Towards Practical Page Coloring-based Multi-core Cache Management *

Xiao Zhang Sandhya Dwarkadas Kai Shen

Department of Computer Science, University of Rochester

{xiao, sandhya, kshen}@cs.rochester.edu

Abstract

Modern multi-core processors present new resource management challenges due to the subtle interactions of simultaneously executing processes sharing on-chip resources (particularly the L2 cache). Recent research demonstrates that the operating system may use the page coloring mechanism to control cache partitioning, and consequently to achieve fair and efficient cache utilization. However, page coloring places additional constraints on memory space allocation, which may conflict with application memory needs. Further, adaptive adjustments of cache partitioning policies in a multi-programmed execution environment may incur substantial overhead for page recoloring (or copying).

This paper proposes a hot-page coloring *approach—enforcing coloring on only a small set of frequently accessed (or* hot*) pages for each process. The cost of identifying hot pages online is reduced by leveraging the knowledge of spatial locality during a page table scan of access bits. Our results demonstrate that hot page identification and selective coloring can significantly alleviate the coloring-induced adverse effects in practice. However, we also reach the somewhat negative conclusion that without additional hardware support, adaptive page coloring is only beneficial when recoloring is performed infrequently (meaning long scheduling time quanta in multi-programmed executions).*

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Design studies, Performance attributes

General Terms Management, Performance

Keywords Multi-core, Resource management, Cache partitioning, Page coloring

1. Introduction

Today's operating systems manage multi-core processors in a time-shared manner similar to traditional single-core uniprocessor systems. While some attention is paid to locality and to balancing load among the multiple cores, the additional challenges due to the subtle interactions of simultaneously executing processes sharing on-chip resources have not been incorporated into mainstream operating systems, largely due to the complex nature of the interactions. For example, the L2 cache is a shared resource that can result in unexpected performance anomalies due to contention or unfair allocation. The performance of a process that would normally have been high due to the cache being large enough to fit its working set could be severely impacted by a simultaneously executing process with high cache demand, resulting in the first process's cache lines being evicted.

Without specific hardware support to control cache sharing, the operating system's only recourse in a physically addressed cache is to control the virtual to physical mappings used by individual processes. In today's processors, the shared cache is normally physically indexed. Traditional page coloring [Taylor 1990] attempts to ensure that contiguous pages in virtual memory are allocated to physical pages that will be spread across the cache. In order to accomplish this, contiguous pages of physical memory are allocated different colors, with the maximum number of colors being a function of the size and associativity of the cache relative to the page size. The free page list is organized to differentiate these colors, and contiguous virtual pages are guaranteed to be assigned distinct colors. A different use of page coloring also allows the operating system to restrict a process's accesses so that it utilizes only a subset of the cache. The shared cache space can thereby be partitioned among multiple simultaneously executing applications on a multi-core platform.

Previous work [Tam 2007, Lin 2008, Soares 2008] has demonstrated the potential for improved performance and fair resource utilization via page color restriction. However,

* This work was supported in part by the U.S. National Science Foundation (NSF) grants CNS-0411127, CAREER Award CCF-0448413, CNS-0509270, CNS-0615045, CNS-0615139, CCF-0621472, CCF-0702505, and CNS-0834451; by NIH grants 5 R21 GM079259-02 and 1 R21 HG004648-01; and by several IBM Faculty Partnership Awards.

EuroSys'09 April 1–3, 2009, Nuremberg, Germany.

several challenges remain. The first issue is that of constraining the allocated memory space. Imposing page color restrictions on an application implies that only a portion of the memory can be allocated to this application. When the system runs out of pages of a certain color, the application is under memory pressure while there still may be abundant memory in other colors. This application can either evict some of its own pages to secondary storage or steal pages from other page colors. The former can result in dramatic slowdown due to page swapping while the latter may yield negative performance effects on other applications due to cache conflicts.

Another issue is the high overhead of online recoloring in a dynamic, multi-programmed execution environment. An adaptive system may require online adjustments of the cache partitioning policy (*e.g.*, context switch at one of the cores brings in a new program with different allocation and requirements from the program that was switched out). Such an adjustment requires a change of color for some application pages. Without special hardware support, recoloring a page implies memory copying, which takes several microseconds on commodity platforms. Frequent recoloring of a large number of application pages may incur excessive overhead that more than negates the benefit of page coloring.

This paper proposes a *hot-page coloring* approach in which cache mapping colors are only enforced on a small set of frequently accessed (or *hot*) pages for each process. We present an efficient approach to tracking application page hotness on-the-fly. We periodically scan page table entries, but reduce the cost by leveraging the spatial locality inherent in access patterns to jump over multiple entries, while at the same time back-tracking when the *guess* is incorrect. Hot-page coloring may realize much of the benefit of all-page coloring, but with reduced memory space allocation constraint and much less online recoloring overhead in an adaptive and dynamic environment.

The rest of this paper is organized as follows. We describe background on the page coloring mechanism and cache partition policies in Section 2. Section 3 presents our page hotness tracking mechanism and discusses its general utility beyond supporting hot-page coloring in this paper. In Section 4, we utilize hotness-based page coloring to mitigate memory allocation constraints and expensive recoloring. Using a Linux-based implementation, Section 5 evaluates the effectiveness and overhead of our hot-page coloring approach. Section 6 describes related work and we conclude in Section 7.

2. Background

Page coloring Page coloring is a software technique that controls the mapping of physical memory pages to a processor's cache blocks. Memory pages that map to the same cache blocks are assigned the same color (as illustrated by Figure 1). By controlling the color of pages assigned to an application, the operating system can manipulate cache blocks at the granularity of the page size times the cache associativity. This granularity is the unit of cache space that can be allocated to an application. The maximum number of colors that a platform can support is the cache line size multiplied by the number of sets and divided by the page size.

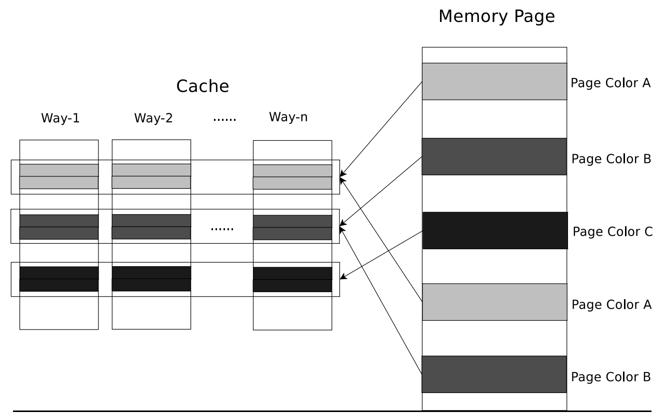

Figure 1. An illustration of the page coloring technique.

Page coloring was first implemented in the MIPS operating system to improve performance stability (by matching virtual and physical page colors, an application's cache behavior remains the same even when pages are remapped in physical memory [Taylor 1990]). More commonly, it was employed to distribute cache accesses evenly across the whole cache and subsequently reduce cache misses within a single application [Kessler 1992, Romer 1994, Bugnion 1996, Sherwood 1999]. Recently, several studies have recognized the potential of utilizing page coloring to manage shared cache space on multi-core platforms [Tam 2007, Lin 2008, Soares 2008].

Cache partitioning policies Given page coloring-based cache partitioning, the operating system must effect an appropriate policy in terms of how the cache space is allocated. Such policies are usually based on certain characterizations of application performance under a given cache allocation. One such characterization, the application cache miss ratio at each possible allocation (called cache miss ratio curve, or MRC), has been used in past research on cache space management (and in particular to minimize the overall cache miss ratio [Stone 1992, Suh 2001]). Additional characterizations such as the stall rate curve [Tam 2007] may also be employed to guide the cache partition policy.

Cache partitioning policies are typically devised to achieve high application performance [Stone 1992, Suh 2001, Tam 2007, Lin 2008] or better fairness [Kim 2004, Hsu 2006]. The fairness objective can mean equal use of resources or equal impact on performance. Policies that gear toward performance alone (*e.g.*, maximizing the whole system instruction throughput or IPC) may suffer on the fairness measure (*e.g.*, starvation of programs that inherently contribute little to the overall IPC). Policies that consider both performance and fairness have also been evaluated. As an example, our previous work [Zhang 2007] evaluates a performance measure (geometric mean of each application's normalized performance) as well as a fairness measure (coefficient of variation of all application performance).

