# Memory Bandwidth Management for Efficient Performance Isolation in Multi-core Platforms

Heechul Yun[†], Gang Yao[‡], Rodolfo Pellizzoni[⋆], Marco Caccamo[‡], Lui Sha[‡]
[†] University of Kansas, USA. heechul@ittc.ku.edu
[‡] University of Illinois at Urbana-Champaign, USA. {gangyao,mcaccamo,lrs}@illinois.edu
[⋆] University of Waterloo, Canada. rpellizz@uwaterloo.ca

**Abstract**—Memory bandwidth in modern multi-core platforms is highly variable for many reasons and it is a big challenge in designing real-time systems as applications are increasingly becoming more memory intensive. In this work, we proposed, designed, and implemented an efficient memory bandwidth reservation system, that we call **MemGuard**. MemGuard separates memory bandwidth in two parts: *guaranteed* and *best effort*. It provides bandwidth reservation for the guaranteed bandwidth for temporal isolation, with efficient reclaiming to maximally utilize the reserved bandwidth. It further improves performance by exploiting the best effort bandwidth after satisfying each core's reserved bandwidth. MemGuard is evaluated with SPEC2006 benchmarks on a real hardware platform, and the results demonstrate that it is able to provide memory performance isolation with minimal impact on overall throughput.

**Index Terms**—Multicore, Performance isolation, Operating system, Memory bandwidth

## 1 Introduction

Computing systems are increasingly moving toward multi-core platforms and their memory subsystem is a crucial shared resource. As applications become more memory intensive and processors include more cores that share the same memory system, the performance of main memory becomes critical for overall system performance.

In a multi-core system, the processing time of a memory request is highly variable as it depends on the location of the access and the state of DRAM chips and the DRAM controller: There is inter-core dependency as the memory accesses from one core could also be influenced by requests from other cores; the DRAM controller commonly employs scheduling algorithms to re-order requests in order to maximize overall DRAM throughput [29]. All these factors affect the temporal predictability of memory intensive real-time applications due to the high variance of their memory access time. This imposes a big challenge for real-time systems because execution time guarantees of tasks running on a core can be invalidated by workload changes in other cores. Therefore, there is an increasing need for memory bandwidth management solutions that provide Quality of Service (QoS).

Resource reservation and reclaiming techniques [32], [1] have been widely studied by the real-time community to solve the problem of assigning different fractions of a shared resource in a guaranteed manner to contending applications. Many variations of these techniques have been successfully applied to CPU management [22], [10], [9], disk management [39], [36], and more recently to GPU management [20], [21]. In the case of main memory resource, there have been several efforts to design more predictable memory controllers [4], [3] providing latency and bandwidth guarantee at the hardware level.

Unfortunately, existing solutions cannot be easily used for managing memory bandwidth on Commercial Off-The-Shelf (COTS) based real-time systems. First, hardware-based solutions are not generally possible in COTS based systems. Moreover, state-of-art resource reservation solutions [10], [13] cannot be directly applied to the memory system, mostly because the achievable memory service rate is highly dynamic, as opposed to the constant service rate in CPU scheduling. To address these limitations and challenges, we propose an efficient and fine-grained memory bandwidth management system, which we call *MemGuard*.

MemGuard is a memory bandwidth reservation system implemented in the OS layer. Unlike CPU bandwidth reservation, under MemGuard, the available memory bandwidth is described as having two components: *guaranteed* and *best effort*. The guaranteed bandwidth is the worst-case bandwidth of the DRAM system, while the additionally available bandwidth is best effort, which cannot be guaranteed by the system. Memory bandwidth reservation in MemGuard is based on the guaranteed part in order to guarantee minimal service to each core. However, to efficiently utilize all the guaranteed memory bandwidth, a reclaiming mechanism is proposed leveraging each core's usage prediction. We further improve the overall system throughput by sharing the best effort bandwidth after the guaranteed bandwidth of each core is satisfied.

Since our reclaiming algorithm is prediction based, misprediction can lead to a situation where guaranteed bandwidth is not delivered to the core. Therefore, MemGuard is intended to support mainly soft real-time systems. However, hard real-time tasks can be accommodated within this resource management framework by selectively disabling the reclaiming feature.

In our previous work [43], we presented a basic reservation,

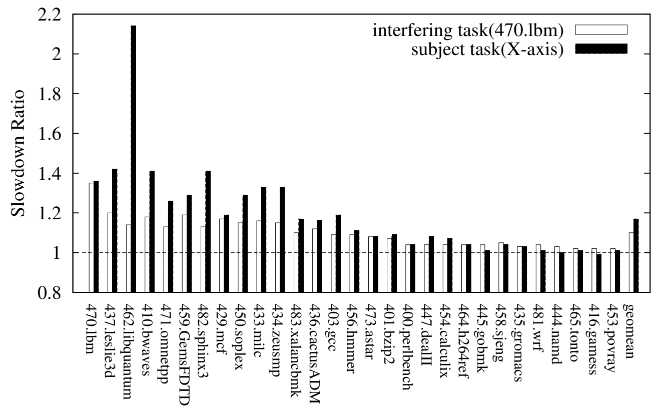
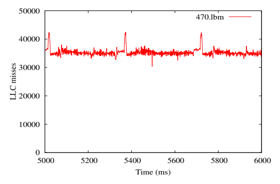
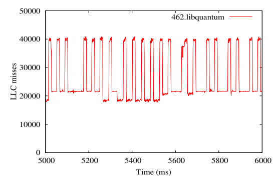


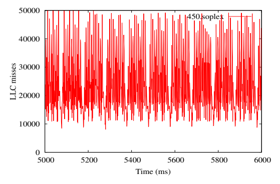
Fig. 1: IPC slowdown of subject tasks (X-axis) and the interfering task (470.lbm) on a dual-core configuration



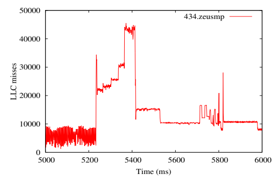
(a) 470.lbm



(b) 462.libquantum



(c) 450.soplex



(d) 434.zeusmp

Fig. 2: Memory access pattern of four representative SPEC2006 benchmarks.

reclaiming, and sharing algorithms of MemGuard. As an extension of our previous work, in this article, we present a new best-effort bandwidth sharing algorithm and its experimental evaluation results. In addition, we also present a new software implementation that eliminates the linear scheduling overhead, and a clear definition of the guaranteed and the best-effort bandwidth. We also present an extensive overhead analysis of the implementation.

In summary, the contributions of this work are: (1) decomposing overall memory bandwidth into a guaranteed and a best effort components; (2) designing and implementing (in Linux kernel) an efficient memory bandwidth reservation system, named MemGuard; (3) evaluating MemGuard with an extensive set of realistic SPEC2006 benchmarks [15] and showing its effectiveness on a real multi-core hardware platform.

The remaining sections are organized as follows: Section 2 describes the challenge of predictability in modern multicore systems. Section 3 describes the overall architecture of MemGuard. Section 4 describes reservation and reclaiming algorithms of MemGuard. Section 5 describes two best-effort bandwidth sharing algorithms of MemGuard. Section 6 describes the evaluation platform and the software implementation. Section 7 presents the evaluation results. Section 8 discusses related work. We conclude in Section 9.

## 2 Problems of Shared Memory in Multicore Systems

Many modern embedded systems process vast amount of data that are collected from various type of sensing devices such as surveillance cameras. Therefore, many real-time applications are increasingly becoming more memory bandwidth intensive. This is especially true in multicore systems, where additional cores increase the pressure on the shared memory hierarchy. Therefore, task execution time is increasingly more dependent on the way that memory resources are allocated among cores. To provide performance guarantees, real-time system have long adopted resource reservation mechanisms. Our work is also based on this approach but there are difficulties in applying a reservation solution in handling memory bandwidth. To illustrate the problems, we performed two set of experiments on a real multicore hardware (described in Section 6.1).

In the first experiment, we measured Instructions-Per-Cycle (IPC) for each SPEC2006 benchmark (subject task) first running on a core in isolation, and then together with a memory intensive benchmark (470.lbm) running on a different core (interfering task). Figure 1 shows IPC slowdown ratio [1] of co-scheduled tasks on a dual core system (See Section 6.1 and Section 7.2 for details). Note that the subject tasks are arranged from the most memory intensive to the least memory intensive on the X-axis. As clearly showed in the figure, both the subject task and the interfering task suffer slowdown since they are interfering with each other in the memory accesses, which is expected. Interestingly, however, the difference of slowdown factors between the two tasks could be as large as factor of two (2.2x against 1.2x). Such effects are typical in COTS systems due to the characteristics of modern DRAM controllers [29]: (1) each DRAM chip is composed of multiple resources, called banks, which can be accessed in parallel. The precise degree of parallelism can be extremely difficult to predict, since it depends, among others, on the memory access patterns of the two tasks, the allocation of physical addresses in main memory, and the addressing scheme used by the DRAM controller. (2) Each DRAM bank itself comprises multiple rows; only one row can be accessed at a time, and switching row is costly. Therefore, DRAM controllers commonly implement scheduling algorithms that re-order requests—typically prioritize row hit requests over row miss requests—to maximize overall memory throughput [29], [33]. In such a DRAM controller, a task that floods the DRAM controller with many row hits would suffer less slowdown than those with many row misses. These results indicate that memory bandwidth is very different compared to CPU bandwidth, in the sense that achievable bandwidth (when requests are backlogged) is not fixed but highly variable

1. The slowdown ratio of a task is defined as *run-alone/co-scheduled IPC*.

depending on the access location of each memory request and on the state of DRAM subsystem.

