COLORIS: A Dynamic Cache Partitioning System Using Page Coloring

Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li
Computer Science Department, Boston University
Boston, MA, USA
yingy@cs.bu.edu, richwest@cs.bu.edu, czq@cs.bu.edu, liye@cs.bu.edu

ABSTRACT

Shared caches in multicore processors are subject to contention from co-running threads. The resultant interference can lead to highly-variable performance for individual applications. This is particularly problematic for real-time applications, requiring predictable timing guarantees. Previous work has applied page coloring techniques to partition a shared cache, so that conflict misses are minimized amongst co-running workloads. However, prior page coloring techniques have not addressed the problem of partitioning a cache on over-committed processors where there are more executable threads than cores. Similarly, page coloring techniques have not proven efficient at adapting the cache partition sizes for threads with varying memory demands.

This paper presents a memory management framework called COLORIS, which provides support for both static and dynamic cache partitioning using page coloring. COLORIS supports novel policies to reconfigure the assignment of page colors amongst application threads in over-committed systems. For quality-of-service (QoS), COLORIS monitors the cache miss rates of running applications and triggers repartitioning of the cache to prevent miss rates exceeding applications-specific ranges. This paper presents the design and evaluation of COLORIS as applied to Linux. We show the efficiency and effectiveness of COLORIS to color memory pages for a set of SPEC CPU2006 workloads, thereby enhancing performance isolation over existing page coloring techniques.

Categories and Subject Descriptors

D.4.2 [**Storage Management**]: Main memory

Keywords

Cache and memory management; dynamic page coloring; multicore; performance isolation

1. INTRODUCTION

PACT'14, August 24–27, 2014, Edmonton, AB, Canada.

ACM 978-1-4503-2809-8/14/08 ...$15.00.
http://dx.doi.org/10.1145/2628071.2628104.

Multicore platforms are gaining popularity these days, both in general computing and embedded domains. They have the ability to support concurrent workloads with less circuit area, lower power consumption and lower cost. Tightly-coupled on-chip resources allow faster data sharing between processing cores, at the same time, suffering from potentially heavy resource contention. Modern processors often devote the largest fraction of on-chip transistors to caches, mostly the shared last-level cache (LLC), the performance of which is crucial for the overall processing capability of processors, especially for running memory-bound applications. Yet, in most commercial off-the-shelf (COTS) systems, only best effort service is provided for accessing the shared LLC. Multiple processes run simultaneously on those systems, interfering with one another on cache accesses. Data stored in a cache line with strong access locality from one process may be evicted at an arbitrary time due to another process's access to different data mapped to the same cache line. Interleaving cache accesses can potentially ruin program locality, the fundamental assumption hardware caches are built upon. Depending on the workload, the degree of interference on application performance can vary significantly [13]. Thus, QoS cannot be guaranteed in such systems.

For safety-critical real-time systems, cache-related performance variations can lead to task timing violations, and potential life-threatening situations. Therefore, performance isolation is desirable in these systems. Although private caches eliminate cache contention for threads mapped to different caches, they offer a smaller storage capacity than a single shared cache, and additionally require coherency protocols to maintain consistency of shared data. Alternatively, shared caches can be used in conjunction with techniques that minimize cache interference, such as page coloring, cache-aware scheduling or hardware cache partitioning.

One of the first uses of page coloring was in virtual address translation in the MIPS system [31]. It was later adopted by operating system memory subsystems [3, 12] and compilers [4], to reduce application cache self-conflicts. Recent work [14, 29, 35] applied page coloring on multicore systems to partition shared caches, and demonstrated enhanced isolation amongst applications. However, the result of previous work has also revealed some open issues related with page coloring, especially with respect to dynamic recoloring and, hence, re-partitioning.

First of all, knowing when to perform re-partitioning is non-trivial. Dynamic phase changing behaviors [1] of applications lead to fluctuating resource demands, which may cause poor cache utilization under static partitioning. To re-partition the shared cache, we want to clearly capture program phase transitions on-the-fly. Even without phase changes, when an application is just started, it is hard to determine its best cache partition size without a-priori knowledge. Existing page coloring techniques either do not adaptively adjust partitions, at least not efficiently, or they fail to identify application phase transitions.

Secondly, finding the optimal partition size to minimize cache misses for a given workload is difficult. If the goal is fairness or QoS, proper performance metrics have to be defined to guide dynamic re-partitioning. For example, some researchers have attempted to construct cache utility curves, which capture the miss rates for applications at different cache occupancies [19, 26, 30, 33, 37], but this is typically expensive to do in a working system.

Another issue with page coloring is the significant overhead with re-partitioning, also called recoloring. Recoloring is cumbersome. It involves allocating new page frames, copying memory between pages and freeing old page frames as necessary. To make the matter worse, naive page selection during recoloring may cause inefficient use of newly allocated cache space. Altogether, the benefit of dynamic partitioning can be undermined.

Lastly, in over-committed systems, excessive recoloring operations may result from the interleaved execution of multiple processes. A simple example would be a 2-core, 6-page-color system with two running processes (P1, P2) and one ready process (P3). The page colors allocated to these three processes might be $\{1,2,3\}, \{4,5,6\}$ and $\{1,2,3\}$, respectively. If a scheduler suspends P2 and dispatches P3 to co-run with P1, two processes will be contending for the same three page colors. This may result in significant contention on a subset of the entire cache space. At this point, either recoloring has to be performed or performance isolation is compromised.

Our work tries to solve the problems associated with implementing dynamic page coloring in production systems. Specifically, this paper describes the implementation of an efficient page recoloring framework in the Linux kernel, called "COLORIS" (COLOR ISolation). COLORIS attempts to isolate cache accesses of different applications, thereby reducing their conflict misses compared to a scheme that lacks cache partitioning. We propose two recoloring policies and compare their performance, while considering factors such as when to trigger recoloring and which processes should be affected. Using the SPEC CPU2006 benchmark, we show the performance of COLORIS on an over-committed multicore platform, where there are more schedulable threads than cores.

The rest of this paper is organized as follows: Section 2 introduces basic page coloring concepts. Section 3 describes the design of the COLORIS framework, with focus on the proposed recoloring mechanism. An evaluation of COLORIS in provided in Section 4. Related work is then described in Section 5, followed by conclusions and future work.

[1]Phase changes might be due to switching between localities such as function loops and procedures within a running program.

