Enabling Software Management for Multicore Caches with a Lightweight Hardware Support

Jiang Lin[1], Qingda Lu[2], Xiaoning Ding[2], Zhao Zhang[1], Xiaodong Zhang[2] and P. Sadayappan[2]

[1]Dept. of Electrical and Computer Engineering
Iowa State University
Ames, IA 50011
{linj,zzhang}@iastate.edu

[2] Dept. of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{luq,dingxn,zhang,saday}@cse.ohio-state.edu

Abstract

The management of shared caches in multicore processors is a critical and challenging task. Many hardware and OS-based methods have been proposed. However, they may be hardly adopted in practice due to their non-trivial overheads, high complexities, and/or limited abilities to handle increasingly complicated scenarios of cache contention caused by many-cores.

In order to turn cache partitioning methods into reality in the management of multicore processors, we propose to provide an affordable and lightweight hardware support to coordinate with OS-based cache management policies. The proposed methods are scalable to many-cores, and perform comparably with other proposed hardware solutions, but have much lower overheads, therefore can be easily adopted in commodity processors. Having conducted extensive experiments with 37 multi-programming workloads, we show the effectiveness and scalability of the proposed methods. For example on 8-core systems, one of our proposed policies improves performance over LRU-based hardware cache management by 14.5% on average and up to 47.5%.

Categories and Subject Descriptors: B.3.2 [Primary Memory]: Design Styles

General Terms: Design, Performance

Keywords: Shared Cache, Cache Management, Multicore

1. Introduction

With a continuous increase of the number of cores on a single processor chip, the management of on-chip resources has become critical and challenging in order to achieve high performance and power efficiency in chip multiprocessors. Since on-chip caches play an important role to bridge the ever-increasing gap between the processor and DRAM, efficient utilization of caches continues to be crucial to achieving high performance by minimizing access conflicts and unfairness, and by maintaining a high quality of services for chip multiprocessors. In existing commercial multicore processors (e.g., [8, 11, 7]), the last-level cache is shared by multiple cores. This design has several merits, such as increasing the cache utilization, reducing cache coherence complexity, and creating a fast communication mechanism among cores by using the shared cache. Given the memory bandwidth limit of multicore processors, shared caches have a better potential to minimize misses than that in private caches. However, existing multicore processors do not have any control over allocating shared cache resources among simultaneously running threads, which has caused significant performance concerns in practice. If this issue is not well and timely addressed, the performance and scalability potentials of multicore processors will be seriously limited.

A number of hardware designs have recently been proposed for shared cache management [20, 10, 16, 5, 17]. The mechanisms in these proposals require to modify the standard LRU replacement policy and to track the thread ownership of each cache line. With these mechanisms, cache partition decisions made by cache management policies can be enforced at the cache set level (way partitioning) or at the whole cache level. There are two major limitations that hinder these hardware solutions to be adopted in practice. First, since the number of cores will continue to increase while the degree of cache associativity (the cache ways) is limited (normally up to 16), way partitioning will not be scalable to future multicore processors. Second, the extra hardware complexity of altering the LRU replacement policy and tracking the ownership of each cache line is non-trivial, not only increasing the chip overhead but also complicating design verification. In addition, for any hardware cache management approach, there inevitably are some worst-case scenarios of cache usage causing severe cache thrashing, which occurs even with uniprocessors. It is expected to be more severe on multicore processors because of reduced cache capacity per core, interference due to shared caches, and limited memory bandwidth per core. Hardware designs still work for scenarios they are designed for, but the lack of flexibility can be an unavoidable issue and inherent weakness, particularly for future multicore processors with an increasingly large number of cores.

In contrast, cache optimization and cache resource management at different levels of software, such as operating systems, compilers, and application programs have shown their effective-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SC09 November 14-20, 2009, Portland, Oregon, USA (c) 2009 ACM 978-1-60558-744-8/09/11... $10.00.

ness to address the limitations of hardware solutions. With a software approach, long-term memory access patterns of most applications can be analyzed or predicted, thus, cache management and optimization decisions can be made more effectively. There have been several successful examples on uniprocessors with simple LRU cache replacement policy (e.g., [12, 15, 4, 1]). However, using a software approach to managing cache resources in multicore processors is much more challenging than in uniprocessors as hardware resources are almost not shared and coordinated for multi-threads. Researchers have evaluated OS-based cache partitioning methods in multicore processors without any additional hardware support [13]. These OS-based methods offer the flexibility in implementing various resource allocation policies. However, since hardware caches are not in the scope of OS management, OS-based methods can inevitably cause non-trivial software overhead, and are not ready to be used as a general management vehicle in practice.

In this paper, we present a hybrid approach to addressing the limitations of both hardware solutions and OS-based methods. Specifically, our multicore shared cache management framework consists of two low-cost and effective components: a *lightweight hardware mechanism* for allocating cache resources and for providing cache usage information; and *OS-based resource allocation policies* for dynamic cache allocation. With a simple and low overhead hardware component, we enable direct OS control over shared caches, software system overhead is minimized. With an OS-based management, we are able to design and implement multiple policies to deal with complicated, difficult cache usage scenarios in multicore systems. Given the large space of software policy design and application scenarios, it is infeasible to provide a comprehensive evaluation of this approach in a single study. Instead, we seek to answer a crucial question: can our hybrid approach perform competitively with pure hardware implementations in their favored scenarios? If the answer is affirmative, then the OS-based cache management with a simple hardware support is a truly viable approach. More research can then be conducted to fully explore its potential for those difficult cache sharing scenarios, which inherently favors software methods over hardware ones.

In this study, we make the following contributions by giving a proof-of-concept design:

1. We have identified several critical system architecture issues concerning the shared cache management, and effectively explored the hardware/software boundary to address these issues.

2. We have proposed a simple and efficient hardware mechanism for cache allocation, whose functionality is sufficient for effective software management.

3. We have shown that sophisticated software cache management is possible, and that its performance overhead is negligible with the hardware support.

Here, we summarize the distinguished features of our design and provide complete design details in Sections 2.1 and 4. First, explicit address mapping from physical address to cache address [18, 14] is employed for cache allocation. While the idea was proposed to reduce cache conflicts on uniprocessors, we use it to maps a memory page explicitly to a cache region. There is virtually no overhead in cache access time, and we use an optimized design to reduce the implementation overhead and to avoid cache coherence issues in the original designs. Second, dynamic cache re-allocation is supported with high efficiency. The execution of software policy is triggered at short intervals (10ms in the default setting) to check for the need for cache re-allocation. The interval is short enough to catch program phase changes, but long enough to avoid significant checking overhead. The hardware support avoids expensive data movement, and the software activates cache re-allocation at carefully selected times to contain cache invalidation/reloading cost.

We propose and evaluate two software policies: Process-level Cache Management policy (*PCM*) and Memory Region-level Cache Management policy (*MCM*). Our simulation-based experiments show that performance improvement over the baseline, shared cache with LRU replacement, is comparable to previously reported results with hardware cache partitioning. The PCM policy improves performance by 4.3% and 13.1% on average, on 2-core and 4-core systems, respectively. More importantly, the policy scales well with the increase in number of processor cores. The improvement on an 8-core system is 14.5% on average and up to 47.5%. We would like to highlight that there is a large design space of software policies and the proposed policies can be further optimized; nevertheless, they have served the purpose to demonstrate a working framework. In addition, we did not attempt to make a direct performance comparison with pure hardware partitioning schemes because of the complexity involved in a fair reproduction of their experiments. Our recent study on real systems [13] does show that the performance improvement of software cache management is comparable to that of hardware partitioning.

