer Architecture*, pp. 45–54, 2009.

[9] Enric Herrero, José González, and Ramon Canal. Elastic cooperative caching: an autonomous dynamically adaptive memory hierarchy for chip multiprocessors. In *Proceedings of the 37th annual international symposium on Computer architecture*, pp. 419–428, 2010.

[10] Dyer Rolan, Basilio B. Fraguela, and Ramon Doallo. Adaptive Set-Granular Cooperative Caching. In *Proceedings of the 18th IEEE International Symposium on High-Performance Com-*

puter Architecture, pp. 1–12, 2012.

[11] Naoki Fujieda, Shimpei Watanabe, and Kenji Kise. A MIPS System Simulator SimMips for Education and Research of Computer Science. *IPSJ Journal*, Vol. 50, No. 11, pp. 2665–2676, 2009.

[12] An-Chow Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proceedings of the 27th annual international symposium on Computer architecture*, pp. 139–148, 2000.

[13] Alvin R. Lebeck and David A. Wood. Dynamic self-invalidation: reducing coherence overhead in shared-memory multiprocessors. In *Proceedings of the 22nd annual international symposium on Computer architecture*, pp. 48–59, 1995.

[14] An-Chow Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proceedings of the 28th annual international symposium on Computer architecture*, pp. 144–154, 2001.

[15] Samira M. Khan, Daniel A. Jiménez, Doug Burger, and Babak Falsafi. Using dead blocks as a virtual victim cache. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pp. 489–500, 2010.

[16] Jaume Abella, Antonio González, Xavier Vera, and Michael F. P. O'Boyle. IATAC: a smart predictor to turn-off L2 cache lines. *ACM Transactions on Architecture and Code Optimization*, Vol. 2, No. 1, pp. 55–77, 2005.

[17] M. Kharbutli and Yan Solihin. Counter-Based Cache Replacement and Bypassing Algorithms. *IEEE Transactions on Computers*, Vol. 57, No. 4, pp. 433–447, 2008.

[18] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pp. 222–233, 2008.

[19] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, Vol. 5, No. 2, pp. 78–101, 1966.

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

[21] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache replacement based on reuse-distance prediction. In *Proceedings of 25th IEEE International Conference on Computer Design*, pp. 245–250, 2007.

[22] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. A Case for MLP-Aware Cache Replacement. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, pp. 167–178, 2006.

[23] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th annual international symposium on Computer architecture*, pp. 381–391, 2007.

[24] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, Jr., and Joel Emer. SHiP: signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 430–441, 2011.

[25] Daniel Sanchez and Christos Kozyrakis. The ZCache: Decoupling Ways and Associativity. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 187–198, 2010.

[26] Sun Microsystems. *UltraSPARC T2 supplement to the UltraSPARC architecture 2007*, 2007.

[27] Anant Agarwal and Stephen D. Pudar. Column-associative caches: a technique for reducing the miss rate of direct-mapped caches. In *Proceedings of the 20th annual international symposium on computer architecture*, pp. 179–190, 1993.

[28] André Seznec. A case for two-way skewed-associative caches. In *Proceedings of the 20th annual international symposium on computer architecture*, pp. 169–178, 1993.

[29] Moinuddin K. Qureshi, David Thompson, and Yale N. Patt. The V-Way Cache: Demand Based Associativity via Global Replacement. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, pp. 544–555, 2005.

[30] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, Jr., and Joel Emer. Adaptive insertion policies for managing shared caches. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pp. 208–219, 2008.

[31] Yuejian Xie and Gabriel H. Loh. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the 36th annual international symposium on Computer architecture*, pp. 174–183, 2009.

[32] G.E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the 8th IEEE International Symposium on High-Performance Computer Architecture*, pp. 117–128, 2002.

[33] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic Partitioning of Shared Cache Memory. *Journal of Supercomputing*, Vol. 28, No. 1, pp. 7–26, 2004.

[34] Haakon Dybdahl, Per Stenström, and Lasse Natvig. A cache-partitioning aware replacement policy for chip multiprocessors. In *Proceedings of the 13th international conference on High Performance Computing*, pp. 22–34, 2006.

