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

Heechul Yun[†], Waqar Ali[†], Santosh Gondi[‡], Siddhartha Biswas[⋆]
[†] University of Kansas, USA. {heechul.yun, wali}@ku.edu
[‡] Bose Corporation, Framingham, MA, USA. santosh_gondi@bose.com
[⋆] Walmart, David Glass Technology Center, Bentonville, AR, USA. siddhartha.biswas@walmart.com

**Abstract**—Soft real-time applications often show bursty memory access patterns—requiring high memory bandwidth for a short duration of time—that are often critical for timely data processing and performance. We call the code sections that exhibit such characteristics as *Memory-Performance Critical Sections (MPCSs)*. Unfortunately, in multicore architectures, non-real-time applications on different cores may also demand high memory bandwidth at the same time. Resulting bandwidth contention can substantially increase the time spent on MPCSs of soft real-time applications, which in turn could result in missed deadlines.

In this paper, we present a memory access control framework called *BWLOCK*, which is designed to protect MPCSs of soft real-time applications. BWLOCK consists of a user-level libarary and a kernel-level memory bandwidth control mechanism. The user-level library provides a lock-like API to declare MPCSs for real-time applications. When a real-time application enters a MPCS, the kernel-level bandwidth control mechanism dynamically throttles memory bandwidth of the rest of the cores to protect the MPCS, until it is completed. We evaluate BWLOCK using CortexSuite benchmarks. By selectively applying BWLOCK, based on the memory intensity of the code blocks in each benchmark, we achieve significant performance improvements, up to 150% reduction in slowdown, at a controllable throughput impact to non real-time applications.

## 1 Introduction

In a multicore system, an application's performance can be significantly affected by co-running 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 [20] 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. 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 [9]. These approaches allow the system to be certified [5] based on the traditional unicore-based certification process [1]. Obviously, however, they do not fully leverage the benefit of using multicore.

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 (e.g., out-of-order cores, non-blocking caches, multi-bank DRAM, and etc.) This allows, to a certain degree, multiple operations to be processed simultaneously without much performance impact [14]. Therefore, it is desirable to develop a solution that can achieve better real-time performance while still exploiting the available parallelism of multicore architectures.

In this paper, we present BWLOCK, a user-assisted and kernel-enforced memory bandwidth control mechanism. It is designed to protect the performance of soft real-time applications from the interference of the other non real-time applications running on different cores. This is accomplished through a close collaboration between the real-time applications and the OS. Our *key observation* is that interference is most visible when multiple cores have high memory bandwidth demands at the same time. In such a case, all participating cores, including the cores that run real-time tasks, will be delayed in accessing memory 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 code sections. We call such a code section as a *Memory-Performance Critical Section (MPCS)*. Fortunately, MPCSs are often easy to identify in many soft real-time applications through application level profiling. Motivated by these observations, BWLOCK provides a lock like API with which programmers can describe certain code sections as MPCSs. When these code sections are executed, BWLOCK limits (throttles) the amount of allowed memory traffic from the rest of the cores, which execute non-real-time tasks, to avoid overloading the memory subsystem. If source-code modification and fine-grained profiling of an application are not desired (or impossible), then the entire execution of the application can be declared as memory performance critical. In that case, whenever the application is

scheduled, the kernel part of BWLOCK will be automatically activated to protect memory performance of the cores that execute the application.

We evaluate BWLOCK using CortexSuite [29], a collection of benchmarks representing various capabilities of human brains (vision, language, learning, and etc.). Our experiments are designed to protect real-time performance of the evaluated applications in the presence of co-running memory intensive non-real-time applications. With CortexSuite, we observe that the memory intensive co-runners can significantly degrade the performance, as we observe up to 150% slowdown. Using BWLOCK, we show that the execution time delays can be considerably decreased; we observe up to 5X slowdown reduction.