This work is unique among the existing studies on multicore cache management involving softwares. Rafique et al. [17] propose an OS cache management approach in which a hardware mechanism enforces cache quota policies and the OS decides the quotas and selects a policy. Here is a critical difference in this study. The hardware mechanism they use is similar to other hardware cache management designs, which requires modificitons of the LRU replacement policy and tracking the cache line ownership. In contrast, our hardware mechanism does not enforce quota but allows the software to enforce it, moving the complexity to software with virtually no performance overhead. We believe this is an optimal hardware/software boundary in multicore cache management. Another study [6] investigates broad design issues in shared cache management through OS-level page allocation. Our study gives a proof-of-concept design, resolves all working details, and demonstrates a particular framework that can be easily adopted in real systems. We have implemented software cache management in real systems using OS page coloring without special hardware support [13]. It was for evaluation only; hardware support is needed for general applications to avoid I/O penalty from page coloring. Furthermore, the hardware support proposed in this study eliminates the data movement overhead during cache re-allocation, and therefore allows cache re-allocation at a much finer time granularity.

The rest of the paper is organized as follows. Section 2.1 presents an overview and our proof-of-concept design of the framework. Section 3 and Section 4 describe the hardware mechanism and a set of software cache management policies of the design. Section 5 discusses the experimental environment. Section 6 presents experimental results and provides detailed analysis. Finally, Section 7 discusses the related work and Section 8 summarizes this study.

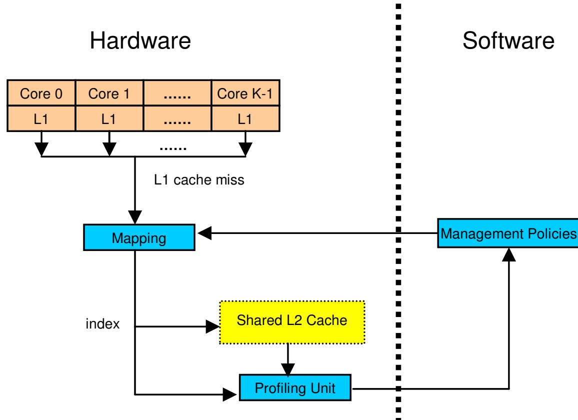


Figure 1. Overview of hybrid cache management scheme.

2. The Hybrid Cache Management Framework

In this section, we give an overview of the framework, and then give a proof-of-concept design of the framework.

2.1 Motivation and Overview of the Framework

Figure 1 illustrates the framework of the proposed hybrid scheme. As shown in the figure, we assume each core has private L1 caches, and the 2nd level (L2) cache is shared by all processor cores.

The key of the framework is exposing shared L2 cache to software (or OS). In an attempt of managing shared cache in OS without any additional hardware support, we have leveraged a well known technique called page coloring to partition shared L2 cache [13]. Intuitively, we can divide a shared cache into N colors, where N is determined by architectural parameters (N = number of cache sets $\times$ set associativity / OS page size). All cache lines in an OS page are cached in one of the N cache colors, determined by the cache color bits of the OS page which is part of physical page number. By controlling the virtual to physical page mapping, OS has the ability to assign cache color of a virtual page. Therefore, OS is able to limit the hardware cache space usage of a process by manipulating the address mapping of all virtual pages of the process. However, there are two limitations of the cache management mechanism in our previous work [13]: 1) the cache and memory co-partitioning (in order to allocate a large cache space to a program, large memory has to be reserved to the program); and 2) a high overhead of page recoloring, which is needed for any dynamic partitioning policy. Page recoloring is expensive because it requires copying the data of whole virtual page from an old physical page to a new physical page. That the source of the above limitations is – physical addresses are directly used to index the L2 cache.

Having observed the root cause of the limitations, we add a new layer of mapping from OS page to cache color to address the issues. An shown in Figure 2, a mapping unit is in charge of the new layer of mapping, which is controlled by software (OS) cache management policies. In this way, we expose the hardware shared cache into the scope of OS management. In addition to the mapping unit, we add a hardware profiling unit to assist software to make effective decisions. The profiling unit provides insightful run-time information of the L2 cache.

Given the above generic framework, designers could have many design choices. Nevertheless, The viable designs should meet the following requirements:

1. *Small Hardware Overhead:* The mapping unit and profiling unit should have small hardware overheads in order to be feasible to add into the processor chip.
2. *Small Performance Overhead:* The mapping unit should not introduce an additional latency for common L2 accesses because of the added layer of mapping.
3. *Small Software Overhead:* The software management policies should not have high execution overhead.
4. *Sufficient Information:* The profiling units should provide sufficient information for the software management policies.

2.2 A Proof-of-Concept Design

A straightforward design of the mapping unit is a mapping table which gives the cache colors of all physical pages managed by OS. Given the large capacity of main memories nowadays, such a mapping table is too large to put into the hardware. In order to meet the small hardware overhead requirement, we introduce the concept of *memory region*. A memory region refers to a group of physical pages that share the k least significant bits of page number, where k is a design choice. As shown in Figure 2, instead of having a mapping entry for each page, a *Region Mapping Table* has a mapping table entry for each memory region, which contains a set of pages. Therefore, the mapping table has 2^k entries. If k is reasonably small, the region mapping table is feasible to be added into the hardware. Note that region mapping table is not the only solution to meet the small hardware overhead requirement. There are other alternatives. For example, we can still maintain a mapping for each individual page, and cache only a small portion of the mapping table in the hardware. We choose the region mapping table-based design in this study, and plan to study other alternatives in the future.

In order to meet the small performance overhead requirement, we make an effort to avoid the extra delay introduced by the mapping table for the majority of L2 cache accesses. We add a cache color field into each TLB entry to buffer the cache color of the corresponding page. It is an optimized design of decoupled cache address mapping [18, 14] (see Section 3 for more discussion).

In order to provide sufficient information to software cache management policy, a *profiling unit* is added to collect runtime information of each memory region. More details of the profiling unit will be given in Section 3.

The *cache management policies* are implemented as operating system modules. OS invokes the chosen cache management policy periodically. A period of 10ms is chosen in this study to meet the small software overhead requirement as well as to capture the dynamic behavior of workloads. During each iteration, the cache management policy first reads runtime statistics provided by the profiling unit, then makes cache partitioning/sharing decisions with its optimization objective, and finally it reconfigures the region mapping table to enforce such decisions.

We discuss our design in detail in the following two sections.

3. Cache Management Mechanism

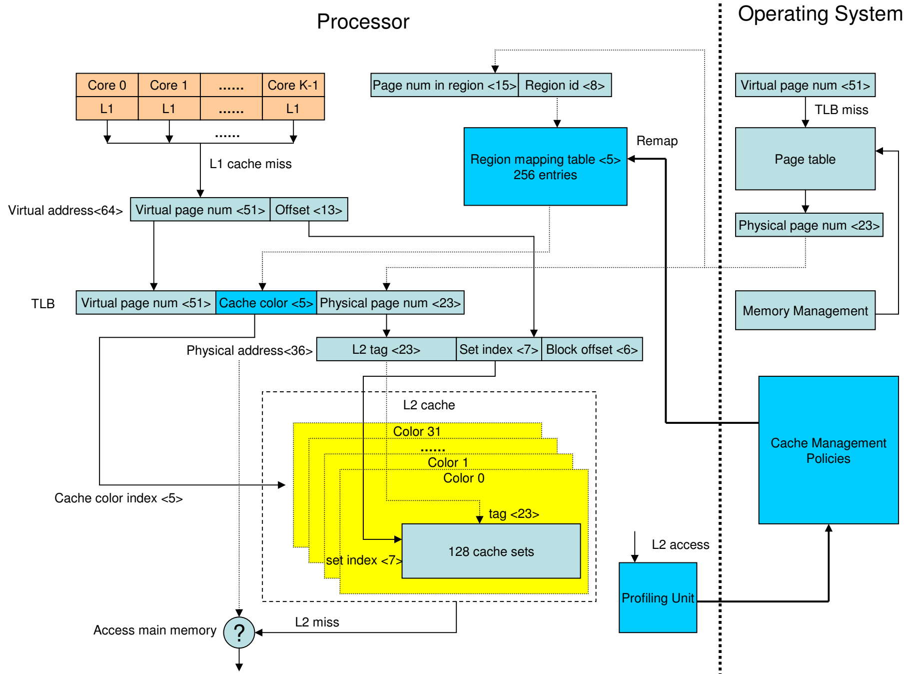


Figure 2. Framework of hybrid cache management scheme.

