BWLOCK: A Dynamic Memory Access Control Framework for Soft Real-Time Applications on Multicore Platforms

Heechul Yun[†], Santosh Gondi[‡], Siddhartha Biswas[†]
[†] University of Kansas, USA. {heechul.yun, sid.biswas}@ku.edu
[‡] Bose Corporation, Framingham, MA, USA. santosh_gondi@bose.com

Abstract—Soft real-time applications (e.g., multimedia processing) often show bursty memory access patterns, regularly requiring high memory bandwidth for a short duration of time. These memory bursts are often originated from a small number of memory-intensive code sections of the applications, which we call *memory-performance critical sections*. In multicore architectures, however, non-real-time applications executing in parallel may also demand high memory bandwidth at the same time, which can substantially degrade the performance of the soft real-time applications—i.e., missing the deadlines.

In this paper, we present a memory access control framework, BWLOCK, which is designed to protect the performance of soft real-time applications on multicore platforms. The framework consists of a user-level API and a kernel-level scheduling and bandwidth control mechanism. With the API, users can describe memory-performance critical sections of their applications. If an application enters a memory-performance critical section, the kernel-level control mechanism *dynamically* throttles memory access rates of the contending cores, which may execute non-real-time applications, to protect the performance of the soft real-time application until the memory-performance critical section is completed.

From case studies with two real-world soft real-time applications, we found that (1) memory-performance critical sections are often easy to identify; and (2) applying BWLOCK significantly improves the performance of the soft real-time applications with small or no throughput penalty of non-real-time applications.

1 Introduction

In a multicore system, an application's performance, running on a core, can be significantly affected by other applications on different cores, due to contention in shared hardware resources such as shared Last-Level Cache (LLC) and DRAM. When the shared resources become bottlenecks, traditional CPU scheduling based techniques such as raising priorities [18] or using CPU reservation based techniques [7], [13], [1] are no longer sufficient to ensure the necessary performance of real-time applications.

In hard real-time systems, one solution adopted in the avionics industry has been disabling all but one core in the system [17] to completely eliminate the shared resource contention problem. This allows the system to be certified [4] based on the traditional unicore-based certification process [2]. Another approach, adopted in PikeOS, is a time partitioning technique in which only one core is allowed to execute during a set of pre-defined time windows [8].

In the context of soft real-time systems, on the other hand, a certain degree of timing variations and deadline violations is often tolerable. Furthermore, modern multicore architectures provide a significant amount of parallelism—via out-of-order cores, non-blocking caches, multi-bank DRAM, and etc.—that can process a considerable degree of concurrent accesses without any noticeable performance impacts [10]. Therefore, it is highly desirable to develop a solution that can provide better real-time performance while still allowing concurrent execution to leverage the full potential of multicore architectures.

In this paper, we present BWLOCK, a user-assisted and kernel-enforced memory-performance control mechanism. It is designed to protect the performance of soft real-time applications from the interference of the other applications running on different cores. This is accomplished through a close collaboration between the applications and the OS. Our key observation is that interference is most visible when multiple cores have high memory demands at the same time. In such a case, all participating cores will be delayed due to queueing and other issues that cannot be hidden by the underlying hardware. Therefore, BWLOCK seeks to avoid overload situations especially when real-time applications are executing memory intensive regions of the code. We call such a region a *memory-performance critical section*. Fortunately, such memory-performance critical sections are often easy to identify in many soft real-time applications—particularly multimedia applications—through application level profiling. For example, using the *perf* tool in Linux [1], one can identify functions that demand high memory bandwidth.

Motivated by these observations, BWLOCK provides a lock like API with which programmers can describe certain code sections that are memory-performance critical. When these code sections execute, BWLOCK limits the amount of allowed memory traffic from the other cores to avoid overloading the memory bus. If source-code modification and profiling of an application is not desired (or impossible), BWLOCK can declare the entire execution of the application as mem-

1. A performance monitoring tool, included in the Linux kernel source tree.

ory performance critical. Then, whenever the application is scheduled, BWLOCK will be activated to protect its memory performance. We call the former as fine-grained bandwidth locking and the latter as coarse-grained bandwidth locking.

We applied BWLOCK in two real-world soft real-time applications—MPlayer (video player) and WebRTC (real-time multimedia communication framework [9])—to protect their real-time performance in the presence of memory intensive non-real-time applications. In the case of MPlayer, we achieve near perfect isolation for the MPlayer at a cost of 17% throughput reduction of the non-real-time applications with the coarse-grained bandwidth locking; we achieve a 17% real-time performance improvement for the MPlayer at the cost of only a 7% throughput reduction of the non-real-time applications with the fine-grained bandwidth locking. Similar improvements are observed for the WebRTC as well.

Our contributions are as follows:

- We propose a user-assisted and kernel-enforced memory-performance control mechanism that can substantially improve performance of soft real-time applications on commodity multicore systems.
- We present extensive evaluation results using real-world soft real-time applications demonstrating the viability and the practicality of the proposed approach.

The remaining sections are organized as follows: Section 2 provides background and motivation. Section 3 presents the design and implementation of BWLOCK. Section 4 describes the evaluation platform and the implementation overhead analysis. Section 5 presents case study results using two real-world soft real-time applications. Section 6 discusses limitations and possible improvements. We discuss related work in Section 7 and conclude in Section 8.

2 Background and Motivation

In [28], we proposed a software based memory bandwidth management system called MemGuard for the Linux kernel. The key idea is to periodically monitor and regulate the memory access rate of each core, using per-core hardware performance counters. If, for example, a group of tasks generates too much memory traffic and delays the critical real-time tasks, MemGuard can regulate the memory access rates of the cores executing the offending tasks. MemGuard provides a bandwidth reservation service that partitions a fraction of the reservable memory bandwidth, which is much lower than the peak bandwidth, in order to guarantee each reserved bandwidth can be delivered even in the worst-case. The bandwidth reservation parameters of the cores are chosen statically. Static partitioning, however, can be inefficient when demands of the cores change over time, because the unused bandwidth will be wasted. To minimize bandwidth waste, MemGuard employs a prediction-based bandwidth reclaiming mechanism that dynamically re-distributes unused bandwidth at runtime.