Many cache partitioning policies determine different allocation for different sets of competing applications. In a dynamic, multi-programmed execution environment where the applications executing simultaneously may change over time, this implies that cache allocations must be adaptively adjusted. Static partitioning may also be employed in such dynamic environments. One example is the equal partitioning policy that allocates a fixed (and equal) proportion of the cache space to each application. Equal partitioning follows the simple fairness heuristic of equal resource usage.

3. Page Hotness Identification

Our hot-page coloring approach builds on effective identification of frequently accessed pages for each application. Its overhead must be kept low for online continuous identification during dynamic application execution.

3.1 Sequential Page Table Scan

The operating system (OS) has two main mechanisms for monitoring access to individual pages. First, on most hardware-filled TLB platforms (*e.g.*, Intel processors), each page table entry has an access bit, which is automatically set by hardware when the page is accessed [Intel]. By periodically checking and clearing this access bit, one can estimate each page's access frequency (or hotness). The second mechanism is via page read/write protection so that accesses to a page will be caught by page faults. One drawback for the page protection approach is the high page fault overhead. On the other hand, it has the advantage (in comparison to the access bit checking) that overhead is only incurred when pages are indeed accessed. Given this tradeoff, Zhou *et al.* [Zhou 2004] proposed a combined method for tracking an application's page accesses—link together frequently accessed pages and periodically check their access bits; invalidate those infrequently accessed pages and catch accesses to them via page faults.

However, traversing the list of frequently accessed pages involves pointer chasing, which exhibits poor efficiency on modern processor architectures. In contrast, a sequential scan of the application (or its corresponding process)'s page table can be much faster on platforms with high peak memory bandwidth and hardware prefetching. For a set of 12 SPECCPU2000 applications, our experiments on a dual-core Intel Xeon 5160 3.0 GHz "Woodcrest" processor shows that the sequential table scan takes tens of cycles (36 cycles on average) per page entry while the list traversal takes hundreds of cycles (258 cycles on average) per entry. Given the trend that memory latency improvement lags memory bandwidth improvement [Patterson 2004], we chose sequential table scan over random pointer chasing in our design.

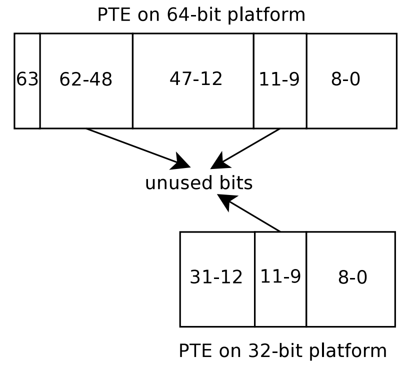

Figure 2. Unused bits of page table entry (PTE) for 4K page on 64-bit and 32-bit x86 platforms. Bits 11-9 are hardware defined unused bits for both platforms [IA32-manual, AMD64-manual]. Bits 62-48 on the 64-bit platform are reserved but not used by hardware right now. Our current implementation utilizes 8 bits in this range for maintaining the page hotness counter.

We consider several issues in the design and implementation of the sequential page table scan-based hot page identification. An accurate page hotness measure requires cumulative statistics on continuous page access checking. Given the necessity of checking the page table entries and the high efficiency of sequential table scan, we maintain the page access statistics (typically in the form of an access count) using a small number of unused bits within the page table entry. Specifically, we utilize 8 unused page table entry bits in our implementation on a 64-bit Intel platform (as illustrated in Figure 2). Some, albeit fewer, unused bits are also available in the smaller page table entry on 32-bit platforms. Fewer bits may incur more frequent counter overflow but do not fundamentally affect our design efficiency. In the worst case when no spare bit is available, we could maintain a separate "hotness counter table" that shadows the layout of the page table. In that case, two parallel sequential table scans are required instead of one, which would incur slightly more overhead.

In our hardware-implemented TLB platform, the OS is not allowed to directly read TLB contents. With hypothetical hardware modification to allow this, we could then sample TLB entries to gather hotness information. Walking through TLBs (*e.g.*, 256 entries on our experimental platform) is much lighter-weight than walking through the page table (usually 1 to 3 orders of magnitude larger than the TLB).

The hotness counter for a page is incremented at each scan that the page is found to be accessed. To deal with potential counter overflows, we apply a fractional decay (*e.g.*, halving or quartering the access counters) for all pages when counter overflows are possibly imminent (*e.g.*, every 128/192 scans for halving/quartering). Applied contin-

Benchmark	# of physically allocated pages	# of excess page table entries
gzip	46181	1141
wupwise	45008	1617
swim	48737	1617
mgrid	14185	1582
applu	45981	4135
mesa	2117	1255
art	903	1028
mcf	21952	1334
equake	12413	1057
parser	10183	699
bzip	47471	954
twolf	1393	88

Table 1. Memory footprint size and number of excess page table entries for 12 SPECCPU2000 benchmarks. The excess page table entries are those that do not correspond to physically allocated pages.

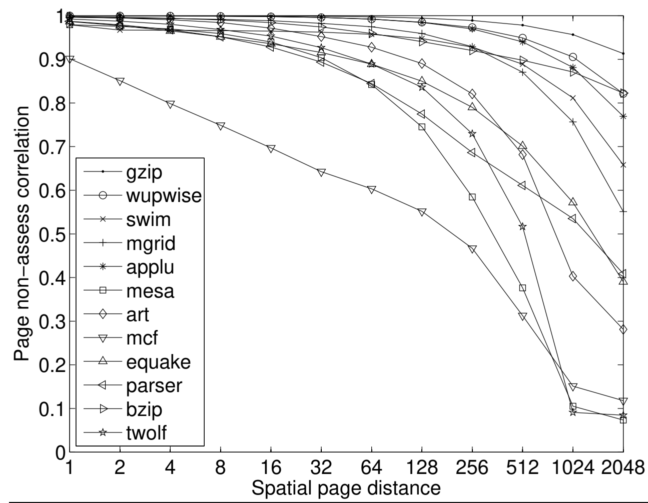

Figure 3. Illustration of page non-access correlation as a function of the spatial page distance. Results are for 12 SPECCPU2000 benchmarks with 2-millisecond sampled access time windows. For each distance value *D*, the non-access correlation is defined as the probability that the next *D* pages are not accessed in a time window if the current page is not accessed. We take snapshots of each benchmark's page table every 5 seconds and present average non-access correlation results here.

uously, the fractional decay also serves the purpose of gradually screening out stale statistics, as in the widely used exponentially-weighted moving average (EWMA) filters.

We decouple the frequency at which the hotness sampling is performed from the time window during which the access bits are sampled (by clearing the access bits at the beginning and reading them at the end of the access time window). We call the former *sampling frequency* and the latter *sampled access time window*. In practice, one may want to enforce an infrequent page table scan for low overhead while at the same time collecting access information over a much smaller time window to avoid hotness information loss. The latter allows distinguishing the relative hotness across different pages accessed in the recent past. Consider a concrete example in which the sampling frequency is once per 100 milliseconds and the sampled access time window is 2 milliseconds. In the first sampling, we clear all page access bits at time 0-millisecond and then check the bits at time 2-millisecond. In the next sampling, the clearing and checking occur at time 100-millisecond and 102-millisecond respectively.

A page table scan is expensive since there is no a priori knowledge of whether each page has been accessed, let alone allocated. There may be invalid page table entries that are not yet mapped and mapped virtual pages that are not yet physically substantiated (some heap management systems may only commit a physical page when it is first accessed). As shown in Table 1, however, such excess page table entries are usually few in practice (particularly for applications with larger memory footprints). We believe the excess checking of non-substantiated page table entries does not constitute a serious overhead.

3.2 Acceleration for Non-Accessed Pages

A conventional page table scan checks every entry regardless of whether the corresponding page was accessed in the last time window. Given that a page list traversal approach [Zhou 2004] only requires continuous checking of frequently accessed pages, the checking of non-accessed page table entries may significantly offset the sequential scan's performance advantage on per-entry checking cost.

We propose an accelerated page table scan that skips the checking of many non-accessed pages. Our acceleration is based on the widely observed data access spatial locality—*i.e.*, if a page was not accessed in a short time window, then pages spatially close to it were probably not accessed either. Intuitively, the non-access correlation of two nearby pages degrades when the spatial distance between them increases. To quantitatively understand this trend, we calculate such non-access correlation as a function of the spatial page distance. Figure 3 illustrates that in most cases (except mcf), the correlation is quite high (around 0.9) for a spatial distance as far as 64 pages. Beyond that, the correlation starts dropping, sometimes precipitously.