3.1 Region Mapping Table

In order to address the limitations with pure software-based cache partitioning [13] by removing expensive page migration costs, we introduce flexible cache indexing by adding a region mapping table, which maps OS pages to cache colors. Because it is impractical to have an entry in the table for every memory page, we group memory pages into M memory regions. For example, in a system shown in Figure 2 which has a shared 4MB L2 cache with 32 cache colors (the OS page size of the system is 8KB), we group pages into M=256 memory regions so that the mapping table has 256 entries. Here M is a design choice: it should be small enough to be put into hardware and big enough to avoid too many pages in a memory region. We refer the eight least significant bits of physical page number as *region id*. Physical pages that share the same region id belong to the same memory region. As the system has 4GB memory with 8K OS page size, there are 2048 pages in each memory region. At runtime, the cache management module assigns a cache color to the 2048 pages by configuring the corresponding entry in the region mapping table. The region mapping table eliminates the expensive costs with page migration.

3.2 Address Translation

In a typical computer system that supports virtual memory, a TLB (Translation Look-aside Buffer) is used to buffer recent virtual-to-physical address translations. To translate a virtual address to a physical address, ATU (Address Translation Unit) in hardware first looks up TLB. If the translation is found in TLB (TLB hit), it is read from TLB directly. Otherwise (TLB miss), the translation is read from the page table, and is buffered in the TLB for future references. The translated physical address is used to index caches and then to address the main memory if the cache access is a cache miss. [1]

With the region mapping table in our design, a physical page may be mapped to any cache color. To support such flexibility, we change the address translation procedure for cache indexing. In a naive design, ATU could look up the region mapping table for every L2 cache access; after TLB translates the virtual address to a physical address, ATU gets the cache color from the color mapping table indexed by the physical address. Nevertheless, this naive design may increase the L2 cache access latency if the region mapping table lookup is on the critical path. To avoid such additional delay, we add a cache color field to each TLB entry to buffer the cache color[2]. With the added field, for an L2 cache access that is a TLB hit, ATU obtains the cache color and the physical address from TLB simultaneously. For an L2 cache access that is a TLB miss, ATU first obtains the physical address from page table, and then obtains the cache color from the region mapping table. ATU uses both cache color and physical address to index cache: as shown in the Figure 2, ATU uses cache color to choose one of 32 colors of the L2 cache, and then uses the physical address to index and tag the cache block inside the cache color.

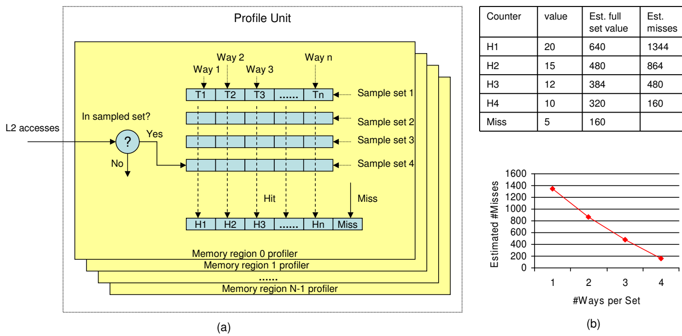
Our design has two advantages over prior page-based mapping designs in decoupling memory addressing and cache indexing [18, 14]. First, a page-based mapping design requires extra space in each page table entry to store the cache color of the page. The change of page table entry is expensive and not acceptable in many systems. In comparison, our memory region-based mapping design does not require changes in the page table, and the region mapping table is a small component. Furthermore, it takes much less time to go through the region mapping table than a page ta-

[1] The TLB access is required if the L1 cache is physically tagged, which is true in most modern systems.

[2] This optimization is similar to the one proposed by prior work [18, 14].

ble when cache remapping is needed. Second, in cache-coherent, shared-memory multi-processor systems that use physically indexed cache, the cache coherence hardware needs to check the last level cache by physical address. In page-based mapping designs [18, 14], to get the cache location (cache color) of a cache coherence access that is a TLB miss, cache coherence hardware needs to address the problem that multiple virtual pages may be mapped onto the same physical page. [14]. In comparison, in our memory region-based design, cache coherence hardware gets the cache color of the physical address from either TLB or the region mapping table, and then locates the data with the physical address and the cache color. Compared with page-based mapping design, however, our design does not have full flexibility in controlling the mapping of every OS page. Instead, we can only control the mapping of each memory region which contains multiple memory pages. We believe that the granularity of the memory region is fine enough for cache management. In summary, compared with page-based mapping designs, our memory region-based mapping design has performance advantages with much lower storage overhead at the cost of giving up full flexibility.

3.3 Profiling Unit



Counter	value	Est. full set value	Est. misses
H1	20	640	1344
H2	15	480	864
H3	12	384	480
H4	10	320	160
Miss	5	160	

(b)

Figure 3. (a) Design of profile unit. (b) Constructing miss curve from the profiled information.

Because cache management is at the level of memory regions, in order to make effective cache partitioning/sharing decisions, software cache management policy needs to know the runtime cache access pattern of each memory region. To capture these patterns, we introduce a profiling unit into the hardware, attempting to answer the following questions for each memory region: 1) how often is the memory region accessed in L2 cache? 2) does the L2 cache accesses to the memory region have strong spatial locality? 3) in what degree do cache misses increase when the memory regions share a cache color with other memory region(s)? the profiling unit provide a set of counters for each memory region, from which the software policies extract the answers of these questions. We will discuss how the software uses these counters in Section 4.

The design of the profiling unit is partly derived from the UMON mechanism proposed by Qureshi and Patt [16]. As shown in Figure 3(a), the profiling unit includes two components for each memory region: a shadow tag array for selected sample sets in a memory region and hit/miss counters. The shadow tag array has the same associativity as the shared cache. We randomly choose 4 sets within the 128 total sets of a memory region for profiling[3]. When an L2 access is sent to the profiling unit, if it is to a selected set, the profiling unit checks if the access is a hit in corresponding set. If the access is a hit in the Kth way, hit counter H_k is increased by one, otherwise the miss counter is increased by one. Figure 3[b] shows how we construct the miss curve of each memory region with the profiled counters (for a 4-way cache). We first multiply counter values by a sampling factor of 32 (number of sets/ number of sampled sets = $128/4 = 32$), to estimate the hit/miss counters for all the 128 cache sets of the memory region. We then construct miss curves against the number of ways per set for each memory region. The construction procedure of miss curves is based on the following property of set-associative caches with the LRU replacement policy: *if $M > N$ and the LRU replacement policy is used, any access that hits in a N-way cache also hits in a M-way cache with the same number of sets.* This property has been used in several other cache studies [20, 16].

3.4 Hardware Overhead

Using the system configuration presented in Figure 2, we have estimated the hardware overhead of the proposed cache management mechanism. The main parameters of the memory subsystem in our simulated 8-core system are shown in Table 1. Because the storage space required by cache color fields in TLB is small, we only calculate the hardware overheads of the region mapping table and profiling unit. As shown in the table, the storage overhead of extra hardware is only 1.25% of the cache tag and data arrays. We want to highlight that, unlike other hardware cache management designs, the added components in our design are decoupled from cache tag and data arrays, so that our design does not complicate the design and verification of the cache and core pipeline.

3.5 Overhead of Re-mapping

When a memory regions is re-mapped, all dirty cache lines in L2 caches need to be written back. Clean cache lines also need to be invalidated. If these cache lines need to be accessed later, they need to be fetched again from main memory. We have found in our experiments that these extra "compulsory" misses do not hurt performance much because re-mapping only happens when the running processes change their L2 cache access pattern significantly.