2. BACKGROUND

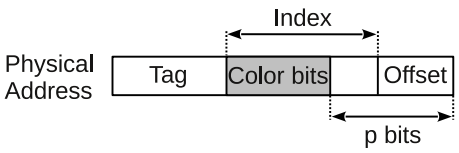

Figure 1: Page Color Bits

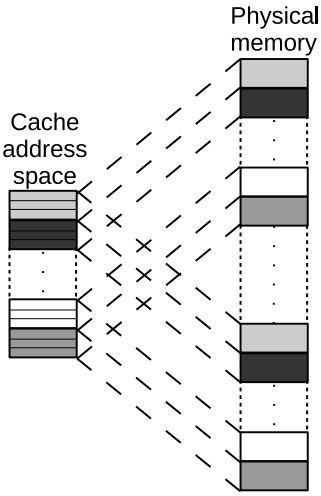

Figure 2: Mapping Between Memory Pages and Cache Space

On most modern architectures, a shared LLC is physically indexed and set associative. When a physical address is used in a cache lookup operation, it is divided into a *Tag*, an *Index* and an *Offset* as shown in Figure 1. The *Offset* bits are used to specify the byte offset location of the desired data within a specific cache line. *Index* bits select a cache set. *Tag* bits are checked against the current cache line to determine a cache hit or miss. Operating systems that manage memory at page granularity use the least significant p bits of a physical address as a byte offset within a page of memory. These p bits typically overlap with the least-significant *Index* bits, leaving the most significant indexing bits to form the *Color* bits (as shown in the shaded area of Figure 1). Two or more pages of physical memory addressed with different color bits are guaranteed *not* to map to the same set of cache lines.

Using color bits, it is possible for an operating system to implement a color-aware memory allocator to control the mapping of physical pages to sets of cache lines. This is illustrated by Figure 2. Pages mapped to the same set of cache lines are said to have the same *page color*. The total number of page colors is derived from the following equation:

$$number\ of\ colors = \frac{cache\ size}{number\ of\ ways \times page\ size}$$

Limitations. Some recent architectures have now introduced *cache slices* [36]. This new cache arrangement imposes a challenge for page coloring techniques since the mapping between memory and cache slices is made opaque. If the hash function for mapping physical addresses to cache sets inside a slice remains the same as above, then it may be possible to treat each slice as though it were a separate cache.

That way, the problem is equivalent to page coloring on N separate caches each of size M, where M is the size of a slice and $N \cdot M$ is the full capacity of the original cache. We will show in later experiments that COLORIS is able to effectively partition caches on architectures featuring slices.

3. COLORIS ARCHITECTURE

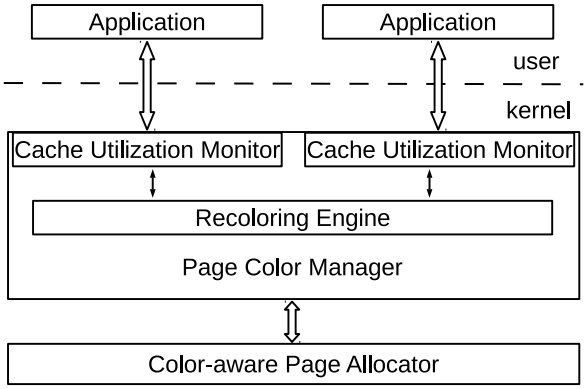

Figure 3: COLORIS Architecture Overview

COLORIS is comprised of two major components: a Page Color Manager and a Color-aware Page Allocator. The Color-aware Page Allocator is capable of allocating page frames of specific colors. The Page Color Manager is responsible for assigning initial page colors to processes, monitoring process cache usage metrics, and performing color assignment adjustment according to system specific objectives (e.g. fairness, QoS, or performance). An architectural overview of COLORIS is shown in Figure 3.

Though the design of COLORIS is system-agnostic, we implemented a prototype version inside the Linux kernel to utilize existing applications and benchmarks for evaluation purposes.

3.1 Color-aware Page Allocator

The Linux kernel allocates single page frames from the per-CPU page frame cache. The page frame cache maintains a list of free pages [2] allocated from the Buddy System. Tam et al [29] made the Linux page frame cache color-aware by splitting the single free list into multiple lists, each corresponding to one particular page color. To refill a list with a specific color, the Buddy System has to be called repeatedly until a page of the correct color is returned. This approach can potentially add significant overhead and compromise the benefit of page coloring.

Instead of using the existing Linux memory allocation framework, we decided to replace it with our own efficient color-aware allocator. In COLORIS, a memory pool is used to serve all page requests. Inside the pool, free pages of the same colors are linked together to form multiple lists. Figure 4 shows the structure of our new page allocator.

In order to reduce the engineering complexity, we decided to keep the original Linux Buddy System interface in COLORIS. This is achieved by maintaining per-process color assignments in the Page Color Manager. Upon receiving an

[2]Strictly speaking, two lists are used for performance optimization.

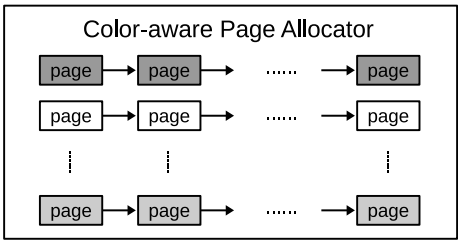

Figure 4: Page Allocator

allocation request, the Color-aware Page Allocator communicates with the Page Color Manager to determine the colors already assigned to the requesting process. The allocator then picks one of these colors in a round-robin manner, and returns a page with that color from the memory pool. If that color does not exist, the allocator attempts to pick another color assigned to the process. If all currently assigned colors are unavailable, then the allocator calls the Buddy System to populate the memory pool with new pages of arbitrary colors. By keeping the memory pool large enough, communication with the Buddy System is minimized. When a page is freed, we first try to place it back into the memory pool and only return it to the Linux page frame cache if the corresponding list is full.

Additionally, instead of managing only data pages [24], COLORIS also supports page coloring of program code pages. This increases cache isolation between processes, except for cases where code is shared. However, we believe shared code should be avoided as much as possible, as a trade off between memory utilization and maximum isolation. For certain libraries, such as *libc*, we acknowledge this is not always practical. For cases where sharing is practically unavoidable, COLORIS randomly selects page colors upon request. An alternative to this is to reserve a dedicated set of page colors for shared memory usage.

3.2 Page Color Manager

The Page Color Manager manages page color resources amongst application processes according to specific policies. In the simplest form, the Page Color Manager statically partitions hardware caches according to various color assignment schemes. The most intuitive approach is to strictly assign different page colors to different processes so that they land on different parts of the shared cache. While this provides the maximum degree of isolation, it also limits the maximum number of processes supported. Consider, for example, a system with 4 cores, 64 page colors and 16 MB memory per color – at most 64 processes can be supported with 1 color each, with the memory footprint of every process limited to no more than 16 MB. Additionally, cache utilization is as low as 6.25% in the worst case when all cores are active.

To mitigate the problem of cache utilization, we want to make optimal color assignments for application processes according to their cache demands. However, this requires a-priori knowledge that is difficult to acquire for most applications. Consequently, we decided to adopt a more flexible color assignment scheme that does not require offline profiling when optimal assignments are not provided.

In this new scheme, the cache is divided into N sections of contiguous page colors on a platform of N processing cores.

Each section is then assigned to a specific core, which we call the local core of all the colors within this section. All the other cores will be referred to as remote cores. When a new process is created, COLORIS searches for a core with the lightest workload and assigns the whole cache section of the core to the process. As a result, in a system with a total of C page colors, every process will be assigned $\frac{C}{N}$ colors. This means that N co-running processes can fully utilize the cache.

For load balancing, process migration can be invoked and the entire address space needs to be recolored. Migration may also be applied to reduce memory bus contention, by placing memory intensive processes to the same cores. However, since migration incurs prohibitive overhead, it should be avoided when possible.