Driven by such page non-access correlation, we propose to quickly bypass cold regions of the page table through an approach we call *locality jumping*. Specifically, when encountering a non-accessed page table entry during the sequential scan, we jump page table entries while assuming that the intermediate pages were not accessed (thus requiring no increment of their hotness counters). To minimize false

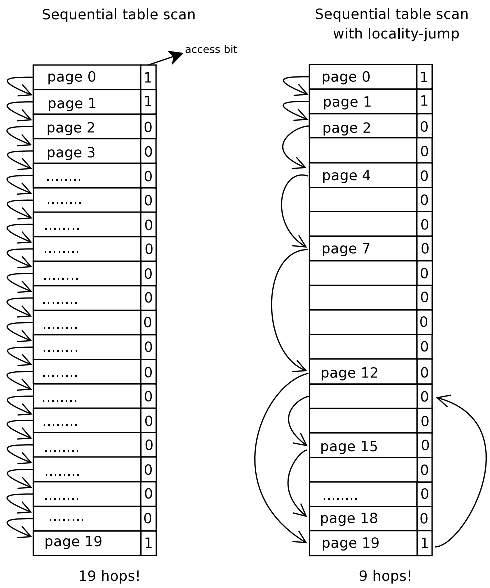

Figure 4. Illustration of sequential page table scan with locality jumping.

jumps, we gradually increase the jump distance in an exponential fashion until we reach a maximum distance (empirically determined to be 64 in our case) or touch an accessed page table entry. In the former case, we will continue jumping at the maximum distance without further increasing it. In the latter case, we jump back to the last seen non-accessed entry and restart the sequential scan. Figure 4 provides a simple illustration of our approach.

Locality jumping that follows a deterministic pattern (*e.g.*, doubling the distance after each jump) runs the risk of synchronizing with a worst-case application access pattern to incur abnormally high false jump rates. To avoid such unwanted correlation with application access patterns, we randomly adjust the jump distance by a small amount at each step. Note, for instance, that the fourth jump in Figure 4 has a distance of 6 (as opposed to 8 in a perfectly exponential pattern).

It is important to note that by breaking the sequential scan pattern, we may sacrifice the per-entry checking cost (particularly by degrading the effectiveness of hardware prefetching). Quantitatively, we observe that the per-entry overhead increases from 36 cycles to 56 cycles on average. Such an increase of per-entry cost is substantially outweighed by the significant reduction of page entry checking.

Finally, it is worth pointing out that spatial locality also applies to accessed pages. However, jumping over accessed page table entries is not useful in our case for at least two reasons. First, in the short time window for fine-grained hotness checking (*e.g.*, 2 milliseconds), the number of non-accessed pages far exceeds that of accessed pages. Second, a jump over an accessed page table entry would leave no chance to increment its hotness counter.

3.3 Additional Uses of Page Hotness Identification

Beyond supporting hot-page coloring in this paper, the page hotness identification has a range of additional utilizations in operating systems. We provide some examples below. Our discussion is at a high level and details of their realizations are beyond the scope of this paper.

The page hotness information we acquire is an approximation of page access frequency. Therefore our approach can support the implementation of LFU (Least-Frequently-Used) memory page replacement. As far as we know, existing LFU systems [Lee 2001, Sokolinsky 2004] are in the areas of storage buffers, database caches, and web caches where each data access is a heavy-duty operation and precise data access tracking does not bring significant additional cost. In comparison, it is challenging to efficiently track memory page access frequency for LFU replacement and our page hotness identification helps tackle this problem.

In service hosting platforms, multiple services (often running inside virtual machines) may share a single physical machine. It is desirable to allocate the shared memory resource among the services according to their needs. The page hotness identification may help such adaptive allocation by estimating the service memory needs at a given hotness threshold. This is a valuable addition to existing methods. For instance, it provides more fine-grained, accurate information than sampling-based working set estimation [Waldspurger 2002]. Additionally, it incurs much less runtime overhead than tracking exact page accesses through minor page faults [Lu 2007].

4. Hot Page Coloring

In this section, we utilize hotness-based partial page coloring to relieve the coloring-induced memory allocation constraints and to alleviate the online recoloring overhead in an adaptive and dynamic environment.

4.1 Relief of Memory Allocation Constraints

Page coloring introduces new constraints on the memory space allocation. When a system has plenty of free memory but is short of pages in certain colors, an otherwise avoidable memory pressure may arise. As a concrete example, two applications on a dual-core platform would like to equally partition the cache by page coloring (to follow the simple fairness goal of equal resource usage). Consequently each can only use up to half of the total memory space. However, one of the applications is an aggressive memory user and would benefit from more than its memory share. At the same time, the other application needs much less memory than its entitled half. The system faces two imperfect choices—to enforce the equal cache use (and thus force expensive disk swapping for the aggressive memory user); or to allow an efficient memory sharing (and consequently let the aggressive memory user pollute the other's cache partition).

In the latter case of memory sharing, a naive approach that colors some random pages from the aggressive application to the victim's cache partition may result in unnecessary contention. Since a page's cache occupancy is directly related to its access frequency, preferentially coloring cold pages to the victim's cache partition would mitigate the effect of cache pollution. Our page hotness identification can be naturally employed to support such an approach. Note that the resulting reduction of cache pollution can benefit adaptive as well as static cache partitioning policies (like the example given above).

4.2 Adaptive Page Recoloring

We consider an adaptive cache partition policy that may modify the partition if the set of co-running applications changes in a dynamic execution environment. We then describe how a page hotness-driven approach can alleviate the overhead associated with cache partition changes.

MRC-driven partition policy For a given set of co-running applications, the goal of our cache partition policy is to improve overall system performance (defined as the geometric mean of co-running applications' performance relative to running independently). The realization of this goal depends on an estimation of the optimization target at each candidate partitioning point. Given the dominance of data access time on modern processors, we estimate that the application execution time is proportional to the combined memory and cache access latency, *i.e.*, roughly $hit + r \cdot miss$, where *hit*/*miss* is the cache hit/miss ratio and r indicates the ratio between cache and memory access latency. For a given application, the cache miss ratio under a specific cache allocation size can be estimated from a cache miss ratio curve (or MRC). Note that while the cache MRC generation requires profiling, the cost per application is independent of the number of processes running in the system. An on-the-fly mechanism to learn the cache MRC is possible. Figure 5 illustrates a simple example of our cache partitioning policy. The cache partition point chosen for best overall system performance may result in either an increase or a decrease of the number of page colors allocated to an application.

Hotness-driven page recoloring The above change in page color allocation may require page recoloring in order to redirect the application's accesses to the prescribed cache partition. Frequent recoloring of all incorrectly mapped pages may incur substantial page copying overhead, in some cases more than negating the benefit of adaptive cache partitioning. Our approach is to recolor a subset of hot (or frequently accessed) pages, which may realize much of the benefit of all-page coloring at reduced cost.

We specify an overhead *budget*, which is the maximum number of recolored pages (or page copying operations) allowed at each recoloring. Given this budget K, we attempt to recolor the hottest (most frequently accessed) K pages. We can locate these pages by using the hotness value of

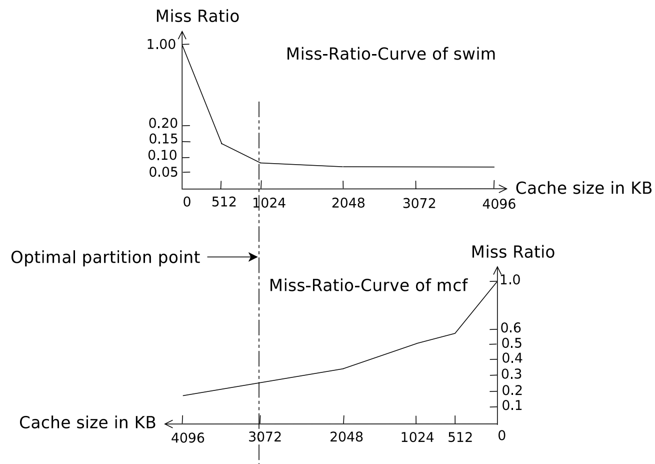

Figure 5. An example of our cache partitioning policy between swim and mcf. The cache miss ratio curve for each application is constructed (offline or during an online learning phase) by measuring the miss ratio at a wide range of possible cache partition sizes. Given the estimation of application performance at each cache partitioning point, we determine that the best partition point for the two applications is if 1 MB cache is allocated to swim and 3 MB cache to mcf.

the K-th hottest page as a threshold. One approach is to maintain a histogram during hotness sampling that counts the number of pages with each possible hotness value. We can scan from the highest hotness value downward until we have accumulated K pages, at which point the hotness threshold is set to the hotness value of the current bin. In our implementation, the histogram is associated with each task (process or thread)'s control structure in the operating system. To better control its space usage, we reduce the number of bins by grouping multiple, similar hotness values together.

