

Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems

Jiang Lin¹, Qingda Lu², Xiaoning Ding², Zhao Zhang¹, Xiaodong Zhang² and P. Sadayappan²

¹Dept. 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

Cache partitioning and sharing is critical to the effective utilization of multicore processors. However, almost all existing studies have been evaluated by simulation that often has several limitations, such as excessive simulation time, absence of OS activities and proneness to simulation inaccuracy. To address these issues, we have taken an efficient software approach to supporting both static and dynamic cache partitioning in OS through memory address mapping. We have comprehensively evaluated several representative cache partitioning schemes with different optimization objectives, including performance, fairness, and quality of service (QoS). Our software approach makes it possible to run the SPEC CPU2006 benchmark suite to completion. Besides confirming important conclusions from previous work, we are able to gain several insights from whole-program executions, which are infeasible from simulation. For example, giving up some cache space in one program to help another one may improve the performance of both programs for certain workloads due to reduced contention for memory bandwidth. Our evaluation of previously proposed fairness metrics is also significantly different from a simulation-based study.

The contributions of this study are threefold. (1) To the best of our knowledge, this is a highly comprehensive execution- and measurement-based study on multicore cache partitioning. This paper not only confirms important conclusions from simulation-based studies, but also provides new insights into dynamic behaviors and interaction effects. (2) Our approach provides a unique and efficient option for evaluating multicore cache partitioning. The implemented software layer can be used as a tool in multicore performance evaluation and hardware design. (3) The proposed schemes can be further refined for OS kernels to improve performance.

1. Introduction

Cache partitioning and sharing is critical to the effective utilization of multicore processors. Cache partitioning usually refers to the partitioning of shared L2 or L3 caches among a set of programming threads running simultaneously on different cores. Most commercial multicore processors today still use cache designs from uniprocessors, which do not consider the interference among multiple cores. Meanwhile, a number of cache partitioning methods have been proposed with different optimization objectives, including performance [17, 11, 5, 2], fairness [8, 2, 12], and QoS (Quality of Service) [6, 10, 12].

Most existing studies, including the above cited ones, were evaluated by simulation. Although simulation is flexible, it possesses several limitations in evaluating cache partitioning schemes. The most serious one is the slow simulation speed – it is infeasible to run large, complex and dynamic real-world programs to completion on a cycle-accurate simulator. A typical simulation-based study may only simulate a few billion instructions for a program, which is equivalent to about one second of execution on a real machine. The complex structure and dynamic behavior of concurrently running programs can hardly be represented by such a short execution. Furthermore, the effect of operating systems can hardly be evaluated in simulation-based studies because the full impact cannot be observed in a short simulation time. This limitation may not be the most serious concern for microprocessor design, but is becoming increasingly relevant to system architecture design. In addition, careful measurements on real machines are reliable, while evaluations on simulators are prone to inaccuracy and coding errors.

Our Objectives and Approach To address these limitations, we present an execution- and measurement-based study attempting to answer the following questions of concern: (1) Can we confirm the conclusions made by the simulation-based studies on cache partitioning and sharing

in a runtime environment? (2) Can we provide additional insights and new findings that simulation-based studies are not able to? (3) Can we make a case for our software approach as an important option for performance evaluation of multicore cache designs?

In order to answer these questions, we first implement an efficient software layer for cache partitioning and sharing in the operating system through virtual-physical address mapping. Specifically, we have modified the Linux kernel for IA-32 processors to limit the memory allocation for each thread by controlling its *page colors*. This flexible cache partitioning mechanism supports static and dynamic partitioning policies. It is worth noting that page coloring may increase I/O accesses, e.g. page swapping or file I/O, which may distort the performance results. We avoided this problem by carefully selecting the workloads and used a machine with large memory. According to the research literature in the public domain, no previous study has implemented dynamic cache partitioning on a real multicore machine. With static policies, this mechanism has virtually zero run-time overhead and is non-intrusive because it only changes the memory allocation and deallocation. With dynamic policies, by employing optimizations such as lazy page migration, on average it only incurs a 2% runtime overhead. We then conducted comprehensive experiments and detailed analysis of cache partitioning using a physical dual-core server. Being able to execute SPEC CPU2006 workloads to completion and collect detailed measurements with performance counters, we have evaluated static and dynamic policies with various metrics.

Novelty and Limitation of Our Work *The novelty of this study is the proposed experimental methodology that enables the examination of existing and future cache partitioning policies on real systems by using a software partitioning mechanism to emulate a hardware partitioning mechanism.* Many hardware cache partitioning schemes have been proposed and new schemes are being studied, but *none has yet been adopted in commodity processors and thus not tested on real machines.* Our software approach is not intended to replace those hardware schemes; instead, our mostly confirmatory results may help them get adopted in real machines. In addition, our evaluation also provides new findings that are very difficult to obtain by simulator due to intolerably long simulation time. A potential concern of this methodology is how closely a software implementation may emulate a hardware mechanism. Indeed, software cannot emulate *all* hardware mechanisms; however, the emulation is close to the hardware mechanisms for most existing and practical hardware-based policies. We discuss it in detail in Section 3.

As a measurement-based study, this work does have a limitation: our experiments are limited by the hardware platform we are using. All experiments are done on two-

core processors with little flexibility in cache set associativity, replacement policy, and cache block size¹. Nevertheless, we can study hours of real-world program execution, while practically a cycle-accurate simulator only simulates seconds of execution. We believe that for a large L2 cache shared by complex programs, one must use sufficiently long execution to fully verify the effectiveness of a cache partitioning policy. As in many cases, measurement and simulation have their own strengths and weaknesses and therefore can well complement to each other.

Major Findings and Contributions Our experimental results confirm several important conclusions from prior work: (1) Cache partitioning has a significant performance impact in runtime execution. In our experiments, significant performance improvement (up to 47%) is observed with most workloads. (2) Dynamic partitioning can adapt to a program’s time-varying phase behavior [11]. In most cases, our best dynamic partitioning scheme outperforms the best static partition. (3) QoS can be achieved for all tested workloads if a reasonable QoS target is set.

We have two new insights that are unlikely to obtain from simulation. First, an application may be more sensitive to main memory latencies than its allocated cache space. By giving more cache space to its co-scheduled application, this application’s memory latency can be reduced because of the reduced memory bandwidth contention. In such a way, both co-scheduled programs can have performance improvement, either from memory latency reduction or increased cache capacity. Simulation-based studies are likely to ignore this scenario because the main memory subsystem is usually not modeled in detail. Second, the strong correlations between fairness metrics and the fairness target, as reported in a simulation-based study [8], do not hold in our experiments. We believe that the major reason is the difference in program execution length: Our experiments complete trillions of instructions while the simulation-based experiments only complete less than one billion instructions per program. This discrepancy shows that whole program execution is crucial to gaining accurate insights.

The contributions of this study are threefold: (1) To the best of our knowledge, this is the most comprehensive execution- and measure-based study for multicore cache partitioning. This paper not only confirms some conclusions from simulation-based studies, but also provides new insights into dynamic execution and interaction effects. (2) Our approach provides a unique and efficient option for performance evaluation of multicore processors, which can be a useful tool for researchers with common interests. (3) The proposed schemes can also be further refined for OS kernels

¹We did not use recent quad-core processors because their cache is statically partitioned into two halves, each of which shared by two cores. In other words, they are equivalent to our platform for the purpose of studying cache partitioning.