In a multi-processor systems, any change in the mapping of the pages need to be coordinated among all the TLBs in the system. This process is called TLB shootdown and it causes a significant overhead. In contrast, changes in region mapping table does not require the shootdown process, because each table is only used to index the local shared L2 cache, and these region mapping tables do not have to be identical.

3.6 Changes to OS

The OS memory management groups physical pages into memory regions; and a memory region can only be allocated to a process or a group of processes as a whole. During page faults, the OS

[3]We generate a random sequence of 128 integers by http://www.random.org/sequences/. We use the first four integers of the sequence to select sample sets for all memory regions. They are 44,105,29, and 51.

Parameters	Values	size
Memory	64-bit virtual address, 36-bit physical address (64GB), 8KB page	4GB main memory
Memory region	256-memory region, 2048-page/memory region	16MB/memory region
L2 cache data (1)	4MB	33554432 bits
L2 cache tag (2)	4096-set, 16-way, 64B line, 23-bit tag (including cache color bits)	1507328 bits
region mapping table (3)	256-entry, 5-bit/entry	1280 bits
Shadow tag of profiling unit (4)	256-memory region, 4-set/region, 16-way, 18-bit tag	294912 bits
Counters of profiling unit (5)	256-memory region, 17-counter per memory region, 32-bit counter	140352 bits
Storage of L2 cache	(1) + (2)	35061760 bits
Storage of proposed added components	(3) + (4) + (5)	436544 bits

Table 1. Parameters of the memory subsystem: storage overhead of added components is only 1.25% of the storage of L2 cache.

allocates physical pages to the process from its assigned memory regions in a round-robin fashion. This is to avoid conflict misses and to balance data distribution in the cache, similar to the idea behind the original bin hopping [9] page allocation algorithm. The cache management policy should make effort to map consecutive memory regions to different cache colors. As our focus in this paper is on the hardware mechanism design and its applications, we do not explore other OS page allocation algorithms in this study and leave it as our future work.

4. Cache Management Policies

In this section, we present two cache management policies: **PCM** (Process-level Cache Management) and **MCM** (Memory region-level Cache Management). The difference between the two policies is that the PCM policy assumes all memory regions of a running process have same access pattern and treats them as a whole, while the MCM policy does not make such assumption and treats each memory region individually.

With the support of the proposed cache management mechanism, both policies dynamically make decisions on the mapping between memory regions and cache colors. If there are M memory regions to be mapped to N cache colors, there are N^M possible mappings, which is a large number that makes any brute-force approach impractical. Therefore, heuristics are needed to make mapping decisions at runtime. The PCM and MCM policies 1) read counter values of profiling unit, and construct the miss rate curves of running processes or memory regions, 2) classify the access pattern for each running process or memory region, according to their miss rate curves, 3) assign a cache color to each memory region, and 4) reset counters, sleep for a given time interval (10ms in this work), and jump to step 1. We discuss the first three steps in this section.

4.1 Constructing Miss Curves

As discussed in Section 3, the profiling unit has an array of counters for each memory region, including a set of hit counters and a miss counter. A hit counter of a memory region H_k summarizes the number of hits on cache way k for the selected four cache sets when the memory region is exclusively mapped to a cache color. The miss counter summarizes the number of misses. Statistically, the selected four sets could represent the whole 128 sets of the cache color. Therefore we can simply multiply the counter values by 32 to estimate the hit and miss counters for the memory region. The miss rate curve of the memory region can be constructed from the counters as shown in Figure 3(b). It is also straightforward to construct the miss rate curve for a running process: we sum all counters of memory regions that belong to the running process before using the same method.

4.2 Classifying Memory Regions and Running Processes

Essentially, a cache management policy in our proposed framework allocates a limited number of cache colors (32 in this study) to a much larger number of memory regions (256 in this study). Before making the cache color allocation decision, PCM and MCM policies seek answers to the three questions asked in Section 3, and classify the running processes or memory regions according to the answers. We adopt Lin et al.'s classification methodology [13], and classify memory regions or running process by four categories (colors): red, yellow, green and black. The classification procedure of memory regions for MCM policy is as follows:

How often is the data to a memory region accessed in L2 cache? The sum of all counter values times the sample ratio (128 sets/4 sample sets=32) is the number of accesses to a memory region: $\#\text{Access} = (\sum_{i=1}^{Assoc} H_i + \#\text{Miss}) \times \text{sample_ratio}$. If the memory region is not accessed frequently enough in L2 cache, the mapping of the memory region does not have big performance impact. MCM classifies this type of memory region as "black" memory region. The threshold we choose in this study is one L2 cache access every 8000 processor cycles to a memory region.

Does the L2 cache accesses of the memory region have strong spatial locality? If a memory region is not "black", the MCM policy checks if the memory region has high miss rate when a dedicated cache color is allocated to the memory region. We refer this miss rate to *MR-Missrate*, which is a lower bound of miss rate of the memory region . The *MR-Missrate* can be obtained from the counter values: $\text{MR-Missrate} = \#\text{Miss}/(\#\text{Miss} + \sum_{i=1}^{Assoc} H_i)$. A high *MR-Missrate* value indicates that the working set of the memory region can not fit into a cache color.

In what degree does the number of cache misses increase when the memory region shares a cache color with other memory region(s)? The *MR-Missrate* value alone is not enough to tell how well a memory region gets along with other memory regions when they share a cache color. Therefore, the MCM needs another metric to indicate whether miss rate of a memory region M is sensitive to the number of sharing memory regions. We examine in depth the physical meanings of the counters of the profiling unit. The hit counter H_k is increased by one when an access to cache line C hits in the kth way of the shadow tag of the profiling unit, which means k distinct cache lines in the same memory region have been accessed between the two consecutive accesses to cache line C. The value k is referred to re-use distance of the

cache line C. If k is large (say 12 for a cache with associativity of 16), it is highly likely that the access to cache line C would be a cache miss when the memory region shares the cache color with a large number of memory regions, and vice versa. Having the above observation, we found that *AHRD* (average re-use distance for hit accesses) is a good indicator. The *AHRD* is calculated as $\text{AHRD} = \sum_{i=1}^{Assoc}(i * H_i) / \sum_{i=1}^{Assoc} H_i$. When *AHRD* is small, cache hits reported by the profiling unit are not likely converted to misses when a large number of memory regions share a cache color.

Combining *MR-Missrate* and *AHRD*, we propose a new metric *MR-Missrate/AHRD* which is the ratio between *MR-Missrate* and *AHRD*. If a memory region M has a large *MR-Missrate/AHRD* value, it is safe to let M share a cache color with a large number of other memory regions, without high risks of increasing the miss rate of M. We quantize *MR-Missrate/AHRD* and classify non-"black" memory regions into three zones: "green", "yellow" and "red". A "green" memory region has a high *MR-Missrate/AHRD* value (> 0.3), and is likely to have streaming access pattern (e.g. its working set is impossible to put into a cache color). The cache management policies may assign a large number of "green" memory regions to a cache color without hurting the performance much. A "red" memory region has a low MR-Missrate/AHRD value ($\leq$ 0.2), which means its working set is likely to fit into a cache color if the cache color is not shared by a large number of memory regions. The cache management policies should assign a large number of cache colors to the "red" memory regions. A "yellow" memory region has a moderate *MR-Missrate/AHRD* value (between 0.2 and 0.3). The cache management policies should allocate a moderate number of cache colors to "yellow" memory regions.

The classification procedure of the running processes for PCM policy is very similar. We can safely replace "memory region" with "running process" in the same procedure, and classify running processes to four categories (colors). The color of a running process is determined by the miss curve constructed for the process. All memory regions allocated to the process are classified as the color of the running process.