There are, however, two major problems when we apply MemGuard to improve the performance of soft real-time applications. First, MemGuard reserves memory bandwidth on a per-core basis. Therefore, when a core hosts different types of applications (with different memory bandwidth demands), it is difficult to choose an appropriate bandwidth reservation parameter for the core. Second, while this restriction can be mitigated—to a certain degree—by the runtime bandwidth re-distribution mechanism, the effectiveness of the mechanism depends on its prediction accuracy, which, in general, varies depending on the characteristics of the applications. In particular, multimedia soft-real-time applications, which we focus on in this paper, often show *bursty* memory access patterns. These patterns are difficult to predict without application level information or using sophisticated learning algorithms, which are difficult to implement efficiently in the kernel.

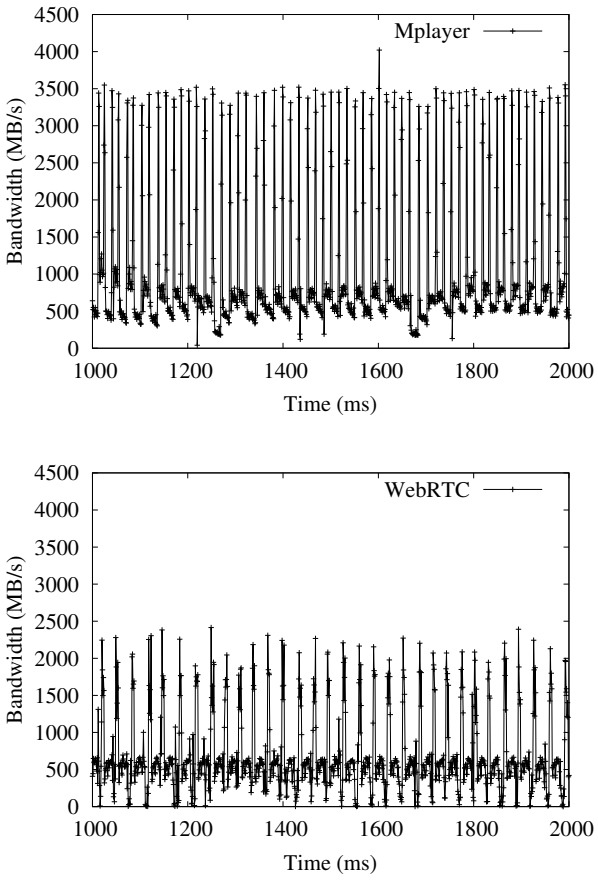


Fig. 1: Memory bandwidth demand changes over time of MPlayer and WebRTC.

Figure 1 shows memory access patterns of two multimedia applications—MPlayer and WebRTC—collected over the duration of one second (sampled at every 1ms.) As both programs process video/audio frames at a regular interval, when processing a new frame, they require high memory bandwidth for a short period of time. At other times, their memory bandwidth demands are low as they are executing compute-intensive instructions or waiting for the next period.

When these soft real-time applications compete for memory bandwidth with the other applications running on different cores, the short code sections that demand high memory bandwidth could suffer a disproportionally high degree of

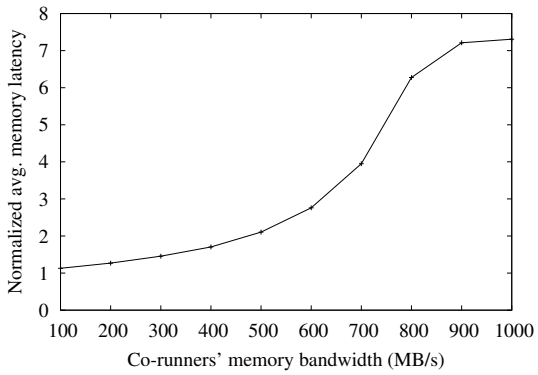


Fig. 2: Average memory access latency of a Latency benchmark as a function of the memory bandwidth of each co-runner on three different cores.

performance degradation. When the overall memory demand is low, memory access latencies often can be hidden due to a variety of latency hiding techniques and the abundant parallelism in modern multicore architectures [10]. However, because memory is much slower than the CPU, when the overall demand is beyond a certain threshold, even the most advanced architecture would no longer be capable of hiding the latencies. Thus, requests pile up in various queues in the system, which substantially slowdowns all the competing tasks that access the memory.

Figure 2 illustrates this phenomenon. In this experiment, we measure the average memory access latency (normalized to run-alone latency) using the *Latency* [28] benchmark (a pointer chasing micro-benchmark) while varying memory bandwidth of the co-runners on the other three cores in a quad-core Intel Xeon system (detailed hardware setup is described in Section 4). When each co-runner's bandwidth is low (100MB/s), the performance impact to the measured Latency benchmark is negligible. However, as the co-runner's memory bandwidth demand increases (from 100 to 900 MB/s), the observed latency of the Latency benchmark increases exponentially.

In summary, our observations are as follows: (1) Soft real-time applications such as multimedia applications often show bursty memory access patterns—regularly requiring high memory bandwidth for a short duration of time. (2) Such a period, which we call a *memory-performance critical section*, is often critical for timely data processing but it can be disproportionally delayed by bandwidth demanding co-runners on different cores.

These observations have motivated us to design BWLOCK, which is described in the next section.

3 BWLOCK

BWLOCK is composed of a user-level API and a kernel-level memory bandwidth control mechanism and it is designed to improve performance of soft real-time time applications (e.g., multimedia applications) on a multicore system. It provides a simple lock like API that can be called by the application programmers to express the importance of memory performance for a given section of code. Once an application acquires the lock, which we call a memory bandwidth lock, the bandwidth control mechanism regulates the maximum allowed memory bandwidth of the other cores, while granting unlimited memory accesses to the requesting task. In this way, the program can avoid excess memory bandwidth contention, which could delay the execution of memory-performance critical sections.