Metric	Formula
Throughput (IPCs)	$\sum_{i=1}^n (\text{IPC}_{\text{scheme}}[i])$
Average Weighted Speedup [21]	$\frac{1}{n} \sum_{i=1}^n (\text{IPC}_{\text{scheme}}[i] / \text{IPC}_{\text{base}}[i])$
SMT Speedup [14]	$\sum_{i=1}^n (\text{IPC}_{\text{scheme}}[i] / \text{IPC}_{\text{base}}[i])$
Fair Speedup [2]	$n / \sum_{i=1}^n (\text{IPC}_{\text{base}}[i] / \text{IPC}_{\text{scheme}}[i])$

Table 1: Comparing different performance evaluation metrics.

to improve system performance.

2. Adopted Evaluation Metrics in Our Study

Cache Partitioning for Multi-core Processors Inter-thread interference with an uncontrolled cache sharing model is known to cause some serious problems, such as performance degradation and unfairness. A cache partitioning scheme can address these problems by judiciously partitioning the cache resources among running programs. In general, a cache partitioning scheme consists of two interdependent parts, *mechanism* and *policy*. A partitioning mechanism enforces cache partitions as well as provides inputs needed by the decision making of a partitioning policy. In almost all previous studies, the cache partitioning mechanism requires special hardware support and therefore has to be evaluated by simulation. For example, many prior proposals use way partitioning as a basic partitioning mechanism on set-associative caches. Cache resources are allocated to programs in units of ways with additional hardware. Basic measurement support can be provided using hardware performance counters. However, many previous studies also introduce special monitoring hardware such as the UMON sampling mechanism in [11].

A partitioning policy decides the amount of cache resources allocated to each program with an optimization objective. An objective is to maximize or minimize an *evaluation metric* of performance, QoS or fairness, while a *policy metric* is used to drive a cache partitioning policy and ideally it should be identical to the evaluation metric [5]. However, it is not always possible to use evaluation metrics as the policy metrics. For example, many evaluation metrics are weighted against baseline measurements that are only available through offline profiling. In practice, online observable metrics, such as cache miss rates, are employed as proxies for evaluation metrics, such as average weighted speedup. Driven by its policy metric, a cache partitioning policy decides a program’s cache quota either statically through offline analysis or dynamically based on online measurements. A dynamic partitioning policy works in an iterative fashion between a program’s execution *epochs*. At the end of an epoch, measurements are collected or predicted by the partitioning mechanism and the policy then recalculates the cache partition and enforces it in the next epoch.

Performance Metrics in Cache Partitioning Table 1 summarizes four commonly used performance evaluation

Metric	X_i
FM1	$\text{Misses}_{\text{base}}[i] / \text{Misses}_{\text{scheme}}[i]$
FM2	$\text{Misses}_{\text{scheme}}[i]$
FM3	$\text{MissRate}_{\text{base}}[i] / \text{MissRate}_{\text{scheme}}[i]$
FM4	$\text{MissRate}_{\text{scheme}}[i]$
FM5	$\text{MissRate}_{\text{scheme}}[i] - \text{MissRate}_{\text{base}}[i]$

Table 2: Comparing different fairness policy metrics. All metrics are calculated as $\sum_i \sum_j |X_i - X_j|$

metrics. Throughput represents absolute IPC numbers. Average weighted speedup is the speedups of programs over their execution with a baseline scheme – a shared L2 cache in this study. SMT speedup is the sum of program speedups over their executions on a dedicated L2 cache, and fair speedup is the harmonic mean of the speedups over a baseline scheme, which in this study uses a shared cache. Fair speedup evaluates fairness in addition to performance [2]. Weighted speedup, SMT speedup and fair speedup cannot be calculated as online statistics and therefore cannot be the direct targets of dynamic partitioning policies. In this study, in addition to throughput, we employ two performance policy metrics with both static and dynamic policies: combined miss rates, which summarizes miss rates; and combined misses, which summarizes the number of cache misses. Both have been chosen by previous studies [18, 11] and can be observed online with hardware performance counters.

Fairness Metrics Ideally, to achieve fairness, the slowdown (speedup) of each co-scheduled program should be identical after cache partitioning. Even though such a perfect fairness is not always achievable, program slowdowns should be as close as possible. This translates to minimizing the fairness evaluation metric (FM0) [8] calculated as follows:

$$\text{FM0} = \sum_i \sum_j |M_i - M_j|,$$

where $M_i = \text{IPC}_{\text{base}}[i] / \text{IPC}_{\text{scheme}}[i]$ (1)

We also use policy metrics FM1~FM5 from [8] and summarize them in Table 2. These fairness metrics are based on miss rates or the number of misses, which take the form of $\sum_i \sum_j |X_i - X_j|$ as FM0. In our study, all the weighted fairness metrics (FM0, FM1 and FM3) are relative to single core executions with a dedicated L2 cache.

To see how well a fairness policy metric correlates with a fairness evaluation metric, we compute the statistical correlation [15] between them by Equation (2).

$$\text{Corr}(\text{FM}_i, \text{FM}_0) = \text{Cov}(\text{FM}_i, \text{FM}_0) / (\sigma(\text{FM}_i)\sigma(\text{FM}_0)),$$

where $\text{Cov}(\text{FM}_i, \text{FM}_0) = E(\text{FM}_i \text{FM}_0) - E(\text{FM}_i)E(\text{FM}_0)$ (2)

In (2), $\sigma(\text{FM}_i)$ and $E(\text{FM}_i)$ denote the standard deviation and expected value of FM_i respectively. Under a policy driven by the policy metric, we obtain data points for both the policy metric and the evaluation metric by running a

workload with different cache partitionings. The value of $Corr(FM_i, FM_0)$ ranges between -1 to 1 and a value of 1 indicates a perfect correlation between two metrics.

QoS Metrics We consider cases where hard QoS constraints are never violated. If every program in a workload has its own QoS requirement, it is possible that we cannot keep all of them. Therefore in this study we only guarantee QoS for a given program of a workload. This is different from prior work [2] in which QoS summarizes the behavior of an entire workload. The QoS evaluation metric used in this study is defined by Equation (3).

$$QoS = \begin{cases} 1 & (\text{IPC}_{\text{scheme}}[i] / \text{IPC}_{\text{base}}[i] \geq \text{Threshold}) \\ 0 & (\text{IPC}_{\text{scheme}}[i] / \text{IPC}_{\text{base}}[i] < \text{Threshold}) \end{cases} \quad (3)$$

In our study, we assume that baseline IPC is profiled offline from dual-core execution of homogeneous workload with half of the cache capacity allocated to each program. For a given program i , maximizing our QoS metric essentially bounds $\text{IPC}_{\text{scheme}}[i]$ within Threshold in (3). To guarantee QoS, we raise the specified threshold in a policy metric to approach a QoS evaluation metric with a low threshold.

3. OS-based Cache Partitioning Mechanism

Instead of using simulation, our unique approach is to use an OS-based cache partitioning mechanism to emulate a hardware mechanism; and then evaluate the cache policies on top of the software mechanism.