Furthermore, the memory access patterns of tasks can be highly unpredictable and significantly change over time. Figure 2 shows the memory access patterns of four benchmarks from SPEC2006. Each benchmark runs alone and we collected memory bandwidth usage of the task from time 5 to 6 sec., sampled over every 1ms time interval (i.e., 1000 samples), using the LLC miss hardware performance monitoring counter (PMC). The Y-axis shows the number of LLC misses for every 1ms interval. The 470.lbm shows relatively uniform access pattern throughout the whole time. On the other hand, 462.libquantum and 450.soplex show highly variable access pattern, while 434.zeusmp show mixed behavior over time.

When resource usage changes significantly over time, a static reservation approach could result in poor resource utilization and performance. If a task reserves for its peak resource demand (in this case, the task's peak memory bandwidth demand), it would significantly waste the resource as the task usually does not consume that amount; while if the task reserves for its average resource demand, it would suffer delay whenever it tries to access more than the average resource demand. The problem is only compounded when the available resource amount also changes over time, as it is the case for memory bandwidth. One ideal solution is to dynamically adjust the resource provision based on its actual usage: when the task is highly demanding on the resource, it can try to reclaim some possible spare resource from other entities; on the other hand, when it consumes less than the reserved amount, it can share the extra resource with other entities in case they need it. Furthermore, if the amount of available resource is higher than expected, we can allocate the remaining resource units among demanding tasks. There have been two types of dynamic adaptation schemes: feedback-control based adaptive reservation approaches [2] and resource reclaiming based approaches [9], [22]. In our work, we choose a reclaiming based approach for simplicity.

The details of our bandwidth management system, MemGuard, are provided in the next section. In summary, based on the discussed experiments, the system will need to: (1) reserve a fraction of memory bandwidth to each core (a.k.a. resource reservation) to provide predictable worst-case behavior; and (2) provide some dynamic resource adjustment on the resource provision (a.k.a. resource reclaiming and sharing) to efficiently exploit varying system resource demands and improve task responsiveness behavior.

## 3 MemGuard Overview

The goal of MemGuard is to provide memory performance isolation while still maximizing memory bandwidth utilization. By memory performance isolation, we mean that the average memory access latency of a task is no larger than when running on a dedicated memory system which processes memory requests at a certain service rate (e.g., 1GB/s). A multicore system can then be considered as a set of unicore systems, each of which has a dedicated, albeit slower, memory subsystem. This notion of isolation is commonly achieved through resource reservation approaches in real-time literature [1] mostly in the context of CPU bandwidth reservation.

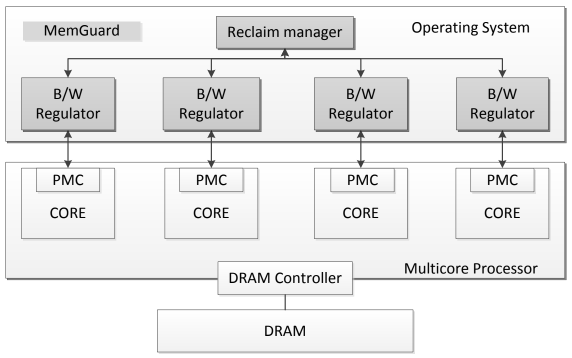


Fig. 3: MemGuard system architecture.

In this paper, we focus on systems where the last level cache is private or partitioned on a per-core basis, in order to focus on DRAM bandwidth instead of cache space contention effects. We assume system operators or an external user-level daemon will configure MemGuard either statically or dynamically via the provided kernel filesystem interface.

Figure 3 shows the overall system architecture of MemGuard and its two main components: the per-core regulator and the reclaim manager. The per-core regulator is responsible for monitoring and enforcing its corresponding core memory bandwidth usage. It reads the hardware PMC to account the memory access usage. When the memory usage reaches a predefined threshold, it generates an overflow interrupt so that the specified memory bandwidth usage is maintained. Each regulator has a history based memory usage predictor. Based on the predicted usage, the regulator can donate its budget so that cores can start reclaiming once they used up their given budgets. The reclaim manager maintains a global shared reservation for receiving and re-distributing the budget for all regulators in the system.

Using the regulators, MemGuard allows each core to reserve a fraction of memory bandwidth, similar to CPU bandwidth reservation. Unlike CPU bandwidth, however, available memory bandwidth varies depending on memory access patterns. In order to achieve performance isolation, MemGuard restricts the total reservable bandwidth to the worst case DRAM bandwidth, *guaranteed bandwidth* $r_{min}$ (see Section 4), similar to the requirement of CPU bandwidth reservation in that the total sum of CPU bandwidth reservation must be equal or less than 100% of CPU bandwidth. Because the memory system can deliver at least the guaranteed memory bandwidth, its fraction reserved by each core can be satisfied *regardless of* the other cores' reservations and their memory access patterns. We will experimentally show performance isolation impact of memory bandwidth reservation in Section 7.1. From the application point of view, the reserved bandwidth can be used to determine its worst-case performance.

Note, however, there are two main problems in the presented reservation approach. First, a core's reserved bandwidth may be wasted if it is not used by the core. Second, the total

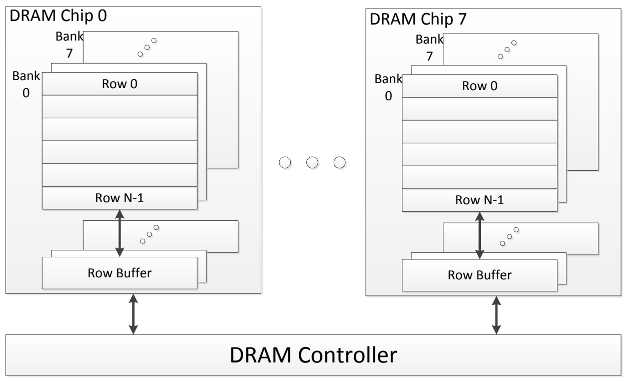


Fig. 5: DRAM based memory system organization—Adopted from Fig. 2 in [27]

reservable bandwidth, the guaranteed bandwidth, is much smaller than the peak bandwidth (e.g., 1.2GBs/s guaranteed vs. 6.4GB/s peak bandwidth in our testbed). Any additional bandwidth, *best-effort bandwidth* (see Section 5), will also be wasted in the reservation only approach. For these two main reasons, reservation alone could significantly waste memory bandwidth utilization and therefore result in low application performance.

To mitigate this inefficiency while still retaining the benefits of bandwidth reservation, MemGuard offers two additional approaches: *bandwidth reclaiming* and *bandwidth sharing*. Bandwidth reclaiming approach (Section 4.3) focuses on managing the guaranteed bandwidth; it dynamically re-distributes bandwidth from cores that under-utilized their reserved bandwidths to cores that need more bandwidths than their reserved ones. On the other hand, bandwidth sharing approaches (Section 5.2 and 5.3) focus on managing the best-effort bandwidth and they only work after all the guaranteed bandwidth is collectively used up by the cores in the system.

Figure 4 shows the high level implementation of MemGuard. Using the figure as a guide, we now describe the memory bandwidth management mechanisms of MemGuard in the following sections.

```
**function** periodic_timer_handler_master ;
**begin**
    $G \leftarrow 0$;
    on_each_cpu( periodic_timer_handler_slave ) ;
**end**
**function** periodic_timer_handler_slave ;
**begin**
    $Q_i^{predict} \leftarrow$ output of usage predictor ;
    $Q_i \leftarrow$ user assigned static budget ;
    **if** $Q_i^{predict} > Q_i$ **then**
        $q_i \leftarrow Q_i$;
    **else**
        $q_i \leftarrow Q_i^{predict}$ ;
        $G$ += $(Q_i - q_i)$;
    program PMC to cause overflow interrupt at $q_i$;
    de-schedule the high priority idle task (if it was
    active);
**end**
**function** overflow_interrupt_handler ;
**begin**
    $u_i \leftarrow$ used budget in the current period ;
    **if** $G > 0$ **then**
        **if** $u_i < Q_i$ **then**
            $q_i \leftarrow \min\{Q_i - u_i, G\}$ ;
        **else**
            $q_i \leftarrow \min\{Q_{min}, G\}$ ;
        $G$ -= $q_i$;
        program PMC to cause overflow interrupt at $q_i$ ;
        Return ;
    **if** $u_i < Q_i$ **then**
        Return ;
    **if** $\sum u_i >= r_{min}$ **then**
        manage best-effort bandwidth (see Section 5) ;
    stall the core by scheduling the high priority idle task;
**end**
```

Fig. 4: MemGuard Implementation

## 4 Guaranteed Bandwidth Management

In this section, we first give necessary background information about DRAM based memory subsystems and define the guaranteed bandwidth which is the basis for our reservation and reclaiming. We then detail the reservation and reclaiming mechanisms of MemGuard.

### 4.1 Guaranteed Memory Bandwidth

Figure 5 shows the organization of a typical DRAM based memory system. A DRAM module is composed of several DRAM chips that are connected in parallel to form a wide interface (64bits for DDR). Each DRAM chip has multiple banks that can be operated concurrently. Each bank is then organized as a 2d array consisting of rows and columns. A location in DRAM can be addressed with the bank, row and column number.