Given the change in the range of colors allocated to a process (which is determined by the cache partitioning policy) and the above hotness threshold, we scan the page table to locate the pages with hotness values above the threshold, and recolor them by uniformly assigning them to the new color range. This uniform recoloring helps to achieve low intra-application cache conflicts. Pseudo-code for our recoloring approach is shown in Figure 6.

5. Evaluation

We implemented the proposed page hotness identification approach and used it to drive hot page coloring (including adaptive recoloring in dynamic, multi-programmed environments) in the Linux 2.6.18 kernel. We have also implemented lazy page copying (proposed earlier by Lin *et al.* [Lin 2008]), which delays the copying to the time of first access, to further reduce the coloring overhead. Specifically, each to-be-recolored page is set invalid in the page table en-

procedure *Recolor*

budget (recoloring budget)
old-colors (thread's color set under old partition)
new-colors (thread's color set under new partition)
if *new-colors* is a subset of *old-colors* **then**
$subtract\text{-}colors = old\text{-}colors - new\text{-}colors.$
Find the hot pages in *subtract-colors* within the *budget* limit and reallocate to *new-colors* in a round-robin fashion.
end if
if *old-colors* is a subset of *new-colors* **then**
$add\text{-}colors = new\text{-}colors - old\text{-}colors.$
Find the hot pages in *old-colors* within the $\frac{|new\text{-}colors|}{|add\text{-}colors|} * budget$ limit, and then move at most *budget* (*i.e.* $\frac{|add\text{-}colors|}{|new\text{-}colors|}$ proportion) of them to *add-colors*.
end if

Figure 6. Procedure for hotness-based page recoloring. A key goal is that hot pages are distributed to all assigned colors in a balanced way.

try, and the actual page copying is performed within the page fault handler triggered by the next access to the page.

We performed experiments on a dual-core Intel Xeon 5160 3.0 GHz "Woodcrest" platform. The two cores share a single 4 MB L2 cache (16-way set-associative, 64-byte cache line, 14 cycles latency, writeback). Our evaluation benchmarks are a set of 12 programs from SPECCPU2000.

5.1 Overhead and Accuracy of Page Hotness Identification

Overhead We compare the page hotness identification overheads of three methods—page linked list traversal [Zhou 2004] and our proposed sequential table scan with and without locality-jumping. In our approach, the page table is traversed twice per scan: once to clear the access bits at the beginning of the sampled access time window and once to check them at the end of the window. We set the access time window to 2 milliseconds in our experiments.

The list traversal approach [Zhou 2004] maintains a linked list of frequently accessed pages while the remaining pages are invalidated and monitored through page faults. The size of the frequently accessed page linked list is an important parameter that requires careful attention. If the size is too large, list traversal overhead dominates; if the size is too small, page fault overhead can be prohibitively high. Raghuraman [Raghuraman 2003, Zhou 2004] suggests that a good list size is 30 K pages. Our evaluation revealed that even a value of 30 K was insufficient to keep the page fault rate low in some instances. We therefore measured performance using both the 30 K list size and no limit for the linked list size (meaning all accessed pages are included into the list), and present the better of the two as a comparison point.

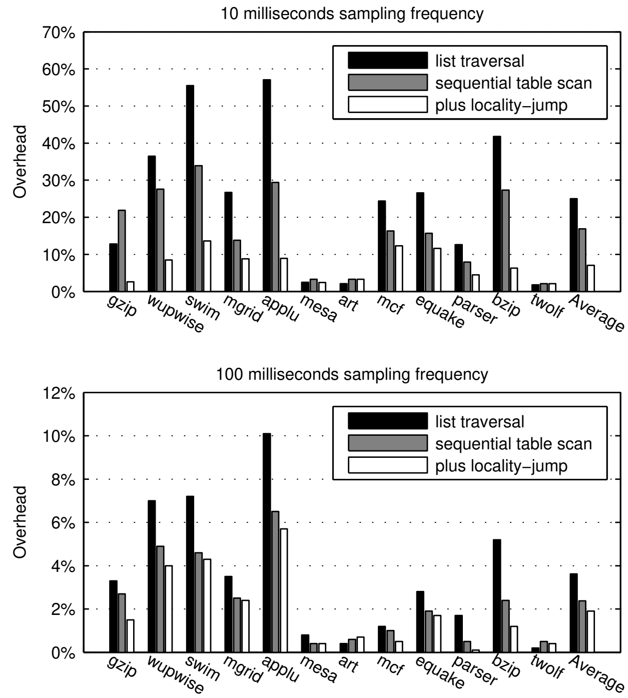

Figure 7. Overhead comparisons under different page hotness identification methods.

The overhead results at two different sampling frequencies (once per 10 milliseconds and once per 100 milliseconds) are shown in Figure 7. When the memory footprint is small, the linked list of pages can be cached and the overhead is close to that of a sequential table scan. As the memory footprint becomes larger, the advantage of spatial locality with a sequential table scan becomes more apparent. On average, sequential table scan with locality jumping involves modest (7.1%, 1.9%) overhead at 10 and 100 millisecond sampling frequencies. It improves over list traversal by 71.7% and 47.2%, and over sequential table scan without locality jumping by 58.1% and 19.6%, at 10 and 100 milliseconds sampling frequencies. To understand the direct effect of locality jumping, Figure 8 shows the percentage of page table entries skipped during the scan. On average we save checking on 63.3% of all page table entries.

Accuracy We measure the accuracy of our page hotness identification methods. We are also interested in knowing whether the locality jumping technique (which saves overhead) would lead to less accurate identification. The ideal measurement goal is to tell how close our identified page hotness is to the "true page hotness". Acquiring the true page hotness, however, is challenging. We approximate it by scanning the page table entries at high frequency without any locality jumping. Specifically, we employ a high sampling frequency of once per 2 milliseconds in this approximation and we call its identified page hotness the *baseline*.

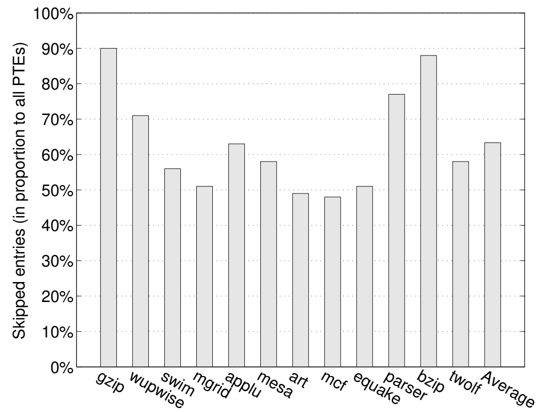

Figure 8. Proportion of skipped page table entries (PTEs) due to our locality-jumping approach in page hotness identification.

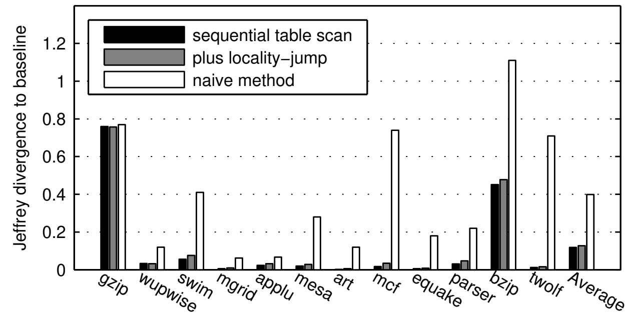

Figure 9. Jeffrey divergence on identified page hotness between various approaches and the baseline (an approximation of "true page hotness").