3.1. Static OS-based Cache Partitioning

A static cache partitioning policy predetermines the amount of cache blocks allocated to each program at the beginning of its execution. Our partitioning mechanism is based on a well accepted OS technique called *page coloring* [20], which works as follows. A physical address contains several common bits between the cache index and the physical page number. These bits are referred to as *page color*. A physically addressed cache is therefore divided into non-intersecting regions by page color, and pages in the same color are mapped to the same cache region. We assign different page colors to different processes, thus, cache space is partitioned between cores for running programs. By limiting the physical memory pages within a subset of colors, the OS can limit the cache used by a given process to cache regions of those colors. In our experiments, the Intel dual-core Xeon processor has a 4MB, 16-way set associative L2 cache and the page size is set to 4KB. Therefore, We can break the L2 cache to 64 colors (cache size / page size / cache associativity). Although a hardware mechanism may support a finer granularity of cache allocation, we have found that most proposed policies [18, 8, 12] work at a coarse granularity at this level. Therefore, the coarse granularity is practically not a limitation in our study.

Our implementation is in Linux kernel 2.6.20.3. The kernel uses a buddy system to maintain free memory: free

physical memory pages are organized into multiple free lists, where the k th list contains memory chunks of size of 2^k pages. We have modified the code to divide each free list into multiple lists, each linking free pages with the same color. When a page fault occurs to the process, the kernel searches for free pages in the free lists with allocated colors in a round-robin fashion. When pages are freed, they are added to a list of its color. The modification has negligible code space and run-time overhead.

3.2. Dynamic OS-based Cache Partitioning

Many proposed cache partitioning policies adjust cache quotas among processes dynamically. To support such policies, we extend the basic color-based partitioning mechanism to support *page recoloring*.

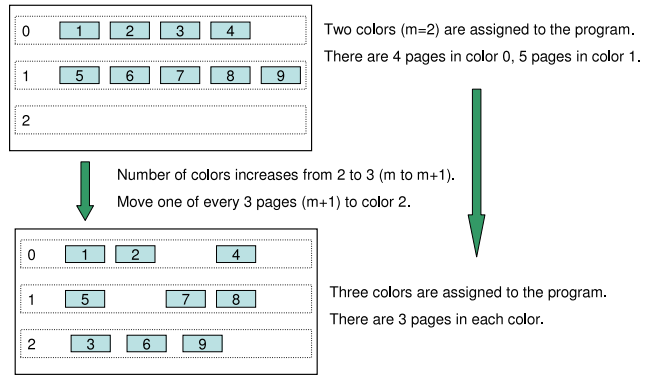


Figure 1: Page recoloring.

Page Recoloring Procedure When a decision is made to increase the cache resource of a given process, i.e. increasing the number of colors used by the process, the kernel will enforce the decision by re-arranging the virtual-physical memory mapping of the process. As shown in Figure 1, if the number of colors used is m , then all virtual memory pages, if not paged out, are mapped onto physical memory pages of those colors. When the number of colors increases to $m + 1$, the kernel assigns one more color to the process and move roughly one of every $m + 1$ of the existing pages to the new color. This process involves allocating physical pages of the new color, copying the memory contents and freeing the old pages. When a decision is made to reduce cache resources, the kernel will also recolor a fraction of virtual memory pages accordingly.

Optimizations to Reduce Runtime Overhead Although the performance overhead of remapping virtual pages is non-negligible, it has little impact on our major research goal of evaluating existing cache partitioning policies. We can measure the run-time overhead and exclude it from the total execution time, so as to emulate a hardware cache partitioning mechanism that does not involve moving memory pages. Nevertheless, we want to make the implementation as efficient as possible.

To reduce the overall overhead, one option is to lower the frequency of cache allocation adjustment. The question is how frequently a dynamic cache partitioning policy needs to adjust the allocation. Using experiments, we find that an *epoch* can be as long as several seconds without affecting the accuracy of evaluation significantly. Additionally, we use a lazy method for page migration: content of a recolored page is moved only when it is accessed. In the implementation, all pages of a process are linked in two arrays of page lists, one indexed by their *current color* and one by their *target color*. When a color is reclaimed from a process, the affected pages are those on the same page list in the first array with index equal to the color, so we do not need to linearly search the virtual address space. To remap these pages, we change their target color (and their location in the data structure). The kernel clears the page’s present bit so that a page fault will occur when the page is accessed. At that time, the accessed virtual page is physically remapped and its current color is changed to match the target color. The performance improvement is significant because in most programs only a small subset of those recolored pages are accessed during an epoch. If a page is re-colored multiple times before it is accessed, its content will be only moved once. With the optimization and use of a five-second epoch length, the average overhead of dynamic partitioning is reduced to 2% of the total execution time. The highest migration overhead we observed is 7%.

3.3. Restrictions of Our Approach

There are two restrictions on our OS-based approach. First, a hardware mechanism may allocate cache blocks at a fine granularity; and it may reallocate cache blocks at a relatively high frequency. However, we believe that for the purpose of evaluation and for most programs, those are not major issues. Almost all existing hardware-based policies are coarse-grain; for example, changing the target cache allocation in unit of 1/16 of the total cache size. The software cache allocation can be as fine as 1/64 of the total cache size on our machine (1/16 is actually used for simplicity). In addition, although a hardware mechanism can dynamically reallocate cache at a high frequency, e.g. every tens of milliseconds, the phase changes of many programs are much less frequent in practice. Most SPEC programs, for example, have phase changes every tens to hundreds of seconds. Our dynamic software mechanism can re-allocate cache every few seconds with small overhead, which is sufficient for this type of workloads. The second restriction is on how to handle the page migration overhead by our approach. Our answer is that the overhead can be measured and excluded for the purpose of evaluating a hardware-based scheme.

4. Dynamic Cache Partitioning Policies for Performance, Fairness and QoS

In this section, we describe the dynamic cache partitioning policies used in our study. All policies adjust the cache partitioning periodically at the end of each epoch. We limit our discussions to the two-core platform used in our experiments. Static partitioning policies are not discussed because our focus there is to evaluate how well a policy metric, if performed with ideal profiling, matches an evaluation metric. The policy design and implementation is not our concern.

Dynamic Partitioning Policy for Performance The policy uses Algorithm 4.1 to dynamically adjust cache partitioning. Four policy metrics can be used to drive the algorithm: throughput (IPCs), combined miss rate, combined misses and fair speedup. We do not consider the non-convexity problem [18] in this study, although the algorithm may be extended to vary the distances of change to address this issue. We did not find that problem in our experiments, which is expected because the unit size of cache allocation is large (256KB). After all, our current algorithm is designed to *evaluate the impact of cache partitioning* but not to handle the worst case. Previous studies use new hardware mechanisms to collect special statistics such as marginal gains [18] or cache miss rate curves [11]. Our algorithm works differently and performance counters available on the processor are sufficient.