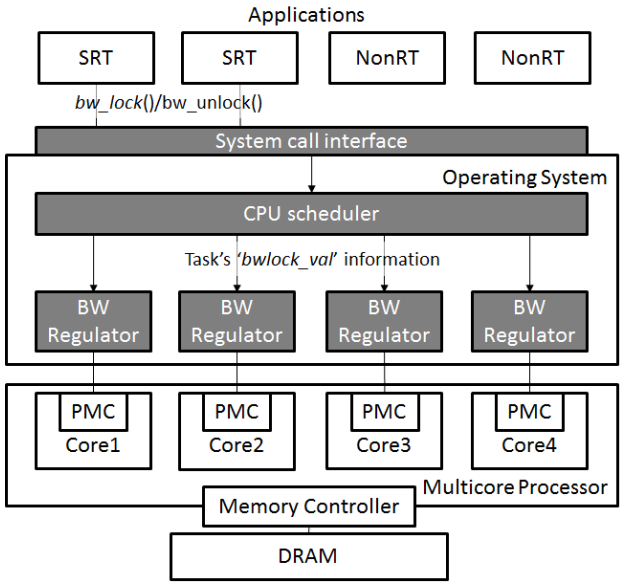


Fig. 3: Overall system architecture of BWLOCK.

API	Description
bw_lock()	begin a memory-performance critical section
bw_unlock()	end a memory-performance critical section

TABLE 1: BWLOCK user-level API

3.1 System Architecture

Figure 3 shows the overall architecture of the proposed system. At the user-level, it provides two functions—`bw_lock()` and `bw_unlock()`—to protect memory-performance critical sections. When `bw_(un)lock()` is called, the kernel updates the calling process's internal state so that whenever the CPU scheduler schedules the task, the kernel can determine whether the task is executing a memory critical section or not. Instead of calling the bwlock functions internally, by modifying the code, a process's bandwidth can be requested externally by another process (e.g., a utility program) via a system call. Inside the kernel, per-core bandwidth regulators are activated when there are cores executing memory-performance critical sections. In our current implementation, the check is periodically (e.g., every 1ms) performed by a timer interrupt handler. Ideally, hardware assisted mechanisms could support more fine-grained memory access control (see Section 6 for discussions on potential hardware support.)

3.2 Design and Implementation

BWLOCK supports fine-grained and coarse-grained bandwidth locking. In fine-grained locking, programmers are

required to use the API in Table 1 to declare memory-performance critical sections. It allows fine-grain control over memory performance but requires detailed profiling information to be effective. Often, such profiling information can easily be obtained using publicly available tools such as *perf* in Linux, as we will show in our case studies in Section 5. The coarse-grained locking is an equivalent of calling `bw_lock` once, by the program itself or by an external utility, and never releasing it. Then, whenever the process is scheduled, it automatically holds the bandwidth lock. We provide an external utility to set the bandwidth lock of any existing process in the system. Therefore, BWLOCK can be applied to any unmodified program, albeit the granularity of control is the entire duration the program's execution. It is important to note that unlike traditional locks, in which only one task can acquire a lock at a given time, a bandwidth lock can be acquired by multiple tasks on different cores. In other words, if there are multiple soft real-time applications requesting a bandwidth lock, all of them will be granted to access to the bandwidth lock. This design is due to the fact that the primary goal of BWLOCK is to protect soft real-time applications from memory intensive non-real-time applications.

In a sense, our design is similar to a two-level priority system in which the lock is used to determine which priority value should be given in accessing the memory. If a task holds a lock, it indicates that the task has a high memory access priority; otherwise, the task has a low memory access priority. If a task's memory access priority is low, the task's maximum memory bandwidth will be limited as long as there is at least one high-memory priority task. The design can be further extended to support multiple levels of memory priorities where each task's maximum memory bandwidth is regulated based on the priority value associated with the bandwidth lock.

Figure 4 shows the kernel-level implementation of BWLOCK. We added an integer value `bwlock_val` to indicate the status (i.e., memory priority) of BWLOCK in the process control block of Linux (`task_struct`). The value can be updated via a system call (see `syscall_bwlock()`). Because it simply updates an integer value (Line 10-17), its calling overhead is very small (overhead analysis is given in Section 4.2).

Each core's bandwidth regulator (see `per_core_period_handler()`) periodically checks how many cores are executing memory-performance critical sections (Line 28). If one or more cores are executing memory-performance critical sections, only the cores that hold the bandwidth lock can access memory freely (Line 30-31, `maxperf_budget` is an infinite value) while the others are regulated (Line 32-33) with the memory bandwidth budget determined by `minperf_budget`. Note that the `minperf_budget` is a system parameter that indicates the maximum amount of memory traffic that can co-exist without significant memory performance interference. Because an appropriate value may vary depending on the platform's architectural characteristics (core architecture, number of DRAM banks, etc.), it should be determined experimentally on each platform. In our test platform, we set the value as 100MB/s, based on the experimental results described in Section 2 (see Figure 2).

```
1  // task structure
2  struct task_struct {
3    ...
4    int bwlock_val; // 1 - locked, 0 - unlocked
5    ...
6  };
7
8  // bwlock system call
9  syscall_bwlock(pid_t pid, int val)
10 {
11   struct task_struct *p;
12   if (pid == 0)
13     p = current; // 'current' <- calling task
14   else
15     p = find_process_by_pid(pid);
16   p->bwlock_val = val;
17   return 0;
18 }
19
20 // periodic handler called by the
21 // bandwidth regulators
22 void per_core_period_handler()
23 {
24   // re-activate the suspended core
25   if (current == kthrottle)
26     deschedule(kthrottle);
27
28   if (nr_bwlocked_cores() > 0) {
29     // one or more cores requested bwlock
30     if (current->bwlock_val > 0)
31       budget = maxperf_budget;
32     else
33       budget = minperf_budget;
34   } else {
35     // no cores requested bwlock
36     budget = maxperf_budget;
37   }
38
39   // program the core's performance counter
40   // to overflow at 'budget' memory accesses
41 }
42
43 // PMC overflow handler
44 void  per_core_overflow_handler()
45 {
46   // stall the core till the next period
47   // kthrottle <- high priority idle thread
48   schedule(kthrottle);
49 }
```

Fig. 4: BWLOCK kernel implementation

If a core exhausts its bandwidth budget within a regulation period, the core's Performance Monitoring Counter (PMC) generates an overflow interrupt (Line 44-49). Then the core will be immediately throttled by scheduling a high priority idle thread (`kthrottle`) [2]. The throttled core will be re-activated at the beginning of the next period interrupt handler (Line 25-26).

4 Evaluation Setup

In this section, we present details on the hardware platform and the BWLOCK software implementation. We also provide detailed overhead analysis and discuss performance trade-offs.