Though the previous static partitioning scheme is simple, it can still potentially lead to low cache utilization for an under utilized system and when there are dynamic program phase changes. A more practical way would be to assign a default-sized partition first, and then gradually approach the best size through re-partitioning (i.e. recoloring). Thus, for a page coloring technique to be truly useful in production systems, dynamic partitioning is essential.

In COLORIS, we extended the Page Color Manager with dynamic partitioning capabilities, by introducing a comprehensive recoloring mechanism. Our primary design focus of this recoloring mechanism is to make it effective and efficient even in over-committed systems.

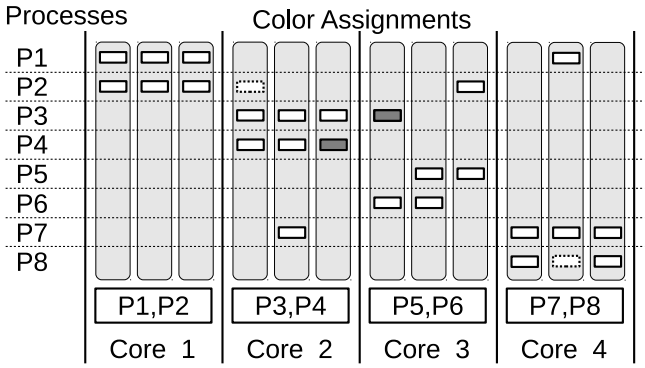

Figure 5: COLORIS Page Coloring Scheme

Starting with the initial color assignments described earlier, the Page Color Manager attempts to make online color assignment changes based on dynamic application behaviors. For applications that do not require the entire local cache section, some colors can be reclaimed; for applications that demand more cache space, colors from other sections can be shared. An example is illustrated in Figure 5. In this example, the twelve columns represent twelve different page colors. Every block within a column is an ownership token denoting that the process having this token (in the same row) is assigned the corresponding color.

Since the local cache section can now be shared, limited cache interference between processes may happen. In this case, a global coordination amongst schedulers on each core is useful. By sharing the information on page color usages and scheduling processes accordingly, the likelihood of cache contention is reduced. While a cache-aware scheduler is not implemented in COLORIS, we leave it as future work.

We have also considered an alternative design for color assignment. Instead of re-partitioning at the granularity of processes, we can adjust the size of a cache section for each core. Before doing that, we need a mechanism for online application classification in terms of cache demands. We want to move applications of the same class to the same core in order to maintain high cache utilization. For online classification, there are several useful areas of study [11, 19, 32, 33, 34].

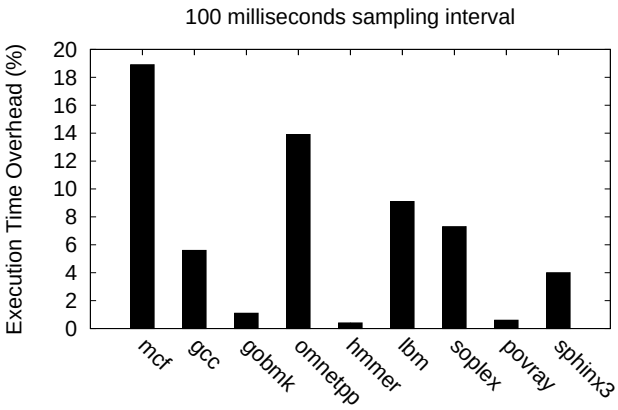

Figure 6: Overhead of Hot Page Identification

Cache Utilization Monitor. The Cache Utilization Monitor is responsible for measuring the cache usage of individual applications, and making decisions about partition adjustments. To start with, we need to introduce the concept of *hotness*, which is the key for partition adjustment. In previous work, page color hotness is defined as the aggregate access frequency of page frames with the same color [35]. It was used to help reduce the cost of cache re-partitioning by only recoloring pages of hot colors. However, hotness identification requires expensive periodic page table scans, which may result in worse performance than if hot page identification is not used at all. We conducted a preliminary experiment on the proposed approach from this work with SPEC CPU2006. The results from Figure 6 clearly support our argument.

Based on this observation, we decided upon a new definition of hotness. One approach is to define it as a function of the number of free pages in the target color. A second approach, which is more appropriate and has been adopted in COLORIS, is to use the number of processes sharing the same color as the definition of hotness. This information can be used to avoid heavy cache contention by assigning cold colors to newly created processes. Due to its simplicity, we believe this to be a suitable metric in practical systems.

Following the re-definition of hotness, we now describe the two recoloring approaches in COLORIS. In the first approach, ALLOC_COLORS(*UNIT*) is invoked to add *UNIT* colors to a process, whenever it runs out of memory with its pre-existing colors. Here, *UNIT* is a configurable number of colors. The second approach triggers recoloring when a process's cache demand exceeds its current assignment. This is determined by monitoring the cache miss rate for a given process, defined as cache misses with respect to total

Algorithm 1 Cache Utilization Monitor

```
procedure MONITOR(cmr)
    assignment = assignment_of(current)
    if cmr > HighThreshold then
        if isCold = False then
            isHot ← True
            return
        end if
        new = ALLOC_COLORS(UNIT)
        /* triggers Recoloring Engine */
        assignment+ = new
        isCold ← False
    else if cmr < LowThreshold then
        if isHot = True then
            isHot ← False
            isCold ← True
            victims = PICK_VICTIMS(UNIT)
            /* triggers Recoloring Engine */
            assignment− = victims
        end if
    end if
end procedure

procedure ALLOC_COLORS(num)
    new ← φ
    while num > 0 do
        if needRemote() then
            new+ = pick_coldest_remote()
        else
            new+ = pick_coldest_local()
        end if
        num ← num − 1
    end while
    return new
end procedure

procedure PICK_VICTIMS(num)
    victims ← φ
    while num > 0 do
        if hasRemote() then
            victims+ = pick_hottest_remote()
        else
            victims+ = pick_hottest_local()
        end if
        num ← num − 1
    end while
    return victims
end procedure
```

accesses over a sample period, using hardware performance counters commonly available on modern processors.

We set up two global cache miss rate thresholds when the system starts up, *HighThreshold* and *LowThreshold*. Applications with miss rates higher than *HighThreshold* are the ones needing more cache space; others with miss rates below *LowThreshold* are willing to provide vacant cache space for re-partitioning. Procedure *MONITOR* in Algorithm 1 is used to trigger recoloring. It takes the cache miss rate (*cmr*) of the current process (*current*) over a period of time (*PERIOD*) as input. The *needRemote()* function returns *True* if *current* has already been using the entire local cache section. The *hasRemote()* function returns *True* if *current* owns colors from remote cache sections. The boolean variable *isHot* is used to indicate when the cache miss rate of a process goes above a specific threshold. It acts as a signal to indicate that a process needs more page colors. Conversely, *isCold* is set when extra page colors are available. Both variables are global to all processes.