Algorithm 4.1: DYNA-PART-FOR-PERF (policy metrics M)

Partition the cache initially as $\frac{n}{2} : \frac{n}{2}$, where n is the total number of colors

```

while Programs are running
do {
  Run one epoch for current partition  $p_0 : p_1$ 
  Try one epoch for each of the two neighboring partitionings:
     $(p_0 - u) : (p_1 + u)$  and  $(p_0 + u) : (p_1 - u)$ 
  Compare  $M$  measurements from
     $p_0 : p_1, (p_0 - u) : (p_1 + u)$  and  $(p_0 + u) : (p_1 - u)$ 
  Choose next partition with the best  $M$  measurement from
     $p_0 : p_1, (p_0 - u) : (p_1 + u)$  and  $(p_0 + u) : (p_1 - u)$ 
}

```

Dynamic Partitioning Policy for Fairness We adopt the methodology for evaluating fairness proposed in a recent study [8]. The evaluation metric is referred to as FM0 (Fairness Metric Zero) and five policy metrics, referred to as FM1 to FM5, are proposed (see Section 2). We adopt their algorithms and implement two dynamic policies based on FM0 and FM4. FM0 is designed to equalize the relative progress of all programs using their baseline execution as the reference point, i.e. to equalize the ratio of the current, cumulative IPC over the baseline IPC. Policy metric FM4 is designed to equalize the cache miss rates. We skip the other metrics to simplify our experiments. This algorithm works in repetitive cycles of two steps: *repartitioning* and *rollback*. The repartitioning step tentatively adjusts the cache partitioning if FM4 difference between two running

programs exceeds $T_{repartition}$, which is a percentage. In the rollback step, the repartitioning decision is committed if the process with more cache allocation has a miss rate reduction of at least $T_{rollback}$. Further details of the algorithm can be found in [8]. $T_{repartition}$ and $T_{rollback}$ are set as 2% and 0.5% respectively, which we found worked well in our experiments.

Dynamic Partitioning Policy for QoS Considerations

The QoS requirement in this study is formulated as follows. For a two-core workload of two programs, the first program is the target program and the second program is the partner program. The QoS guarantee is to ensure that the performance of the target program is no less than X% of a baseline execution of homogeneous workload on a dual-core processor with half of the cache capacity allocated for each program, and meanwhile the performance target is to maximize the performance of the partner program. We assume that the IPC in the baseline execution is known. Prior work [10, 7] uses new hardware support including both capacity and bandwidth sharing for QoS in multicores. Our OS-based implementation can control cache partitioning but not bandwidth partitioning. Instead, it uses cache partitioning to counter the effect of cache sharing. Algorithm 4.2 shows the design details.

Algorithm 4.2: DYNA-PART-FOR-QOS (target program’s baseline IPC)

```

Partition the cache initially as  $\frac{n}{2} : \frac{n}{2}$ , where  $n$  is the total number of colors
while target program is running
do
  Wait until end of current epoch
  Compute the target program’s accumulated IPC
  if the accumulated IPC is less than the baseline IPC
    by certain percentage (2% in this study)
  then
    if the cache allocation to the target program does not
      exceed the maximum
    then Use partitioning  $(p_0 + u) : (p_1 - u)$  for next
      epoch,  $(p_0 + u)$  for the target program
    else stall the partner program
  if the accumulated IPC is greater than the baseline IPC
  then
    if the partner program is stopped
    then resume the partner program
    else Use partitioning  $(p_0 - u) : (p_1 + u)$  for next
      epoch,  $(p_0 - u)$  for the target program

```

5. Experimental Methodology

Hardware and Software Platform We conducted our experiments on a Dell PowerEdge 1950 machine. It has two dual-core, 3.0GHz Intel Xeon 5160 processors and 8GB Fully Buffered DIMM (FB-DIMM) main memory. Each Xeon 5160 processor has a shared, 4MB, 16-way set associative L2 cache. Each core has a private 32KB instruction cache and a private 32KB data cache. The machine has four 2GB 667MT FB-DIMMs. We use default configuration of the processor, which disables next line prefetch but enables hardware prefetch. The snoop filter in the chipset was turned on. We ran benchmarks only on one of the two processors by setting processor affinities for programs.

We use Red Hat Enterprise Linux 4.0 with kernel linux-2.6.20.3. Execution performance results were collected by `perfmon` using `perfmon` kernel interface and `libpfm` library [4]. We divided physical pages into 64 colors. The cache allocation granularity was actually 4 colors bundled together, as it simplified the experiment and we found that it made little difference in the results. When a page needed to be re-colored, a lazy recoloring policy was used as discussed in Section 3. We protected the page by clearing the present bit and setting re-color bit (an unused bit in original linux kernel) in the page table entry corresponding to the page. After that, any access to the page would generate a page fault. In the page fault handler, if the re-color bit was set, page was re-colored.

Workloads We selected the SPEC CPU2006 benchmark suite [16] and compiled them with Intel C++ Compiler 9.1 and Intel FORTRAN Compiler 9.1 for IA32. We selected programs and constructed workloads as follows. First, all 29 programs were classified to four categories. By varying the number of colors that a benchmark uses, we were able to measure the execution time of each program for each possible cache size. To simplify the experiment, we grouped four colors into *super color*. For simplicity, we use color to refer to super color hereafter. We ran each benchmark with both four colors (super colors) and sixteen colors (super colors). Our page-level cache partitioning mechanism partitioned physical memory as well. To avoid page thrashing, we allocated at least two colors (super colors) to each program in our evaluation. Consequently, each program had at least one eighth of total physical memory (1GB) which is larger than memory footprint of any program from SPEC CPU2006 on our experimental machine. This ensures that I/O overhead does not increase for page coloring, which was confirmed in the experiments. As shown in Table 3, six programs had more than 20% performance slowdown when using only four colors (1MB cache), compared with using all sixteen colors (4MB). We referred to them as “red” application as they were sensitive to the L2 cache size. It is predicted that the working set of a red application can be fit into 4MB cache but not 1MB cache. Nine programs had a performance slowdown between 5% and 20%; they were referred to as “yellow” programs. The remaining fourteen programs were further divided into two classes by using the number of L2 cache accesses obtained by performance counter. Programs with more than fourteen L2 cache accesses per 1000 processor cycles were referred to as “green” programs and the rest were referred to as “black” programs; the number fourteen is an arbitrary threshold.

We then constructed the workloads as follows. The red programs had intensive cache accesses and they also demanded a large cache size. The green programs accessed L2 cache extensively but were insensitive to L2 cache size. Their working sets could be too large to fit into a 4MB cache

Class	Slowdown(1M/4M)	L2 access rate	Benchmarks
Red	> 20%	average: 13.7 per 1K cycle	401.bzip2 429.mcf 471.omnetpp 473.astar 482.sphinx3 483.xalancbmk
Yellow	> 5%	average: 13.4 per 1K cycle	403.gcc 437.leslie3d 450.soplex 459.GemsFDTD 465.tonto 470.lbm (400.perlbench 436.cactusADM 464.h264ref)
Green	≤ 5%	≥ 14 per 1K cycle	410.bwaves 434.zeusmp 435.gromacs 453.povray 462.libquantum 481.wrf
Black	≤ 5%	< 14 per 1K cycle	416.gamess 433.milc 444.namd 445.gobmk 447.dealII 456.hammer 458.sjeng 454.calculix

Table 3: Benchmark classification.