In each bank, there is a buffer, called *row buffer*, to store a single row (typically 1-2KB) in the bank. In order to access data, the DRAM controller must first copy the row containing the data into the row buffer (i.e., opening a row). The required latency for this operation is denoted as $tRCD$ in DRAM specifications. The DRAM controller then can read/write from the row buffer with only issuing column addresses, as long as the requested data is in the same row. The associated latency is denoted as $tCL$. If the requested data is in a different row, however, it must save the content of the row buffer back to the originating row (i.e., closing a row). The associated latency is denoted as $tRP$. In addition, in order to open a new row in the same bank, at least $tRC$ time should have been passed since the opening of the current row. Typically $tRC$ is slightly larger than the sum of $tRCD$, $tCL$, and $tRP$. The access latency to a memory location, therefore, varies depending on whether the data is already in the row buffer (i.e., row hit) or not (i.e., row miss) and the worst-case performance is determined by $tRC$ parameter.[2]

Considering these characteristics, we can distinguish the maximum (peak) bandwidth and the guaranteed (worst-case) bandwidth as follows:

- **Maximum (peak) bandwidth:** This is stated maximum performance of a given memory module where the speed is only limited by I/O bus clock speed. For example, a PC6400 DDR2 memory's peak bandwidth is 6.4 GB/s where it can transfer 64bit data at the rising and falling edge of bus cycle running at 400MHz.
- **Guaranteed (worst-case) bandwidth:** The worst-case occurs when all memory requests target a single memory bank, and each successive request accesses different row, causing a row switch (row miss). For example, the worst-case bandwidth of the PC6400 DDR2 memory used in our evaluation is 1.2GB/s, based on parameters in [26].

Because the guaranteed bandwidth can be satisfied regardless of memory access locations, we use the guaranteed bandwidth as the basis for bandwidth reservation and reclaiming as we will detail in the subsequent subsections.

## 4.2 Memory Bandwidth Reservation

MemGuard provides two levels of memory bandwidth reservation: system-wide reservation and per-core reservation.

1) System-wide reservation regulates the total allowed memory bandwidth such that it does not exceed the guaranteed bandwidth — denoted as $r_{min}$.
2) Per-core reservation assigns a fraction of $r_{min}$ to each core, hence each core reserves bandwidth $B_i$ and $\sum_{i=0}^{m-1} B_i = r_{min}$ where $m$ is the number of cores.

Each regulator reserves memory bandwidth represented by memory access budget $Q_i$ (in bytes) for every period $P$ (in seconds): i.e. $B_i = \frac{Q_i}{P}$ (bytes/sec). Regulation period $P$ is a system-wide parameter and should be small to effectively enforce specified memory bandwidth. Although small regulation period is better for predictability, there is a practical limit on reducing the period due to interrupt and scheduling overhead; we currently configure the period as 1ms. The reservation follows the common resource reservation rules [32], [35], [37]. For accurately accounting memory usage, we use per-core PMC interrupt: specifically, we program the PMC at the beginning of each regulation period so that it generates an interrupt when the core's budget is depleted. Once the interrupt is received, the regulator calls the OS scheduler to schedule a high-priority real-time task to effectively idle the core until the next period begins. At the beginning of the next period the budget is replenished in full and the real-time "idle" task will be suspended so that regular tasks can be scheduled. Because the reservation mechanism is based on hardware PMC interrupts, it can accurately regulate the maximum number of memory accesses of each core for each period $P$.

2. We do not consider auto-refresh as its impact on the bandwidth is small [8], due to the long referesh interval ($tREFI$). We also do not consider the bus turn-arounds because COTS dram controllers typically process reads and writes in batches which amortize the bus turn-around delay [28].

## 4.3 Memory Bandwidth Reclaiming

Each core has statically assigned bandwidth $Q_i$ as the baseline. It also maintains an instant budget $q_i$ to actually program the PMC which can vary at each period based the output of the memory usage predictor. We currently use an Exponentially Weighted Moving Average (EWMA) filter as the memory usage predictor which takes the memory bandwidth usage of the previous period as input. The reclaim manager maintains a global shared budget $G$. It collects surplus bandwidth from each core and re-distributes it when in need. Note that $G$ is initialized by the master core (Core0) at the beginning of each period (Line 3 in Figure 4) and any unused $G$ in the previous period is discarded. Each core only communicates with the central reclaim manager for donating and reclaiming its budget. This avoids possible circular reclaiming among all cores and greatly reduces implementation complexity and runtime overhead.

The details of the reclaiming rules are as follows:

1) At the beginning of each regulation period, the current per-core budget $q_i$ is updated as follows:

$$q_i = \min\{Q_i^{predict}, Q_i\}$$

If the core is *predicted* not to use the full amount of the assigned budget $Q_i$, the current budget is set the predicted usage, $Q_i^{predict}$ (See Line 12-13 in Figure 4). Any predicted surplus budget of the core is donated to the global budget $G$ (Line 14 in Figure 4). Hence, $G$ is:

$$G = \sum_i \{Q_i - q_i\}.$$

2) During execution, the core can reclaim from the global budget if its corresponding budget is depleted. The amount of reclaim depends on the requesting core's condition: If the core's used budget $u_i$ is less than the statically reserved budget $Q_i$ (this can happen when $Q_i^{predict} < Q_i$), then it tries to reclaim up to $Q_i$ as follows (Line 23 in Figure 4):

$$q_i = \min\{Q_i - u_i, G\}.$$

Or, if $u_i \geq Q_i$ (getting more than its reserved amount), it only gets up to a small fixed amount of budget, $Q_{min}$ (See Line 25 in Figure 4):

$$q_i = \min\{Q_{min}, G\}.$$

If $Q_{min}$ is too small, too many interrupts can be generated within a period, increasing overhead. On the other hand, if, it is too big, it might increase the probability of reclaim under-run error, described in the next item. As such, it is a configuration parameter. We currently assign 1% of the peak bandwidth as the value of $Q_{min}$. After the core updates its budget $q_i$, it reprograms the performance counter and return (Line 27-28 in Figure 4).

3) Since our reclaim algorithm is based on prediction, a core may not be able to use its assigned budget $Q_i$. This can happen when the core donated its budget too much (due to mis-prediction) and other cores already reclaimed all donated budgets ($G = 0$) before the core tries to reclaim.

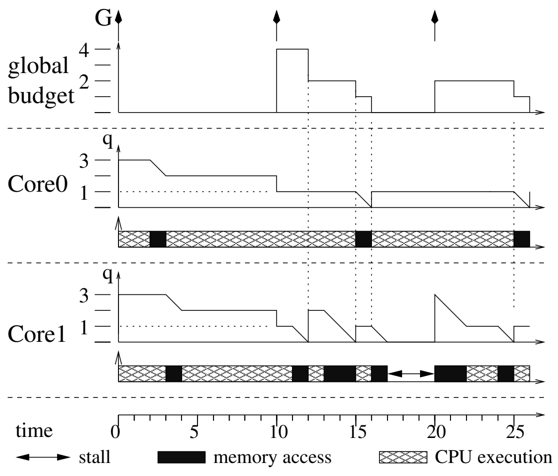


Fig. 6: An illustrative example with two cores

When this happens, our current heuristic is to allow the core to continue execution, hoping it may still use $Q_i$, although it is not guaranteed (Line 29-30 in Figure 4). At the beginning of the next period, we verify if each core was able to use its budget. If not, we call it a *reclaim under-run error* and notify the predictor of the difference $(Q_i - u_i)$. The predictor then tries to *compensate* it by using $Q_i + (Q_i - u_i)$ as its input for the next period.

### 4.4 Example

Figure 6 shows an example with two cores, each with an assigned static budget 3 (i.e., $Q_0 = Q_1 = 3$). The regulation period is 10 time units and the arrows at the top of the figure represent the period activation times. The figure shows the global budget $G$ two cores' instant budgets $q_1$ and $q_2$

When the system starts, each core starts with the assigned budget 3. At time 10, the prediction $Q_{predict}$ of both cores is 1 as both have used only 1 budget (i.e., $u_1 = u_2 = 1$) within the period [0,10]; hence, the instant budgets $q_1$ and $q_2$ become 1 and the global budget $G$ becomes 4 (Line 13-14 in Figure 4). At time 12, Core 1 depletes $q_i$. Since $Q_1$ is 3, Core1 reclaims 2 ($Q_1 - q_1$, Line 23 in Figure 4) from $G$ and $q_1$ becomes 2. At time 15, Core1 depletes its budget again. This time Core1 already has used $Q_i$, it only reclaims 1 ($Q_{min}$, Line 25 in Figure 4) from $G$ and $q_1$ becomes 1. At time 16, Core0 depletes its budget. Since remaining $G$ is 1 at this point, Core0 only reclaims 1 and $G$ drops to 0. At time 17, Core1 depletes its budget again and it is suspended (Line 33 in Figure 4) as it cannot reclaim from $G$. When the third period starts at time 20, the $Q_1^{predict}$ is larger than $Q_1$. Therefore, Core1 gets the full amount of the assigned budget, $q_1 = Q_1 = 3$ (Line 11 in Figure 4), while Core0 only gets 1, and donates 2 to $G$. At time 25, after Core1 depletes $q_1$, Core1 reclaims $Q_{min}$ from G.

## 5 Best-effort Bandwidth Management

In this section, we first define the best-effort memory bandwidth in MemGuard and describe two proposed management schemes: *spare sharing* and *proportional sharing*.

### 5.1 Best-effort Memory Bandwidth

We define best-effort memory bandwidth as any additionally achieved bandwidth above the guaranteed bandwidth $r_{min}$. Because $r_{min}$ is smaller (e.g., 1.2GB/s) than the peak bandwidth (e.g., 6.4GB/s), efficient utilization of best-effort bandwidth is important, especially for memory intensive tasks.