4.3 Cache Color Allocation

We use a simple strategy to allocate cache colors to memory regions. As shown in Algorithm 1, the cache color allocation algorithm segregate the cache spaces between memory regions within conflicting categories. After calculating the average memory region density (*regions-per-color*) of the system, we first allocate a small number of cache colors to "green" memory regions, letting the densities of these memory regions reach four times of the average density. We then allocate a number of cache colors to "yellow" memory regions, letting the densities of these memory regions be twice of the average density. Finally, we allocate the rest cache colors to "red" and "black" processes. These decisions are enforced by a reconfiguration of the region mapping table.

Both PCM and MCM policies use the same cache color allocation algorithm after all memory regions are classified. We remark that the software policies module have very small overhead: the complexity is O(N) where N is the total number of memory regions.

5. Experimental Setup

Algorithm 1: The cache color allocation algorithm.

Data: Counter values of reported by profiling units
Result: configuration of region mapping table

green-list.clear(); yellow-list.clear(); other-list.clear()

forall *memory regions* **do**
  /* classify(memory region) is a function to classify the memory region to four categories */
  switch *classify(memory region)* **do**
    case *GREEN* add the memory regions to *green-list*
    case *YELLOW* add the memory region to *yellow-list*
    otherwise add the memory region to *other-list*

region-per-color = *num-memory-region* / *num-cache-color*
if *other-list is empty* **then**
  move all memory regions from *green-list* and *yellow-list* to *other-list*.
green-color = *streaming-list*.size() / (4 $\times$ *region-per-color*)
if *num-green-color* == *0* **then**
  move all memory regions from *green-list* to *other-list*.
yellow-color = *yellow-list*.size() / (2 $\times$ *region-per-color*)
if *num-yellow-color* == *0* **then**
  move all memory regions from *yellow-list* to *other-list*.
num-other-color = *num-cache-color* $-$ *num-green-color* $-$ *num-yellow-color*

/* map-regions($list$,i,j) maps regions in $list$ to j cache colors, start from color i */
map-regions(*other-list*, 0, *num-other-color*)
map-regions(*yellow-list*, *num-other-color*, *num-yellow-color*)
map-regions(*green-list*, *num-other-color* + *num-yellow-color*, *num-green-color*)

5.1 Simulation Environment

We use M5 [2] as the base architectural simulator. The simulated processor core is 4-way issue, out-of-order and with a 16-stage pipeline. We add page tables into M5 for each process to enhance its virtual-to-physical address translation functionality. The size of each OS page is 8KB. Because our study focuses on the last-level cache (L2 cache) which has strong interaction with the main memory, we extend M5 to simulate DDR2 DRAM systems in detail. The simulated memory transactions are pipelined whenever possible. The processor cores have private L1 data and instruction caches. The L2 cache is shared by all cores and uses the true LRU replacement policy. We increase the L2 cache capacity as the number of cores increases but keep a constant degree of associativity. An 1MB L2 cache is used for 2-core systems, a 2MB cache for 4-core systems, and a 4MB cache for 8-core system. Therefore, on average, each core has eight cache colors and each color has 128 cache sets. We allocate 32 memory regions to each process. Table 2 summarizes the major simulation parameters.

5.2 Workloads

In order to make the simulation time tolerable while still emulating the representative behavior of program executions, we select representative simulation points of one billion instructions, each for every benchmark. The simulation points are picked up according to SimPoint 3.0 [19]. In our experiments, each processor core is single-threaded and runs a distinct application. Following the methodology of a previous study [13], we classify the twenty-six benchmarks of the SPEC2000 suite into Red, Yellow, Green

Parameters	Values
Processor	2/4/8 cores, 3.2 GHz, 4-issue per core, 16-stage pipeline
Functional units	4 IntALU, 2 IntMult, 2 FPALU, 1 FPMult
IQ, ROB and LSQ size	IQ 64, ROB 196, LQ 32, SQ 32
Num of physical register	228 Int, 228 FP
Branch predictor	Hybrid, 8k global + 2K local, 16-entry RAS, 4K-entry and 4-way BTB
L1 caches (per core)	64KB Inst/64KB Data, 2-way, 64B line, hit latency: 1 cycle Inst/3-cycle Data
L2 cache (shared)	1MB/2MB/4MB, 16-way, 64B line, 15-cycle hit latency
Memory regions and cache color	32-memory region/process, 8 cache colors/core
MSHR entries	Inst:8, Data:32, L2:64
Memory	2-channel, 2-DIMM/channel, 1-rank/DIMM, 4-bank/rank,
DDR2 channel bandwidth	667MT/s (Mega Transfers/second), 8byte/channel, 5.3GB/s/channel
DDR2 DRAM latency	5-5-5, precharge 15ns, row access 15ns, column access 15ns

Table 2. Major simulation parameters.

Class	Slowdown (512KB/2MB)	L2 access rate per 1K cycle (512KB)	Benchmarks (*MR-Missrate/AHRD* = *MR-Missrate* / *AHRD*)	
R-type (4)	$\geq$ 50%	average: 74.9	$R1$: 178.galgel (0.001=0.001/1.928) $R3$: 188.ammp (0.087=0.149/1.718)	$R2$: 179.art (0.000=0.001/1.928) $R4$: 300.twolf (0.000=0.001/1.689)
Y-type(4)	$\geq$ 25%	average: 30.7	$Y1$: 172.mgrid (0.177=0.253/1.428) $Y3$: 176.gcc (0.017=0.023/1.382)	$Y2$: 175.vpr (0.020=0.049/2.437) $Y4$: 187.facerec (0.151=0.472/2.832)
G-type (5)	$<$ 25%	$\geq$ 40.0	$G1$: 171.swim (0.414=0.435/1.052) $G3$: 183.equake (0.203=0.486/2.388)	$G2$: 173.applu (0.364=0.460/1.266) $G4$: 189.lucas (0.525=0.542/1.033)
B-type (13)	$<$ 25%	$<$ 40.0	$B1$: 168.wupise $B3$: 252.eon	$B2$: 177.mesa $B4$: 253.perlbmk

Table 3. Benchmark classification.

Num. of cores	Workload	Benchmarks	Wo.	Bench.	Wo.	Bench.	Wo.	Bench.
2-core	2C-RR1	R1R2	2C-RR2	R3R4	2C-YY1	Y1Y2	2C-YY1	Y3Y4
	2C-RY1	R1Y1	2C-RY2	R2Y2	2C-RY3	R3Y3	2C-RY4	R4Y4
	2C-RG1	R1G1	2C-RG2	R2G2	2C-RG3	R3G3	2C-RG4	R4G4
	2C-YG1	Y1G1	2C-YG2	Y2G2	2C-YG3	Y3G3	2C-YG4	Y4G4

Num. of cores	Workload	Benchmarks	Wo.	Bench.	Wo.	Bench.	Wo.	Bench.
4-core	4C-RRRR1	R1R2R3R4	4C-YYYY1	Y1Y2Y3Y4	4C-RRYY1	R1R2Y1Y2	4C-RRYY2	R3R4Y3Y4
	4C-RRGG1	R1R2G1G2	4C-RRGG2	R3R4G3G4	4C-YYGG1	Y1Y2G1G2	4C-YYGG2	Y3Y4G3G4
	4C-RYGB1	R1Y1G1B1	4C-RYGB2	R2Y2G2B2	4C-RYGB3	R3Y3G3B3	4C-RYGB4	R4Y4G4B4

Num. of cores	Workload	Benchmarks	Wo.	Bench.	Wo.	Bench.
8-core	8C-RRRR1	R1R2R3R4R1R2R3R4	8C-YYYY1	Y1Y2Y3Y4Y1Y2Y3Y4	8C-RRYY1	R1R2G1G2Y1Y2Y3Y4
	8C-RRGG1	R1R2G1G2G1G2G3G4	8C-YYGG1	Y1Y2Y1Y2G1G2G3G4	8C-RYGB1	R1R2Y1Y2G1G2B1B2
	8C-RYGB2	R3R4Y3Y4G3G4B3B4	8C-RYGB3	R0R2Y0Y2G0G2B0B2	8C-RYGB4	R1R3Y1Y3G1G3B1B3