2. It is a simple busy-waiting loop designed to prevent any further memory accesses.

Period (us)	Overhead (%)
100	3.5
250	1.5
500	0.9
1000	0.7
2500	0.5

TABLE 2: Period interrupts handling overhead

4.1 Hardware Platform

We use a quad-core Intel Xeon W3530 based desktop computer as our testbed. The processor has private 32K-I/32K-D L1 caches and a private 256 KiB L2 cache for each core and a shared 8MiB L3 cache. The memory controller (MC) is integrated in the processor and connected to a 4GiB 1066 MHz DDR3 memory module. We disabled turbo-boost, dynamic power management, and hardware prefetchers for better performance predictability and repeatability.

4.2 Implementation Overhead Analysis

We implemented BWLOCK in Linux version 3.6.0. There are two major sources of overhead in BWLOCK: system call and interrupt handling. First, in the fine-grained setting, two system calls are required for each memory-performance critical section. In our current implementation, a single system call is used to implement both `bw_lock()` and `bw_unlock()`. The system-call overhead is small—the average value out of 10,000 iterations is 125.24ns—as it simply changes a single integer value in the task's `task_struct`.

Second, as shown in Figure 4, a timer interrupt handler is used to periodically reset each core's budget. The actual access control is performed by a performance counter overflow interrupt handler. Unlike the overflow handler, which is not in the critical path of normal program execution, the execution time of the periodic timer interrupt is pure overhead which is directly added to the task's execution time, just like the OS tick timer handler in the OS. We quantified the period interrupt handling overhead by measuring the execution time increase of the Latency benchmark, compared to its execution time without using BWLOCK. Table 2 shows the measured overhead (percentage of the increased execution time) under different period lengths. Based on this result, we use 1ms period, unless noted otherwise.

5 Evaluation Results

In this section, we present case-study results using two real-world soft real-time applications—MPlayer (a video player) and WebRTC [9] (a multimedia real-time communication framework for browser based web applications)—to evaluate the effectiveness of BWLOCK.

5.1 MPlayer

MPlayer is a widely used open-source video player. In the following set of experiments, our goal is to protect real-time performance of the MPlayer(s) in the presence of memory intensive co-running applications while still maximizing overall throughput of the co-runners.

In the first set of experiments, one MPlayer instance plays a H264 movie clip with a frame resolution of 1920×816 and a frame rate of 24fps. We modified the source code of MPlayer slightly to get the per-frame processing time and other statistics. Decoded video frames are displayed on screen via a standard X11 server process. Therefore, the MPlayer and the X11 processes have soft real-time characteristics.

LLC misses	Cycles	Function
51.6%	27.8%	yuv420_rgb32_MMX
18.8%	9.3%	prefetch_mmx2
4.5%	7.3%	hl_decode_mb_simple_8

TABLE 3: Profiled information of MPlayer

LLC misses	Cycles	Function
53.29%	32.85%	sse2_blt
24.13%	24.19%	fbBlt
14.10%	19.61%	sse2_composite_over_8888_88888

TABLE 4: Profiled information of X11

Average duration	99 pct. duration	Function	Application
2.9ms	4.2ms	yuv420_rgb32_MMX	MPlayer
1.1ms	2.9ms	sse2_blt	X11

TABLE 5: Timing statistics of memory intensive functions

5.1.1 Profiling

To understand their memory-performance characteristics, we collect function level profiling information—cache-misses and cycles of each function—with the *perf* tool, which uses hardware performance counters. The profiled information of MPlayer and X11 is shown in Table 3 and 4, respectively. In both cases, the functions that generate most of the memory traffic are clearly identified: `yuv420_rgb32_MMX` in MPlayer and `sse2_blt` in X11. Note that both functions are responsible for more than 50% of total LLC-misses, while using relatively small CPU cycles (27.8% and 32.85% respectively). Therefore, they are prime candidates for applying BWLOCK. Due to the restrictions of our current software based implementation, it is also important to know the duration of each function: if it is too short, BWLOCK may not be able to regulate co-runners' memory accesses when needed. Table 5 shows the average and 99 percentile execution times of the functions. Fortunately, both functions are long enough to be regulated by the bandwidth control mechanism of BWLOCK.

5.1.2 Performance comparison

To investigate the effectiveness of BWLOCK, we conducted a set of experiments. We first ran the MPlayer alone (with the X-server) to get the baseline performance. In order to generate memory interference, we used two instances of a memory intensive synthetic benchmark [28], referred as Bandwidth(wr) [3]. We also measured their baseline performance in isolation. We then co-scheduled all four processes—MPlayer,

3. The benchmark modifies a huge one-dimensional array in a loop.

```
1  static inline int yuv420_rgb32_MMX
2  (SwsContext *c, const uint8_t *src[],
3  ..
4  {
5    bw_lock(); // added
6
7    YUV2RGB_LOOP(4)
8    ...
9    bw_unlock(); // added
10 }
```

Fig. 5: Code modification example for fine-grained application of BWLOCK.

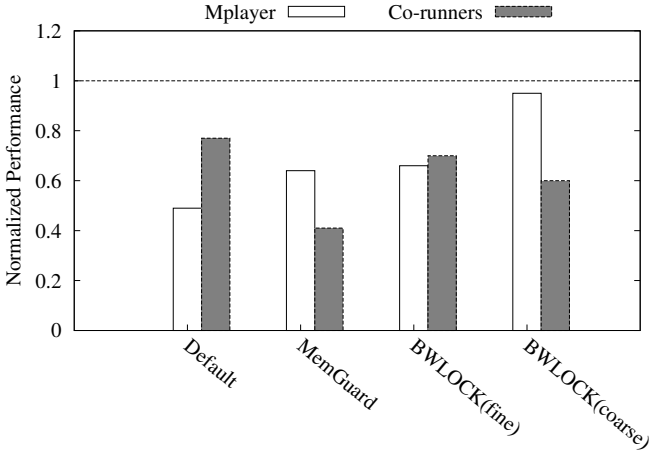


Fig. 6: Normalized performance of MPlayer (average frame time) and co-running Bandwidth(wr) benchmarks (MB/s): 1×MPlayer and 2×Bandwidth instances.