Combinations	Workload	Benchmarks	Wo.	Benchmarks	Wo.	Benchmarks
Red + Red	RR1	401.bzip2 473.astar	RR2	429.mcf 482.sphinx3	RR3	471.omnetpp 483.xalancbmk
Red + Yellow	RY1	401.bzip2 403.gcc	RY2	429.mcf 437.leslie3d	RY3	471.omnetpp 450.soplex
	RY4	473.astar 459.GemsFDTD	RY5	482.sphinx3 465.tonto	RY6	483.xalancbmk 470.lbm
Red + Green	RG1	401.bzip2 410.bwaves	RG2	429.mcf 434.zeusmp	RG3	471.omnetpp 435.gromacs
	RG4	473.astar 453.povray	RG5	482.sphinx3 462.libquantum	RG6	483.xalancbmk 481.wrf
Yellow + Yellow	YY1	403.gcc 459.GemsFDTD	YY2	437.leslie3d 465.tonto	YY3	450.soplex 470.lbm
Yellow + Green	YG1	403.gcc 410.bwaves	YG2	437.leslie3d 434.zeusmp	YG3	450.soplex 435.gromacs
	YG4	459.GemsFDTD 453.povray	YG5	465.tonto 462.libquantum	YG6	470.lbm 481.wrf
Green + Green	GG1	410.bwaves 453.povray	GG2	434.zeusmp 462.libquantum	GG3	435.gromacs 481.wrf

Table 4: Workload mixes.

or small enough to fit into a 1MB cache. Intuitively, when a red program is co-scheduled with a green program, we should give the red program a larger cache capacity since it benefits more from an increased cache quota than the green program. Yellow programs were included because they are moderately affected by cache performance. Those black programs were not interesting for this study because their performance are not sensitive to the shared cache performance. We constructed 27 workloads using six red, six yellow and six green programs. The workloads are shown in Table 4. Some programs had more than one reference input; we ran all inputs and considered them as a single run. Each workload consisted of two runs from two programs. If one run finished earlier, we re-ran it until every run finished once. Only statistics for the first run were used to compute performance, fairness and QoS.

6. Evaluation Results

6.1. Cache Partitioning for Performance

Evaluation Approach The design of our policies was adapted from previous hardware-based studies [18, 11], and we implemented both static and dynamic policies. The three static policies were evaluated using an offline approach: we tested all possible cache partitionings for each program and collected the data for each evaluation metric and data for each policy metric. Then, we decided the choice of cache partitioning according to each policy and compare it with the optimal choice. As for dynamic policies, for simplicity we only present results with a dynamic policy using the total misses as the policy metric.

Performance Impact of Static Policies For each type of workloads, Figure 2 shows the average improvement with the best static partitioning of each workload over the shared cache. The difference is most significant in the RG-type

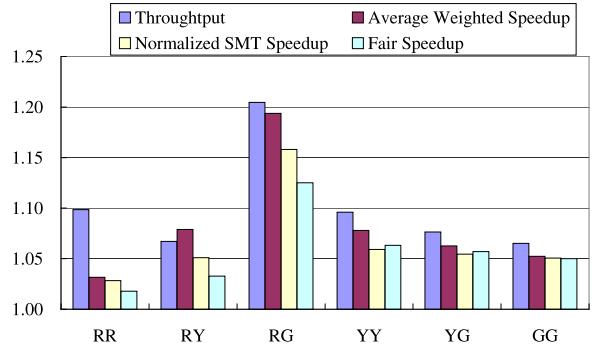


Figure 2: Performance of optimal partitioning for four evaluation metrics.

workloads, where the throughput is improved by more than 20%. Cache partitioning is also beneficial to the other workloads, where the average improvement ranges from 2% to 10%. RG-type workloads are most sensitive to cache partitioning because both programs in a workload are memory intensive, while the R-type program is sensitive to cache capacity and the G-type program is insensitive. We have detailed analysis below to show that the G-type program may have better performance in two-core execution even if it receives less cache capacity. Therefore, allocating more cache space to the R-type program improves the overall performance significantly. Among all evaluation metrics, the improvement on throughput is the most significant. This is because the other metrics, while a consideration of with the overall instruction throughput, have the effect to balance performance of all programs. A point worth mentioning is that our experiments were conducted on a dual-core processor. We believe that the performance improvement by cache partitioning will increase as additional cores are added on real machines.

Table 5 shows the average weighted speedup of all RG-type (one R-type program plus one G-type program) and

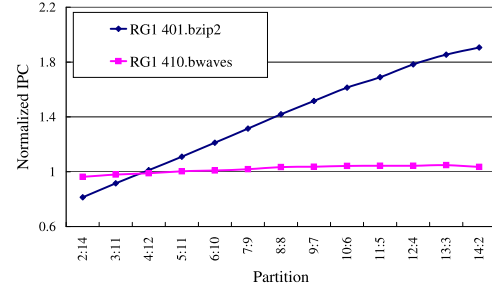
Partition	2:14	3:13	4:12	5:11	6:10	7:9	8:8	9:7	10:6	11:5	12:4	13:3	14:2
RG1	0.89	0.95	1.00	1.06	1.11	1.17	1.23	1.28	1.33	1.37	1.41	1.45	1.47
RG2	0.85	0.89	0.91	0.94	0.95	0.97	0.99	1.00	1.01	1.01	1.01	1.03	1.03
RG3	0.91	0.92	0.94	0.96	0.97	0.98	0.99	1.00	1.00	1.00	1.01	1.01	0.96
RG4	0.85	1.06	1.08	1.10	1.11	1.13	1.12	1.14	1.14	1.10	1.17	1.17	1.18
RG5	0.94	0.93	0.93	0.95	0.98	1.01	1.05	1.11	1.14	1.19	1.24	1.30	1.37
RG6	0.91	0.95	0.97	0.99	1.01	1.03	1.04	1.06	1.07	1.09	1.09	1.11	1.11
RR1	0.86	0.89	0.91	0.93	0.95	0.97	0.99	1.01	1.01	1.01	1.02	1.01	0.98
RR2	1.05	1.05	1.03	1.02	1.01	1.01	0.98	0.97	0.99	0.95	0.97	1.00	1.01
RR3	1.00	1.01	1.01	1.02	1.02	1.02	1.02	1.01	1.01	1.00	0.98	0.97	0.93

Table 5: The average weighted speedups of all selected RG- and RR-type workloads for all cache partitionings experimented. We have raw data for all selected workloads, all evaluation metrics and all partitionings (not shown).

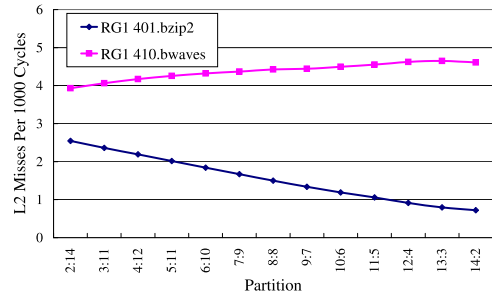
RR-type (two R-type programs) workloads, respectively, for all cache partitionings from 2:14 to 14:2. We have raw data for all workloads, all evaluation metrics and all partitionings, but only present this set of data due to space limitations. Average weighted speedup here is the average of the speedups of each program over their execution on a baseline configuration that uses a shared L2 cache. Therefore, each number in Table 5 represents the relative performance of a cache partitioning over the shared cache. Overall, the RG-type workloads have the most significant improvements and the RR-type workloads have the smallest ones. This is expected because R-type programs are sensitive to cache capacity and G-type programs are insensitive. RR-type workloads are relatively insensitive to cache allocation because both programs demand large cache capacity.