[35] Aamer Jaleel, Hashem H. Najaf-abadi, Samantika Subramaniam, Simon C. Steely, and Joel Emer. CRUISE: cache replacement and utility-aware scheduling. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating*

Systems, pp. 249–260, 2012.

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

[37] Jichuan Chang and Gurindar S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proceedings of the 21st annual international conference on Supercomputing*, pp. 242–252, 2007.

[38] Ravi Iyer. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *Proceedings of the 18th annual international conference on Supercomputing*, pp. 257–266, 2004.

[39] Kyle J. Nesbit, James Laudon, and James E. Smith. Virtual private caches. *SIGARCH Comput. Archit. News*, Vol. 35, No. 2, pp. 57–68, 2007.

[40] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proceedings of the 14th IEEE International Symposium on High-Performance Computer Architecture*, pp. 367–378, 2008.

[41] Lisa R. Hsu, Steven K. Reinhardt, Ravishankar Iyer, and Srihari Makineni. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pp. 13–22, 2006.

[42] Derek Chiou, Larry Rudolph, Srinivas Devadas, and Boon S. Ang. Dynamic Cache Partitioning via Columnization. Technical Report 430, MIT Laboratory for Computer Science Computation Structures Group, 2000.

[43] Parthasarathy Ranganathan, Sarita Adve, and Norman P. Jouppi. Reconfigurable caches and their application to media processing. In *Proceedings of the 27th annual international symposium on Computer architecture*, pp. 214–224, 2000.

[44] Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. Architectural support for operating system-driven CMP cache management. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pp. 2–12, 2006.

[45] Chun Liu, Anand Sivasubramaniam, and Mahmut Kandemir. Organizing the Last Line of Defense before Hitting the Memory Wall for CMPs. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, pp. 176–185, 2004.

[46] Edouard Bugnion, Jennifer M. Anderson, Todd C. Mowry, Mendel Rosenblum, and Monica S. Lam. Compiler-directed page coloring for multiprocessors. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pp. 244–255, 1996.

[47] Daniel Sanchez and Christos Kozyrakis. Vantage: Scalable and Efficient Fine-Grain Cache Par-

titioning. In *Proceedings of the 38th annual international symposium on Computer architecture*, pp. 57–68, 2011.

[48] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pp. 211–222, 2002.

[49] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. Distance Associativity for High-Performance Energy-Efficient Non-Uniform Cache Architectures. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, pp. 55–, 2003.

[50] B.M. Beckmann and D.A. Wood. Managing Wire Delay in Large Chip-Multiprocessor Caches. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pp. 319–330, 2004.

[51] Jaehyuk Huh, Changkyu Kim, Hazim Shafi, Lixin Zhang, Doug Burger, and Stephen W. Keckler. A NUCA substrate for flexible CMP cache sharing. In *Proceedings of the 19th annual international conference on Supercomputing*, pp. 31–40, 2005.

[52] Mahmut Kandemir, Feihui Li, Mary Jane Irwin, and Seung Woo Son. A novel migration-based NUCA design for chip multiprocessors. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pp. 28:1–28:12, 2008.

[53] Michael Zhang and Krste Asanovic. Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, pp. 336–345, 2005.

[54] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th annual international symposium on Computer Architecture*, pp. 364–373, 1990.

[55] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. Optimizing Replication, Communication, and Capacity Allocation in CMPs. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, pp. 357–368, 2005.

[56] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive NUCA: near-optimal block placement and replication in distributed caches. In *Proceedings of the 36th annual international symposium on Computer architecture*, pp. 184–195, 2009.

[57] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, Vol. 25, No. 3, pp. 63–79, 1992.

[58] E. Hagersten, A. Landin, and S. Haridi. DDM - a cache-only memory architecture. *IEEE Computer*, Vol. 25, No. 9, pp. 44–54, 1992.