Our contributions are as follows:

- We propose a user-assisted and kernel-enforced memory-performance control mechanism to protect performance of soft real-time applications on commodity multicore systems.
- We present extensive evaluation results using realistic benchmarks, demonstrating the viability of the proposed approach. Also, we provide the source code and test scripts used in our evaluation as open-source [1].

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 evaluation results. Section 6 discusses limitations and possible improvements. We discuss related work in Section 7 and conclude in Section 8.

## 2 Background and Motivation

Soft real-time applications often show *bursty* memory access patterns, requiring high memory bandwidth for a short duration of time—for example, certain phases of video decoding process. At other times, however, their memory bandwidth demand may be low, for example, when they execute compute-intensive instructions or waiting for next periods. When a soft real-time application competes for memory bandwidth with other applications running on different cores, the short code sections that demand high memory bandwidth can suffer a disproportionally high degree of performance degradation. Although memory access latency of a single application often can be hidden due to a variety of latency hiding techniques and the abundant parallelism in modern multicore architectures [14], if the overall memory bandwidth demand reaches beyond a certain threshold, even the most advanced architecture would no longer be capable of hiding the latency. Thus, requests pile up in various queues in the system and all tasks that access the memory would be delayed.

Figure 1 illustrates this phenomenon. In this experiment, we measure the average memory access latency (normalized to run-alone latency) of a synthetic benchmark in the presence of memory intensive co-runners on the other three cores in a quad-core Intel Xeon system (detailed hardware setup is described in Section 4). All benchmarks are configured so that each memory access generates a DRAM transaction (an LLC miss). The Y-axis shows the normalized average memory access latency of the subject benchmark and the X-axis shows each co-runner's controlled memory bandwidth usage. Note that when the co-runner's bandwidth is low (100MB/s), the performance impact to the measured subject benchmark is negligible. However, as the co-runner's memory bandwidth demand increases (from 100 to 1000 MB/s), the observed memory access latency of the benchmark increases exponentially and then saturates. This shows that memory bandwidth contention can cause disproportionate memory access delay to applications.

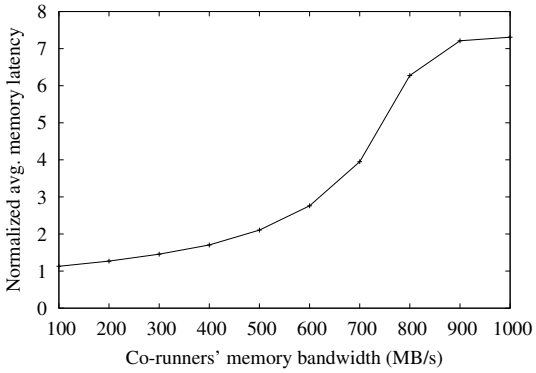



Fig. 1: Normalized average memory access latency of a *Latency* benchmark as a function of each co-runner's memory bandwidth demand (on three different cores.)

Such bandwidth contention is especially problematic for a soft real-time application executing memory-performance critical code sections (MPCSs). In the following, we describe a simple and explicit bandwidth control mechanism, called BWLOCK, to eliminate the memory bandwidth contention problem for soft real-time applications.

## 3 BWLOCK

The goal of BWLOCK is to selectively eliminate (reduce) memory bandwidth contention of soft real-time applications. It is application driven in the sense that the application explicitly specifies which parts of the program are memory performance critical sections (MPCSs), using a lock-like API, which we call a memory bandwidth lock. Bandwidth contention is eliminated by regulating (throttling) memory bandwidth of the rest of the cores that execute memory intensive non real-time (NRT) applications. At run-time, BWLOCK always satisfies the following two simple rules:

- **Rule 1:** All (soft) RT tasks[2] will not be throttled in any cases.
- **Rule 2:** All NRT tasks' maximum memory bandwidth will be limited to a certain threshold, if there is at least one RT task that holds the bandwidth lock.

In other words, NRT tasks will be throttled if they exceed a given bandwidth threshold while RT tasks execute MPCSs,

1. https://github.com/CSL-KU/bwlock

2. In this paper, we use the terms "core" and "task" interchangeably, unless noted otherwise, as only one task is executed on a certain core at a time.

