MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-core Platforms

Heechul Yun[‡], Gang Yao[‡], Rodolfo Pellizzoni[⋆], Marco Caccamo[‡], Lui Sha[‡]
[‡] University of Illinois at Urbana-Champaign, USA. {heechul,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 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 distinguishes memory bandwidth as 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.**

I. Introduction

Computing systems are increasingly moving toward multi-core platforms and their memory subsystem represents 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 more 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 [16]. All these factors affect the temporal predictability of memory intensive real-time applications due to the high variance of their memory access time. Therefore, there is an increasing need for memory bandwidth management solutions that provide Quality of Service (QoS).

This problem has already been recognized by many researchers, and recent work has focused on designing more predictable memory controllers. For example, a reservation based approach has been applied to design a DRAM controller that supports real-time features [4]. Resource reservation and reclaiming techniques [17], [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. Proposed techniques have been successfully applied to CPU management [10], [7], [6] and more recently to GPU management [14], [15].

Unfortunately, existing solutions cannot be easily used for managing memory bandwidth due to inherent limitations. An increasing number of real-time systems are built with Commercial Off-The-Shelf (COTS) components, and hardware-based solutions are not generally possible in such systems. Moreover, state-of-art resource reservation solutions [7], [8] 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.

In our previous work, we investigated memory bandwidth reservation at the operating system level [21]. However, our existing solution has significant limitations. First of all, it assumes a constant available memory bandwidth, which is not true in DRAM-based systems. Second, it can not adapt to dynamic changes in memory resource usage. Third, while the work in [21] provides safe performance guarantees for hard real-time tasks, it makes no effort to optimize memory throughput for soft real-time tasks, possibly resulting in severely reduced system performance. To address these limitations and challenges, we propose a new, efficient and fine-grained memory bandwidth management system, which we call *MemGuard*.

Unlike CPU bandwidth reservation, under MemGuard the available memory bandwidth can be described as having two components: *guaranteed* and *best effort*. The guaranteed bandwidth represents the minimum service rate the DRAM system can provide, while the additionally available bandwidth is best effort and can not be guaranteed by the system. Memory bandwidth reservation is based on the guaranteed part in order to achieve temporal isolation. However, to efficiently utilize all the guaranteed memory bandwidth, a reclaiming mechanism is proposed leveraging each core's usage prediction. The system throughput is further improved by exploiting 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. We evaluate the performance of MemGuard under different configurations and we present detailed results in the evaluation section.

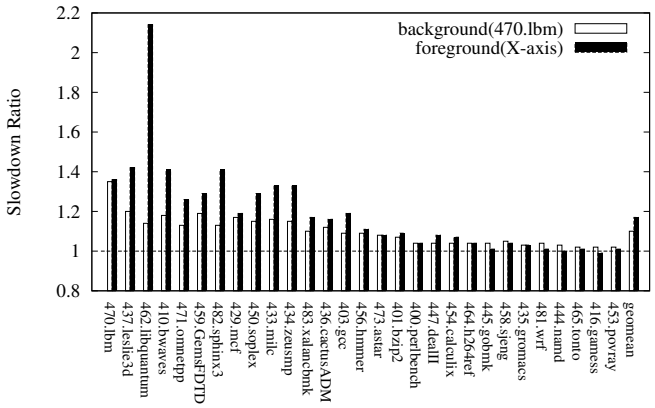
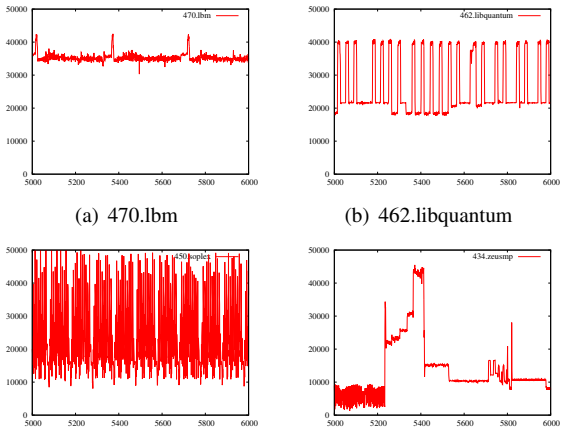