Functions *pick_coldest_remote()* and *pick_hottest_remote()* choose a color in a remote cache section belonging to the *current* process, with the smallest or largest global hotness value, respectively. Similarly, *pick_coldest_local()* and *pick_hottest_local()* return a color from the local cache section, owned by *current*, with the smallest or largest remote hotness value, respectively. Here, we define global hotness as the number of owners of a color running on all cores, while remote hotness is the number of owners of a color running on remote cores. We also define a local color to be any color within a local cache section.

The key insight of using remote hotness is that, when adding or taking away a local color to/from a process, we do not care about other processes running on the same core. Since they cannot run simultaneously with one another, there is no cache interference amongst them for sharing colors. In Figure 5, we show all four cases where the four functions above are called respectively. The figure illustrates a state transition during recoloring. Solid white blocks indicate the process owns the corresponding color before and after recoloring. Dashed white blocks indicate the process has the color before recoloring. Similarly, dark blocks indicate the process is assigned the color after recoloring.

Despite the global thresholds, we also allow individual applications to provide their private threshold pair as part of QoS specification. When QoS specification is not available, global thresholds are used.

While dynamic partitioning can benefit from the information provided by cache utility curves, such information is not easily obtainable in a running system. Without this information, COLORIS' objective is to enhance QoS by attempting to maintain application miss rates below *HighThreshold*, given sufficient capacity. Sufficient capacity here means there are some other applications, with miss rates lower than *LowThreshold*, that are able to provide enough free colors. If there exists a cache partitioning scheme that guarantees no application's miss rate exceeds *HighThreshold*, COLORIS attempts to achieve such guarantee.

Notice that the Cache Utilization Monitor takes into account both cache miss rates and the number of cache references. When the number of references is small, miss rate for a process is set to zero, to indicate the cache is not being used significantly. Likewise, in situations where frequent page recoloring is not beneficial to overall performance, it is possible to disable recoloring for individual processes.

Recoloring Engine. The Recoloring Engine performs two tasks: (1) shrinkage of color assignments, and (2)expansion of color assignments. Lazy recoloring [14] is adopted for shrinking a color assignment. Basically, we look for pages of specific colors that are going to be taken away and clear the present bits of their page table entries. At the same time, an unused bit of every page table entry whose present bit is cleared is set to identify the page as needing recoloring. However, we do not set a target color for each page to be recolored. In the page fault handler, we allocate a new page from the page allocator and copy the content of the old page to it. Since round-robin is used in page allocation, as described in Section 3.1, pages to be recolored are eventually spread out uniformly across the cache partition assigned to the process.

Assignment expansion is more complicated than shrinkage. The major reason being that it is difficult to figure out

which pages should be moved to the new colors. Ideally, we want memory accesses to be redistributed evenly across the new cache partition. In COLORIS, we currently consider two selection policies.

1) *Selective Moving* – In this policy, we take the set associativity of the cache into consideration. Assuming an n-way set associative cache, we know that one page color of the cache can hold up to n pages at the same time. We therefore scan the whole page table of the current process and recolor one in every $n + 1$ pages of the same color, trying to minimize cache evictions when big data structures with contiguous memory are being accessed. These pages will be immediately moved to the newly assigned colors in a round-robin manner.

2) *Redistribution* – When the expansion is triggered, we first go through the entire page table of the current process and clear the access bit of every entry. The access bit is a special bit in page table entry on x86 platforms; whenever the page in that entry is accessed, this bit is automatically set by hardware. After a fixed time window *WIN*, we scan the page table again and find all entries with the access bit set. Pages in those entries have all been accessed during the time window. Since it is hard to re-balance the cache hotness by moving selected pages, we let the process itself perform the redistribution. That is, all accessed pages are recolored using lazy recoloring as mentioned above.

Comparing these two policies, *Selective Moving* is simpler and more light-weight. However, *Redistribution* is likely to be more powerful for re-balancing cache hotness. Their effectiveness is evaluated in the next section.

4. EVALUATION

We conducted a series of experiments to evaluate the performance and effectiveness of the COLORIS page coloring framework. We implemented a prototype system for a 32-bit Ubuntu 12.04 Linux OS with kernel version 3.8.8. All the experiments except the last one in Section 4.5 were conducted using the SPEC CPU2006 benchmark suite on a Dell PowerEdge T410 machine with a quad-core Intel Xeon E5506 2.13GHz processor and 8GB of RAM. A total of 4MB 16-way set-associative L3 cache was shared amongst the 4 cores of the processor. As a result, there were 64 page colors available in the system with 4KB page size. Since we set the size of the memory pool in our Color-aware Page Allocator to be 1GB, each color provided up to 16MB of memory for application use.

4.1 Allocator Implementation Overhead

In the first experiment, we compared the performance of the original Linux memory allocator with COLORIS's page allocator to evaluate the efficiency of our implementation. In Linux, we ran each SPEC benchmark alone in the system without co-runners and recorded their execution times. In COLORIS, we assigned all colors, effectively the entire cache, to every program and ran them with the same configuration. In the latter case, COLORIS picked page colors in round-robin manner for each application page request. Table 1 shows that our page allocator achieved similar performance as the original Linux Buddy System.

	Linux (sec)	**COLORIS (sec)**
gobmk	736	738
gcc	507	495
libquantum	773	778
bzip2	961	956
sphinx3	864	866
omnetpp	476	480
povray	366	367
hmmer	849	850
h264ref	1131	1128
mcf	450	444
soplex	429	428
leslie3d	797	799
gromacs	1386	1384
namd	790	792
milc	671	679
gamess	1503	1510
zeusmp	842	841
soplex	429	428
tonto	977	977
wrf	1328	1323
calculix	1517	1512
sjeng	871	868
astar	777	773
perlbench	569	569
cactusADM	1796	1788
GemsFDTD	725	728
lbm	533	535

Table 1: Allocator Implementation Overhead

4.2 Effectiveness of Page Coloring

The benefit of page coloring is the performance isolation between running processes. In the following experiments, we tried to evaluate the effectiveness of page coloring for applications of different characteristics. We first selected three groups of benchmarks: {*sphinx3*, *leslie3d*, *libquantum*}, {*leslie3d*, *h264ref*, *gromacs*}, and {*povray*, *h264ref*, *gromacs*}. According to their memory access intensity, we call them the heavy background workload (H), the medium background workload (M) and the light background workload (L), respectively. Three foreground benchmarks were then selected: *omnetpp*, *gobmk* and *hmmer*. *omnetpp* has a very large memory footprint and is memory-intensive. *gobmk*, with a small footprint, is less memory-intensive but cache-sensitive. *hmmer* is similar to *gobmk* except for cache-sensitivity, due to its small working set.

For every experiment, we chose one foreground workload and one background workload. The three background programs were started first, each pinned to a different core. The foreground workload started execution after a delay of one second on the fourth core. The background workload remained executing during the entire life of the foreground workload. We denote experiments with page coloring as P experiments, and experiments with heavy, medium, and light background workloads as H, M, and L experiments, respectively. For example, an experiment running foreground and heavy background workloads together with page coloring is labeled as H + P. In any experiment with page coloring, every workload was assigned 16 page colors.