As described in the previous section, memory reservation and reclaiming algorithms in MemGuard operate on the guaranteed bandwidth $r_{min}$. If all cores exhaust their given bandwidths before the current period ends, however, all cores would wait for the next period doing nothing under the rules described in Section 4. Since MemGuard already delivered reserved bandwidth to each core, any additional bandwidth is now considered as best-effort bandwidth.

### 5.2 Spare Sharing

When all cores collectively use their assigned budgets (Line 31 in Figure 4), the spare sharing scheme simply let all cores compete for the memory until the next period begins. This strategy maximizes throughput as it is effectively equivalent to temporarily disabling the MemGuard. In another perspective, it gives an equal chance for each core to utilize the remaining best-effort bandwidth, regardless of its reserved memory bandwidth.

Figure 7(a) shows an example operation of the scheme. At time 5, Core0 depletes its budget and it suspend the core (Line 33 in Figure 4), assuming the $G$ is zero in the period. At time 7, Core1 depletes its budget. At this point, there are no cores that have remaining budgets. Therefore, Core1 sends a broadcast message to wake-up the cores; and both cores compete for the memory until the next period begins at time 10.

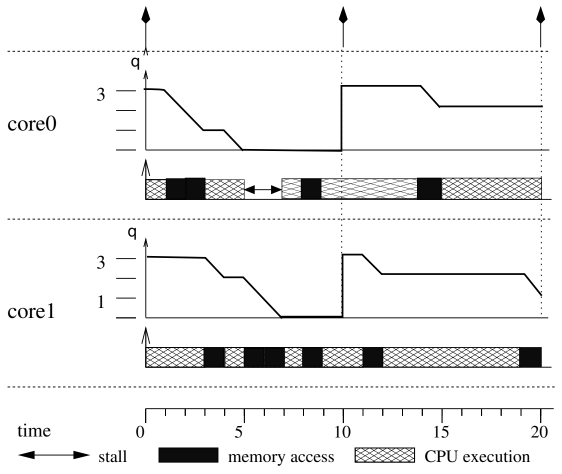
Note that this mechanism only starts *after* all the cores have depleted their assigned budgets. The reason is that if a core has not yet used its budget, allowing other cores to execute may bring intensive memory contention, preventing the core from using the remaining budget.

### 5.3 Proportional Sharing

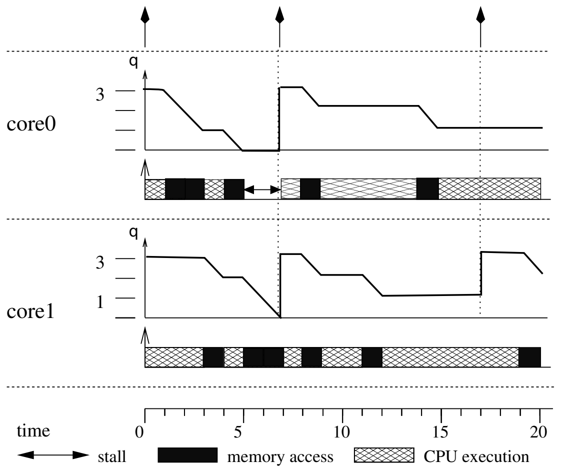
The proportional sharing scheme also starts after all cores use their budgets like the spare-sharing scheme, described in the previous subsection, but it differs in that it starts a new period immediately instead of waiting the remaining time in the period. This effectively makes each core to utilize the best-effort bandwidth proportional to its reserved bandwidth—i.e., the more the $Q_i$, the more best-effort bandwidth the core gets.

Figure 7(b) shows an example operation of the scheme. At time 7, when all cores have used their budgets, it starts a new period and each core's budget is recharged immediately. This means that the length of each period can be shorter depending on workload. However, if the condition is not met, i.e., there is at least one core that has not used its budget, then the same fixed period length is enforced.

This scheme bears some similarities with the IRIS algorithm, a CPU bandwidth reclaiming algorithm [23], which extends CBS [1] to solve the deadline aging problem of CBS; in the original CBS, when a server exhausts its budget, it



(a) Spare Sharing



(b) Proportional Sharing

Fig. 7: Comparison of two best-effort bandwidth management schemes

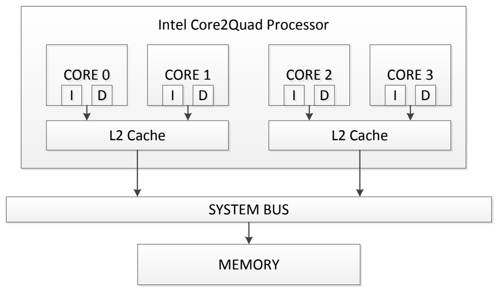


Fig. 8: Hardware architecture of our evaluation platform.

immediately recharges the budget and extends the deadline. This can cause very long delays to CPU intensive servers that extend deadlines very far ahead while other servers are inactive. The IRIS algorithm solves this problem by introducing a notion of recharging time in which a server that exhausts its budget must wait until it reaches the recharging time. If there is no active server and there is at least one server that waits for recharging, IRIS updates server's time to the earliest recharging time. This is similar to the proportional sharing scheme presented in this subsection in that servers' budgets are recharged when all servers have used their given budgets. They are, however, significantly different in the sense that proportional sharing is designed for sharing memory bandwidth between multiple concurrently accessing cores and is it is not based on CBS scheduling method.

## 6 Evaluation Setup

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

### 6.1 Evaluation platform

Figure 8 shows the architecture of our testbed. We use an Intel Core2Quad Q8400 processor based desktop computer as our testbed. The processor is clocked at 2.66GHz and has four physical cores. It contains two separate 2MB L2 caches, each of which is shared by two cores. As our focus is not the shared cache, we use cores that do not share the same LLC for experiments. We do, however, use four cores when tasks are not sensitive to shared LLC by nature (e.g., working set size of each task is bigger than the LLC size). We use two 2GB PC6400 DDR2 DRAM (2 ranks/module, 8banks/rank, and 16K rows/bank). We experimentally measured the guaranteed bandwidth $r_{min}$ as 1.2GB/s, which is also in a match with the computed value using parameters in [26].

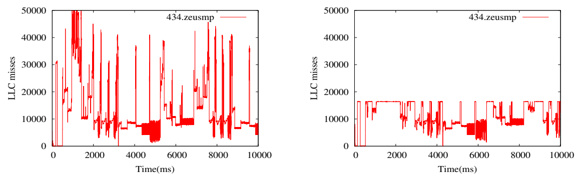
In order to account per-core memory bandwidth usage, we used a LLC miss performance counter [3] per each core. Since the LLC miss counter does not account prefetched memory traffic, we disabled all hardware prefetchers [4].

Note that LLC miss counts do not capture LLC write-back traffic which may underestimate actual memory traffic, particularly for write-heavy benchmarks. However, because SPEC2006 benchmarks, which we used in evaluation, are read heavy (only 20% of memory references are write [19]); and memory controllers often implement write-buffers that can be flushed later in time (e.g., when DRAM is not in use by other outstanding read requests) and writes are considered to be completed when they are written to the buffers [17]; therefore, write-back traffic do not necessarily cause additional latency in accessing DRAM. Analyzing the impact of accounting write-back traffic in terms of performance isolation and throughput is left as future work.

### 6.2 Implementation Details and Overhead Analysis

We implemented MemGuard in Linux version 3.6 as a kernel module [5]. We use the *perf_event* infrastructure to install the counter overflow handler at each period. The logic of both handlers is already shown in Figure 4.

3. LAST_LEVEL_CACHE_MISSES: event=2e, umask=41. See [18]
4. We used http://www.eece.maine.edu/~vweaver/projects/prefetch-disable/
5. MemGuard is available at https://github.com/heechul/memguard



(a) w/o MemGuard (b) MemGuard (1GB/s)

Fig. 9: Memory access patterns of a SPEC2006 benchmark.

With MemGuard, system designers can assign an absolute bandwidth value for each core (e.g., 1GB/s) or a relative weight expressing its relative importance. In the latter case, the actual bandwidth is calculated at every period by checking active cores (cores that have runnable tasks in their runqueues). Figure 9 shows memory access patterns of a benchmark with (reservation only) and without using MemGuard.

Compared to our previous implementation [43], we made several changes. First, as explained in Section 4, we now schedule a high-priority real-time task to "throttle" the core instead of de-scheduling all the tasks in the core. This solves the problem of linearly increasing overhead as the number of tasks increase. Second, we now use a dedicated timer for pacing the regulation period, instead of using the OS tick timer to easily re-program the length of each period—it is needed to support the proportional sharing mode in Section 5.3. To synchronize the period among all cores, a core (the master core) is designated to receive the timer interrupts. On receiving a timer interrupt, the master core broadcasts the new period to the rest of the cores via inter-processor interrupts (IPIs). Although this master-slave architecture incurs some propagation delay, we found it is negligible (typically a few micro seconds) compared to the period length. Regarding IPI overhead, similar findings were reported in [11]. The PMC programming overhead is also negligible as it only requires a write to each core's local register. The overhead of scheduling a real-time idle task is equal to a single context switch overhead, which is typically less than 2μs in the tested platform [6], regardless of the number of tasks in the core; this is a notable improvement compared to our previous implementation in which overhead grows linearly as the number of tasks increase.