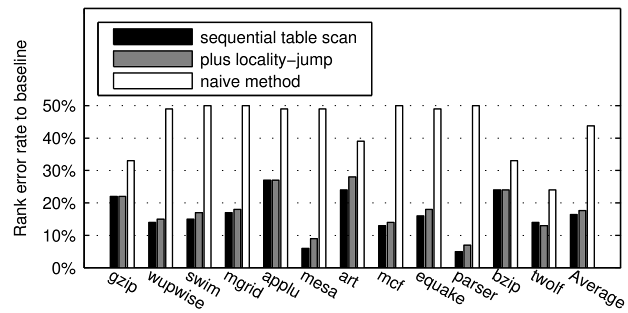

Figure 10. Rank error rate on identified page hotness between various approaches and the baseline (an approximation of "true page hotness").

For a given hotness identification approach, we measure its accuracy by calculating the difference between its identified page hotness and the baseline. To mitigate any potential weakness of using a single difference metric, we use two difference metrics in our evaluation. The first is the Jeffrey-divergence, which is a numerically robust variant of Kullback-Liebler divergence. More precisely, the Jeffrey-divergence of two probability distributions p and q is defined as:

$$JD(p,q) = \sum_i \left(p(i) \log \frac{2p(i)}{p(i)+q(i)} + q(i) \log \frac{2q(i)}{p(i)+q(i)} \right).$$

$JD(p,q)$ measures the divergence in terms of relative entropy from p and q to $\frac{p+q}{2}$ and it is in the range of [0, 2]. In order to calculate Jeffrey-divergence, page hotness is normalized such that hotness sums up to 1. Here $p(i)$ and $q(i)$ represent page i's measured hotness from the two methods being compared.

The second difference metric we utilize is the rank error rate. Specifically, we rank pages in hotness order (pages of the same hotness are ranked equally at the highest rank available) and sum up the absolute rank difference between the two methods being compared. The rank error rate is the average absolute rank difference per page divided by the total number of pages.

We measure the page hotness identification of our sequential table scan approach and its enhancement with locality-jumping. These approaches employ a sampling frequency of once per 100 milliseconds. As a point of comparison, we also measure the accuracy of a naive page hotness identification approach that considers all pages to be equally hot. Note that under our rank order definition, all pages under the naive method have the highest rank.

Figure 9 shows the accuracy of various page hotness identification approaches (in terms of Jeffrey divergence to the baseline). Figure 10 shows similar results on the rank error rate. Results demonstrate that our proposed methods achieve substantially better accuracy than the naive method. Further, the locality-jumping technique does not appear to degrade the accuracy of our results.

Figure 11 visually presents the deviation of our identified page hotness from the baseline for all 12 applications. A visual analysis suggests that our hotness identification results are fairly accurate overall. The high Jeffrey divergence observed for gzip in Figure 9 is due to a large relative difference (but a small absolute difference as Figure 11 shows) for the large number of pages with extremely low hotness values.

5.2 Effectiveness of Hot-Page Coloring— Relieving Memory Allocation Constraints

As explained in Section 4.1, page coloring introduces new memory allocation constraints that may cause otherwise avoidable memory pressure or cache pollution. We examine the effectiveness of hot-page coloring in reducing the negative effect of such coloring-induced memory allocation constraints. In this experiment, two applications on a dual core platform would like to equally partition the cache using page coloring (to follow the simple fairness goal of equal resource usage). Consequently each can only use up to half of the total memory space. However, one of the applications

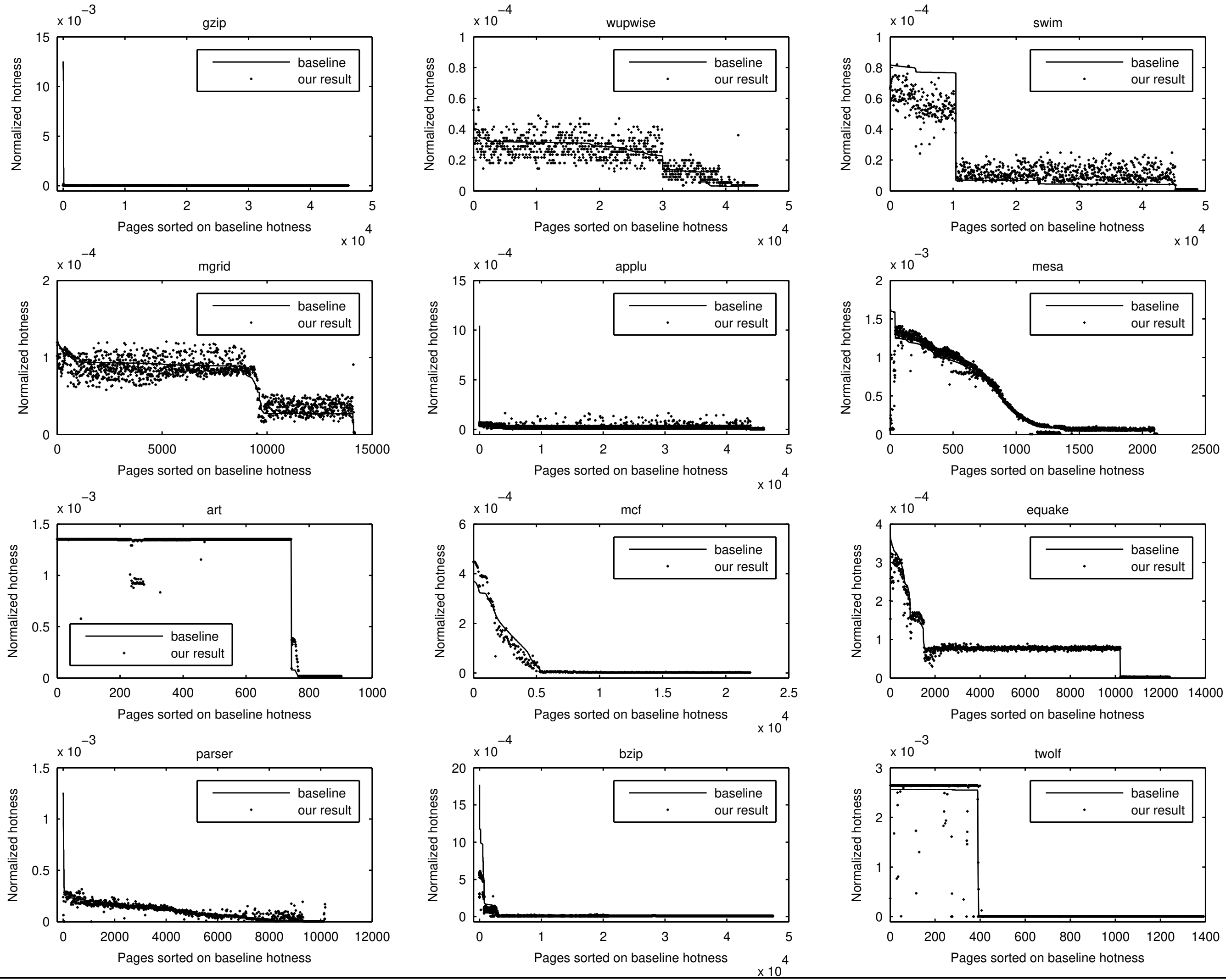

Figure 11. All-page comparison of page hotness identification results for our sequential table scan with locality-jumping approach (at once-per-100-millisecond sampling frequency) and the baseline page hotness. Pages are sorted by their baseline hotness. The hotness is normalized so that the hotness of all pages in an application sum up to 1.

uses more memory than its entitled half. Without resorting to expensive disk swapping, this application would have to use some memory pages beyond its allocated colors and therefore pollute the other application's cache partition.

Specifically, we consider a system with 256 MB memory[1]. We pick swim as the polluting application with a 190 MB memory footprint. When only half of the total 256 MB memory is available, swim has to steal about 62 MB from the victim application's page colors. Figure 11 shows that in swim, 20% of the pages are exceptionally hotter than the other 80% of the pages, which provides a good opportunity for our hot-page coloring. We choose six victim applications with small memory footprints that, without the coloring-induced allocation constraint, would fit well into the system memory together with swim. They are mesa, mgrid, equake, parser, art, and twolf.