Fig. 1: IPC slowdown of foreground (X-axis) and background task (470.lbm) on a dual-core configuration



(a) 470.lbm (b) 462.libquantum

(c) 450.soplex (d) 434.zeusmp

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

In summary, the contributions of this work are: (1) decomposing overall memory bandwidth into a guaranteed and a best effort components. Then, we experimentally identify the boundary so we can apply the proposed reservation technique; (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 [11] and showing its effectiveness on a real multi-core hardware platform.

The remaining sections are organized as follows: Section II describes the challenge of predictability in modern multi-core systems. Section III describes the details of the proposed MemGuard approach. Section IV describes the evaluation platform and the software implementation. Section V presents the evaluation results. Section VI discusses related work. We conclude in Section VII.

II. Problems of Shared Memory in Multi-core 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 multi-core 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 reservation solution in handling memory bandwidth. To illustrate the problems, we performed two set of experiments on a real multi-core hardware (described in Section IV-A).

In the first experiment, we measured Instructions-Per-Cycle (IPC) for each SPEC2006 benchmark (foreground task) first running on a core in isolation, and then together with a memory intensive benchmark (470.lbm) running on a different core (background task). Figure 1 shows IPC slowdown ratio (run-alone IPC/co-scheduled IPC) of both foreground and background tasks; foreground 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 foreground and the background task suffer slowdown since they are interfering each other in the memory accesses, which is expected. Interestingly, however, most of the times the foreground task slowdown more than the background task, and the difference of slowdown factors between that two tasks could be as large as factor of two (2.2x against 1.2x). Furthermore, note that the slowdown factor is not necessarily proportional to how memory intensive the foreground task is. Such effects are typical in COTS systems due to the characteristics of modern DRAM controllers [16]: (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, sequential accesses to the same row are much more efficient that random accesses to different rows within a bank. DRAM controllers commonly implement scheduling algorithms that re-order requests depending on the DRAM state and the backlogged requests inside the DRAM controller, in order to maximize throughput [16]. 470.lbm tends to suffer less slowdown than foreground tasks because it is memory intensive and it floods the request queue in the DRAM controller with sequential access requests. These results indicate that memory bandwidth is very different compared to CPU bandwidth, in the sense that maximum achievable bandwidth is not fixed but highly variable depending on 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 using hardware Performance Measuring Counters (PMC) between time 5 to 6 seconds, sampled over every 1ms time interval. 470.lbm shows highly 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 results in poor resource utilization and poor performance. If the resource is reserved with the maximum request, it would lead to significant waste as the task usually does not consume that amount; while if it is with the average value, the task would suffer slowdown whenever it tries to access more than that average value during one sampling period. 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. 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 [6], [10]. 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 memory bandwidth resource to one specific core (a.k.a resource reservation) to provide predictable and guaranteed worst-case behavior; and (2) provide some dynamic resource adjustment on the resource provision (a.k.a resource reclaiming) to efficiently exploit varying system resources and improve task responsiveness.

III. MemGuard

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 multi-core system can then be considered as a set of uni-core systems, each of which has a dedicated, albeit slower, memory subsystem. This notion of isolation is commonly achieved through resource reservation approaches in many real-time literature [1] mostly in the context of CPU bandwidth reservation.

In order to achieve performance isolation, MemGuard regulates each core's memory request rate. Specifically, we regulate the total sum of request rates among all cores to be less than the minimum service rate of the DRAM subsystem, denoted as r_{min}. Our observation is that by regulating the aggregated requests rate to the DRAM controller, we can minimize the potential delay in the DRAM controller as requests are likely to be processed immediately. This is 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. In our case, the memory bandwidth available for reservation is restricted to the minimum DRAM service rate as it would make memory access delay caused by concurrent accesses from other cores negligible.