X11, and two Bandwidth(wr) instances—at the same time in four different configurations. For convenience of monitoring and measurement, each process was assigned to a dedicated core using the CPU affinity facility in Linux. Note that all four processes are single-threaded. In *Default*, we used a standard vanilla Linux kernel. In *MemGuard*, we used MemGuard [28]; the memory bandwidth budgets were statically configured as 500, 500, 100, and 100MB/s for Core 0 to 3, respectively[4], and predictive bandwidth re-distribution was enabled to minimize unused bandwidth [5]. In *BWLOCK(fine)*, we manually inserted `bw_lock` and `bw_unlock` in the previously identified memory intensive functions of MPlayer and X11 (Table 5), as shown in Figure 5. Lastly, in *BWLOCK(coarse)*, both MPlayer and X11 were *not* modified but configured to automatically hold the bandwidth lock whenever they were scheduled.

For MPlayer, performance is measured by the reciprocal of the average frame processing time. For co-running Bandwidth(wr) benchmarks, performance is measured by their aggregated throughput (MB/s).

Figure 6 shows the results. In the figure, performance

4. Out of the reservable bandwidth of 1200MB/s [28], we allocate 100MB/s to each co-runner to minimize interference (see Figure 2), and evenly split the remaining bandwidth of 1000MB/s to the MPlayer and the Xserver.

5. The re-distribution algorithm leverages a history-based bandwidth usage prediction; for more details, please refer [28].

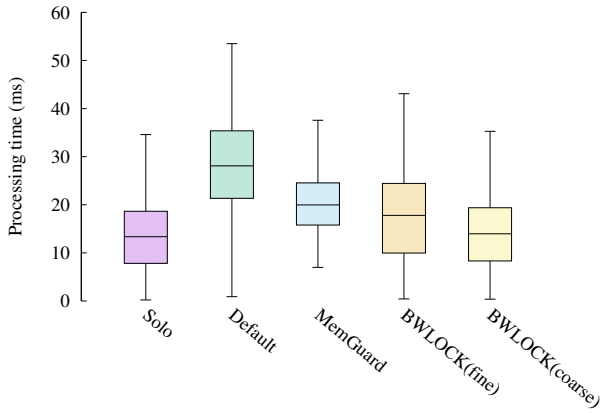
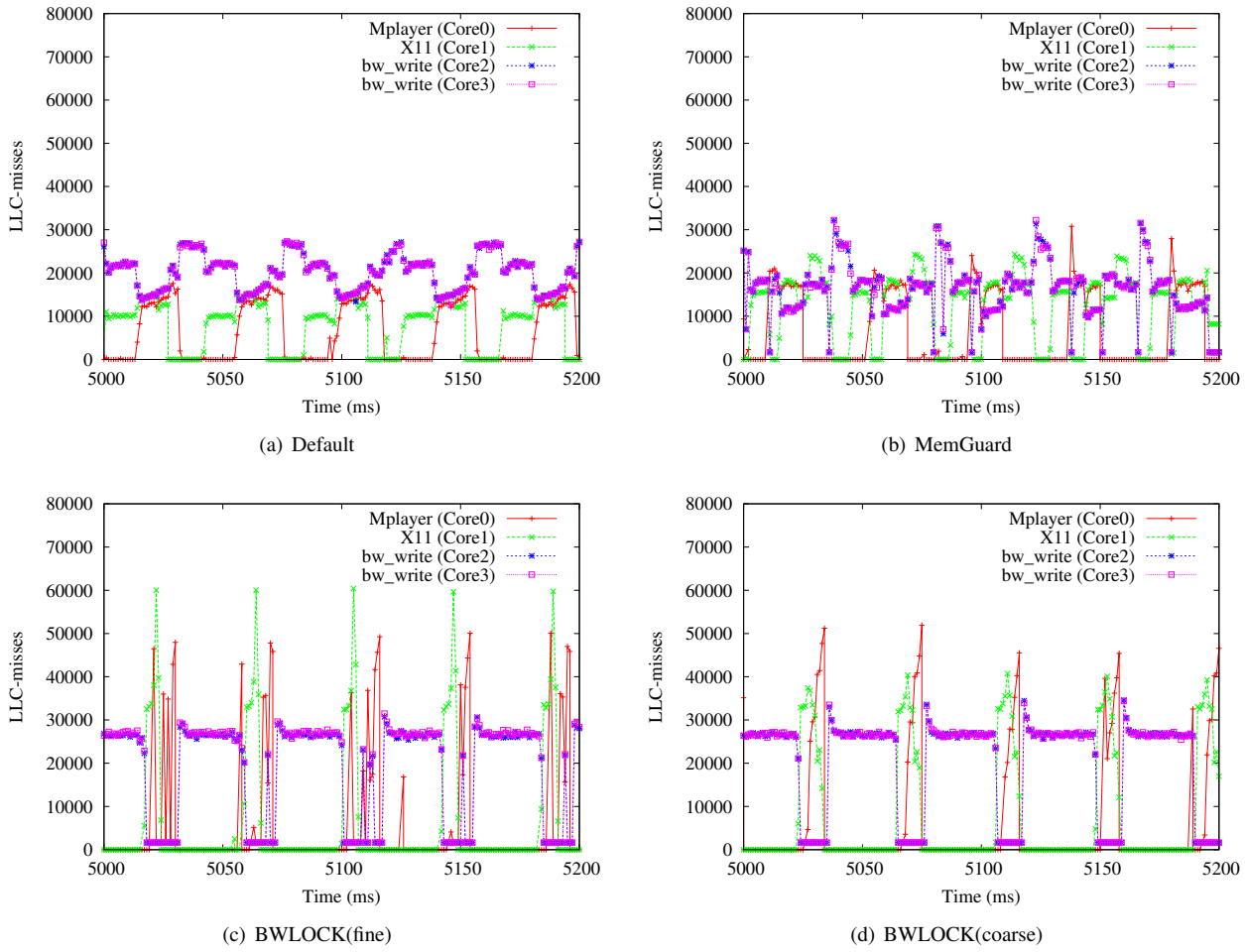


Fig. 7: MPlayer's frame processing time distribution.

is normalized to each application's baseline performance measured in isolation. In Default, MPlayer's performance is significantly degraded—dropped by 51%—due to memory contention, while the co-running Bandwidth(wr) instances are not affected as much—dropping by 22%. This kind of disproportional performance impact is common in COTS multicore systems and is caused by a combination of application memory characteristics and the DRAM controller's scheduling policy [22], [16]. In MemGuard, MPlayer's performance is better protected—dropped by 36%—as a fraction of memory bandwidth is reserved for it. However, this comes at a cost of considerable performance reduction of the co-runners. In BWLOCK(fine), on the other hand, MPlayer's performance is similarly protected with much less performance degradation to the co-runners. This is because co-runners are regulated only when MPlayer executes its memory-performance critical sections, which are short and bursty, and are explicitly informed to the kernel by calling the bandwidth lock. Lastly, in BWLOCK(coarse), MPlayer achieves near perfect performance isolation. This is because whenever it is scheduled, the kernel scheduler automatically calls the bandwidth lock. However, because the entire duration of MPlayer's processing is protected by the bandwidth lock, the performance of co-runners is further degraded.