Note, however, that MemGuard does incur some performance overhead for several reasons. First, each core receives a period interrupt (either timer or IPI interrupt) at every period. So, computation time of the period interrupt would be added to the task execution time. Second, a core can receive a number of performance counter overflow interrupts, especially when reclaiming is enabled. In that case, computation time of the overflow handler would also increase the task execution time. Third, executing the interrupt handlers could evict some cache-lines used by tasks, which would indirectly increase the task execution time. Lastly, MemGuard maintains a few shared global variables (e.g., $G$) which are protected by a spinlock. As such, when multiple cores try to access the lock, there can be some lock contention effect.

6. Measured using lat_ctx of LMBench3[25].

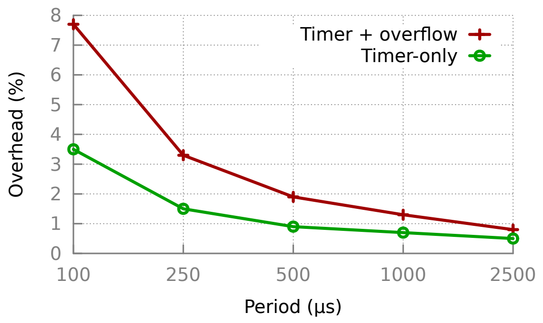


Fig. 10: Runtime overhead vs. Period

We conduct an experiment to quantify overhead caused by MemGuard. In this experiment, we first run four instances of the Latency benchmark [43] (one for each core) and measure the execution time of the one at Core 0. We then repeat the experiment with using MemGuard on two different settings: *Timer-only* and *Timer+overflow*. In Timer-only setting, we assign very high bandwidths to cores so that only timer interrupts occur periodically. In Timer+overflow setting, however, we assign very small bandwidths to cores so that each core generates an overflow interrupt and a timer interrupt handler in each period. To exclude the execution time increase due to core throttling and to measure only the overhead of both interrupt handlers themselves, we disabled throttling (i.e., commenting out Line 30 in Figure 4) during the experiment.

Figure 10 shows performance overhead (i.e., increased task execution time due to the interrupt handlers) as a function of the period length. Note that with 1ms (1000μs) period, performance overhead is less than 2%. But as we reduce the period length further, overhead increases—up to 8% with 100μs period. Based on the results, we pick 1ms as the default period length and use it for the rest of our evaluation.

## 7 Evaluation Results and Analysis

In this section, we evaluate MemGuard in terms of performance isolation guarantee and throughput with SPEC2006 benchmarks and a set of synthetic benchmarks.

To understand characteristics of SPEC2006 benchmarks, We first profile them as follows: We run each benchmark for 10 seconds with the reference input and measure the instruction counts and LLC miss counts, using *perf* tool in Linux kernel, to calculate the average IPC and the memory bandwidth usage. We multiply the measured LLC miss count with the cache-line size (64 bytes in our testbed) to get the total memory bandwidth usage. Note that MemGuard is not being used in this experiment.

Table 1 shows the characteristics of each SPEC2006 benchmark, in decreasing order of average memory bandwidth usage, when each benchmark runs alone on our evaluation platform. Notice that the benchmarks cover a wide range of memory bandwidth usage, ranging from 1MB/s (453.povray) up to 2.1GB/s (470.lbm).

In the subsequent experiments, we use four different modes of MemGuard: reserve only (MemGuard-RO), b/w reclaim

| Benchmark | Avg. IPC | Avg. B/W(MB/s) | Memory Intensity |
|---|---|---|---|
| 470.lbm | 0.52 | 2121 | High |
| 437.leslie3d | 0.51 | 1581 | |
| 462.libquantum | 0.60 | 1543 | |
| 410.bwaves | 0.62 | 1485 | |
| 471.omnetpp | 0.83 | 1373 | |
| 459.GemsFDTD | 0.50 | 1203 | |
| 482.sphinx3 | 0.58 | 1181 | |
| 429.mcf | 0.18 | 1076 | |
| 450.soplex | 0.54 | 1025 | |
| 433.milc | 0.59 | 989 | Medium |
| 434.zeusmp | 0.93 | 808 | |
| 483.xalancbmk | 0.54 | 681 | |
| 436.cactusADM | 0.68 | 562 | |
| 403.gcc | 0.98 | 419 | |
| 456.hmmer | 1.53 | 317 | |
| 473.astar | 0.58 | 307 | |
| 401.bzip2 | 0.97 | 221 | |
| 400.perlbench | 1.36 | 120 | |
| 447.dealII | 1.41 | 118 | |
| 454.calculix | 1.53 | 113 | |
| 464.h264ref | 1.42 | 101 | |
| 445.gobmk | 0.97 | 95 | Low |
| 458.sjeng | 1.10 | 74 | |
| 435.gromacs | 0.86 | 60 | |
| 481.wrf | 1.73 | 38 | |
| 444.namd | 1.47 | 18 | |
| 465.tonto | 1.38 | 2 | |
| 416.gamess | 1.34 | 1 | |
| 453.povray | 1.17 | 1 | |

TABLE 1: SPEC2006 characteristics

(MemGuard-BR), b/w reclaim + spare share (MemGuard-BR+SS), and b/w reclaim + proportional sharing (MemGuard-BR+PS). In MemGuard-RO mode, each core only can use its reserved b/w as described in Section 4.2. The MemGuard-BR mode uses the predictive memory bandwidth reclaiming algorithm described in Section 4.3. Both MemGuard-BR+SS mode and MemGuard-BR+PS use the bandwidth reclaiming but differ in how to manage the best-effort bandwidth after all cores collectively consume the guaranteed bandwidth as described in Section 5.2 and Section 5.3 respectively.

In all experiments, we repeat each experiment at least three times and choose the highest performing result.

### 7.1 Performance Isolation Effect of Reservation

In this experiment, we illustrate the effect of memory bandwidth reservation on performance isolation. Specifically, our goal is to validate whether the bandwidth reservation rule (i.e., $\sum_{i=0}^{m-1} B_i = r_{min}$ in Section 4.2) provides adequate performance isolation among the cores.

The experiment setup is as follows. We first measure the IPC of each SPEC2006 benchmarks alone on Core0 with 1.0GB/s memory bandwidth reservation. We then repeat the experiment but co-schedule 470.lbm, the most memory intensive benchmark, on Core2 with several different memory bandwidth reservations—varying from 0.2 to 2.0 GB/s. Note that assigning more than 0.2GB/s on Core 2 makes the total reserved bandwidth exceeds the estimated $r_{min}$ of 1.2GB/s, which violates our reservation rule. Also note that, in this experiment, we use the reservation only mode (MemGuard-RO) and disable reclaiming and sharing. In other words, each core can use only up to the reserved bandwidth but no more.

Figure 11 shows the normalized IPC of 462.libquantum benchmark on Core0 as a function of reserved bandwidth

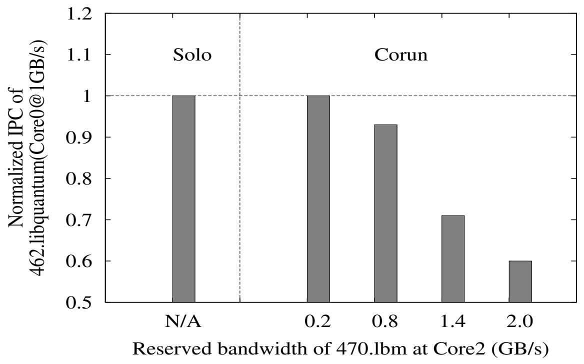


Fig. 11: Normalized IPC of 462.libquantum (Core0@1.0GB/s)

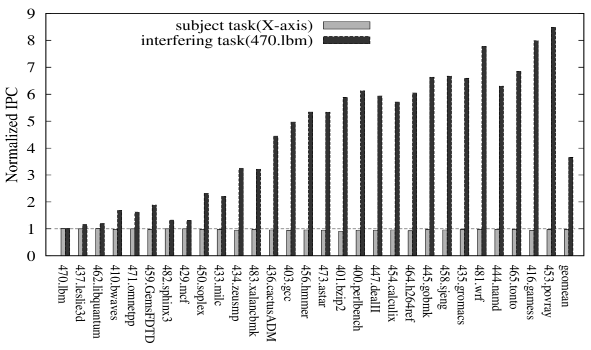
of the interfering 470.lbm on Core2. First, note that the benchmark's solo and co-run performance results are almost identical when the reserved bandwidth for the co-running 470.lbm at Core2 is 0.2GB/s. In other words, performance isolation is achieved as performance of 462.libquantum is not affected by the co-scheduled 470.lbm. As we increase 470.lbm's assigned memory bandwidth, however, performance of 462.libquantum gradually decreases; when the reserved bandwidth for 470.lbm is 2.0GB/s (i.e., 3.0GB/s aggregate bandwidth reservation), more than 40% IPC reduction is observed due to increased memory contention. Results for the rest of SPEC benchmarks also show consistent behavior but are omitted due to space limitations. We also performed experiments with different bandwidth assignments for the subject tasks and found consistent behavior. The full figures that include all benchmarks and different bandwidth assignments are available in [44].

### 7.2 Effect of Reclaiming and Sharing

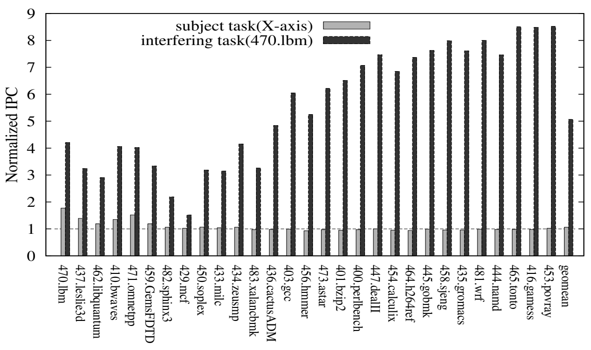
In this experiment, we evaluate the effectiveness of bandwidth reclaiming (MemGuard-BR) and sharing (MemGuard-BR+SS and MemGuard-BR+PS) mechanisms on a dual-core system configuration.