We have the following observations. First, cache partitioning does make significant performance improvement to RG-type workloads. For example, workload RG1 (of 401.bzip2 and 410.bwaves) has the largest average weighted speedup (47% improvement) at 14:2 partitioning over shared cache. The 14:2 partitioning is 20% better than the 8:8 even partitioning (typical hardware private partitioning) and 65% better than the worst 2:14 partitioning. The best partitioning for workload RG5 is 14:2 and it is 37%, 30% and 46% better than the shared cache, the even partitioning and the worst case 3:13 partitioning. Second, the even partitioning is usually better than uncontrolled cache management. Additionally, for RG-type workloads, it is usually the case, but not always, that giving the R-type program the maximum cache capacity will yield the best performance. Finally, RR-type workloads are generally insensitive to cache partitioning, although the performance of each individual program is sensitive to the partitioning as we observed in the raw data (not shown).

We also have an interesting finding after looking into individual workloads in more detail. Figure 3(a) shows the normalized speedups of programs 401.bzip2 and 410.bwaves in workload RG1. Surprisingly, shifting more cache capacity from 410.bwaves to 401.bzip2 improves the performance of *both* programs. This is counter-intuitive because the performance of 410.bwaves should drop with less cache capacity. Here is our explanation: when de-



(a) Normalized IPC



(b) Misses Per 1000 Cycles

Figure 3: Workload RG1 under all possible static partitions

creasing 401.bzip2’s cache allocation, its cache miss rate increases significantly and so does the memory bandwidth utilization. Because of the queuing delay at the memory controller, the cache miss penalty increases for both programs, which degrades the performance of 410.bwaves even though its cache miss rate drops. To confirm it, Figure 3(b) shows the cache miss rate of the two programs for all partitionings: The miss rate of 401.bzip2 decreases sharply with increasing capacity and that of 410.bwaves remains almost constant. Program 410.bwaves has a relative high cache miss rate, and therefore its performance is sensitive to the cache miss penalty. Therefore, the performance of both programs is improved when more cache capacity is allocated to 401.bzip2. We have the same observation in two other RG-type workloads and one YG-type workload, namely RG4, RG5 and YG5.

A Comparison of Policy Metrics and Evaluation Metrics Figure 4 compares the best cache partitioning under each policy metric with the best partitioning under evalua-

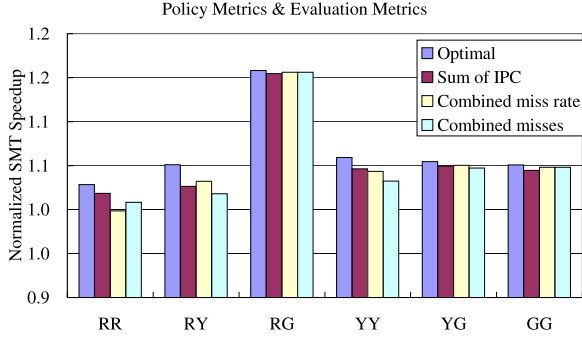


Figure 4: Comparison of the performance of static OS-based cache partitioning by the three policy metrics (sum of IPC, combined miss rate, and combined misses) with the optimal performance by evaluation metrics normalized SMT speedup.

tion metric normalized SMT speedup². To simplify the discussion, it only shows the average improvements of the partitioning policies over the shared cache for each workload type. We have the following observations. First, all three policy metrics are good indications of the performance under the four evaluation metrics. They are close to the best partitioning under each metric in most cases. Second, the average performance improvement by using a policy metric is sensitive to the type of workload. The RG-type workloads have the highest improvements under all evaluation metrics by using any of the three policy metrics. By comparison, the RR-type workloads have smallest improvement by any policy metric and under any evaluation metric. Finally, the three policy metrics are very comparable to all evaluation metrics. For the evaluation metric throughput, sum of IPC is the best policy metric overall because both metrics are based on the combined instruction throughput. We also found that combined miss rates are slightly better than combined misses as a policy metric, although the latter one was used in several recent studies.

Results of Dynamic Policy It is expected that dynamic schemes would generally outperform static ones. However, note that the previous results of static partitioning assume complete profiling results while dynamic partitioning here starts with no knowledge of the workload. Figure 5 compares the average weighted speedups by using combined miss rates as the policy metric. Due to space limitations, we only present this policy metric and only with the evaluation metric of average weighted speedup. We have the following observations. First, the dynamic policy, which does not have any pre-knowledge about a workload, is very comparable to the static policy with ideal profiling input. It outperforms the static policy for some workloads and under-performs for some others. For RG-type workloads, static policy slightly outperforms the dynamic one because it always allocates most cache capacity to the R-

²We also have data for other three evaluation metrics. Due to space limitations, we do not show them in the figure.

type program. The dynamic policy has a disadvantage for not sticking to the best partitioning. When compared with the shared cache, the dynamic policy improves the performance for many workloads (by up to 39%). Occasionally it degrades the performance but only slightly. Note that even the best static partitioning may occasionally degrade the performance slightly.

To analyze the behavior of the dynamic policy, we look into workloads RG1 and YY2 for more details. For workload RG1 consisting of 401.bzip2 and 410.bwaves, the best static partitioning is 14:2, which is predicted correctly by the static partitioning policy. The dynamic policy swings between 13:3 and 14:2, therefore it uses the suboptimal 13:3 partitioning for a large fraction of time. Additionally, it also takes some time to reach the optimal 14:2 partitioning. Workload YY2 is an example that favors the dynamic partitioning. Its second program 465.tonto has alternative high IPC phases with low cache miss rate and low IPC phases with high cache miss rate. The dynamic policy responds to the phase changes and therefore outperforms the static partitioning.

6.2. Fairness of Cache Partitioning Policies

Evaluation Approach We first collected the fairness statistics by using static partitionings. Then, as discussed in Section 2, we have implemented two dynamic partitioning policies that targets metrics FM0 and FM4, respectively. We want to answer the following questions: (1) Are there strong correlations between policy metrics FM1-FM5 with evaluation metric FM0 on real machines? A previous simulation-based study [8] has reported strong correlations from those policies except FM2. (2) What would happen if profiling data for FM0 is available and the data is used to decide cache partitioning?

We use hardware performance counters to collect the cache miss, cache access and IPC data and then use Equation (2) to quantify the correlation. As done in [8], for each workload we use n static cache partitionings to get the data series, where $n = 13$ is the number of selected static cache partitionings.