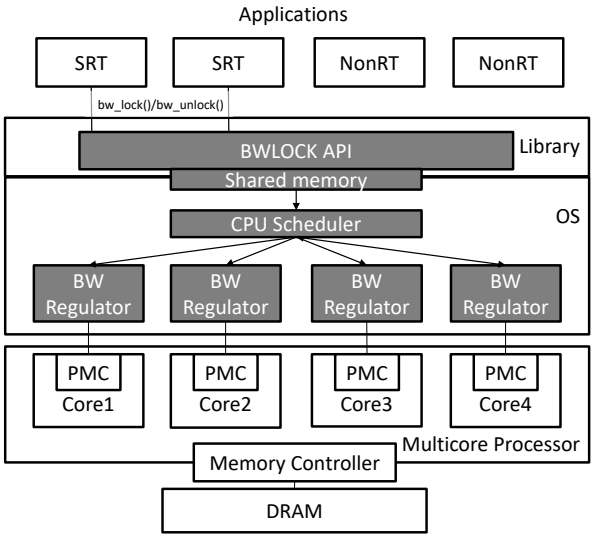



Fig. 2: Overall system architecture of BWLOCK.

to minimize memory bandwidth contention. But if no RT task executes a MPCS, both RT and NRT tasks will be executed without any bandwidth throttling.

### 3.1 Assumptions and Semantics

We assume that the taskset is composed of two tiers of tasks: real-time (RT) tasks and non real-time (NRT) tasks. We assume partitioned fixed-priority real-time scheduling for RT tasks. RT tasks are strictly prioritized over NRT tasks and we do not assume any particular scheduling model for NRT tasks. It is important to note that, in our model, RT tasks do not migrate while NRT tasks can.

The locking semantic of BWLOCK can be viewed as a $k$-exclusion lock where $k$ is equal to the number of cores, as it allows up to $k$ active lock holders at a time—one lock holding RT task per core. Unlike other k-exclusion lock designs [11], [4], [30], [8], however, in BWLOCK, when a lock holder task is preempted by another task on the same CPU core, the lock is automatically released so that the newly scheduled task on the core can subsequently acquire the lock to execute its MPCSs; when the preempted task is re-scheduled, it automatically re-acquires the lock again. This implicit lock acquisition/release by CPU scheduling is possible because memory bandwidth, which BWLOCK tries to protect, is a performance property and not a logical one.

### 3.2 System Architecture

Figure 2 shows the overall architecture of BWLOCK, which is composed of a user-level library and a kernel module. The user-level library provides an API for memory bandwidth control. Real-time tasks use the API to explicitly declare memory-performance critical sections (MPCSs). Additionally, any real-time task's entire execution can be declared as a single big MPCS via an external helper utility without needing to modify the application program code, although it would affect throughput of NRT tasks. The library shares memory pages (one per core) with the kernel for sharing information. A task's use of MPCS is updated directly in the shared memory pages to eliminate the need of system calls for communicating the information with the kernel. This makes it possible to use BWLOCK even in small sections of code which get invoked repeatedly without paying excessive system call overhead.

Inside the kernel, per-core bandwidth regulators check whether there are RT tasks executing MPCSs by looking at the shared memory pages of the cores. In our current implementation, the check is periodically (every 1ms) performed by a timer interrupt handler. If such RT tasks exist, the the regulators are automatically activated to protect memory performance of the RT tasks by throttling memory bandwidth usage of the rest of the cores that execute NRT tasks.