[59] Milo M. K. Martin, Mark D. Hill, and David A. Wood. Token coherence: decoupling perfor-

mance and correctness. In *Proceedings of the 30th annual international symposium on Computer architecture*, pp. 182–193, 2003.

[60] Hongzhou Zhao, Arrvindh Shriraman, and Sandhya Dwarkadas. SPACE: sharing pattern-based directory coherence for multicore scalability. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pp. 135–146, 2010.

[61] John H. Kelm, Matthew R. Johnson, Steven S. Lumettta, and Sanjay J. Patel. WAYPOINT: scaling coherence to thousand-core architectures. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pp. 99–110, 2010.

[62] Blas A. Cuesta, Alberto Ros, María E. Gómez, Antonio Robles, and José F. Duato. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *Proceedings of the 38th annual international symposium on Computer architecture*, pp. 93–104, 2011.

[63] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative caching: using remote client memory to improve file system performance. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, 1994.

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

[65] Naoki Fujieda and Kenji Kise. A Partitioning Method of Cooperative Caching with Hit Frequency Counters for Many-Core Processors. In *Proceedings of the 3rd Workshop on Ultra Performance and Dependable Acceleration Systems held in conjunction with ICNC'11*, pp. 160–165, 2011.

[66] William J. Dally and Brian Towles. Route packets, not wires: on-chip inteconnection networks. In *Proceedings of the 38th annual Design Automation Conference*, pp. 684–689, 2001.

[67] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using SimPoint for accurate and efficient simulation. In *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pp. 318–319, 2003.

[68] Koh Uehara, Shimpei Sato, and Kenji Kise. A Practical Infrastructure for Researches and Education of Many-Core Processors. *IEICE Transactions on Information and Systems*, Vol. J93-D, No. 10, pp. 2042–2057, 2010.

[69] Aamer Jaleel, Robert S. Cohn, Chi-Keung Luk, and Bruce Jacob. CMP$im: A Pin-Based On-The-Fly Multi-Core Cache Simulator. In *Proceedings of the 4th Annual Workshop on Modeling, Benchmarking, and Simulation*, 2008.

[70] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general

execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Computer Architecture News*, Vol. 33, No. 4, pp. 92–99, 2005.

[71] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, Vol. 35, No. 2, pp. 50–58, 2002.

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

[73] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, Vol. 39, No. 2, pp. 1–7, 2011.

[74] Alaa R. Alameldeen, Aamer Jaleel, Moinuddin Qureshi, and Joel Emer. 1st JILP Workshop on Computer Architecture Competitions (JWAC-1) Cache Replacement Championship. http://www.jilp.org/jwac-1/.

[75] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions (Extended Abstract). In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pp. 106–112, 1977.

[76] D. E. Culler, A. Gupta, and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1999.

[77] Advanced Center for Computing and Communication, RIKEN. Himeno benchmark. http://accc.riken.jp/2444.htm.

[78] Naoki Fujieda and Kenji Kise. A Method for Efficient Use of CMP Cooperative Caching with Extra Entries of Directory. *IPSJ Transaction on Advanced Computing Systems*, Vol. 5, No. 3, pp. 86–100, 2012.

[79] Dan Nicolaescu, Alex Veidenbaum, and Alex Nicolau. Reducing data cache energy consumption via cached load/store queue. In *Proceedings of the 2003 international symposium on Low power electronics and design*, pp. 252–257, 2003.

[80] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis. Weighted round-robin cell multiplexing in a general-purpose ATM switch chip. *Selected Areas in Communications, IEEE Journal on*, Vol. 9, No. 8, pp. 1265 –1279, 1991.

[81] Enric Herrero, José González, and Ramon Canal. Power-Efficient Spilling Techniques for Chip Multiprocessors. In *International European Conference on Parallel and Distributed Computing*, pp. 256–267, 2010.

[82] Assaf Shacham, Keren Bergman, and Luca P. Carloni. On the Design of a Photonic Network-

on-Chip. In *Proceedings of the First International Symposium on Networks-on-Chip*, pp. 53–64, 2007.