Figure 7 shows the distribution of frame processing times in each configuration. First, note that each frame in the video inherently takes a different amount of decoding time, which is reflected in the variation shown in Solo. Second, as expected, the variation and the average of decoding times are significantly increased in Default—due to the memory interference caused by co-running Bandwidth(wr) benchmarks—while they are generally much smaller in MemGuard, BWLOCK(fine), and BWLOCK(coarse). In particular, the results of BWLOCK(coarse) are very similar to the ideal performance observed in Solo.

Figure 8 shows the memory access pattern of each core, each of which executes a different task. The y-axis shows the number of memory accesses of each core for every one millisecond period. Note that Core2 and Core3, which execute the memory intensive Bandwidth(wr) benchmarks, have a very



(a) Default (b) MemGuard

(c) BWLOCK(fine) (d) BWLOCK(coarse)

Fig. 8: Per-core memory access patterns.

high constant memory demand when they run in isolation. In Default, whenever MPlayer (red line) and/or X11 (green line) demand more memory bandwidth (increase in the LLC misses), they suffer considerable delays due to the bandwidth contention (compare, for example, the height and the duration of the red and green lines at around 5050 ms with that of BWLOCK(coarse)). In MemGuard, we can observe that both MPlayer and X11 are getting more bandwidth (both red and green lines are taller than those of Default), although they still suffer significant memory interference (pink and blue lines are still quite high). In BWLOCK(fine) and BWLOCK(coarse), on the other hand, we can observe that the co-runners are immediately regulated (again observe the pink and blue lines) upon arrivals of memory bandwidth demand spikes of MPlayer and X11. This results in improved performance.

5.1.3 Overloaded System

So far, we have assigned one task per core and both MPlayer and X11 do not consume 100% of CPU cycles on the assigned core. In other words, the system is under-utilized. In order to investigate how BWLOCK performs in an over-loaded situation, we performed another set of experiments in which each core executes one MPlayer instance and one Bandwidth(wr) instance to fully load the system. Performance metrics are the same: the average frame processing time of all MPlayer instances and the aggregate bandwidth of the Bandwidth(wr) instances. Figure 9 shows the results. Notice that, in this experiment setup, all cores run both real-time and non-real-time tasks. Therefore, MemGuard's core-based bandwidth partitioning, which prioritizes certain cores over the others, is not appropriate. Hence, we only compare the results of Default and the two BWLOCK settings (fine and coarse). As shown in the figure, both BWLOCK settings provide good performance isolation for the MPlayer instances at the cost of degraded performance for the Bandwidth(wr) co-runners, which do not request bandwidth locks. Note that our current BWLOCK implementation does not limit the number of tasks that can hold bandwidth locks at a given time. Therefore, memory contention among the MPlayer instances, which hold bandwidth locks on different cores, could potentially delay with each other. The performance reduction of MPlayer in BWLOCK(coarse) is not from the contention from the Bandwidth(wr) instances, but is entirely from the co-running MPlayer instances; we verified this by comparing it with the result obtained by running only four MPlayer instances without any co-runners.

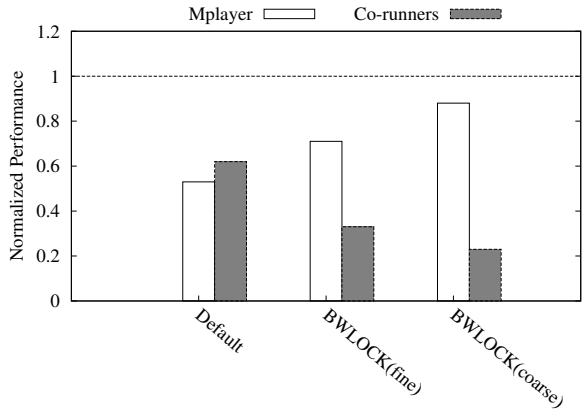


Fig. 9: Normalized performance of MPlayer (average frame time) and co-running Bandwidth benchmarks (MB/s): 4×MPlayer and 4×Bandwidth instances.

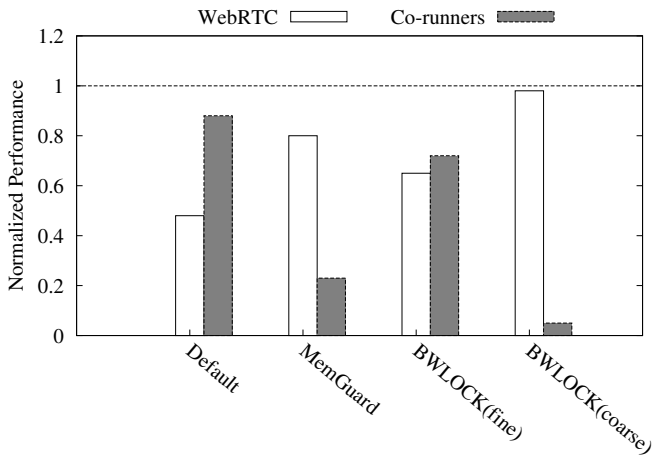


Fig. 10: Normalized performance of WebRTC (average bandwidth) and co-running LBM benchmarks (MB/s)

5.2 WebRTC

In this subsection, we further evaluate BWLOCK using the WebRTC framework[9], which is an open source, plug-in free, RTC (real-time communication) platform for enabling audiovisual applications in a web browser.