The experiment setup is as follows: We first measure the IPC of each SPEC2006 benchmarks (subject task) alone on Core0 with 1.0GB/s memory bandwidth reservation and the IPC of 470.lbm (interfering task) alone on Core2 with 0.2GB/s reservation using MemGuard-RO. We then repeat the experiment but this time co-schedule each subject task and the interfering task pair together using MemGuard-RO, MemGuard-BR+SS, and MemGuard-BR+PS. Note that in all cases, the statically reserved bandwidths of Core0 and Core2 are 1.0GB/s and 0.2GB/s respectively. Notice that the Core2 is under-reserved as 470.lbm's unregulated average bandwidth is above 2GB/s (see Table 1).

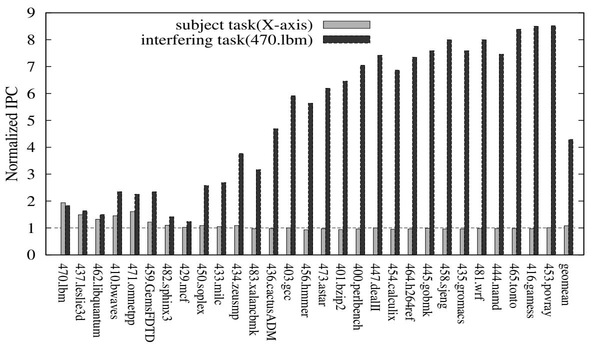
Figure 12 shows the normalized IPCs (w.r.t. MemGuard-RO) of each co-scheduled subject and interfering task pair. The subject tasks in X-axis are sorted in decreasing order of memory intensity. The value of 1 or above means that the performance of the benchmark is equal to or better than the performance with its reserved memory bandwidth only; if the value is less than 1, performance of the benchmark is suffered due to interferences from the other core. MemGuard-BR shows



(a) MemGuard-BR (b/w reclaim)

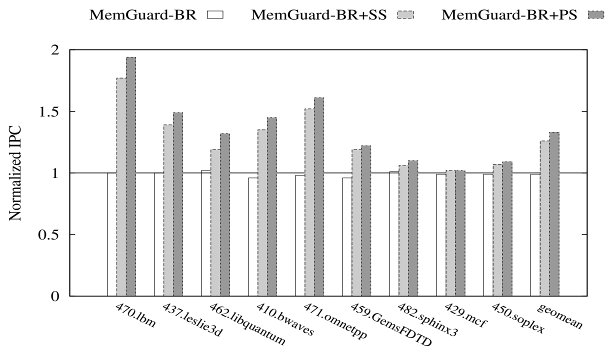


(b) MemGuard-BR+SS (b/w reclaim + spare share)

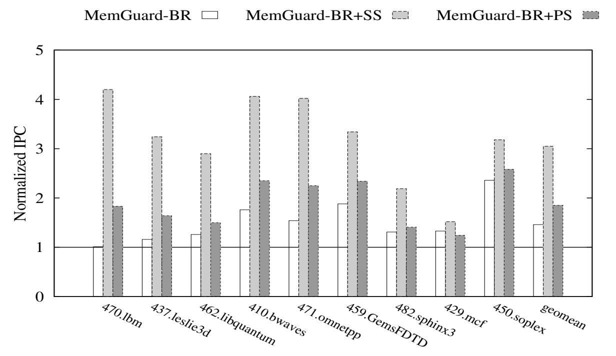


(c) MemGuard-BR+PS (b/w reclaim + proportional share)

Fig. 12: Normalized IPCs of SPEC2006 benchmarks.



(a) Subject tasks (X-axis)@Core0



(b) Interfering task (470.lbm)@Core2

Fig. 13: Normalized IPC of nine memory intensive SPEC2006 benchmarks (a) and the co-running 470.lbm (b). The X-axis shows the subject task on Core 0 in both sub-figures.

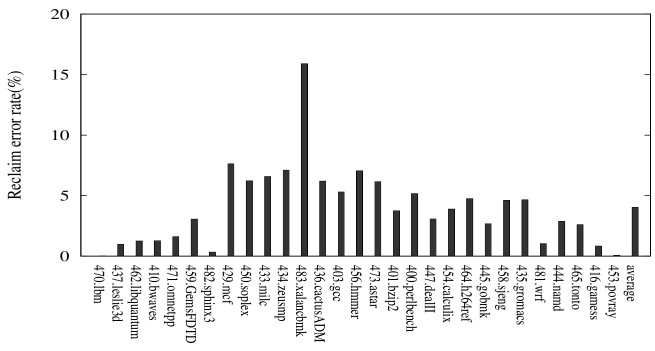


Fig. 14: Reclaim under-run error rate in MemGuard-BR mode

the effect of our bandwidth reclaiming algorithm. For most pairs, the interfering task achieves a higher IPC compared to the baseline (i.e., MemGuard-RO). This can be explained as follows: if the subject task does not use the assigned budget, the corresponding interfering task can effectively reclaim the unused budget and make more progress. In particular, the interfering tasks in the right side of the figure (from 433.milc on the X-axis) show significant performance improvements. This is because the corresponding subject task uses considerably smaller average bandwidth than their assigned budgets. Consequently, interfering tasks can reclaim more budgets and achieve higher performance. The average IPC of all interfering tasks is improved by 3.7x, compared to the baseline, showing the effectiveness of the reclaiming algorithm.

Note that the slowdowns of the subject tasks, due to reclaiming of the interfering task, are small—less than 3% on average. The slight performance reduction, i.e., reduced performance isolation, can be considered as a limitation of our prediction based approach that can result in reclaim under-run error as described in Section 4.3. To better understand this, Figure 14 shows reclaim under-run error rates (error periods / total periods) of the experiments used to draw Figure 12(a). On average, the error rate is 4% and the worst case error rate is 16% for 483.xalancbmk. Note that although 483.xalancbmk suffers higher reclaim under-run error rate, it

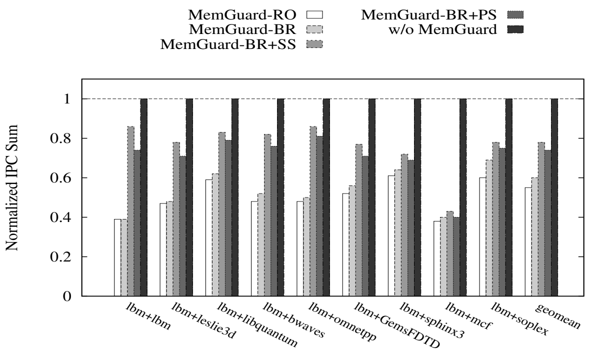
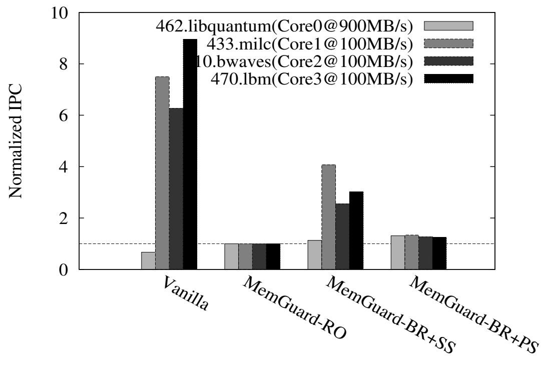
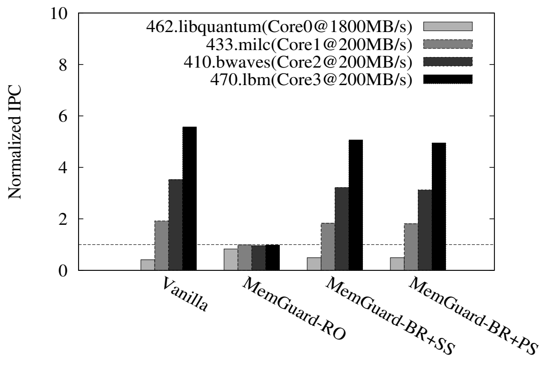


Fig. 15: Aggregated throughput comparison



(a) $r_{min}$=1.2GB/s



(b) $r_{min}$=2.4GB/s

Fig. 16: Isolation and throughput impact of $r_{min}$.

does not suffer noticeable performance degradation because the absolute difference between the reserved bandwidth and the achieved bandwidth is relatively small in most periods that suffered reclaim under-run errors.

MemGuard-BR+SS enables the spare bandwidth sharing algorithm (Section 5.2) on top of MemGuard-BR. Compared to Figure 12(a), the tasks in the left side of the figure—i.e., task pairs coupled with more memory intensive subject tasks—show noticeable improvements. This is because after both tasks (the subject task and the interfering task) collectively consume the total reserved bandwidth ($r_{min}$), the spare bandwidth sharing mode allows both tasks to continue until the beginning of the next period, making more progress on both tasks. On average, the performance is improved by 5.1x for interfering tasks and by 1.06x for subject tasks, compared to the baseline.