### 3.3 Design and Implementation

**Algorithm 1:** BWLOCK kernel implementation

```
1  struct task_struct {
2      ...
3      int bwlock_val;
4      ...
5  }
6  procedure nr_bwlocked_cores()
7      int locked_cores := 0 ;
8      for_each_core (c)
9          if cpu_rq(c)→bwlock_val then
10             locked_cores++;
11         end
12     return (locked_cores)
13 procedure per_core_period_handler()
14     int new_budget := max_perf_budget;
15     // current ← this core's running task pointer
16     if (current == kthrottle) then
17         deschedule(kthrottle);
18     end
19     if is_bwlock_requested_by(current) then
20         current→bwlock_val = 1
21     else
22         current→bwlock_val = 0
23     end
24     if nr_bwlocked_cores() > 0 then
25         if !rt_task(current) then
26             new_budget = min_perf_budget;
27         end
28     end
29     program_pmc(new_budget);
30     return
31 procedure pmc_overflow_interrupt()
32     // kthrottle ← per-core high priority RT idle thread
       schedule(kthrottle);
33     return
```

Algorithm 1 shows part of implementation of the BWLOCK kernel module. Up on initialization, the kernel module creates a shared memory page for each processing core. These pages

serve as tables for allocating entries to user-space tasks which need to use bandwidth lock. When a (RT) task initializes the user-level library, it maps the shared memory page of the core on which the task is executing to the task's memory space. As noted earlier, each RT task is pinned to a particular core on initialization (i.e., no run-time migration.) Once the initialization is completed, the task can request or release the bandwidth lock using the following API: `bw_lock()` and `bw_unlock()`, for lock acquisition and release, respectively.

Each core's bandwidth regulator periodically checks the core's bandwidth lock memory page to determine the tasks that request/release the bandwidth lock. The regulator updates the tasks' task control blocks (TCBs) to indicate whether the memory bandwidth lock is requested or not (Line 19-23). We added an integer variable `bwlock_val` in Linux's TCB (`task_struct`) for storing the information (Line 3).

Then, the regulator determines how many cores are executing MPCSs. If one or more cores are executing MPCSs, then memory bandwidth of the cores that currently execute NRT tasks will be throttled—limited by `minperf_budget`, which denotes the minimum bandwidth budget (Line 24-28). Note that, in our current implementation, `minperf_budget` is a system parameter, which indicates the maximum amount of per-core memory traffic that can co-exist without significant memory bandwidth contention. Because an appropriate value may vary depending on the platform's architectural characteristics (out-of-order pipeline, number of DRAM banks, etc.) as well as applications' memory access characteristics, it should be carefully determined. In our test platform, we set the value as 100MB/s, based on our empirical observations in Section 2 (see Figure 1). This is a conservative (safe) estimation as we used a memory bandwidth sensitive synthetic benchmark in the experiments. On the other hand, `maxperf_budget` denotes the maximum budget of the core, which is, in our current implementation, effectively an infinite number so that the core will not be throttled.

If a core exhausts its bandwidth budget within a regulation period, the core's Performance Monitoring Counter (PMC) generates an overflow interrupt (Line 31-33). Then the core will be immediately throttled by scheduling a high priority idle thread (`kthrottle`) [3]. The throttled core is re-activated at the beginning of the next period (Line 9-11).

As noted earlier, unlike traditional locks, in which only one task can acquire a lock at a given time, the bandwidth lock can be acquired by multiple tasks on different cores—at most one task per core at a time. In other words, if there are multiple soft real-time tasks requesting the bandwidth lock, all of them can be granted access to the bandwidth lock. This design is due to the fact that the primary goal of BWLOCK is to protect soft real-time tasks from memory intensive non-real-time tasks.

# 4 Evaluation Setup

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

## 4.1 Hardware Platform

We use a quad-core Intel Xeon W3530 (Nehalem architecture) based desktop computer as our test-bed. Each core has private 32K-I/D L1 caches and a private 256 KiB L2 cache and all four cores share a 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 Software Implementation

We use a standard Linux 4.0 kernel for evaluation. The kernel part of BWLOCK is implemented as a separate kernel module and loaded (and unloaded) dynamically to conduct experiments. Once the kernel module is loaded, it exposes a device driver interface. The user-level library uses the device interface to directly share memory pages with the kernel module, which eliminates the need of system calls for acquiring and releasing the bandwidth lock.