The goal of this experiment is to provide real-time performance isolation to WebRTC sessions, in the presence of memory intensive co-running applications. We also investigate the side effects of different isolation mechanisms on the performance of co-runners. The basic setup is as follows: Two hosts, each of which executes a chrome browser and initiates a peer-to-peer WebRTC session, are directly connected to a dedicated Gigabit Ethernet switch to minimize network jitters. A separate host on the local LAN acts as WebRTC server to facilitate the initial handshake between the two hosts. Hence, the performance variability observed is almost entirely due to resource contention in the host itself. One host executes a WebRTC instance (chrome browser), and two LBM benchmark (part of SPEC2006) instances, which generates high memory traffic. And we measure the performance of WebRTC and the LBM instances.

WebRTC utilizes a GCC (Google Congestion Control) algorithm to derive target bit-rate of audiovisual streams based on the resource contention in the network, and the end hosts [5]. The frame rate and sending bandwidth are adjusted to match the available resources at any given time. The default video resolution of 640×480, and the frame rate of 30 FPS are used for experimentation, while the threshold bandwidth is increased to 4 Mb/s from default 2 Mb/s. All other system parameters in the experiment setup are identical to the previous section, unless noted otherwise.

5.2.1 Profiling

Similar to MPlayer, we collected function level profiling information for WebRTC using the *perf* tool to understand its memory access pattern. The result shows that `sk_memset32_SSE2` and `S32A_Opaque_BlitRow32_SSE2` from the Skia library (a graphic library) causes more than 50% (29.29% and 22.95% respectively) of cache misses of WebRTC. The mean execution length of each of the functions is around 7 us and more than 99% of the sample values are less than 100 us. Note that this is much smaller than the memory intensive functions we observed in MPlayer and Xserver. Because the period length of BWLOCK is 1 ms, we initially thought that using `bw_lock/unlock()`—in BWLOCK(fine) setting—to those functions may not provide performance benefits, especially considering the overhead of frequent system calls.

5.2.2 Performance

Figure 10 shows the normalized performance of WebRTC and co-running LBM(s) in four different configurations. Similar to the previous experiment, WebRTC experiences disproportionally high performance degradation due to the co-runners. In MemGuard, WebRTC achieves much improved performance but at the cost of significantly reduced throughput of the co-runners. In BWLOCK(fine), despite our initial concerns, the performance of WebRTC improves considerably without much penalty to the co-running tasks. Initially, this was surprising because the lengths of the memory-performance critical sections are much smaller than the period length of BWLOCK. Further investigation reveals that it is due to the fact that the short memory-intensive functions we instrumented are called in a batch, effectively coalescing a single large memory-intensive function. Hence, we get similar performance benefits as in the previous section. Lastly, in BWLOCK(coarse), WebRTC achieves near perfect performance isolation, but at the cost of heavy performance reduction of the co-runners. The WebRTC process consists of ~20 threads, out of which a couple of threads are involved in decoding and encoding of video. These threads consume a significant percentage of CPU cycles. Therefore, in BWLOCK(coarse), co-runners are regulated most of the time.

Table 6 shows round-trip delay (RTT) and frame rate (FPS) of WebRTC, which are collected using its own built-in performance monitoring tool. Compared to Default, which

shows substantially worse performance in both RTT and FPS, all isolation configurations show markedly better performance in both metrics.

Config.	RTT (ms)	FPS (frames/sec)
Solo	1.85	30.00
Default	17.20	21.34
MemGuard	2.24	29.98
BWLOCK(fine)	4.22	29.58
BWLOCK(coarse)	2.22	30.00

TABLE 6: WebRTC internal performance metrics

6 Discussion

In this section, we discuss limitations of our approach and future improvements.

6.1 Hardware Assisted Memory Bandwidth Control

A significant limitation of our current approach is our software-based periodic monitoring and bandwidth controlling mechanism in which the control granularity is limited to a millisecond range due to the interrupt handling overhead. This means that the detection and application of bandwidth lock can be delayed up to a period or even ignored completely if the lock duration is too short. While this may not be a serious issue in many soft real-time applications as we have shown in this paper, there may be other applications in which such delays are not tolerated. We believe this limitation can be addressed by implementing the regulation mechanism at the hardware level, which would allow fine-grained memory access control with little overhead. We left this as our future work.

6.2 Application to Hard Real-Time Systems

Although we focus on soft real-time applications in this paper, we believe BWLOCK can also be applied to hard real-time systems in some cases. For example, consider a case in which all hard-real-time tasks can be hosted in a single designated core. We can then apply BWLOCK (coarse grain mode) to all the hard real-time tasks to ensure that whenever any of them executes on the core, the other cores' maximum memory bandwidth usage would be bounded. Then, both hard- and non-real-time tasks can safely co-exist without needing to worry about excessive memory contention because the non-real-time tasks will be automatically throttled whenever any of the hard real-time tasks is activated. Note that non-real-time tasks can be hosted in any cores, including the designated core, as long as their priorities are lower than the hard-real-time ones.

6.3 Protection and Security

BWLOCK can be abused if granted to any process. For example, one might always want to hold the bandwidth lock regardless of its necessity to get the best possible performance at the cost of others who do not use the bandwidth lock. Hence, this ability to access the bandwidth lock must be limited to the users with root privileges; much like utilizing the real-time CPU schedulers requires a root privilege. In the case where BWLOCK is applied in a multi-tenant system, which uses a hypervisor, the mechanism must be implemented at the hypervisor level and only certain virtual machines, allowed by the hypervisor, should be allowed to request the bandwidth lock.

7 Related Work

OS level memory access control was first discussed in literature by Bellosa [3]. The basic idea is to reserve a fraction of memory bandwidth for each core [3], [27], [28] (or task [11]) by means of software mechanisms—e.g., TLB handler [3] or hardware performance counters [28], [11]. One problem of memory bandwidth reservation is that the reserved bandwidth can be wasted if it is not fully utilized by the reserved core. The work in [28] partly solves the problem by dynamically re-distributing the reserved bandwidth of under-utilized cores to the other more demanding cores. However, its effectiveness depends on the accuracy of future usage predictions, which can be challenging to achieve especially when the access pattern is bursty. In contrast, BWLOCK allows unrestricted memory accesses most of the time, hence, leveraging full benefits of memory-level parallelism available in modern multicore architectures, while limiting excessive concurrent memory accesses from non-real-time tasks only when soft-real-time tasks are executing memory-performance critical sections. We find that this selective, on-demand regulation approach in BWLOCK is more efficient the static reservation approaches[28], [11].