We evaluate three policies: *random*, in which the polluting application randomly picks the pages to move to the victim application's entitled colors; *hot-page coloring*, which uses the page hotness information to pollute the victim application's colors with the coldest (least frequently used) pages; and *no pollution*, a hypothetical comparison base that is only possible with expensive disk swapping, which we idealize by providing enough memory so that swapping is avoided. Figure 12 shows the victim applications' slowdowns under different cache pollution policies. Compared to random pollution, the hotness-aware policy reduces the slowdown for applications with high cache space sensitivity. Specifically,

[1] The relatively small system memory size is chosen to match the small memory usage in our SPECCPU benchmarks. We expect that the results of our experiment should also reflect the behaviors of larger-memory-footprint applications in larger systems.

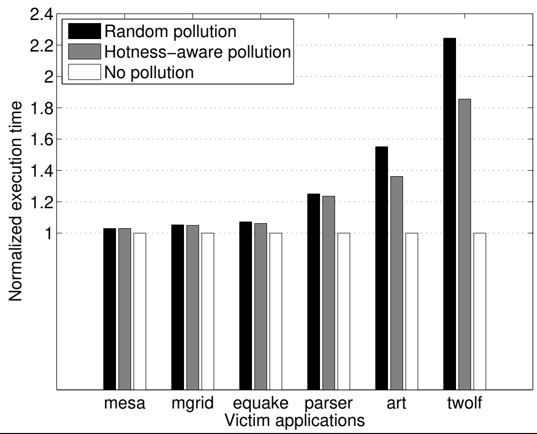

Figure 12. Normalized execution time of different victim applications under different cache pollution schemes. The polluting application is swim.

for the two most sensitive victims (art and twolf), the random cache pollution yields 55% and 124% execution time increases (from no pollution) while the hotness-aware pollution causes 36% and 86% execution time increases.

5.3 Effectiveness of Hot-Page Coloring— Alleviating Page Recoloring Cost

In a multi-programmed system where context switches occur fairly often, an adaptive cache partitioning policy may need to recolor pages to reflect the dynamically changing co-running applications. Each of our multi-programmed experiments runs four applications on a dual-core processor. Specifically, we employ two such four-application groups with significant intra-group cache contention. These two groups are {swim, mgrid, bzip, mcf} and {art, mcf, equake, twolf}, and their contention relations are shown in Figure 13. Within each group, we assign two applications to each sibling core on a dual-core processor and run all possible combinations. In total, there are 6 tests:

test1 = {swim, mgrid} vs. {mcf, bzip};
test2 = {swim, mcf} vs. {mgrid, bzip};
test3 = {swim, bzip} vs. {mgrid, mcf};
test4 = {art, mcf} vs. {equake, twolf};
test5 = {art, equake} vs. {mcf, twolf};
test6 = {art, twolf} vs. {mcf, equake}.

We compare system performance under several static cache management policies.

- In *default sharing*, applications freely compete for the shared cache space.
- In *equal partition*, the two cores statically partition the cache evenly and applications can only use their cores' entitled cache space. Under such equal partition, there is no need for recoloring when co-running applications change in a dynamic execution environment.

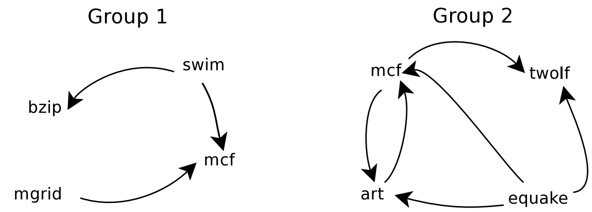

Figure 13. Contention relations of two groups of SPEC-CPU2000 benchmarks. If A points to B, that means B has more than 50% performance degradation when running together with A on a shared cache, compared to running alone when B can monopolize the whole cache.

We then consider several adaptive page coloring schemes. As described in Section 4.2, adaptive schemes utilize the miss-ratio-curve (MRC) to determine a desired cache partition between co-running applications. Whenever an application's co-runner changes, the application re-calculates an optimal partition point and recolors pages.

- In *all-page coloring*, we recolor all pages necessary to achieve the new desired cache partition after a change of co-running applications. This is the obvious alternative without the guidance of our hot-page identification.
- The *ideal page coloring* is a hypothetical approach that models the all-page coloring but without incurring any recoloring overhead. Specifically, consider the test of {A,B} vs. {C,D}. We run each possible pairing (A-C, A-D, B-C, and B-D) on two dedicated cores (without context switches) and assume that the resulting average performance for each application would match its performance in the multi-programmed setting.
- In *hot page coloring*, we utilize our page hotness identification to only recolor hot pages within a target recoloring budget that limits its overhead. The recoloring budget is defined as an estimated relative slowdown of the application (specifically as the cost of each recoloring divided by the time interval between adjacent recoloring events, which is estimated as the CPU scheduling quantum length). Our experiments consider two recoloring-caused application slowdown budgets—5% (conservative) and 20% (aggressive). In our implementation, a given recoloring budget is translated into a cap on the number of recolored pages according to the page copying cost. Copying one page takes roughly 3 microseconds on our experimental platform.

The recoloring overhead in the adaptive schemes depends on the change frequency of co-running applications, and therefore it is directly affected by the CPU scheduling quantum. We evaluate this effect by experimenting with a range of scheduling quantum lengths (100–800 milliseconds). Figure 14 presents the system performance of the 6 tests under different cache management policies. Our performance metric is defined as the geometric mean of individual applica-

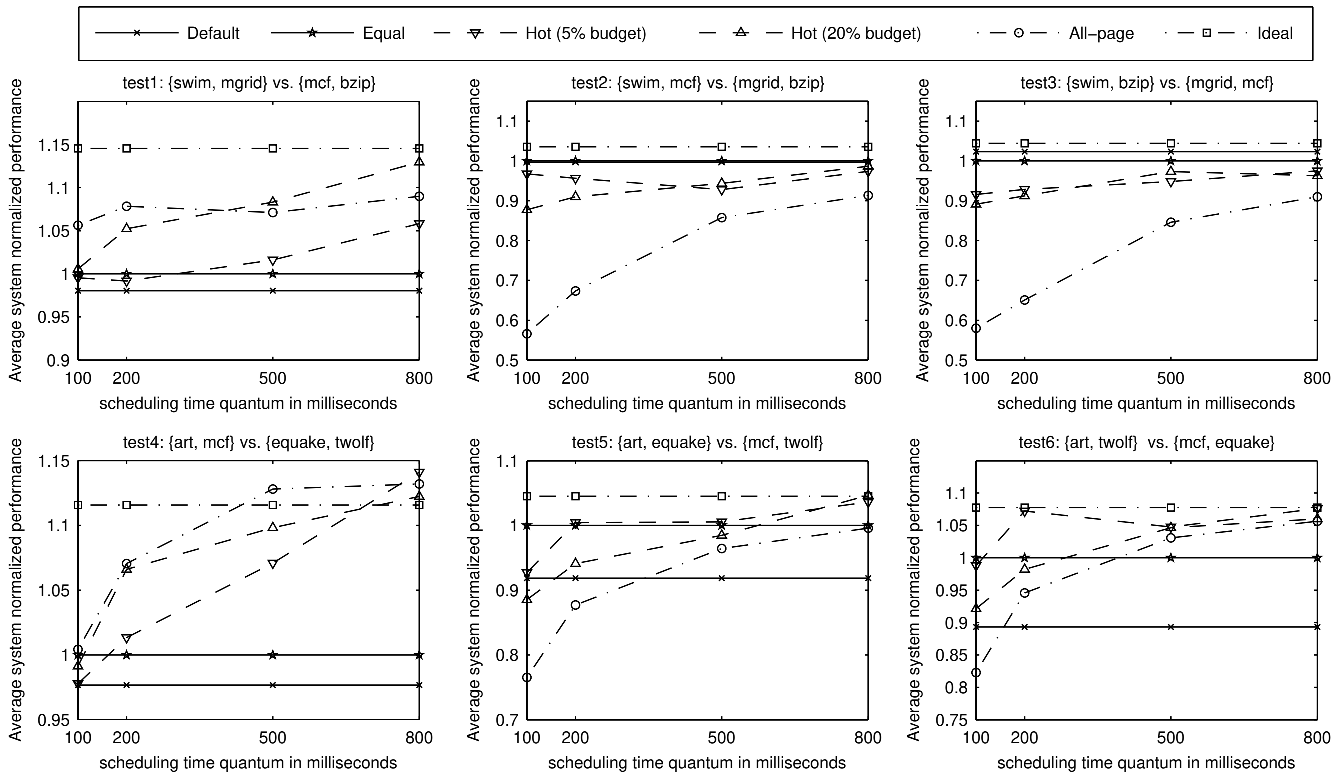

Figure 14. Performance comparisons under different cache management policies for 6 multi-programmed tests (four applications each) on a dual-core platform.

tions' relative performance compared to when running alone and utilizing the whole cache. All performance numbers are normalized to that of the equal partition policy.