[83] Shinya Takamaeda, Shimpei Sato, Naoki Fujieda, Takefumi Miyoshi, and Kenji Kise. ScalableCore System: Hardware Environment for Many-core Architectures Evaluation. *IPSJ Transaction on Advanced Computing Systems*, Vol. 4, No. 1, pp. 24–42, 2011.

List of Publications

Journal Papers

1. Naoki Fujieda, and Kenji Kise: A Method for Efficient Use of CMP Cooperative Caching with Extra Entries of Directory, IPSJ Transaction on Advanced Computing Systems, Vol.5, No.3, pp.86-100 (2012) (In Japanese).
2. Naoki Fujieda, Shimpei Watanabe, and Kenji Kise: A MIPS System Simulator SimMips for Education and Research of Computer Science, IPSJ Journal, Vol.50, No.11, pp.2665-2676 (2009) (In Japanese).
3. Shinya Takamaeda, Shimpei Sato, Naoki Fujieda, Takefumi Miyoshi, and Kenji Kise: ScalableCore System: Hardware Environment for Many-core Architectures Evaluation, IPSJ Transaction on Advanced Computing Systems, Vol.4, No.1, pp. 24-42 (2011) (In Japanese).
4. Shimpei Sato, Naoki Fujieda, Akira Moriya, and Kenji Kise: SimCell: A Processor Simulator for Multi-Core Architecture Research, IPSJ Transactions on Advanced Computing Systems, Vol.2, No.1, pp. 146-157 (2009), **IPSJ 論文船井若手奨励賞**.

International Conference Papers

5. Naoki Fujieda and Kenji Kise: A Partitioning Method of Cooperative Caching with Hit Frequency Counters for Many-Core Processors, *Proc. 3rd Workshop on Ultra Performance and Dependable Acceleration Systems held in conjunction with ICNC'11*, pp.160-165 (2011).
6. Naoki Fujieda, Takefumi Miyoshi, and Kenji Kise: SimMips: A MIPS System Simulator, *Proc. Workshop on Computer Architecture Education(WCAE) held in conjunction with MICRO-42*, pp. 32-39 (2009).
7. Takakazu Ikeda, Shinya Takamaeda-Yamazaki, Naoki Fujieda, Shimpei Sato, and Kenji Kise: Request Dentity Aware Fair Memory Scheduling, *Proc. 3rd JILP Workshop on Computer Architecture Competitions: Memory Scheduling Championship in conjunction with ISCA-39*, Available at http://www.cs.utah.edu/ rajeev/jwac12/ (2012), **Performance Track Award**.
8. Shinya Takamaeda, Shintaro Sano, Yoshito Sakaguchi, Naoki Fujieda, and Kenji Kise: ScalableCore System: A Scalable Many-core Simulator by Employing Over 100 FPGA, *Proc. 8th*

International Symposium on Applied Reconfigurable Computing (ARC2011), Lecture Notes in Computer Science, Vol.7199/2012, pp.138-150 (2012).

9. Mochamad Asri, Naoki Fujieda, and Kenji Kise: Rethinking Processor Instruction Fetch: Inefficiencies-Cracking Mechanism, *Proc. 2011 International SoC Design Conference (ISOCC2011)*, pp.207-210 (2011).
10. Yuhta Wakasugi, Naoki Fujieda, Shinya Takamaeda, and Kenji Kise: MipsCoreDuo: A Multi-function Dual-core Processor, *Proc. International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS)*, pp. 587-590 (2009).
11. Shimpei Sato, Naoki Fujieda, Akira Moriya, and Kenji Kise: Processor Simulator SimCell to Accelerate Research on Many-core Processor Architectures, *Proc. Workshop on Cell Systems and Applications (WCSA 2008) held in conjunction with the ISCA-2008*, pp. 119-127 (2008).