MemGuard-BR+PS enables the proportional sharing mode (Section 5.3) on top of MemGuard-BR. While it also improves performance of both subject and interfering tasks as in MemGuard-BR+SS, the average improvement of interfering tasks is only 4.3x, compared to 5.1x in MemGuard-BR+SS. On the other hand, the average improvement of subject tasks is 1.08x, compared to 1.06x in the MemGuard-BR+SS mode. This is because the proportional sharing mode provides much less bandwidth to the interfering tasks as it begins a new period immediately after the guaranteed bandwidth is consumed, while the spare sharing mode let the interfering task freely compete with the subject task until the next period begins, hence achieves more bandwidth.

The differences of the two modes—MemGuard-BR+SS and MemGuard-BR+PS—can be seen more clearly by investigating the "High" memory intensity subject tasks and the corresponding interfering tasks separately as shown in Figure 13. In all cases, the proportional sharing mode improves subject tasks' performance at the cost of reduced interfering tasks' performance. Hence, the proportional sharing mode is useful when we want to prioritize certain cores with more guaranteed bandwidth reservations over other cores with less reservations.

Figure 15 compares throughput of four MemGuard modes (MemGuard-RO, MemGuard-BR, MemGuard-BR+SS, and MemGuard-BR+PS) and the vanilla kernel without using MemGuard (Vanilla.) Here we define throughput simply as the sum of IPCs of each subject task and the interfering task pair. The Y-axis shows the normalized IPC sum (w.r.t. Vanilla) of each pair of subject tasks and interfering tasks that represents the system throughput of the pair. Compared to MemGuard-RO, MemGuard-BR achieves 11% more throughput on average (geometric mean). Both MemGuard-BR+SS and MemGuard-BR+PS achieve additional 11% and 9% improvement respectively. Although Vanilla achieves higher throughput in general, it does not provide performance isolation while MemGuard provides performance isolation at a reasonable throughput cost.

## 7.3 Results on Four Cores

In this experiment, we evaluate MemGuard using all four cores in our testbed. We use four SPEC benchmarks—462.libquantum, 433.milc, 410.bwaves, and 470.lbm—each of which runs on one core in the system.

Because the testbed has two shared LLC caches, each of which is shared by two cores, we carefully choose the benchmarks in order to minimize cache storage interference effect. To this end, we experimentally verify each benchmark by running it together with one synthetic cache trash task in both shared and separate LLC configurations; if performance of the two configurations differ less than 5%, we categorize the benchmark as LLC insensitive.

Figure 16(a) shows the normalized IPC (w.r.t. MemGuard-RO where each task is scheduled in isolation) of each task when all four tasks are co-scheduled using Mem-

Guard in three different modes (MemGuard-RO, MemGuard-BR+SS, and MemGuard-BR+PS) and without using MemGuard (Vanilla). The weight assignment is 9:1:1:1 (for 462.libquantum, 433.milc, 410.bwaves, and 470.lbm respectively) and the $r_{min}$ is 1.2GB/s. Vanilla is unaware of the weight assignment. Hence, the high-priority 462.libquantum on Core 0 is 33% slower than the baseline reservation due to contentions from other low priority tasks. Although it is clear that overall throughput is higher in Vanilla, it cannot provide isolated performance guarantee for one specific task, in this case 462.libquantum. In contrast, MemGuard-RO delivers exactly the performance that is promised by each task's reserved bandwidth (i.e., baseline) without experiencing noticeable slowdowns by interferences from co-running tasks. Hence it can guarantee performance of the high-priority task (462.libquantum) at the cost of significant slowdowns of low-priority tasks. MemGuard-BR+SS improves performance of all tasks beyond their guaranteed performances by sharing best-effort bandwidth—through bandwidth reclaiming and spare sharing. This is especially effective for low-priority tasks as they are improved by 4.07x, 2.55x, 3.02x (433.milc, 410.bwaves, 470.lbm respectively) compared to the baseline. The high-priority task (462.libquantum) is also improved by 1.13x. MemGuard-BR+PS also improves performance of all tasks above their guaranteed performances, but different in that it favors the high-priority task over low-priority tasks: the high-priority task is improved by 1.31x while low-priority tasks are improved by 1.34x, 1.27x, and 1.25x. This is because MemGuard-BR+PS enforces the assigned weight all the time, by starting a new period immediately when the guaranteed bandwidth is used, while MemGuard-BR+SS doesn't between the time it satisfies the guaranteed bandwidth and the time when the next period starts (the interval is fixed in MemGuard-BR+SS).

Figure 16(b) follows the same weight settings but doubles the $r_{min}$ value to 2.4GB/s in order to compare its effect on throughput and performance isolation. Because the $r_{min}$ is doubled, each core's reserved bandwidth is doubled and the baseline of each task (Y-axis value of 1) is changed accordingly. Note first that MemGuard-RO does not guarantee performance isolation anymore as 462.libquantum is 17% slower than the baseline. It is because the 2.4GB/s bandwidth can not be guaranteed by the given memory system, causing additional queuing delay to the 462.libquantum. This is consistent with our finding in Section 7.1. In both MemGuard-BR+SS and MemGuard-BR+PS, the IPC of 462.libquantum is further reduced, because other cores can generate more interference using reclaimed bandwidth that 462.libquantum donated. On the other hand, both modes achieve higher overall throughput as they behave very similar to Vanilla. This shows the trade-off between throughput and performance isolation when using MemGuard.

### 7.4 Effect on Soft Real-time Applications

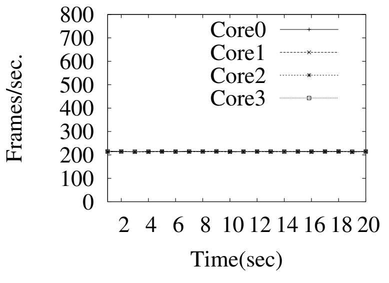
We illustrate the effect of MemGuard for soft real-time applications using a synthetic soft real-time image processing benchmark *fps*. The benchmark processes an array of two HD images (each image is 32bpp HD data: 1920x1080x4 bytes = 7.9MB) in sequence. It is greedy in the sense that it attempts to process as quickly as possible.

Figure 17 shows frame-rates of *fps* instances on our 4-core system using MemGuard in two different modes (MemGuard-BR+SS, MemGuard-BR+PS) and without using MemGuard (Vanilla). The weight assignment is 1:2:4:8 (for Core0,1,2,3 respectively) and the $r_{min}$ is 1.2GB/s. Vanilla is unaware of the weight assignment. Hence, all instances show almost identical frame-rates. MemGuard-BR+SS reserves memory bandwidth for each core, as long as the total sum of the reservations is smaller than the guaranteed bandwidth. Therefore the frame-rates are different depending on the amount of reservations. However, the ratio of frame-rates is different from the ratio of weight assignments because the total reserved bandwidth is small compared to the peak memory bandwidth; after all cores are served their reserved bandwidths, they equally compete for the remaining bandwidth. On the other hand, the frame-rate ratios in MemGuard-BR+PS are in line with the ratios of assigned weights. This is because MemGuard-BR+PS enforces the given bandwidth assignment all the time (by starting a new period immediately after cores use the $r_{min}$), resulting in better prioritization over MemGuard-BR+SS.

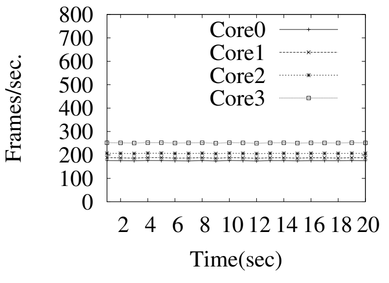
## 8 Related Work

Resource reservation has been well studied especially in the context of CPU scheduling [32], [1] and has been applied to other resources such as GPU [20], [21]. The basic idea is that each task or a group of tasks reserves a fraction of the processor's available bandwidth in order to provide temporal isolation. Abeni and Buttazzo proposed Constant Bandwidth Server (CBS) [1] that implements reservation by scheduling deadlines under EDF scheduler. Based on CBS, many researchers proposed reclaiming policies in order to improve average case performance of reservation schedulers [22], [10], [9], [23]. These reclaiming approaches are based on the knowledge of task information (such as period) and the exact amount of extra budget. While our work is inspired by these works, we apply reservation and reclaiming on memory bandwidth which is very different from CPU bandwidth in several key ways.

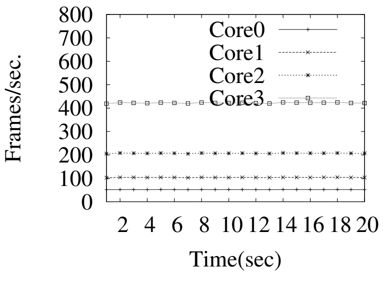
DRAM bandwidth is different from CPU bandwidth in the sense that the attainable capacity (bandwidth) varies significantly while it is a constant in case of CPU bandwidth. Unlike CPU, a DRAM chip consists of multiple resources called banks that can be accessed in parallel and each bank has its own state that affects required access time for the bank. Therefore, the maximum capacity (bandwidth) of DRAM fluctuates depending on the access pattern and the bank states. Some DRAM controllers, specially designed for predictable memory performance, solve this problem by forcing to access all DRAM banks—in a pipelined manner—at a time for each memory transaction, thereby eliminating the capacity variation [30], [3], [31]. Some of the DRAM controllers also provide native support for bandwidth regulation [3], [31], which allow them to analyze DRAM performance based on network calculus [12] as shown in [38]. While they may be



(a) Vanilla



(b) MemGuard-BR+SS



(c) MemGuard-BR+PS

Fig. 17: Frame-rate comparison. The weight assignment is 1:2:4:8 (Core0,1,2,3) and $r_{min}$ = 1.2GB/s for (b) and (c).

acceptable for specialized hardware architectures such as the TM3270 media processor [40], however, they require a very large transaction length to accommodate many DRAM banks. For example, a typical modern DRAM module consists of 8 to 16 banks to achieve high throughput. To support such a DRAM module, the DRAM controllers in [3], [31] would need a transaction length of 1KB (or 512B, depending on configurations), which is too big for most general purpose CPU architectures. Therefore, these predictable DRAM controller designs are not available in most COTS DRAM controllers.