We also had two control groups: the first group, S, ran only the foreground workloads, with page coloring limiting

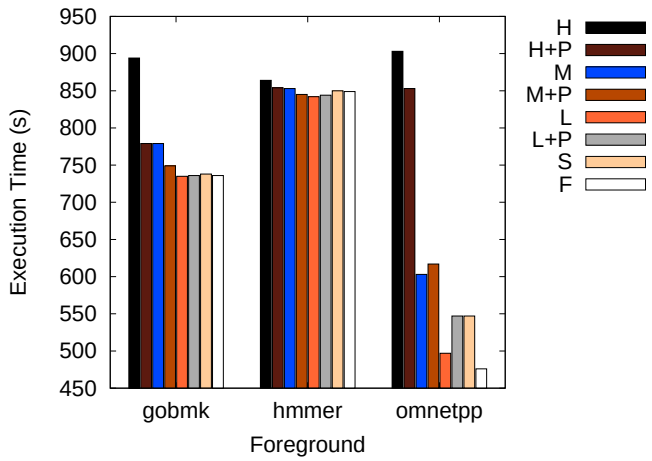

Figure 7: Execution Time of Foreground Workload

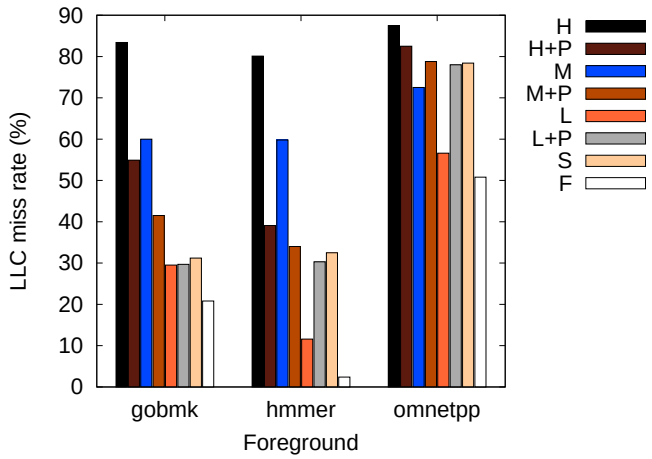

Figure 8: Cache Miss Rate of Foreground Workload

access to a quarter of the last-level cache; the second group, F, also ran only the foreground workloads but without page coloring, thereby allowing access to the full cache. We conducted all experiments three times and recorded the average execution times and LLC miss rates of each foreground workload. The results are shown in Figure 7 and Figure 8.

As can be seen in Figure 7 (H and H + P), the cache-sensitive workload *gobmk* experienced a performance gain of as much as 13% under the interference of a heavy background workload with page coloring. For non-cache-sensitive workload *hmmer*, page coloring was less effective.

With ideal performance isolation, given the same cache partition size, a foreground workload running with background workload should exhibit the same behavior as when it is running alone in the system (which is represented by the control group S). As Figure 7 shows, with the presence of interference, COLORIS was able to constrain the performance variations of *gobmk*, *hmmer* and *omnetpp* to within 5%, 0.5% and 55% respectively, as compared to S. Without page coloring, the variations can be as much as 21%, 2% and 90%, as compared to F. For *omnetpp*, a 39% reduction in variation was achieved.

Notice that *omnetpp* suffered higher miss rate when page coloring was enabled (comparing L and L+P). This is mainly caused by the high self-conflict from limiting its memory accesses to just $\frac{1}{4}$ of the entire cache (as shown in Figure 8).

4.3 Recoloring Evaluation

Configurations	LowThreshold (%)	HighThreshold (%)
C1	30	65
C2	30	75
C3	0	100
C4	(40, 0, 30, 40)	(80, 60, 65, 80)
C5	-	-
C6	30	75

Table 2: Experimental Configurations

To demonstrate the benefit of recoloring, we designed a set of experiments with four benchmarks, *povray*, *tonto*, *omnetpp* and *gamess*. Amongst them, only *omnetpp* is memory-intensive and cache-hungry, meaning the more cache space there is, the lower its cache miss rate will be. If the other three benchmarks are similar to *omnetpp* in terms of cache demand, then there will be no possibility for re-partitioning. Configurations for different experiments are listed in Table 2. In all the configurations *WIN* was set to 3ms (except for C6), *UNIT* to 4, and *PERIOD* to 5 seconds. (40, 0, 30, 40) in C4 means assigning these four *LowThresholds* to *povray*, *tonto*, *omnetpp* and *gamess*, respectively, instead of using global *LowThreshold*. Note that the thresholds of configuration C3 prevent page recoloring altogether. Configuration C5 was used for the special case in which every benchmark ran alone with full cache space access.

All the experiments were carried out by fixing each benchmark to a different core. The four benchmarks were started at the same time, each with 16 non-overlapping page colors. Each experiment was run for more than an hour. After the first minute, we set up the performance counters to collect LLC miss rates and the numbers of instructions retired from user-level (to exclude kernel overhead) over a measured *60 minutes* interval. At the same time, the Cache Utilization Monitor was also enabled. For these experiments, the LLC was fully utilized, which meant colors needed to be removed from one application before they could be re-assigned to another. The final result was produced by repeating the same experiments three times and taking the average, which is shown in Figure 9, Table 3 and Table 4.

With static cache partitioning (C3), the miss rate of *omnetpp* reached 77.8%. With the help of recoloring (C1 and C2), the miss rate was successfully limited to below *HighThreshold* thus meeting the system default QoS requirement. By comparing miss rates of *omnetpp* between C1 and C2, we can tell that the degree of improvement for cache-hungry applications depends on how we set up the thresholds. As shown in Table 3, despite the recoloring overhead, *omnetpp* had received 3.3% − 9.4% performance gain, proving that our approaches are practical. *povray* managed to keep the same miss rate with fewer cache lines since its working set is small enough to fit into private L2 cache. Although the other two benchmarks, *tonto* and *gamess*, were negatively affected by reduced cache sizes, the system overall performance remained roughly the same.