Domestic Conference Papers and Technical Reports

12. Naoki Fujieda and Kenji Kise: The cache partitioning method for CMP cooperative caching, *IPSJ SIG Technical Reports 2011-ARC-196*, pp. 1-8 (2011) (In Japanese).
13. Naoki Fujieda and Kenji Kise: Implementation and Evaluation of a Fast and Handy LCD Module Using an FPGA, *IEICE Technical Reports RECONF2010-82*, pp. 193-198 (2011) (In Japanese).
14. Naoki Fujieda, Shimpei Watanabe, and Kenji Kise: SimMips: A 5000-line MIPS System Simulator that Runs Linux for Education and Research of Computer Science, *Proc. 20th Computer System Symposium (ComSys2008)*, pp. 143-150 (2008) (In Japanese).
15. Shinya Takamaeda, Shimpei Watanabe, Ken Kyou, Koh Uehara, Naoki Fujieda, Takefumi Miyoshi, and Kenji Kise: Development of ScalableCore System to Evaluate Many-core Architectures, *Proc. 8th Symposium on Advanced Computing Systems and Infrastractures (SACSIS2010)*, pp.287-294 (2010) (In Japanese).
16. Shinya Takamaeda, Shimpei Watanabe, Ken Kyou, Naoki Fujieda, Koh Uehara, Takefumi Miyoshi, and Kenji Kise: ScalableCore system: Scalable HW Evaluation Environment for Many-core Architecture Researches, *IPSJ SIG Technical Reports 2009-ARC-185*, pp. 1-10 (2009) (In Japanese).
17. Yuhta Wakasugi, Shimpei Sato, Koh Uehara, Naoki Fujieda, Shimpei Watanabe, Shinya Takamaeda, Yosuke Mori, and Kenji Kise: Development of Low-Cost Verification Scheme for VDEC-assisted Prototype Chip and its Application, *IPSJ SIG Technical Reports 2009-ARC-183*, pp. 1-8 (2009) (In Japanese).
18. Yuhta Wakasugi, Naoki Fujieda, and Kenji Kise: アーキテクチャ研究をサポートする低コ

ストで効率的な VDEC チップ試作・検証システムの開発と応用, *Proc. 7th Symposium on Advanced Computing Systems and Infrastractures (SACSIS2009)*, Poster Session, pp. 134-135, (2009) (In Japanese), **Most Interesting Poster Award**.

19. Yosuke Mori, Akira Moriya, Naoki Fujieda, and Kenji Kise: The Cache-Core optimization on Multi-Core Processors considering several overheads, *IPSJ SIG Technical Reports 2009-ARC-181*, pp.105-110 (2009) (In Japanese).
20. Kenji Kise, Shimpei Sato, Akira Moriya, Naoki Fujieda, Yuhta Wakasugi, Shimpei Watanabe, Koh Uehara, Yosuke Mori, Shinya Takamaeda, Tomohide Takahashi, Tomonari Muneoka, Yusuke Yamada, Katsuhiko Gondow, Ryotaro Kobayashi, Takefumi Miyoshi, and Hironori Nakajo: MieruPC プロジェクト: 中身が見える計算機システムを構築する研究・教育プロジェクト, *20th Computer System Symposium (ComSys2008)*, Poster Session (2008) (In Japanese), **Best Poster Award**.
21. Shimpei Watanabe, Naoki Fujieda, Yuhta Wakasugi, Shinya Takamaeda, Yosuke Mori, and Kenji Kise: Development of Simple Embedded System with MIPS System Simulator SimMips, *IPSJ SIG Technical Reports 2008-EMB-10*, pp. 23-28 (2008) (In Japanese).
22. Koh Uehara, Shimpei Sato, Akira Moriya, Naoki Fujieda, Shinya Takamaeda, Shimpei Watanabe, Takefumi Miyoshi, Ryotaro Kobayashi, and Kenji Kise: Development of Simple and Effective Many-Core Architecture, *IPSJ SIG Technical Reports 2008-ARC-180*, pp. 39-44 (2008) (In Japanese).
23. Akira Moriya, Naoki Fujieda, Shimpei Sato, and Kenji Kise: The multi-function cache core architecture to enhance the memory performance on many-core processors, *Proc. 6th Symposium on Advanced Computing Systems and Infrastractures (SACSIS2008)*, pp. 421-430 (2008) (In Japanese).