Our first observation is that the simple policy of equal cache partition achieves quite good performance generally speaking. It does so by reducing inter-core cache conflicts without incurring any adjustment costs in multi-programmed environments. On average, it has a 3.5% performance improvement over default sharing and its performance is about 7.7% away from that of ideal page coloring.

All-page coloring achieves quite poor performance overall. Compared to equal partitioning, it degrades performance by 20.1%, 11.7%, and 1.7% at 100, 200, and 500 milliseconds scheduling time quanta respectively. It only manages to achieve a slight improvement of 1.6% at the long 800 milliseconds scheduling quantum. The poor performance of all-page coloring is due to the large recoloring overhead at context switches. To provide an intuition of such cost, we did a simple back-of-the-envelope calculation as follows. The average working set of the 7 benchmarks used in these experiments is 82.1 MB. If only 10% of the working set is recolored at every time quantum (default 100 milliseconds), the page copying cost alone would incur 6.3% application slowdown, negating most of the benefit gained by the ideal page coloring.

The hot page coloring greatly improves performance over all-page coloring. It can also improve the performance over equal partitioning at 500 and 800 milliseconds scheduling time quanta. Specifically, the conservative hot page coloring (at 5% budget) achieves 0.3% and 4.3% performance improvement while aggressive hot page coloring (at 20% budget) achieves 2.9% and 4.0% performance improvement. However, it is somewhat disappointing that the page copying overhead still outweighs the adaptive page coloring's benefit when context switches occur at a finer granularity (every 100 or 200 milliseconds). Specifically, the conservative hot page coloring yields 3.8% and 0.5% performance degradation compared to equal partition while the aggressive hot page coloring yields 7.1% and 2.3% performance degradation.

We notice that in test4 of Figure 14, the ideal scheme does not always provide the best performance. One possible explanation for this unintuitive result is that our page recoloring algorithm (described in Section 4.2) also considers intra-thread cache conflicts by distributing pages to all assigned colors in a balanced way. Such intra-thread cache conflicts are not considered in our ideal scheme.

5.4 An Evaluation of Fairness

We also study how these cache management policies affect the system fairness. We use an *unfairness* metric, defined as

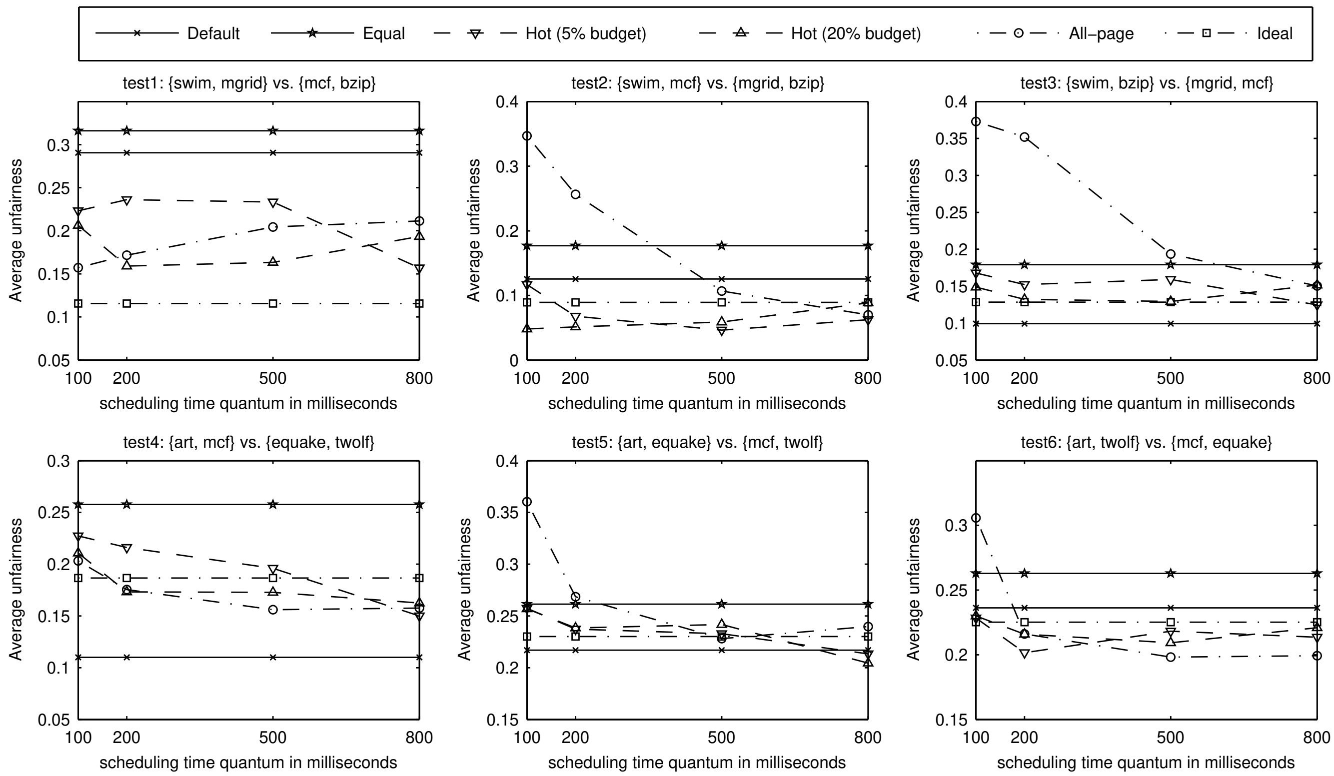

Figure 15. Unfairness comparisons (the lower the better) under different cache management policies for 6 multi-programmed tests (four applications each) on a dual-core platform.

the coefficient of variation (standard deviation divided by the mean) of all applications' normalized performance. Here, each application's performance is normalized to its execution time when it monopolizes the whole cache resource. If normalized performance is fluctuating across different applications, *unfairness* tends to be large; if every application has a uniform speedup/slowdown, then *unfairness* tends to be small.

We evaluate the execution unfairness of the 6 tests we examined in Section 5.3. Figure 15 shows the results under different cache management policies. Results show that equal partition performs poorly, simply because it allocates cache space without knowledge of how individual applications' performance will be affected. The *unfairness* of default sharing is not as high as one may expect, because this set of benchmarks exhibits contention in both directions for most pairs, resulting in relatively uniform poor performance for individual ones. Ideal page coloring is generally better (lower unfairness metric value) than default sharing and equal partition. Hot and all-page coloring perform similarly to what they did in the performance results: they gradually approach the fairness of ideal page coloring as the page copying cost is amortized over longer scheduling time quanta. It also suggests that expensive page coloring may be worthwhile in cases where the quality-of-service (like those in service level agreements) is the first priority and customized resource allocation is needed. Note that our cache partition policy does not directly take fairness into consideration. It should be possible to derive other metrics to optimize fairness and to use other metrics for fairness.

6. Related Work

Hardware-based cache partitioning Hardware-based cache partitioning schemes mainly focus on modifying cache replacement policies and can be categorized by partition granularity: way-partitioning [Chiou 2000, Qureshi 2006] and block-partitioning [Suh 2001, Zhao 2007, Rafique 2006]. Way-partitioning (also called column partitioning in [Chiou 2000]) restricts cache block replacement for a process to within a certain way, resulting in a maximum of n slices or partitions with an n-way associative cache. Block-partitioning allows partitioning blocks within a set, but is more expensive to implement. It usually requires hardware support to track cache line ownership. When a cache miss occurs, a cache line belonging to an over-allocated owner is preferentially evicted.