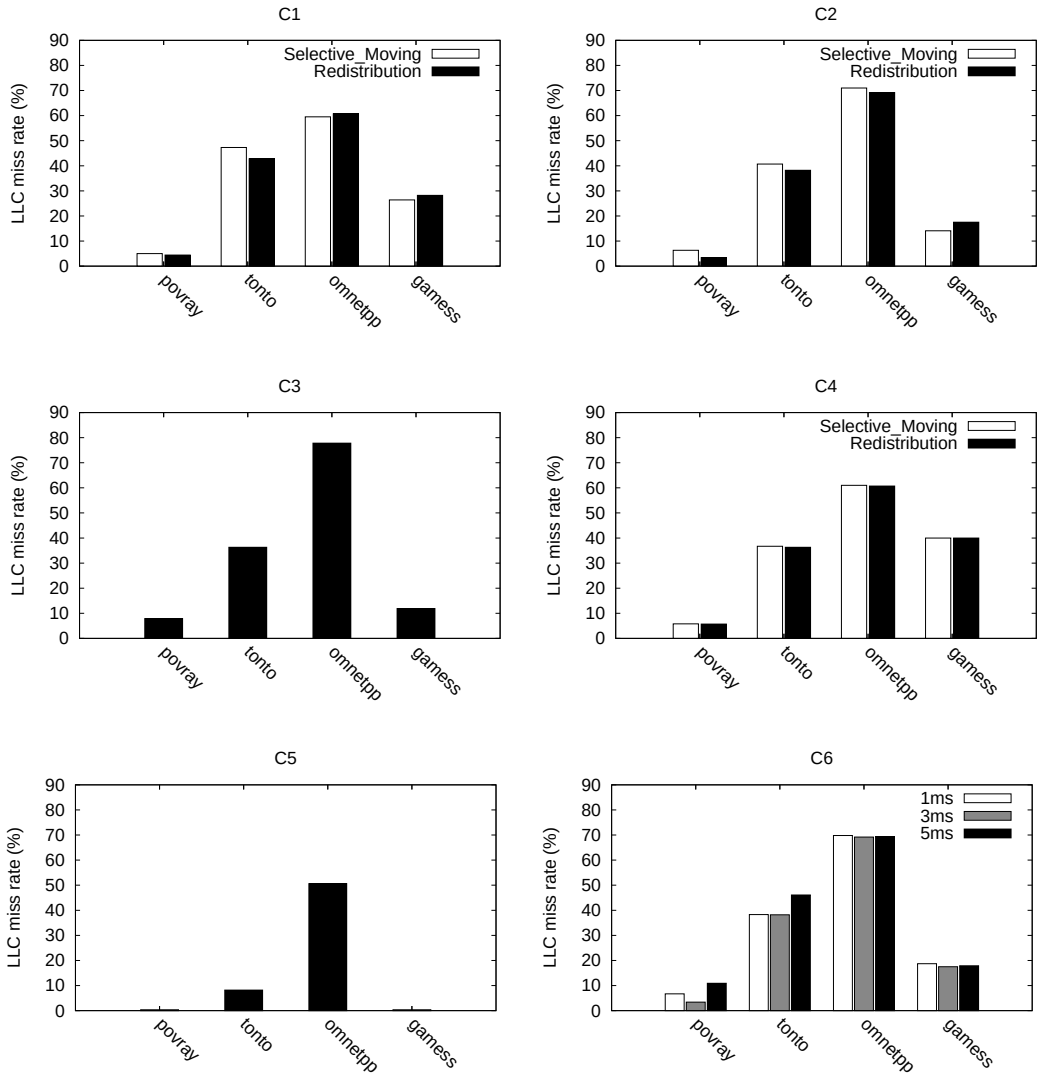

Figure 9: LLC Miss Rates

		C1	C2(C6.WIN=3)	C3	C4	C6.WIN=1	C6.WIN=5
Selective Moving	**povray**	100.1	101.8	102.2	100.7	-	-
	tonto	88.9	90.7	91.6	91.4	-	-
	omnetpp	46.2	43.6	42.2	45.8	-	-
	gamess	120.9	122.8	123.1	119.2	-	-
	total	356.1	358.9	359.1	357.1	-	-
Redistribution	**povray**	100.2	101.7	-	100.7	101.7	101.8
	tonto	90.0	91.4	-	91.4	91.2	89.9
	omnetpp	46.0	44.1	-	46.0	43.9	43.8
	gamess	121.1	122.3	-	119.3	121.3	122.5
	total	357.3	359.5	-	357.4	358.1	358.0

Table 3: Instructions Retired in One Hour ($\times 10^{11}$)

Instead of using system default QoS specification, reasonable individual specifications can also be satisfied by the COLORIS framework as shown in C4. By setting *LowThreshold* to 0 for *tonto*, we avoided decreasing its cache space, and limited its miss rate to below 40%. The result shows that recoloring is effective for guaranteeing QoS requirements of individual applications.

Table 4 lists the total number of pages recolored in all the experiments. It can be seen that *Redistribution* incurred higher overhead as compared to *Selective Moving*. How-

	C1	C2(C6.WIN=3)	C4	C6.WIN=1	C6.WIN=5
Selective Moving	10746	3845	8791	-	-
Redistribution	32749	22243	33547	6476	30174

Table 4: Recoloring Overhead (Total # Pages Recolored)

ever, *Redistribution* achieved a slightly better overall system performance if we look at the total instructions retired in Table 3. The result indicates that *Redistribution* is more effective in utilizing newly assigned cache space.

Following this observation, we then evaluated the impact of window size *WIN* on the effectiveness of the *Redistribution* policy in C6. *omnetpp* was the only application that received an expanded cache space during the experiment. Its performance under different window sizes stayed the same, which suggests that larger window size may not help improve performance, as recoloring overhead will increase as well.

Finally, Table 5 shows the stable color assignments for each repetition of the above experiments. Within each 4-element tuple, the numbers are the sizes of color assignments for *povray*, *tonto*, *omnetpp* and *gamess* respectively. Across multiple repetitions of the same experiment, stable-state color assignments are fairly consistent. This indicates the COLORIS recoloring mechanism is stable.

4.4 Performance in Over-Committed Systems

In these experiments, we tried to evaluate the design of COLORIS for over-committed systems. We first created four groups of SPEC benchmarks (G1, G2, G3, G4): {*gobmk*, *sphinx3*}, {*gromacs*, *leslie3d*}, {*h264ref*, *omnetpp*}, {*povray*, *gamess*}. Programs were started with ten seconds interval in the following order: *gobmk*, *gromacs*, *h264ref*, *povray*, *sphinx3*, *leslie3d*, *omnetpp*, *gamess*. One minute after the last benchmark program was started, we enabled hardware performance counters and the Cache Utilization Monitor (if used). All experiments were then run for an hour, at the end of which results were collected.

When running experiments in COLORIS, benchmark programs received their initial color assignments in a round-robin fashion, since all colors had zero hotness at the beginning. According to their launching order, programs in group Gx were assigned the cache section belonging to core x. After assignment, they were pinned automatically to those cores. For experiments in Linux, we pinned each group onto the same core similar to the COLORIS experiments. By doing this, we eliminated the difference of memory bus contention between the two sets of experiments. The same experiments were carried out with three different configurations, C7, C8 and C9. In C7, recoloring used the *Redistribution* policy, *WIN* = 3ms, UNIT = 4, PERIOD = 5s and system-wide thresholds were set to (*LowThreshold*, *HighThreshold*) = (20%, 75%). An individual QoS specification (0%, 100%) was also provided to *leslie3d*, which essentially disabled recoloring. The reason being *leslie3d* is an LLC-thrashing [11] application that does not benefit from assignment expansion (known through offline profiling). In C8, a static partitioning scheme was used. For C9, applications were run in Linux without page coloring. Results from Figure 10 show that COLORIS performs no worse than Linux in heavy over-committed cases, while at the same time guaranteeing QoS for applications.