## 4.3 Implementation Overhead Analysis

A major source of overhead of BWLOCK is the periodic timer interrupt handler, shown in Algorithm 1. 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. We quantified the period interrupt handling overhead by measuring the execution time increase of a benchmark, compared to its execution time without using BWLOCK; we found the overhead is less than 1% execution time increase with the regulation period of 1 ms (longer periods result in even less overheads). In addition, the performance counter overflow interrupt handler, which actually performs access control, is not in the critical path of normal program execution and does not occur at all if the memory budget of the core, which executes a given task, is sufficient. Therefore, we believe the overhead of BWLOCK is acceptable for soft real-time systems.

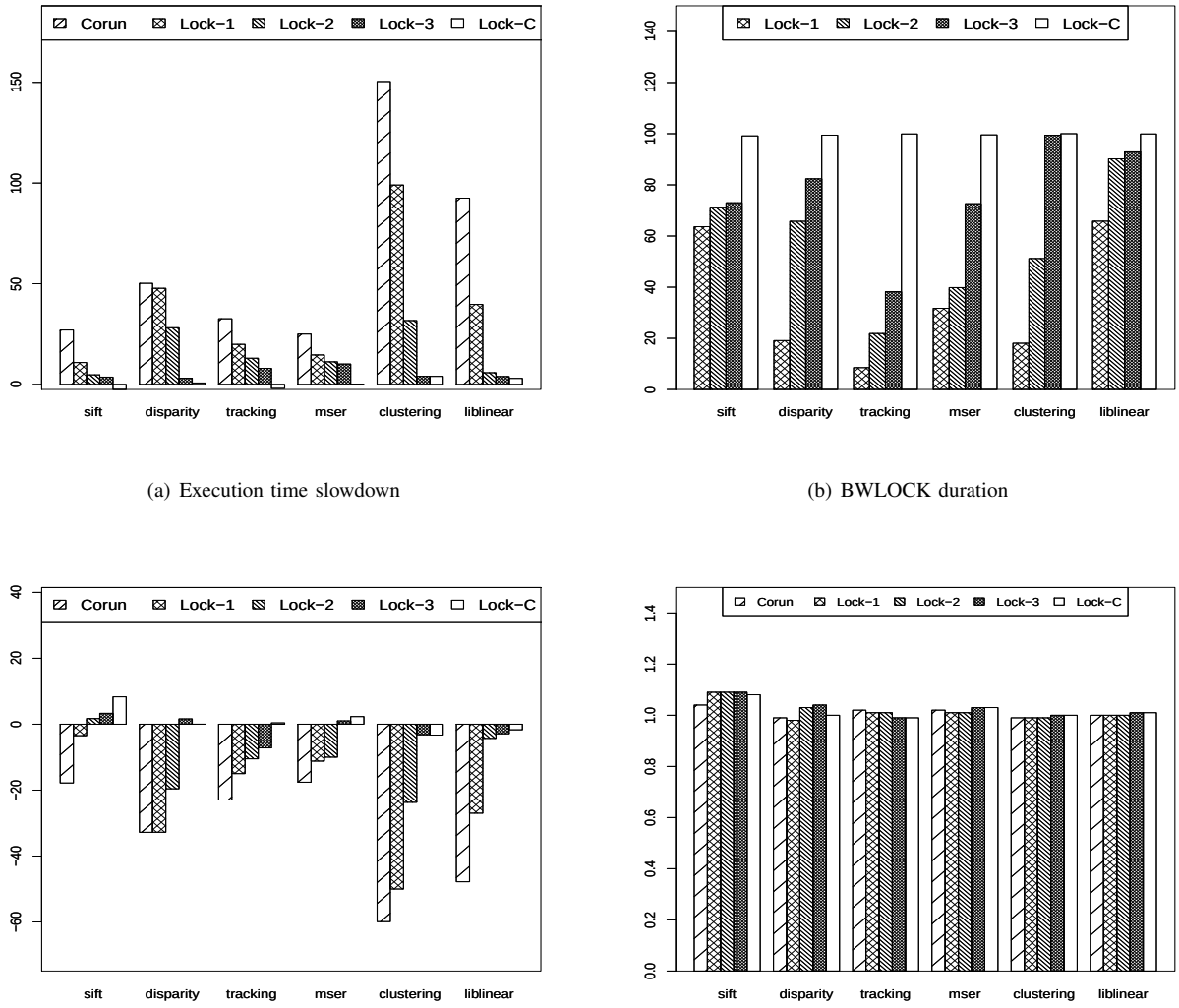
# 5 Evalution Results

In this section, we present our evaluation results of BWLOCK using the CortexSuite [29], a synthetic brain benchmark suite that includes popular vision and machine-learning algorithms. We choose six benchmarks from the CortexSuite that require

| Configuration | Description |
|---|---|
| Lock-1 | Apply BWLOCK to *one* most memory intensive function of the subject benchmark. |
| Lock-2 | Apply BWLOCK to *two* most memory intensive functions of the subject benchmark. |
| Lock-3 | Apply BWLOCK to *three* most memory intensive functions of the subject benchmark |
| Lock-C | Apply BWLOCK to the *entire* program execution of the subject benchmark |

TABLE 1: BWLOCK configurations used in evaluation.

3. It is currently a simple busy-waiting loop that is designed to prevent any further memory accesses. Precise sleep using a high-resolution timer can be used instead to reduce energy consumption, although energy saving is not our primary concern in this paper.




(a) Execution time slowdown

(b) BWLOCK duration

(c) Memory bandwidth reduction

(d) Cache (LLC) miss-rate

Fig. 3: Effect of using BWLOCK on CortexSuite. All sub-figures are normalized to the baseline solo execution.

high memory bandwidth. Using the benchmarks, we conduct a set of experiments to evaluate the effectiveness of BWLOCK in the presence of memory intensive co-runners.