Table 4. Workload mixes.

and Black applications. As shown in Table 3, four applications have more than 50% performance slowdown when using only four cache colors (512KB cache), compared with using sixteen cache colors (2MB). We refer to them as R-type (RED) applications as they were sensitive to the L2 cache size. It is predicted that the working set of a red application can fit into 2MB cache but not 512KB cache. Four applications have a performance slowdown between 25% and 50%; they are referred to as Y-type (YELLOW) applications. The remaining sixteen applications are further divided into two classes by L2 cache access intensity. Applications with more than 40 L2 cache accesses per 1000 processor cycles are referred to as G-type (GREEN) applications[4] and the rest are referred to as B-type (BLACK) applications. The number 40 is an arbitrary threshold, we select four out of thirteen black applications to construct the workloads. The values of proposed metric are also shown in the Table 3. As shown in the table, the metric *MR-Missrate/AHRD* successfully differentiates G-type applications from R-type and Y-type applications.

The workloads are shown in Table 4. Sixteen 2-core workloads, twelve 4-core workloads and nine 8-core workloads are are randomly chosen by composing selected applications. Each workload is named by the workload type and its workload index. For example, the 4-core workload 4C-RYGB2 consists of four applications R-type *179.art*, Y-type *175.vpr*, G-type *173.applu* and B-type *177.mesa*.

5.3 Metrics

Table 5 summarizes three commonly used performance evaluation metrics. *Throughput* represents absolute IPC numbers. *Weighted speedup* is the sum of speedups of all programs over their execution with a baseline scheme. *Fair speedup* is the harmonic mean of the speedups over a baseline scheme. Fair speedup evaluates fairness in addition to performance [5]. In this study, the baseline scheme for both weighted speedup and fair speedup is single-core execution with a 512KB L2 cache, other configurations of the system remain same as Table 2. We use *weighted speedup* metrics throughout the performance discussion, and present some key per-

[4] 181.mcf is classified as G-type because its working set does not fit into both 512KB and 2MB cache, and because it accesses L2 cache intensively. We do not include it in our workload construction because of its long simulation time.

formance results using *throughput* and *fair speedup*.

Metric	Formula
Throughput (IPCs)	$\sum_{i=1}^{n}(\text{IPC}_{\text{scheme}}[i])$
Weighted Speedup [22]	$\sum_{i=1}^{n}(\text{IPC}_{\text{scheme}}[i]/\text{IPC}_{\text{base}}[i])$
Fair Speedup [5]	$n/\sum_{i=1}^{n}(\text{IPC}_{\text{base}}[i]/\text{IPC}_{\text{scheme}}[i])$

Table 5. Performance evaluation metrics.

6. Performance Evaluation and Analysis

6.1 Performance Comparison on Weighted Speedup Metric

Figure 4 shows normalized weighted speedups. Weighted speedups are normalized to those with shared cache.

We first compare the performance of two baseline cache management policies: private cache and shared cache. As shown in Figure 4, shared cache has performance advantages over private cache because it adapts to the demands of competing processes. Compared with shared cache, the average performance degradation with private cache is 4.8%, 10.2%, and 17.6% on 2-, 4-, and 8-core systems, respectively. Nevertheless, in a few cases private cache outperforms shared cache because it isolates cache usage of processes. For example, for 2C-RG4, 2C-YG2, 4C-YYGG2 and 4C-RYGB4, private cache improves performance by 7.9%, 8.4%, 4.6% and 2.5%, respectively, compared with shared cache. Private cache improves the performance of these four workloads because it segregates the cache space of G-type application (with streaming access pattern) from that of the other types of applications.

The PCM policy provides much better performance than shared cache. PCM improves performance on average by 4.3%, 13.1% and 14.5% compared with shared cache on 2-, 4-, and 8-core systems, respectively. PCM outperforms shared cache performance for 32 out of 37 workloads, and for the other 5 workloads PCM decreases performance by up to 3.0% and only 1.5% on average. If we look into performance of workloads in different groups, for workloads without G-type and Y-type applications, PCM performs similarly as shared cache because mostly no process is classified as green or yellow process (so that all cache colors are shared by all simultaneously running processes). For workloads containing a G-type application, PCM improves performance by 8.8%, 18.2% and 20.9% on average and up to 28.3%, 42.6% and 47.5% on 2-, 4-, and 8-core systems, respectively. This difference gives two clear messages in designing the hardware support and software policies in our framework: 1) the main task of profiling unit is to identify those memory regions or running processes with streaming access pattern (G-type); and 2) the software policy can simply reserve a small portion of the shared cache for G-type applications, and let other types of applications share the rest of the cache.

MCM, classifying memory region individually, does not perform better than PCM. Compared with shared cache, MCM improves performance by 4.0%, 12.0%, 14.2% on average on 2-, 4- and 8-core systems, respectively. MCM can not bring better performance than PCM because our round-robin page allocation policy lets the memory regions of the same process have similar behaviors. MCM performs slightly worse than PCM because MCM occasionally classifies memory regions to wrong categories, due to cache set sampling of profiling unit and short dynamic behavior of memory regions. PCM suffers less because it averages profiling unit counters for all memory regions before a process is classified. Despite the drawbacks of MCM, we believe that if memory region variation does exist, either by different OS page allocation policy or by MCM-aware compiler/OS optimizations, MCM may outperform PCM.

6.2 Performance Comparison on Throughput Metric

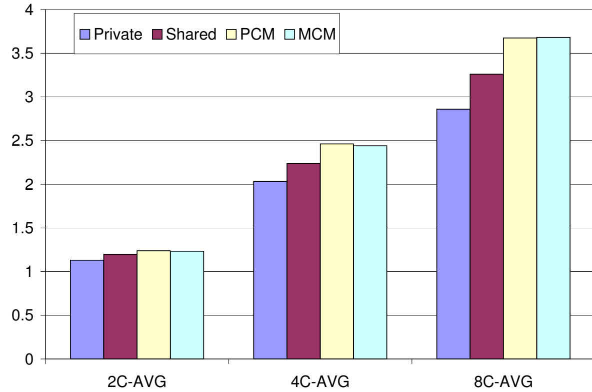


Figure 5. Throughput (Sum of IPCs).

Figure 5 compares the performance of cache management policies on the throughput metric, which is the sum of IPCs. On average, the throughputs are 1.130, 1.198, 1.238 and 1.234 on 2-core systems with private cache, shared cache, PCM, and MCM, respectively. The throughputs are 2.033, 2.237, 2.462 and 2.441 on 4-core systems and 2.859, 3.260, 3.675 and 3.680 on 8-core systems. Although the absolute numbers of performance improvement/degradation are different from weighted speedups, the trends are largely similar. If we compare throughputs of systems with different number of cores, although the cache capacity is increased with the increase of number of cores, we do not see a linear increase of throughput because the configurations of 2-, 4- and 8-core systems have the same main memory subsystem. The average memory access latencies are increased from 106ns on 2-core to 156ns on 4-core, and to 261ns on 8-core systems with PCM. As the result, the average throughput of 8-core systems is only 1.49 times of that of 4-core systems[5].

If we look into the performance of individual workloads using two metrics, weighted speedup and throughput agree with each other for most workloads. We remark that for all workloads weighted speedup and throughput agree with each other when we compare PCM and MCM with shared cache.

PCM and MCM policies may improve performance for all applications of a workload when compared with shared cache. For example, Table 6 shows detail data of 4C-RRGG2, including IPC, L2 cache miss rate, average memory bandwidth utilization and average memory access latency. PCM not only largely improves the IPCs of two R-type applications (*ammp* and *twolf*), but also significantly improves the performance of two G-type applications (*equake* and *lucas*). We find that during runtime PCM correctly

[5] We do not directly compare the throughputs of 2-core workloads, because they do not include black applications as 4-core and 8-core workloads do.