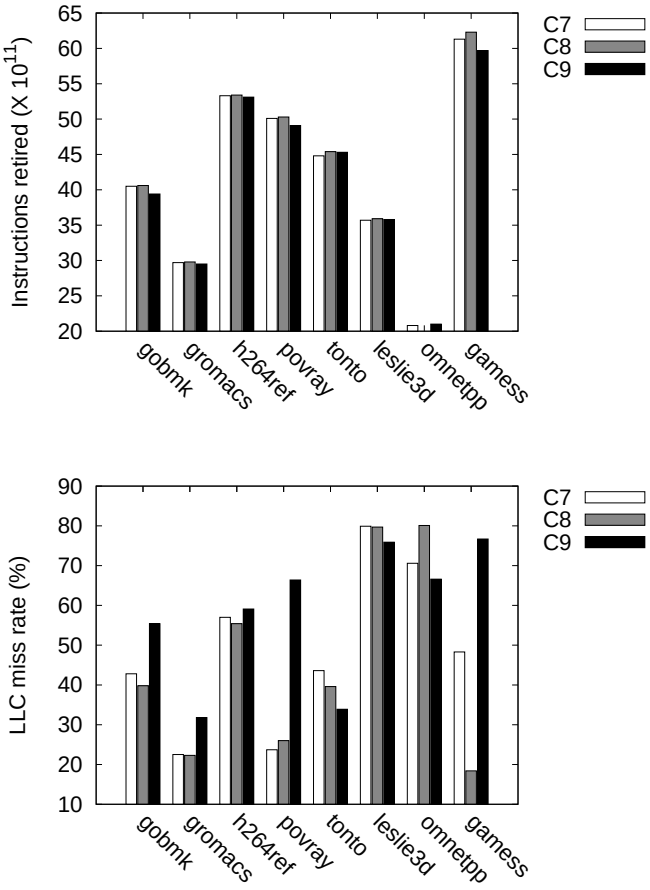

Figure 10: Over-Committed System Performance

4.5 Page Coloring on Cache Slices

In order to prove that page coloring can work with the latest cache-slice architecture, we conducted another set of experiments similar to the ones in Section 4.2, but on an Intel Core i3-2100 Sandy Bridge dual-core processor. It has two 1.5MB, 12-way set-associative L3 cache slices. The system also features 4GB DDR3 RAM. The three background workloads were: H = {*libquantum*}, M = {*h264ref*}, L = {*povray*}. The results are shown in Figure 11. Note that for *omnetpp* with a light background workload (case L), the execution time is exactly 350 seconds. The performance with a heavy background workload clearly demonstrates the isolation provided by page coloring.

5. RELATED WORK

5.1 Software Cache Partitioning

Page coloring [31] was first introduced as a way to manage shared caches on multicore processors in [23]. Cho and

		C1	C2(C6.WIN=3)	C4	C6.WIN=1	C6.WIN=5
Selective Moving	**EXP(1)**	(4,8,44,8)	(8,16,28,12)	(4,16,40,4)	-	-
	EXP(2)	(4,12,40,8)	(16,12,24,12)	(4,16,40,4)	-	-
	EXP(3)	(4,8,44,8)	(12,12,28,12)	(4,16,40,4)	-	-
Redistribution	**EXP(1)**	(4,12,40,8)	(12,16,28,8)	(4,16,40,4)	(8,12,28,16)	(16,8,28,12)
	EXP(2)	(4,12,40,8)	(8,16,28,12)	(4,16,40,4)	(12,16,28,8)	(16,12,28,8)
	EXP(3)	(4,12,40,8)	(12,12,28,12)	(4,16,40,4)	(16,16,28,4)	(12,12,28,12)

Table 5: Stable-State Color Assignments (povray, tonto, omnetpp, gamess)

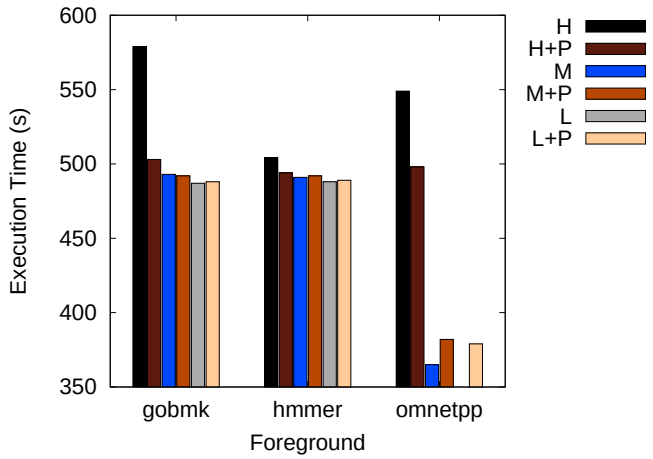

Figure 11: Page Coloring with Cache Slices

Jin [6] applied page coloring to a multicore system with a distributed shared cache, with the goal to place data in cache slices that are closer to the CPU running the target application. Tam et al [29] implemented static page coloring in a prototype Linux system.

Lin et al [14] later investigated recoloring policies for fully committed systems, having the same number of co-running workloads as cores. While these policies can be applied to a 2-core platform, their complexity grows exponentially as the number of cores increases. Moreover, page hotness [35] is ignored when picking pages for recoloring, leading to inefficient use of newly assigned colors when all recolored pages are barely accessed. Using hot pages can effectively reduce the number of pages to be recolored, but it pays additional overhead from hotness identification.

Recent cache management work has attempted to divide the cache into different usages. Soares et al [24] determined cache-unfriendly pages via online profiling and mapped them to a pollute buffer, which is a small portion of the cache. Lu et al [16] let a user-level allocator cooperate with a kernel page allocator to provide object-level cache partitioning, in which the partitions are decided offline. The basic idea is to force user-level data structures with weak locality to use a reserved part of the cache, while other data structures use the rest.

Another interesting work proposed an SRM-Buffer [7], which reduces cache interference from kernel address space. In systems like Linux, the page cache usually occupies a significant amount of memory – one burst of accesses to it may incur large-scale cache evictions, hurting application performance. To address this problem, the authors limited the range of page colors that can be used by the Linux page cache during a file IO burst.

5.2 Hardware Cache Partitioning

Unlike software partitioning, which divides a shared cache among applications, hardware partitioning usually assigns different sections of cache to different cores. Way partitioning [5, 21, 28] and block partitioning [27] are the two common approaches for partitioning a shared cache. Way partitioning restricts cache replacement for a core to a specific set of ways in an n-way set-associative cache. Block partitioning enforces partitions within individual sets. Rafique et al [20] developed a loose form of block partitioning, which only limits the total number of cache lines each core can occupy. Others [10, 22] partitioned cache blocks statistically, allowing minor interference while retaining cache occupancies. A hybrid approach was proposed by Mancuso et al [17], which combines page coloring with way partitioning, effectively turning a shared LLC into scratchpad memory [1] for a fixed set of hot data. Srikantaiah et al [25] argued that cache partitions should be assigned to CPU sets instead of individual CPUs, facilitating constructive sharing for multithreaded applications. Current trends also favor the design of configurable or hybrid (private/shared) caches [2, 8, 9, 15, 18, 36], which assign cache slices for different uses.