Because COTS DRAM controllers typically do not provide any intrinsic mechanisms to provide memory performance guarantees, our main focus is to develop software (OS) mechanisms to provide better performance isolation on COTS systems. OS level memory access control was first discussed in literature by Bellosa [7]. Similar to our work, this work also proposed a software mechanism to prevent excess memory bandwidth usage of each core, by adjusting the number of idle loops in the TLB miss handler. There are, however, three major limitations, which are addressed in this work: First, it defines the maximum reservable bandwidth in an ad-hoc manner—i.e., 0.9 times of the measured bandwidth of the Stream benchmark [24]—that can be violated depending on memory access patterns as shown in Section 2. Second, it does not address the problem of wasted memory bandwidth in case cores do not use their reserved bandwidth. Finally, its bandwidth control is "soft" in the sense that it allows cores to overuse their reserved bandwidth until a feed-back control mechanism stabilizes the cores' bandwidth usages by increasing/decreasing the idle loops in the TLB handler. It is, therefore, not appropriate to apply the technique for hard real-time applications.

In contrast, our work clearly defines the maximum reservable bandwidth, the guaranteed bandwidth (Section 4), based on the understanding of DRAM architecture; can provide "hard" bandwidth reservation for hard real-time applications; and provides reclaiming and sharing mechanisms to better utilize the memory bandwidth while still providing each core's reserved bandwidth. A recent work also presented a similar memory bandwidth reservation system focusing on individual server oriented reservation, in contrast to our core oriented reservation approach [16]. However, they do not address the problem of wasted bandwidth, which is addressed in this work.

WCET analysis in multicore is difficult because service times of the shared resources depend on their internal states and access histories. Although there is an analytic method to deal with history dependent service times in unicore [5], such an approach may not be applicable in multicore because multiple cores can concurrently access the shared resources and software do not have enough control on accessing and scheduling of the shared resources.

We believe the basic reservation mechanism of MemGuard can be used with other multicore WCET analysis techniques ( [34], [6]) as shown in our previous work [42] in the context of mixed critical systems. A tighter WCET analysis technique, exploiting MemGuard's globally synchronized regulation, can be found in [41]. The focus of this paper is, however, not on the WCET analysis but on the development of practical reservation mechanisms that can benefit not only real-time applications but also non real-time applications.

## 9 Conclusion

We have presented MemGuard, a memory bandwidth reservation system, for supporting efficient memory performance isolation on multicore platforms. It decomposes memory bandwidth as two parts: guaranteed bandwidth and best effort bandwidth. Memory bandwidth reservation is provided for the guaranteed part for achieving performance isolation. An efficient reclaiming mechanism is proposed for effectively utilizing the guaranteed bandwidth. It further improves system throughput by sharing best effort bandwidth after each core satisfies its guaranteed bandwidth. It has been implemented in Linux kernel and evaluated on a real multicore hardware platform.

Our evaluation with SPEC2006 benchmarks showed that MemGuard is able to provide memory performance isolation under heavy memory intensive workloads. It also showed that the proposed reclaiming and sharing algorithms improve overall throughput compared to a reservation only system under time-varying memory workloads. As future works, we plan to apply our work on a 8-core P4080 platform [14], together with our industrial sponsor. We will also continue to reduce implementation overhead to accommodate more fine-grained real-time tasks.

## Acknowledgements

This research is supported in part by ONR N00014-12-1-0046, Lockheed Martin 2009-00524, Rockwell Collins RPS#6 45038, NSERC DG 402369-2011, NSF CNS-1302563, and NSF CNS-1219064. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the ONR, NSERC, NSF or the supporting companies.

## References

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

[2] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole. Analysis of a reservation-based feedback scheduler. In *Real-Time Systems Symposium (RTSS)*, 2002.

[3] B. Akesson, K. Goossens, and M. Ringhofer. Predator: a predictable SDRAM memory controller. In *Hardware/software codesign and system synthesis (CODES+ISSS)*, pages 251–256. ACM, 2007.

[4] 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.

[5] B. Andersson, S. Chaki, D. de Niz, B. Dougherty, R. Kegley, and J. White. Non-preemptive scheduling with history-dependent execution time. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

[6] B. Andersson, A. Easwaran, and J. Lee. Finding an upper bound on the increase in execution time due to contention on the memory bus in cots-based multicore systems. *SIGBED Review*, 7(1):4, 2010.

[7] 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.

[8] B. Bhat and F. Mueller. Making DRAM refresh predictable. *Real-Time Systems*, 47(5):430–453, 2011.

[9] M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Real-Time Systems Symposium (RTSS)*. IEEE, 2000.

[10] M. Caccamo, G. Buttazzo, and D. Thomas. Efficient reclaiming in reservation-based real-time systems. In *Real-Time Systems Symposium (RTSS)*, pages 198–213. IEEE, 2005.

[11] F. Cerqueira, M. Vanga, and B. Brandenburg. Scaling Global Scheduling with Message Passing. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[12] R. Cruz. A calculus for network delay. I. Network elements in isolation. *Transactions on Information Theory*, 37(1):114–131, 1991.

[13] T. Cucinotta, L. Abeni, L. Palopoli, and G. Lipari. A robust mechanism for adaptive scheduling of multimedia applications. *ACM Trans. Embed. Comput. Syst. (TECS)*, 10(4):46:1–46:24, Nov. 2011.

[14] Freescale. *P4080: QorIQ P4080/P4040/P4081 Communications Processors with Data Path*, 2014.

[15] J. Henning. SPEC CPU2006 Benchmark Descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[16] 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.

[17] Intel. *Intel ®64 and IA-32 Architectures Optimization Reference Manual*, April 2012.

[18] Intel. *Intel®64 and IA-32 Architectures Software Developer Manuals*, 2012.

[19] A. Jaleel. *Memory Characterization of Workloads Using Instrumentation-Driven Simulation*, 2010.

[20] S. Kato, K. Lakshmanan, Y. Ishikawa, and R. Rajkumar. Resource Sharing in GPU-accelerated Windowing Systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011.

[21] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. Time-Graph: GPU Scheduling for Real-Time Multi-Tasking Environments. In *USENIX Annual Technical Conference (ATC)*. USENIX, 2011.

[22] G. Lipari and S. Baruah. Greedy reclaimation of unused bandwidth in constant bandwidth servers. In *Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2000.

[23] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo. IRIS: A new reclaiming algorithm for server-based real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2004.

[24] J. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, December 1995.

[25] L. McVoy, C. Staelin, et al. Lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*. USENIX, 1996.

[26] Micron Technology, Inc. *1Gb DDR2 SDRAM: MT47H128M8, Rev. Z 03/14 EN*, 2003.

[27] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *International Symposium on Microarchitecture (MICRO)*, pages 146–160. IEEE, 2007.

[28] C. Natarajan, B. Christenson, and F. Briggs. A study of performance impact of memory controller features in multi-processor server environment. In *Proceedings of the 3rd workshop on Memory performance issues*, pages 80–87. ACM, 2004.

[29] K. Nesbit, N. Aggarwal, J. Laudon, and J. Smith. Fair queuing memory systems. In *International Symposium on Microarchitecture (MICRO)*, pages 208–222. IEEE, 2006.

[30] M. Paolieri, E. Quiñones, J. Cazorla, and M. Valero. An analyzable memory controller for hard real-time cmps. *Embedded Systems Letters, IEEE*, 1(4):86–90, 2009.

[31] M. Pérez, C. Rutten, L. Steffens, J. van Eijndhoven, and P. Stravers. Resource reservations in shared-memory multiprocessor socs. In *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, pages 109–137. Springer, 2005.

[32] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Multimedia Computing and Networking (MNCN)*, January 1998.

[33] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. Owens. Memory access scheduling. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 128–138. ACM, 2000.

[34] S. Schliecker and R. Ernst. Real-time performance analysis of multiprocessor systems with shared memory. *ACM Transactions on Embedded Computing Systems (TECS)*, 10(2):22, 2010.

[35] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Real-Time Systems Symposium (RTSS)*. IEEE, 2003.

[36] D. Skourtis, S. Kato, and S. Brandt. QBox: Guaranteeing I/O performance on black box storage systems. In *High-Performance Parallel and Distributed Computing (HPDC)*, pages 73–84. ACM, 2012.

[37] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.

[38] L. Steffens, M. Agarwal, and P. Wolf. Real-time analysis for memory access in media processing SOCs: A practical approach. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 255–265. IEEE, 2008.

[39] P. Valente and F. Checconi. High throughput disk scheduling with fair bandwidth distribution. *Transactions on Computers*, 59(9), 2010.

[40] J. Waerdt et al. The TM3270 media-processor. In *International Symposium on Microarchitecture (MICRO)*, pages 331–342. IEEE, 2005.

[41] G. Yao, H. Yun, Z. Wu, R. Pellizzoni, M. Caccamo, and L. Sha. Schedulability Analysis for Memory Bandwidth Regulated Multicore Real-Time Systems. *Transactions on Computers*, December 2014 (accepted).

[42] 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.

[43] 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.

[44] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory Bandwidth Management for Efficient Performance Isolation in Multi-core Platforms. Technical report, University of Kansas, 2014.