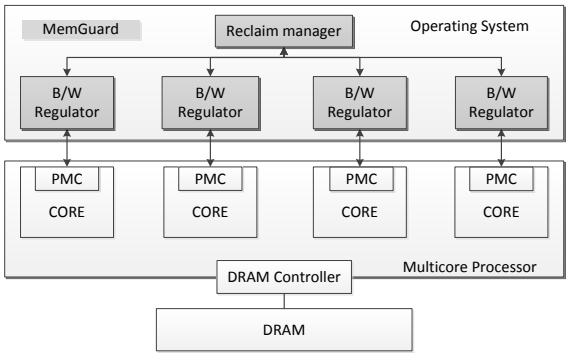


Fig. 3: MemGuard system architecture.

A. System Architecture

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 budget. The reclaim manager maintains a global shared reservation for receiving and re-distributing the budget for all regulators in the system.

In this paper, we focus on systems that 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 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 subsections.

B. Memory Bandwidth Reservation

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

```
1  function periodic_timer_handler ;
2  begin
3      Q_i^predict ← output of usage predictor ;
4      Q_i ← user assigned static budget ;
5      if Q_i^predict > Q_i then
6          q_i ← Q_i;
7      else
8          q_i ← Q_i^predict ;
9      G += max{0, Q_i − q_i};
10     program PMC to cause overflow interrupt at q_i;
11     re-schedule all dequeued tasks;
12 function overflow_interrupt_handler ;
13 begin
14     u_i ← used budget in the current period ;
15     if G > 0 then
16         if u_i < Q_i then
17             q_i ← min{Q_i − u_i, G} ;
18         else
19             q_i ← min{Q_min, G} ;
20         G -= q_i;
21         program PMC to cause overflow interrupt at q_i ;
22         Return ;
23     if u_i < Q_i then
24         Return ;
25     if Σ u_i = r_min then
26         wake up all cores ;
27         Return ;
28     de-schedule tasks in the CPU run-queue ;
```

Fig. 4: MemGuard Implementation

1) System-wide reservation regulates the total allowed memory bandwidth such that it does not exceed the minimum DRAM service rate r_{min}, providing performance isolation on the reserved bandwidth. We describe an experimental method to estimate r_{min} in Section V-A.
2) Per-core reservation assigns a fraction of r_{min} to each core, hence each core reserves bandwidth B_i and $r_{min} = \sum_{i=0}^{m} B_i$.

Each regulator reserves memory bandwidth represented by memory access budget Q_i for every period P, i.e., $B_i = \frac{Q_i}{P}$. Regulation period P is a system-wide parameter and should be small to effectively enforce specified memory bandwidth. In our current implementation, P is equal to a scheduler tick length as MemGuard is called at each scheduler tick handler. 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 scheduler tick as 1ms. The reservation follows the common resource reservation rules [17], [19], [20]. Per-core instant budget q_i is deducted as the core consumes memory bandwidth. 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 q_i is depleted for Core i. Once the interrupt is received, the regulator calls the OS scheduler to dequeue all tasks from the run-queue of the core so that the core can not access memory any more until the next period. At the beginning of the next period the budget is replenished in full and all dequeued tasks, if any, are enqueued to the run-queue.

C. 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; the q_i can vary at each period depending on predictor and the actual bandwidth usage of the core. The predictor uses an Exponentially Weighted Moving Average (EWMA) filter to estimate the predicted memory usage $Q^{predict}$ of the current period; it 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 at the beginning of each period and any unused G is discarded at the end of this period. 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 5-8 in Figure 4).
2) At the beginning of each regulation period, the global budget G is updated as follows:

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

Each core donates its spare budget to G (See Line 9 in Figure 4).
3) 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 static reserved bandwidth Q_i (this happens when the prediction is smaller than Q_i), then it tries to reclaim amount equal to the difference between Q_i and the current usage u_i; if the core used equal or greater than the assigned budget Q_i, it only gets up to a small fixed amount of budget, Q_{min} (See Line 16-19 in Figure 4). If Q_{min} is too small, too many interrupts can be generated within a period, increasing overhead. As such, it is a configuration parameter and empirically determined for each particular system.
4) Since our reclaim algorithm is based on prediction, it is possible a core may not be able to use the originally assigned budget Q_i. This can happen when the core donates its budget too much (due to mis-prediction) and