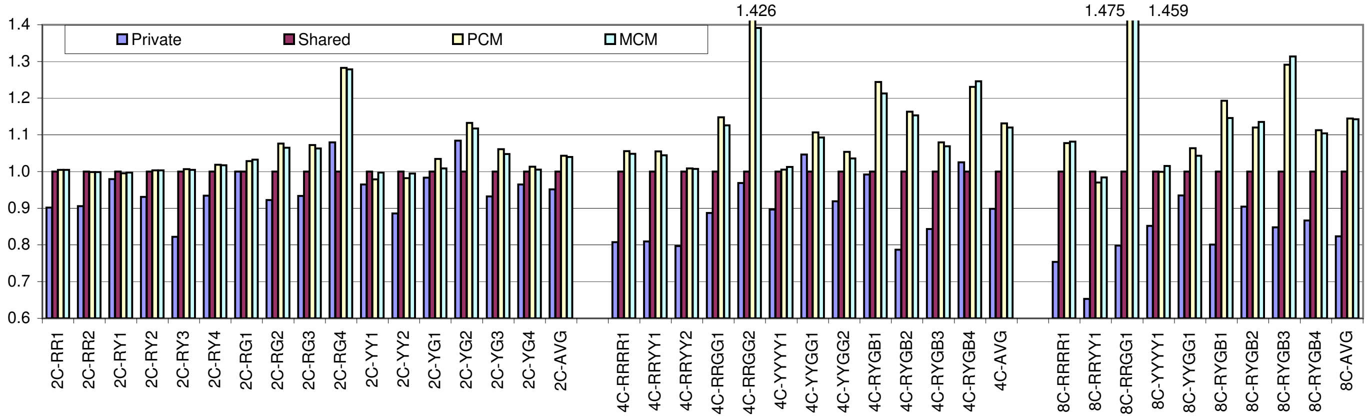


Figure 4. Normalized weighted speedups.

Applications of 4C-RRGG2	IPC		L2 miss rate		Mem BW (GB/s)		Mem Latency (ns)	
	Shared	PCM	Shared	PCM	Shared	PCM	Shared	PCM
R3:188.ammp	0.646	1.059	28.5%	14.9%	1.56	1.15	132	129
R4:300.twolf	0.227	0.446	39.9%	17.8%	1.16	0.81	118	113
G3:183.equake	0.246	0.261	56.2%	56.9%	1.72	1.87	117	107
G4:189.lucas	0.360	0.376	60.3%	64.5%	4.64	5.02	125	117
Overall	1.479	2.143	47.1%	37.3%	9.08	8.85	125	116

Table 6. Detail data of 4C-RRGG2 by shared cache and PCM: IPCs, L2 cache miss rates average memory bandwidth utilization, and average memory access latency.

classifies equake and lucas as green and yellow processes and ammp and twolf as other processes. Therefore, only one or two cache colors (128 or 256KB) are allocated to *equake* and *lucas* each. IPCs of the two processes are increased with such small cache capacity, compared with their IPCs with shared cache. This surprising finding is also reported by the prior cache partitioning study on real systems [13]. Their explanation for this interesting finding roots on the reduction of bandwidth utilization when number of overall cache misses is reduced by cache partitioning scheme, but no detail data in the paper supports the explanation[6]. To confirm their explanation, we look into the detailed statistics of 4C-RRGG2 with shared cache and PCM. We find that *ammp* and *twolf* enjoy the large cache capacity allocated to them by PCM: their L2 cache miss rates are decreased significantly, hence their memory bandwidth utilization is reduced. For equake and lucas, despite visible cache miss rate increases with PCM policy due to the small cache capacity allocated to them, their IPCs are increased significantly. We confirm the explanation in prior work [13] that as the overall L2 cache miss rate is reduced (from 47.1% to 37.3%), the overall bandwidth utilization is reduced (from 9.08GB/s to 8.85GB/s). Therefore the L2 cache miss penalty is reduced. *Equake* and *lucas* extensively access the main memory, so that their performances are sensitive to the memory access latency. Consequently, *equake* and *lucas* enjoy the lower memory access latency and their performance is also improved. We have the same observation for other seven workloads: 2C-RG3, 2C-RG4, 2C-YG3, 4C-RYGB2, 4C-RYGB3, 4C-RYGB4, and 8C-RYGB3. The large performance improvement of these cases shows the effectiveness of our design and the overall framework.

[6]We believe they did not report data to support the explanation because some statistics are hard to get on real systems.

6.3 Performance Comparison on Fair Speedup Metric

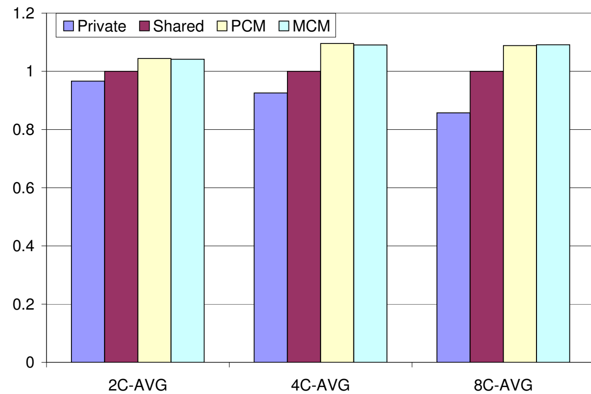
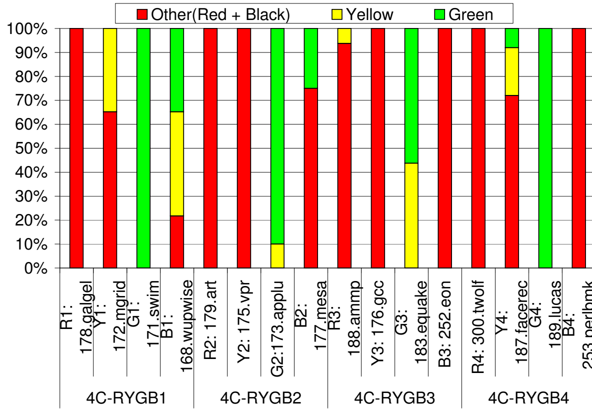
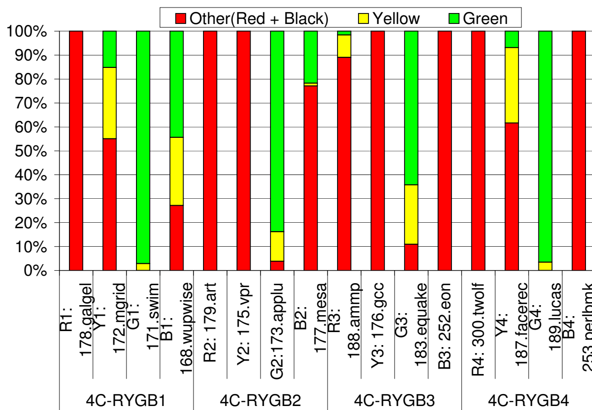


Figure 6. Normalized fair speedup.

A cache management policy may improve the overall performance of the system at the cost of severely degrading the performance of some applications of a workload. Fair speedup, the harmonic mean of normalized IPCs, considers both fairness and performance [5]. Figure 6 shows the performance on the fair speedup



(a) PCM



(b) MCM

Figure 7. Distribution of classification for 4C-RYGB-type workloads.

metric. The fair speedups are normalized to those with shared cache. On average, MCM improves performance on this metric by 4.2%, 9.1% and 9.1% on average compared with shared cache, respectively. PCM performs similarly as MCM.