Page coloring based cache partitioning Cho and Jin [Cho 2006] first proposed the use of page coloring to manage data placement in a tiled CMP architecture. Their goal was to reduce a single application's cache miss and access latency. Tam *et al.* [Tam 2007] first implemented page coloring in the Linux kernel for cache partitioning purposes, but restricted their implementation and analysis to static partitioning of the cache among two competing threads. Lin *et al.* [Lin 2008] further extended the above to dynamic page coloring. Their evaluation acknowledged that page recoloring is clearly an expensive operation and should be attempted rarely in order to make page coloring beneficial. Soares *et al.* [Soares 2008] remap high cache miss pages to dedicated cache sets to avoid polluting pages with low cache misses. These previous works either only consider a single application or two co-running competing threads, where frequent page recoloring is not incurred. Also, they mainly target one beneficial aspect of page coloring, rather than developing a practical and viable solution within the operating system. Our approach alleviates two important obstacles: memory pressure and frequent recoloring when using the page coloring technique.

Cache management policies Kim *et al.* [Kim 2004] proposed 5 different metrics for L2 cache fairness. They use cache miss or cache miss ratio as performance (or normalized performance) and define fairness as the difference between the maximum and minimum performance of all applications. Our fairness metric in Section 5.4 takes all applications' performance into consideration and tends to be more numerically robust than only considering max and min. Iyer *et al.* [Iyer 2007] proposed 3 types of quality-of-service metrics (resource oriented, individual, or overall performance oriented) and statically/dynamically allocated cache/memory resources to meet these QoS goals. Hsu *et al.* [Hsu 2006] studied various performance metrics under communist, utilitarian, and capitalist cache polices and made the conclusion that thread-aware cache resource allocation is required to achieve good performance and fairness. All these studies focus on resource management in the space domain. Another piece of work by Fedorova *et al.* [Fedorova 2007] proposed to compensate/penalize threads that went under/over their fair cache share by modifying their CPU time quanta.

7. Conclusion

We present an efficient approach to tracking application page hotness on-the-fly. Driven by the page hotness information, we propose new approaches to mitigate practical obstacles faced by current page coloring-based cache partitioning on multi-core platforms. The results of our work make page coloring-based cache management a more viable option for general-purpose systems, although with cost amortization time-frames that are still higher than typical operating system time slices. We envision that our approach will show even greater potential on many-core platforms, where resource contention/constraints are likely more severe. In parallel, computer architecture researchers are also investigating new address translation hardware to make page coloring extremely lightweight. We expect features provided by new hardware in the near future to allow more efficient operating system control. In the meanwhile, we hope our proposed approach could aid performance isolation in existing multi-core processors on today's market.

Experiments in this paper target multiple sequential applications running in multi-programmed environments. A further place to employ our approach is virtual machine-driven shared service hosting platforms. In particular, cloud computing platform providers (*e.g.*, Amazon [Amazon] and GoGRID [GoGrid]) charge customers by the amount of time certain hardware resources are used (*e.g.*, memory, processor, and network bandwidth). Without carefully managing various resource conflicts (especially those at the chip level), the service platform cannot guarantee full performance delivery as desired by customers.

Acknowledgments

We thank the anonymous EuroSys referees and our shepherd Timothy Roscoe for their helpful comments on a preliminary version of this paper. Additional thanks to Tongxin Bai who pointed us toward the locality-based page table scan in the page hotness identification.

References

[Amazon] Amazon. Amazon elastic compute cloud. http://aws.amazon.com/ec2/.

[AMD64-manual] AMD64-manual. AMD-64 architecture programmer's manual, 2008.

[Bugnion 1996] E. Bugnion, J. M. Anderson, and M. S. Lam. Compiler-directed page coloring for multiprocessors. In *7th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 244–255, Cambridge, MA, October 1996.

[Chiou 2000] D. Chiou, P. Jain, L. Rudolph, and S. Devadas. Application-specific memory management for embedded systems using software-controlled caches. In *37th Conf. on Design Automation*, pages 416–419, Los Angeles, CA, 2000.

[Cho 2006] S. Cho and L. Jin. Managing distributed, shared L2 caches through OS-level page allocation. In *39th Int'l Symp. on Microarchitecture (Micro)*, pages 455–468, Orlando, FL, December 2006.

[Fedorova 2007] A. Fedorova, M. Seltzer, and M.D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *16th Int'l Conf. on Parallel Architecture and Compilation Techniques (PACT)*, pages 25–36, Brasov, Romania, September 2007.

[GoGrid] GoGrid. http://www.gogrid.com.

[Hsu 2006] 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 *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 13–22, 2006.

[IA32-manual] IA32-manual. IA-32 Intel architecture software developer's manual, 2008. http://www.intel.com/products/processor/manuals/.

[Intel] Intel. TLBs, paging-structure caches, and their invalidation, 2008. http://www.intel.com/design/processor/applnots/317080.pdf.

[Iyer 2007] R. Iyer, L. Zhao, F. Guo, R. Illikkal, Don Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS policies and architecture for cache/memroy in CMP platforms. In *ACM SIGMETRICS*, pages 25–36, San Diego, June 2007.

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

[Kim 2004] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 111–122, 2004.

[Lee 2001] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. on Computers*, 50(12):1352–1361, December 2001.

[Lin 2008] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 367–378, Salt Lake, UT, February 2008.

[Lu 2007] P. Lu and K. Shen. Virtual machine memory access tracing with hypervisor exclusive cache. In *USENIX Annual Technical Conf. (USENIX)*, pages 29–43, Santa Clara, CA, June 2007.

[Patterson 2004] D. A. Patterson. Latency lags bandwith. *Communications of the ACM*, 47(10):71–75, October 2004.

[Qureshi 2006] M. Qureshi and Y. Patt. Utility-based cache partitioning: A low-overhead, hight-performance, runtime mechanism to partition shared caches. In *39th Int'l Symp. on Microarchitecture (Micro)*, pages 423–432, Orlando, FL, December 2006.

[Rafique 2006] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven CMP cache management. In *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 2–12, 2006.

[Raghuraman 2003] A. Raghuraman. Miss-ratio curve directed memory management for high performance and low energy. Master's thesis, Dept. of Computer Science, UIUC, 2003.

[Romer 1994] T. H. Romer, D. Lee, B. N. Bershad, and J. B. Chen. Dynamic page mapping policies for cache conflict resolution on standard hardware. In *First USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 255–266, Monterey, CA, November 1994.

[Sherwood 1999] T. Sherwood, B. Calder, and J. Emer. Reducing cache misses using hardware and software page replacement. In *13th Int'l Conf. on Supercomputing (ICS)*, pages 155–164, Rhodes, Greece, June 1999.

[Soares 2008] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In *41th Int'l Symp. on Microarchitecture (Micro)*, pages 258–269, Lake Como, ITALY, November 2008.

[Sokolinsky 2004] L. B. Sokolinsky. LFU-K: An effective buffer management replacement algorithm. In *9th International Conference on Database Systems for Advanced Applications*, pages 670–681, 2004.

[Stone 1992] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. *IEEE Trans. on Computers*, 41 (9):1054–1068, September 1992.

[Suh 2001] G. E. Suh, L. Rudolph, and Srini Devadas. Dynamic cache partitioning for simultaneous multithreading systems. In *Int'l Conf. on Parallel and Distributed Computing and Systems*, pages 116–127, Anaheim, CA, August 2001.

[Tam 2007] D. Tam, R. Azimi, L. Soares, and M. Stumm. Managing shared L2 caches on multicore systems in software. In *Workshop on the Interaction between Operating Systems and Computer Architecture*, San Diego, CA, June 2007.

[Taylor 1990] G. Taylor, P. Davies, and M. Farmwald. The TLB slice – a low-cost high-speed address translation mechanism. In *17th Int'l Symp. on Computer Architecture (ISCA)*, pages 355–363, Seattle, WA, June 1990.

[Waldspurger 2002] C. A. Waldspurger. Memory resource management in vmware ESX server. In *5th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 181–194, Boston, MA, December 2002.

[Zhang 2007] X. Zhang, S. Dwarkadas, G. Folkmanis, and K. Shen. Processor hardware counter statistics as a first-class system resource. In *11th Workshop on Hot Topics in Operating Systems (HotOS)*, San Diego, CA, May 2007.

[Zhao 2007] L. Zhao, R. Iyer, R. Illikkal, J. Moses, D. Newell, and S. Makineni. CacheScouts: Fine-grain monitoring of shared caches in CMP platforms. In *16th Int'l Conf. on Parallel Architecture and Compilation Techniques (PACT)*, pages 339–352, Brasov, Romania, September 2007.

[Zhou 2004] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *11th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 177–188, Boston, MA, October 2004.