Fairness of Static Cache Partitioning The first five columns of table 6 show the average fairness, measured by the evaluation metric FM0, of the static partitionings selected by the five static fairness policies. The first column represents an upper bound on the fairness that a static policy can achieve. The number in bold type indicates the best static policy on average for a given type workloads. Our major finding is that that none of the five policies can approach the fairness of the best static partitioning. This will be discussed in more detail as we present the metric correlations. The static policies driven by FM2 and FM4, which equalize the number of cache misses and cache miss rates, respectively, are not as good as the other three policies. This

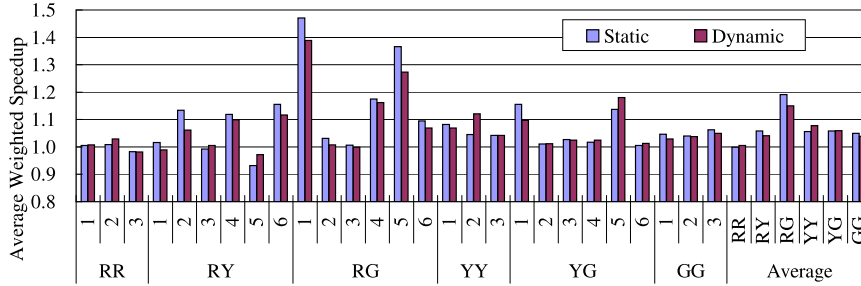


Figure 5: Performance comparison of static and dynamic partitioning policies which use combined miss rates as policy metrics.

Policy	Static-Best	Static-FM1	Static-FM2	Static-FM3	Static-FM4	Static-FM5	Dynamic-Best	Dynamic-FM4
RR	0.015	0.281	0.260	0.281	0.348	0.202	0.015	0.204
RY	0.071	0.284	0.481	0.291	0.359	0.214	0.041	0.242
RG	0.025	0.119	0.195	0.119	0.162	0.075	0.037	0.225
YY	0.093	0.168	0.713	0.168	0.465	0.190	0.055	0.276
YG	0.115	0.217	0.349	0.226	0.204	0.174	0.056	0.185
GG	0.008	0.038	0.097	0.038	0.097	0.037	0.011	0.054

Table 6: Fairness of the static and dynamic partitioning policies. Lower value mean less difference in program slowdown and better fairness.

is consistent with the previous study [8] and is expected because FM2 and FM4 do not use any profiled IPC data from single-core execution, on which the evaluation metric FM0 is defined. Between the two, FM4 is better than FM2. Additionally, the policy driven by FM5 achieves the best fairness overall. Finally, the RR-, RY- and YY-type workloads are more difficult targets for fairness than the others because in those workloads both programs are sensitive to L2 cache capacity.

Correlations Between the Policy Metrics and the Evaluation Metrics Figure 6 shows the quantified correlations (see Section 2) between FM1, FM2, FM4, FM5 and the evaluation metric FM0. A number close to 1.0 indicates a strong correlation. FM2 is not included because it has been shown to have a poor correlation with FM0 [8], and it is confirmed by our data. In contrast to the previous study, we found that none of the policies had a consistently strong correlation with FM0. Overall FM5 has a stronger correlation with FM0 than the other three policies metrics for RY-, RG-, YY- and YG-type workloads. However, it is the worst one for RR- and GG-type workloads.

There are three reasons why our findings are different from simulation results. First of all, our workloads are based on SPEC CPU2006 benchmark suite, while the previous study uses SPEC CPU2000 plus *mst* from Olden and a tree-related program. Most importantly, we are able to run the SPEC programs with the reference data input, while the previous study run their SPEC programs with the test data input. We believe that the use of test input was due to the simulation time limitation. Second, most runs in our experiments complete trillions of instructions (micro-ops on Intel processor) for a single program, while the previous study only completes less than one billion instructions on average. Additionally, our hardware platform has 4MB L2 cache per processor compared with 512KB L2 cache in the previous

study, which may also contribute to the difference. After all, our results indicate that better understanding is needed in the study of fairness policy metrics.

Fairness by Dynamic Cache Partitioning We intend to study whether a dynamic partitioning policy may improve fairness over the corresponding static policy. First, we have implemented a dynamic policy that directly targets the evaluation metric FM0. It assumes the pre-knowledge of single-core IPC of each program, and uses it to adjust the cache partitioning at the end of each epoch. Specifically, if a program is relatively slow in its progress, i.e. its ratio of the current IPC (calculated from the program’s start) over the single-core IPC is lower than that of the other program, then it will receive one more color for the next epoch. We have also implemented a dynamic partitioning policy based on the FM4 metric. To simplify our experiments, we did not include the other policy metrics in the experiments. The right part of Table 6 shows the performance of the dynamic policies. As it shows, the dynamic policy driven by FM0 achieves almost ideal fairness. Note that the policy does require the profiling data for single-core execution and we assume the data are available. The dynamic policy driven by FM4 outperforms the static one for all types of workloads except the RG type. The exception is possible if one program in the workload always makes relatively faster progress than the other one with any possible partitioning, and therefore the partitioning that mostly counters the imbalance should always be used.

6.3. QoS of Cache Partitioning

Evaluation Approach In our experiment, the QoS threshold is set to 95% (see Section 2). Note that during multicore execution the performance of the target program will be affected by not only the cache capacity used by the partner program but also by its usage of L2 cache and memory bandwidth. We assume that the L2 cache controller and

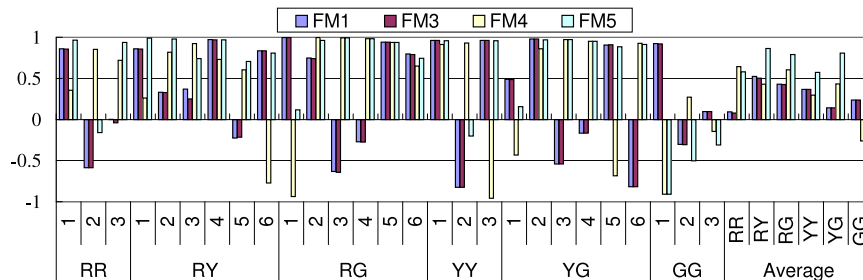


Figure 6: Correlation between fairness policy metrics (FM1, FM3, FM4 and FM5) and fairness evaluation metrics (FM0). FM2 is not shown because it has poor correlation with FM0.

memory controller use some fair access scheduling, which is true in our hardware platform. To counter the effect of bandwidth sharing, the target program may need to have more than half of the cache capacity, and in the worst case the partner program may have to be stopped temporarily.

Evaluation Results Figure 7 shows the performance of the target programs, the partner programs and the overall performance of all workloads. The target program is always the first program in the program pair. The performance of the target and partner program is given by the IPC of each program normalized to its baseline IPC. The baseline IPC is profiled offline from dual-core execution of homogeneous workload with half of the cache capacity allocated for each program. The overall performance is given by the throughput (combined IPC) normalized by that of the performance-oriented dynamic policy. The IPCs are collected when the target program completes its first run. Figure 7(a) shows the performance with static cache capacity partitioning (8:8). By static capacity partitioning only, without bandwidth partitioning, twelve target programs do not meet the 95% QoS requirement. The normalized performance of the target program, 429.mcf, of RY2 is only 67%. On average for all workloads, it achieves 95% of the throughput of the performance-oriented policy. With dynamic cache partitioning policy designed for QoS, as shown in the figure 7(b), all target programs meet the 95% QoS requirement. The normalized performance of target program ranges from 96% to 188%, and the average is 113%. The normalized performance of the partner program ranges from 69% to 171%, and the average is 95%. Furthermore, the QoS-oriented policy does sacrifice a fraction of performance to meet the QoS requirement. On average for all workloads, it achieves 90% of the throughput of the performance-oriented policy.