To isolate the effects of cache-space contention from memory bandwidth contention, in this experiment, we partition the last level cache (LLC) into four equal sets using PALLOC [32] [4]. We assign three cache-sets (3/4 of the LLC) to the core that runs the subject benchmark and assign one cache-set (1/4 of the LLC) to the remaining three cores. We profile the solo execution of the subject benchmarks, using the *perf* tool in Linux, to identify their memory performance critical sections (MPCSs) at the granularity of functions. We rank all functions of each benchmark according to their memory bandwidth usage (i.e., the percentage of LLC misses of the function over all LLC misses). We then choose top three functions from each benchmark as MPCSs. For each benchmark, we apply the memory bandwidth lock to an increasing number of the MPCSs, from one to three functions, denoted as Lock-1, Lock-2, and Lock-3, respectively, in Table 1. In addition, we also evaluate the course-grained memory bandwidth locking, denoted as Lock-C in Table 1, which protects the entire duration of the benchmark execution without any application code changes. In each BWLOCK configuration, we run the subject benchmark with three instances of a memory intensive benchmark, Bandwidth(wr) to evaluate the effectiveness of BWLOCK.

Figure 3(a) shows the percentage of performance slowdown. Note first that when BWLOCK is not used—Corun—performance of each benchmark is substantially delayed, up to 150% in *clustering*, due to memory bandwidth contention from the memory intensive co-runners. However, as we apply

4. PALLOC is a kernel level memory allocator which can be used to partition the last level cache and DRAM banks. In our evaluation, we use PALLOC for partitioning the LLC.

more memory bandwidth locks—from Lock-1 (1 function is protected using BWLOCK) to Lock-C (entire program is protected using BWLOCK)—performance improves accordingly. The performance improvement is most pronounced in memory intensive benchmarks like *clustering* and *liblinear*, which suffer 150% and 90% slowdown, respectively, in Corun but suffer virtually no slowdown in Lock-C. Furthermore, incrementally increasing the aggressiveness (i.e. number of locked functions) results in incremental performance improvements. For example, in the *clustering* benchmark, locking only two functions reduces the slowdown from 150% to 32%.