In the context of providing performance isolation in multicore systems, page coloring based cache-partitioning techniques have been extensively studied [19], [29], [6], [14], [21], [25], [26]. The basic idea is to allocate memory pages of certain physical addresses such that each core accesses a different part of the cache-sets. This way, a cache can be effectively partitioned without any special hardware support. However, the downside of this approach is that it is very costly to change the size of partitions at runtime. More recently, page coloring has been applied to partition DRAM banks [20], [24], [26]. These space-partitioning techniques can reduce the degree of interference experienced by concurrent tasks and are orthogonal to our approach that focuses on bandwidth control.

There have been many hardware proposals that allow collaborations between the system software (OS) and the hardware to make better resource scheduling/allocation decisions. For example, many DRAM controller design proposals utilize task-level priority information in scheduling memory requests [15], [16], [23], [12]. Such hardware support can be especially useful for BWLOCK because the software based periodic bandwidth control mechanism can be replaced by more efficient hardware mechanisms with lower overhead, which we plan to explore in the future.

8 Conclusion

We have presented BWLOCK, a user-assisted and kernel-enforced memory-performance control mechanism, designed to protect performance of soft real-time applications, such as multimedia applications. It provides simple a lock-like

API to declare memory-performance critical sections in the application code. When an application accesses a memory critical section, BWLOCK automatically regulates the other cores so that they cannot cause excessive memory interference.

We applied BWLOCK in two real-world soft real-time applications—MPlayer and WebRTC framework—to protect their real-time performance in the presence of memory-intensive non-real-time applications that share the same machine. In both cases, we were able to achieve near perfect real-time performance, or to choose not perfect—but still better than the vanilla Linux—real-time performance for minimal throughput reductions of non-real-time applications.

Our future work includes hardware assisted bandwidth control for better control quality and compiler-based automatic identification of memory-performance critical sections in soft real-time applications.

APPENDIX

The source code of BWLOCK is available at https://github.com/heechul/bwlock

ACKNOWLEDGEMENTS

This research is supported in part by NSF CNS 1302563. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Real-Time Systems Symposium (RTSS)*, pages 4–13. IEEE, 1998.

[2] Aeronautical Radio Inc. *Avionics Application Standard Software Interface (ARINC) 653*, 2013.

[3] F. Bellosa. Process cruise control: Throttling memory access in a soft real-time environment. Technical Report TR-I4-97-02, University of Erlangen, Germany, July 1997.

[4] Certification Authorities Software Team (CAST). Position Paper CAST-32: Multi-core Processors (Rev 0). Technical report, Federal Aviation Administration (FAA), May 2014.

[5] Luca De Cicco et al. Experimental investigation of the google congestion control for real-time flows. In *Proceedings of the 2013 ACM SIGCOMM workshop on Future human-centric multimedia networking*, pages 21–26. ACM, 2013.

[6] X. Ding, K. Wang, and X. Zhang. SRM-buffer: an OS buffer management technique to prevent last level cache from thrashing in multicores. In *European Conf. on Computer Systems (EuroSys)*. ACM, 2011.

[7] D. Faggioli, M. Trimarchi, F. Checconi, M. Bertogna, and A. Mancina. An implementation of the earliest deadline first algorithm in linux. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1984–1989. ACM, 2009.

[8] S. Fisher. Certifying Applications in a Multi-Core Environment: a New Approach Gains Success. Technical report, SYSGO AG., 2012.

[9] Google. WebRTC. https://http://www.webrtc.org/.

[10] J.L. Hennessy and D.A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2011.

[11] R. Inam, N. Mahmud, M. Behnam, T. Nolte, and M. Sjödin. The Multi-Resource Server for Predictable Execution on Multi-core Platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, April 2014.

[12] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. *ACM SIGMETRICS Performance Evaluation Review*, 35(1):25–36, 2007.

[13] S. Kato, R. Rajkumar, and Y. Ishikawa. Airs:supporting interactive real-time applications on multicore platforms. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2010.

[14] H. Kim, A. Kandhalu, and R. Rajkumar. A coordinated approach for practical os-level cache management in multi-core real-time systems. In *Real-Time Systems (ECRTS)*, pages 80–89. IEEE, 2013.

[15] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *High Performance Computer Architecture (HPCA)*, pages 1–12. IEEE, 2010.

[16] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 65–76. IEEE, 2010.

[17] O. Kotaba, J. Nowotsch, M. Paulitsch, S. Petters, and H Theilingx. Multicore in real-time systems temporal isolation challenges due to shared resources. In *Workshop on Industry-Driven Approaches for Cost-effective Certification of Safety-Critical, Mixed-Criticality Systems (at DATE Conf.)*, 2013.

[18] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Real Time Systems Symposium (RTSS)*, pages 166–171. IEEE, 1989.

[19] 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 *High Performance Computer Architecture (HPCA)*. IEEE, 2008.

[20] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *Parallel Architecture and Compilation Techniques (PACT)*, pages 367–376. ACM, 2012.

[21] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-Time Cache Management Framework for Multi-core Architectures. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2013.

[22] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, page 18. USENIX Association, 2007.

[23] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu. Mise: Providing performance predictability and improving fairness in shared main memory systems. In *High Performance Computer Architecture (HPCA2013)*, pages 639–650. IEEE, 2013.

[24] N. Suzuki, H. Kim, D. de Niz, B. Andersson, L. Wrage, M. Klein, and R. Rajkumar. Coordinated Bank and Cache Coloring for Temporal Protection of Memory Accesses. In *Computational Science and Engineering (CSE)*, pages 685–692. IEEE, 2013.

[25] Y. Ye, R. West, Z. Cheng, and Y. Li. COLORIS: a dynamic cache partitioning system using page coloring. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 381–392. ACM, 2014.

[26] H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni. PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[27] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory Access Control in Multiprocessor for Real-time Systems with Mixed Criticality. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

[28] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-core Platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

[29] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *European Conf. on Computer Systems (EuroSys)*, 2009.