In summary, without bandwidth partitioning, the static cache capacity partitioning can not guarantee to meet the QoS requirement. The results also indicate that L2 cache and memory bandwidth partitioning as proposed in [7] is needed to meet the QoS requirement. When such a bandwidth partitioning mechanism is not available, our dynamic cache partitioning policy can serve as an alternative approach to meet the QoS requirement of target programs and

let the partner programs utilize the rest of cache resources.

7. Related Work

Cache Partitioning for Multicore Processors Most multicore designs have chosen a shared last-level cache for simple cache coherence and for minimizing overall cache miss rates and memory traffic. Most proposed approaches have added cache partitioning support at the micro-architecture level to improve multicore performance [9, 18, 11]. Several studies highlighted the issues of QoS and fairness [6, 10, 8, 5, 2]. There have been several studies on OS-based cache partitioning policies and their interaction with the micro-architecture support [12, 3]. Our research is conducted on a real system with a dual-core processor without any additional hardware support. Our work evaluates multicore cache partitioning by running programs from SPEC CPU2006 to completion, which is not feasible with the above simulation-based studies.

Page Coloring Page coloring [20] is an extensively used OS technique for improving cache and memory performance [1]. Sherwood et al. [13] proposed compiler and hardware approaches to eliminate conflict misses in physically addressed caches. To the best of our knowledge, it is the first work proposing the use of page coloring in multicore cache management. In their paper, only cache miss rates for a 4-benchmark workload on a simulated multicore processor were presented. In comparison, our recoloring scheme is purely based on software and we are able to conduct a comprehensive cache partitioning study on a commodity multicore processor with the page coloring scheme. A very recent study by Tam et al. [19] implemented a software-based mechanism to support static cache partitioning on multicore processors. Their work is based on page coloring and thus shares several similarities with ours. Our work differs significantly from [19] in the following aspects: (1) In addition to static partitioning, our software layer also supports dynamic partitioning policies with low overhead. We have therefore been able to capture programs' phase-changing behavior and draw important conclusions regarding dynamic cache partitioning schemes. (2) We have conducted one of the most comprehensive cache partitioning studies with different policies optimizing performance,

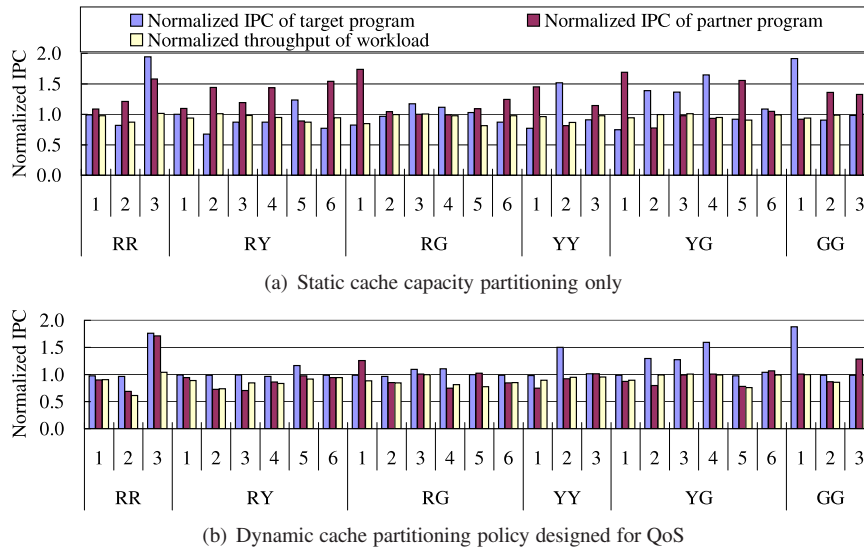


Figure 7: Normalized performance

fairness and QoS objectives.

8. Conclusions and Future Directions

We have designed and implemented an OS-based cache partitioning mechanism on multicore processors. Using this mechanism, we have studied several representative cache partitioning policies. The ability of running workloads to completion has allowed us to confirm several key findings from simulation-based studies. We have also gained new insights that are unlikely to obtain by simulation-based studies.

Ongoing and future work is planned along several directions. First, we will refine our system implementation, to further reduce dynamic cache partitioning overhead. Second, we plan to make our software layer available for the architecture community by adding an easy user interface. Third, our software provides us with the ability to control data locations in the shared cache. With a well defined cache partitioning interface, we are conducting cache partitioning research at the compiler level, for both multiprogramming and multithreaded applications.

Acknowledgments

We thank the constructive comments from the anonymous referees. This research was supported in part by the National Science Foundation under grants CCF-0541366, CNS-0720609, CCF-0602152, CCF-072380 and CHE-0121676.

References

- [1] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam. Compiler-directed page coloring for multiprocessors. In *Proc. ASPLOS'96*, pages 244–255, 1996.
- [2] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proc. ICS'07*, 2007.
- [3] S. Cho and L. Jin. Managing distributed, shared L2 caches through OS-level page allocation. In *Proc. MICRO'06*, pages 455–468, 2006.
- [4] Hewlett-Packard Development Company. *Perfmon project*. <http://www.hpl.hp.com/research/linux/perfmon>.
- [5] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *Proc. PACT'06*, pages 13–22, 2006.
- [6] R. Iyer. CQoS: a framework for enabling qos in shared caches of cmp platforms. In *Proc. ICS'04*, pages 257–266, 2004.
- [7] R. Iyer, L. Zhao, F. Guo, Y. Solihin, S. Markineni, D. Newell, R. Ilkikal, L. Hsu, and S. Reinhardt. QoS policy and architecture for cache/memory in CMP platforms. In *Proc. SIGMETRICS'07*, 2007.
- [8] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proc. PACT'04*, pages 111–122, 2004.
- [9] C. Liu, A. Sivasubramaniam, and M. Kandemir. Organizing the last line of defense before hitting the memory wall for cmps. In *Proc. HPCA'04*, page 176, 2004.
- [10] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *Proc. ISCA'07*, 2007.
- [11] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proc. MICRO'06*, pages 423–432, 2006.
- [12] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven CMP cache management. In *Proc. PACT'06*, pages 2–12, 2006.
- [13] T. Sherwood, B. Calder, and J. Emer. Reducing cache misses using hardware and software page placement. In *Proc. ICS'99*, pages 155–164, 1999.
- [14] A. Snaveley, D. M. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Proc. ASPLOS'02*, pages 66–76, June 2002.
- [15] G. W. Snedecor and W. G. Cochran. *Statistical Methods*, pages 172–195. Iowa State University Press, sixth edition, 1967.
- [16] Standard Performance Evaluation Corporation. *SPEC CPU2006*. <http://www.spec.org>.
- [17] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proc. HPCA'02*, pages 117–128, 2002.
- [18] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.
- [19] D. Tam, R. Azimi, L. Soares, and M. Stumm. Managing shared L2 caches on multicore systems in software. In *WIOSCA'07*, Jun. 2007.
- [20] G. Taylor, P. Davies, and M. Farmwald. The TLB slice—a low-cost high-speed address translation mechanism. In *Proc. ISCA'90*, pages 355–363, 1990.
- [21] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *Proc. MICRO'01*, pages 318–327, 2001.