6.4 Effectiveness of Proposed Metrics

Figure 7(a) shows the distribution of process classification with proposed metrics *MR-Missrate/AHRD* by the PCM policy for four 4C-RYGB-type workloads. Executions of G-type and Y-type applications are classified as "green" and yellow processes mostly. Executions of other types (R-type and B-type) of applications are classified as "other" processes mostly. Therefore, a large portion of the total cache capacity is shared by these types of workloads. If we look into the applications individually, all the executions of R-type applications are classified as "other" processes because their sensitivity to the cache capacity. The only exception is 188.ammp which is classified as "yellow" process occasionally. The executions of Y-type applications are classified as "green" or "yellow" processes much more often than R-type applications. 172.mgrid and 187.facerec are classified as "yellow" processes with a large portion of time. 187.facerec is even classified as "green" process occasionally. It is surprising that executions of two B-type applications, 168.wupwise and 177.mesa, are classified as "green" or "yellow" processes. We believe that it is because we use a conservatively threshold to differentiate black process from the other types of processes. Nevertheless, it may not significantly impact the overall system performance because B-type applications are not sensitive to the cache capacity.

Figure 7(b) shows distribution of memory region classification with the metrics *MR-Missrate/AHRD* by MCM policy. The distributions are very close to that of process classification in Figure 7(a). Nevertheless, it has more variations: the memory regions in 7 out of 16 applications are classified as all three categories during applications' executions. In comparison, only 2 processes are classified as all three categories during their executions. As discussed earlier, MCM may occasionally classify memory regions to wrong categories due to cache set sampling of the profiling unit and short dynamic behaviors of memory regions. In comparison, PCM suffers less from these two factors because it averages counters of all memory regions of a process before it classifies the process.

6.5 Effect of Varying the Number of Sampled Sets in the Profiling Unit

We select 4 sets out of 128 total sets for each memory region in the default hardware mechanism configuration. We want to analyzes the sensitivity of the mechanism to the number of sampled sets. We change the number of sample sets from 4 to 2, 8 and all 128 sets, and then compare the performance of 4-core workloads with the weighted speedup. We have found that the performance improvement is less than 1% for all workloads when 8 or all 128 sets are used. When the sample set number is reduced to 2 sets, the performance degradation is less than 2% for those workloads. We have also found that MCM is more sensitive to the number of sampled sets than PCM.

7. Related Work

Hardware Cache Management. There have been several studies focusing on hardware-based cache partitioning for multicore processors [20, 10, 16, 17, 5]. In general, these designs include new hardware support to enforce cache partitioning decisions by changing the LRU cache replacement policy, and to trace the ownership of each cache lines. Those proposed approaches have several limitations as mentioned in Section 1, including the implementation complexity and lack of flexibility. The hardware mechanism in this study uses decoupled cache address mapping proposed for reducing cache conflict misses on single-threaded processors [18, 14]. In those designs, page remapping is activated when excessive cache misses occur on certain pages with the same cache color. Extra hardware is used to decouple physical memory addressing and cache indexing so that a physical page can be mapped to any cache color. A cache color field, which records the cache color of an OS page, is added to each page table entry as well as to each TLB entry. By so doing, expensive data migration is eliminated from the page remapping procedure. Our framework utilizes decoupled cache address mapping as the hardware mechanism to enforce cache allocation, including partitioning and sharing, for multicore processors[7]. Instead of passively reacting to

[7]One study [18] included a brief evaluation of their design in the context of multicore/multithreaded processors but did not extend the design itself.

cache conflicts, the software cache management in our framework is proactive in dynamic cache re-allocation. We further optimize the hardware mechanism to eliminate the need of large storage overhead in page tables, minimize the runtime overhead of cache remapping, and avoid complication on cache coherence in multiprocessor environment from the original designs at the cost of less flexibility on page color mapping. Our simulation results show that, even without the full flexibility, our scheme using memory region-level mapping achieves comparable or higher performance improvements.

Software Cache Management. Several studies [3, 18] have used software-based approaches to managing the cache by controlling virtual to physical address mapping in the OS at page level. Their cache management mechanisms are based on *page coloring* [21], an OS technique which works as follows. A physical address contains several common bits between the cache index and the physical page number, referred to as *page color*. One can divide a physically addressed cache into non-intersecting regions (*cache color*) by page color, and pages with the same page color are mapped to the same cache color. Bugnion et al. [3] and Sherwood et al. [18] use profiling information to map OS pages to cache colors. Their goal is to reduce cache misses from conflicting cache accesses of different OS pages. The profiling information is generated at compile time and passed to OS at run time. While these studies target for single threaded processors, this study is target for multicore processors. Lin et al. [13] limited the cache usage of a process by limiting the number of cache colors that the pages of the process are mapped to. Therefore, a physically addressed cache can be partitioned among simultaneously running processes on multicore processors. There are two limitations in their cache partitioning schemes: First, the physical memory space is co-partitioned with the shared cache. If a process demands a large cache space, even if the process has a small memory footprint, a large memory space may have to be reserved for the process, Second, it is expensive to remap a page because of the needs to move data between two physical pages. In comparison, our scheme eliminates the co-partitioning limitation and the expensive data movement overhead. As discussed in Section 1, a recent study [6] proposed the approach of shared cache management through OS-level page allocation. This study gives design details and demonstrates a particular framework that can be easily adopted in real systems.

8. Conclusion

We have proposed a flexible and effective framework to manage cache resources for multicore processors. A scalable and low-overhead hardware-based cache management mechanism built on *memory region* forms the basis of our proposed framework. The proposed mechanism addresses many limitations of previously proposed cache management mechanisms. Enabled by this hardware mechanism, two software-based cache management policies, PCM and MCM, have been proposed and evaluated. Our simulation results show that these new policies significantly improve system performance when compared with shared cache and private cache.

Acknowledgments

We thank the constructive comments from the anonymous referees. This research was supported in part by the National Science Foundation under grants CNS-0834476, CCF-0514085, CNS-0834393, and CCF-0913050.

9. References

[1] B. Bershad, D. Lee, T. Romer, and B. Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 158–170, 1994.

[2] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt. Network-oriented full-system simulation using M5. In *Proceedings of the Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads*, 2003.

[3] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam. Compiler-directed page coloring for multiprocessors. In *Proceedings of the seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 244–255, 1996.

[4] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Proceedings of the 1992 Supercomputing Conference*, pages 114–124, 1992.

[5] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proceedings of the 21st annual International Conference on Supercomputing*, pages 242–252, 2007.

[6] S. Cho and L. Jin. Managing distributed, shared L2 caches through os-level page allocation. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 455–468, 2006.

[7] Intel Corporation. Inside intel core microarchitecture. `ftp://download.intel.com/technology/architecture/new_architecture_06.pdf`.

[8] R. N. Kalla, B. Sinharoy, and J. M. Tendler. Ibm power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

[9] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Trans. Comput. Syst.*, 10(4):338–359, 1992.

[10] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, 2004.

[11] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[12] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 94–105, 1991.

[13] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multi-core cache partitioning: Bridging the gap between simulation and real systems. In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture*, pages 367–278, 2008.

[14] R. Min and Y. Hu. Improving performance of large physically indexed caches by decoupling memory addresses from cache addresses. *IEEE Trans. Comput.*, 50(11):1191–1201, 2001.

[15] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, 1992.

[16] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 423–432, 2006.

[17] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven cmp cache management. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pages 2–12, 2006.

[18] T. Sherwood, B. Calder, and J. Emer. Reducing cache misses using hardware and software page placement. In *Proceedings of the 13th International Conference on Supercomputing*, pages 155–164, 1999.

[19] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, 2002.

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

[21] G. Taylor, P. Davies, and M. Farmwald. The TLB slice–a low-cost high-speed address translation mechanism. In *Proceedings of the 25th annual International Symposium on Computer architecture*, pages 355–363, 1990.

[22] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *Proceedings of the 34th annual International Symposium on Microarchitecture*, pages 318–327, 2001.