5.3 Cache Utility Curves

Cache utility curves provide a rich set of information to determine the optimal-sized cache partitions. Zhou et al [37] described a software approach to generate miss-rate curves (MRCs) online. The approach maintains an LRU page list for each address space. Page accesses are determined by either periodically scanning page table entries or clearing present bits of entries and causing page faults. Tam et al [30] built RapidMRC by using a sampled data address register (SDAR) available on the PowerPC to record all addresses of memory requests over sampled periods. In other work on cache-aware fair and efficient scheduling (CAFÉ [33]), utility curves were generated through online cache occupancy modeling.

Finally, there has been work focused on the development of new hardware to construct utility curves. Suh et al [26] proposed way counters and set counters for computing the marginal reduction in cache misses as cache space shrinks. However, their approach requires an application to run on its own, with access to the entire cache, in order to accumulate utility data. To overcome this limitation, Qureshi and Patt [19] implemented the UMON monitoring circuit, which acts like a copy of the cache. UMON collects complete utility data when only a portion of the cache is allowed for access, regardless of contention from other cores.

6. CONCLUSIONS AND FUTURE WORK

On multicore platforms, processes compete for a shared last-level cache, causing conflict misses and performance variations. COLORIS addresses this problem by extending traditional OSes with static and dynamic cache partitioning capabilities. In the static scheme, COLORIS exclusively assigns different page colors to processes on different cores. In the dynamic scheme, a Cache Utilization Monitor measures application cache usage online and triggers re-partitioning to satisfy QoS demands of individual applications. To achieve efficient re-partitioning, two page selection policies are described: *Selective Moving* and *Redistribution*. When applied to over-committed systems, COLORIS tries to maintain good cache isolation amongst processes, attempting to minimize cache interference, by carefully managing page colors in a Page Color Manager. Our experimental results show that COLORIS is able to reduce the performance variation due to co-running workload interference by up to 39%, and successfully provides QoS for memory-bound applications.

COLORIS supports dynamic partitioning in systems that are over-committed. In such cases, COLORIS allows a limited amount of sharing of page colors with application processes on remote cores. The performance of COLORIS is at least as good as vanilla Linux implementations for heavily over-committed situations, while significantly outperforming vanilla Linux for cases where the system is less loaded.

Future work will investigate cache-aware scheduling algorithms that cooperate with COLORIS, as we attempt to eliminate cache conflict misses by different workloads altogether. In particular, we will investigate ways to provide hard guarantees on the isolation between workloads, which is important for real-time applications with deadlines. In addition to cache contention, we will also consider factors such as shared memory buses and NUMA interconnects on multi-socket and manycore architectures [3].

7. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1117025. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

8. REFERENCES

[1] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign*, CODES '02, pages 73–78, New York, NY, USA, 2002.

[2] N. Beckmann and D. Sanchez. Jigsaw: Scalable software-defined caches. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 213–224, Piscataway, NJ, USA, 2013.

[3] B. K. Bray, W. L. Lunch, and M. J. Flynn. Page allocation to reduce access time of physical caches. Technical report, Stanford, CA, USA, 1990.

[3]COLORIS source code is available upon request.

[4] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam. Compiler-directed page coloring for multiprocessors. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 244–255, New York, NY, USA, 1996.

[5] D. Chiou, P. Jain, L. Rudolph, and S. Devadas. Application-specific memory management for embedded systems using software-controlled caches. In *Proceedings of the 37th Annual Design Automation Conference*, pages 416–419, New York, NY, USA, 2000.

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

[7] X. Ding, K. Wang, and X. Zhang. SRM-buffer: an OS buffer management technique to prevent last level cache from thrashing in multicores. In *Proceedings of the 6th ACM European Conference on Computer Systems*, pages 243–256, New York, NY, USA, 2011.

[8] H. Dybdahl and P. Stenstrom. An adaptive shared/private NUCA cache partitioning scheme for chip multiprocessors. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, pages 2–12, Washington, DC, USA, 2007.

[9] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. A NUCA substrate for flexible CMP cache sharing. In *Proceedings of the 19th Annual International Conference on Supercomputing*, pages 31–40, New York, NY, USA, 2005.

[10] R. Iyer. CQoS: A framework for enabling QoS in shared caches of CMP platforms. In *Proceedings of the 18th Annual International Conference on Supercomputing*, pages 257–266, New York, NY, USA, 2004.

[11] A. Jaleel, H. H. Najaf-abadi, S. Subramaniam, S. C. Steely, and J. Emer. CRUISE: cache replacement and utility-aware scheduling. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 249–260, New York, NY, USA, 2012.

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

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

[14] 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 *Proceedings of the 14th International Symposium on High-Performance Computer Architecture*, pages 367–378, 2008.

[15] C. Liu, A. Sivasubramaniam, and M. Kandemir. Organizing the last line of defense before hitting the memory wall for CMPs. In *Proceedings of the 10th*

International Symposium on High-Performance Computer Architecture, pages 176–185, 2004.

[16] Q. Lu, J. Lin, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Soft-OLP: Improving hardware cache performance through software-controlled object-level partitioning. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 246–257, 2009.

[17] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *Proceedings of the 19th Real-Time and Embedded Technology and Applications Symposium*, pages 45–54, 2013.

[18] R. Manikantan, K. Rajan, and R. Govindarajan. Nucache: An efficient multicore cache organization based on next-use distance. In *Proceedings of the 17th International Symposium on High-Performance Computer Architecture*, pages 243–253, Feb 2011.

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

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

[21] P. Ranganathan, S. Adve, and N. P. Jouppi. Reconfigurable caches and their application to media processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 214–224, New York, NY, USA, 2000.

[22] D. Sanchez and C. Kozyrakis. Vantage: Scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, pages 57–68, New York, NY, USA, 2011.

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

[24] 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 *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 258–269, Washington, DC, USA, 2008.

[25] S. Srikantaiah, R. Das, A. K. Mishra, C. R. Das, and M. Kandemir. A case for integrated processor-cache partitioning in chip multiprocessors. In *Proceedings of the Conference on High-Performance Computing Networking, Storage and Analysis*, pages 6:1–6:12, New York, NY, USA, 2009.

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

[27] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic cache partitioning for simultaneous multithreading systems. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 116–127, 2001.

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

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

[30] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 121–132, New York, NY, USA, 2009.

[31] G. Taylor, P. Davies, and M. Farmwald. The TLB slice–a low-cost high-speed address translation mechanism. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 355–363, New York, NY, USA, 1990.

[32] R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang. Online cache modeling for commodity multicore processors. *SIGOPS Oper. Syst. Rev.*, 44(4):19–29, Dec. 2010.

[33] R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang. CAFÉ: Cache-aware fair and efficient scheduling for CMPs. In *Multicore Technology: Architecture, Reconfiguration and Modeling*. CRC Press, 2013.

[34] Y. Xie and G. Loh. Dynamic classification of program memory behaviors in CMPs. In *the 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2008.

[35] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems*, pages 89–102, New York, NY, USA, 2009.

[36] L. Zhao, R. Iyer, M. Upton, and D. Newell. Towards hybrid last level caches for chip-multiprocessors. *SIGARCH Comput. Archit. News*, 36(2):56–63, May 2008.

[37] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 177–188, New York, NY, USA, 2004.