Of course, the improvement of the subject benchmark comes at the cost of co-runners. The more an application uses BWLOCK, the more the co-runners, if they are memory intensive, will suffer as their memory bandwidth will be throttled. Figure 3(b) shows the percentage of time during which each co-runner's memory bandwidth is throttled. As can be seen in the figure, as the lock level increases, the throttled time also increases, which would result in lower performance of the co-runners.

Note that, because the shared LLC is partitioned between the subject and the co-runners, performance of the subject benchmarks is mainly affected by memory bandwidth. Figure 3(c) shows the achieved memory bandwidth differences, compared to solo execution, in each BWLOCK configuration that closely track the performance slowdowns shown in Figure 3(a). In addition, Figure 3(d) shows the measured LLC miss-rates of the subject benchmarks, normalized to solo execution, showing no significant variations due to cache partitioning. In other words, bandwidth reductions are the chief cause of performance slowdowns in our experiments. Therefore, reducing bandwidth contention by using BWLOCK improves application performance.

## 6 Discussion

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

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 adopting hardware-based bandwidth regulation mechanisms, which can be found in [7], [2], [21], [13].

BWLOCK can be abused if granted to any task. 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 the 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. Even so, if all simultaneously running tasks are real-time ones, then BWLOCK does not currently offer any protection between them. We argue, however, that the fact that real-time tasks can only be interfered by fellow real-time tasks—but not by unknown non real-time tasks—makes it much easier to analyze the system because the real-time tasks can be known in advance. Similar arguments were made in [12] in the context of mixed-criticality scheduling.

## 7 Related Work

OS level memory bandwidth control was first discussed in literature by Bellosa [3]. The basic idea is to control memory bandwidth usage of the cores by means of software mechanisms. For example, in Bellosa's original work, TLB miss handler was used for (approximate) memory bandwidth control. More recently, MemGuard [33], [34] and several works [15], [10], [25] re-introduced OS-level memory bandwidth control for real-time systems, providing stronger bandwidth guarantees (bandwidth reservation) by using hardware performance counters on modern COTS multicore processors. Some works have investigated dynamic budget adjustment techniques for soft real-time systems [34], [24]. In [10], Fldoin et al. also present a bandwidth reservation scheme but at the granularity of tasks instead of cores. The scheme is primarily focused on hard real-time tasks but also allows dynamic bandwidth adjustments if the hard real-time tasks finish early. In contrast, BWLOCK enables more fine-grained and direct memory bandwidth control—albeit requiring programmers' efforts and best only for soft real-time tasks—that can take advantage of memory-level parallelism available in modern multicore architectures, while limiting excessive concurrent memory accesses when necessary. We show that this application driven, explicit, fine-grained memory access control approach can be more efficient, especially for soft real-time applications.

Another OS-based technique is page coloring, which has been extensively studied, especially in the context of partitioning shared caches [23], [31], [32]. 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 [22], [28], [32] and TLB [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 several hardware proposals that allow collaborations between the system software (OS) and the hardware to make better resource scheduling/allocation decisions. For example, several DRAM controller proposals utilize task-level priority information in scheduling memory requests [18], [27], [16]. Also, hardware based predictable memory system designs have been actively studied in the real-time systems community. Some researchers investigated LR server [6] based memory controller and interconnection network designs that provide latency and bandwidth guarantees [7], [2]. Others

investigated TDMA based memory controllers that can provide fine-grained bandwidth reservation capability in hardware [21], [13]. These hardware designs can be integrated with BWLOCK, replacing the current OS based regulators, to provide finer protections to real-time applications.

## 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. It provides a lock-like API to declare memory-performance critical sections (MPCSs) in the application code. When a real-time task's MPCS is executed on a core, BWLOCK automatically regulates memory traffic of the rest of the cores, executing non real-time tasks, to avoid memory bandwidth contention.

We evaluated BWLOCK using CortexSuite benchmarks. By selectively applying BWLOCK on MPCSs, we achieved significant performance improvements, up to 150%, of real-time tasks at controllable throughput impact to non-real-time tasks. Our future work includes integrating hardware assisted bandwidth control mechanisms (e.g.,[2], [21]) and developing automated MPCSs identification tools at the granularities of functions and superblocks [19].

## References


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

[2] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens. Real-time scheduling using credit-controlled static-priority arbitration. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 3–14. IEEE, 2008.

[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] B. B. Brandenburg. *Scheduling and locking in multiprocessor real-time operating systems*. PhD thesis, University of North Carolina at Chapel Hill, 2011.

[5] C. A. S. T. (CAST). Position Paper CAST-32: Multi-core Processors (Rev 0). Technical report, Federal Aviation Administration (FAA), May 2014.

[6] R. L. Cruz. A calculus for network delay. i. network elements in isolation. *IEEE Transactions on information theory*, 37(1):114–131, 1991.

[7] B. D. de Dinechin, Y. Durand, D. van Amstel, and A. Ghiti. Guaranteed services of the noc of a manycore processor. In *Proceedings of the 2014 International Workshop on Network on Chip Architectures*, pages 11–16. ACM, 2014.

[8] G. Elliott, B. Ward, and J. Anderson. Gpusync: A framework for real-time gpu management. In *Real-Time Systems Symposium (RTSS)*, pages 33–44. IEEE, 2013.

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

[10] J. Flodin, K. Lampka, and W. Yi. Dynamic budgeting for settling dram contention of co-running hard and soft real-time tasks. In *Industrial Embedded Systems (SIES)*, pages 151–159. IEEE, 2014.

[11] P. Gai, L. Abeni, and G. Buttazzo. Multiprocessor DSP scheduling in system-on-a-chip architectures. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 231–238. IEEE, 2002.

[12] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *Embedded Software (EMSOFT)*, pages 1–15. IEEE, 2013.

[13] M. D. Gomony, B. Akesson, and K. Goossens. A real-time multichannel memory controller and optimal mapping of memory clients to memory channels. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(2):25, 2015.

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

[15] 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.

[16] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. *ACM SIGMETRICS Performance Evaluation Review*, 35(1):25–36, 2007.

[17] N. Kim, B. C. Ward, M. Chisholm, C.-Y. Fu, J. H. Anderson, and F. D. Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 149–160. IEEE, 2016.

[18] 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.

[19] K. Lampka, G. Giannopoulou, R. Pellizzoni, Z. Wu, and N. Stoimenov. A formal approach to the wcrt analysis of multicore systems with memory contention under phase-structured task sets. *Real-Time Systems*, 50(5-6):736–773, 2014.

[20] 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.

[21] Y. Li, B. Akesson, and K. Goossens. Architecture and analysis of a dynamically-scheduled real-time memory controller. *Real-Time Systems*, pages 1–55, 2015.

[22] 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.

[23] 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.

[24] J. Nowotsch and M. Paulitsch. Quality of service capabilities for hard real-time applications on multi-core processors. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, pages 151–160. ACM, 2013.

[25] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 109–118. IEEE, 2014.

[26] S. A. Panchamukhi and F. Mueller. Providing task isolation via tlb coloring. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 3–13. IEEE, 2015.

[27] 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.

[28] N. Suzuki, H. Kim, D. d. 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.

[29] S. Thomas, C. Gohkale, E. Tanuwidjaja, T. Chong, D. Lau, S. Garcia, and M. B. Taylor. Cortexsuite: A synthetic brain benchmark suite. *IISWC*, Oct. 2014.

[30] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson. Making shared caches more predictable on multicore platforms. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 157–167. IEEE, 2013.

[31] 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.

[32] 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)*, pages 155–166. IEEE, 2014.

[33] 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)*, pages 55–64. IEEE, 2013.

[34] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory Bandwidth Management for Efficient Performance Isolation in Multi-core Platforms. *Transactions on Computers (TC)*, 2015.