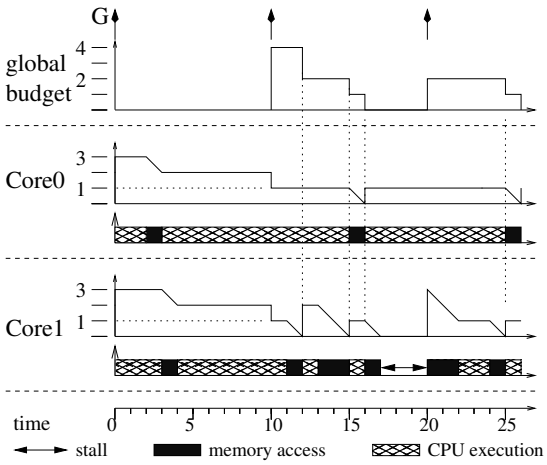


Fig. 5: An illustrative example with two cores

other cores already reclaimed all the donated budget (i.e., $G = 0$) before the core tries to reclaim. When this happens, our current heuristic is to allow the core continue execution, hoping that it may use its Q_i, although it is not guaranteed (See Line 23-24 in Figure 4). At the beginning of the next period, we verify if it was able to use the budget. If not, we call it a *reclaim underrun 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.

D. Spare Memory Bandwidth Sharing

Since system-wide reservation only accounts up to the minimal service rate r_{min} (i.e., guaranteed bandwidth), we try to exploit the spare memory bandwidth exceeding r_{min} (i.e., best-effort bandwidth) whenever it is possible, in order to improve the overall system throughput.

When all cores collectively use their assigned budgets, the remaining time (until the next period begins) is considered as spare time and MemGuard tries to maximize memory throughput by allowing cores to continue—we call it spare memory bandwidth sharing (See Line 25-27 in Figure 4)

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

E. Example

Figure 5 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 demonstrates the global budget together with these two cores.

When the system starts, each core starts with the assigned budget 3. At time 10, the prediction for each core is 1 as it only used budget 1 within the period [0,10], hence, the instant budget becomes 1 and the global budget G becomes 4 (each core donates 2). At time 12, Core 1 depletes its instant budget. Since its assigned budget is 3, Core 1 tries to reclaim 2 from G and G becomes 2. At time 15, Core 1 depletes its budget again. This time Core 1 already used its assigned budget, only a fixed amount of extra budget (Q_{min}) 1 is reclaimed from G and G becomes 1. At time 16, Core 0 depletes its budget. Since G is 1 at this point, Core 0 only reclaims 1 and G drops to 0. At time 17, Core 1 depletes its budget again then it dequeues all the tasks as it can not reclaim additional budget from G. When the third period starts at time 20, the $Q_1^{predict}$ is larger than Q_1. Therefore, Core 1 gets the full amount of assigned budget 3, according to Rule 1 in Section III-C, while Core 0 only gets 1, and donates 2 to G. At time 25, after Core 1 depletes its budget, Core 1 reclaims an additional budget Q_{min} from G.

IV. Evaluation Setup

In this section, we introduce the platform used for the evaluation and the software system implementation.

A. Evaluation platform

Figure 6 shows the architecture of our testbed, an Intel Core2Quad Q8400 processor. The processor is clocked at 2.66GHz and has four physical cores. It contains two separate 2MB L2 caches; each L2 cache is shared between two cores.

As our focus is not the shared cache, we use cores that do not share the same LLC for experiments (i.e., Core 0 and Core 2 in Figure 6). 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).

In order to account per-core memory bandwidth usage, we use a LLC miss performance counter per each core. Since the LLC miss counter does not account prefetched memory traffic, we disabled all hardware prefetchers [1]. 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 [13]); 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 [12], 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.

B. Software Implementation

We implemented MemGuard in Linux version 3.6. Most code is developed as a kernel module with small modifications in the core kernel scheduler [2]. We use the kernel scheduler tick handler for the period handler. We use the *perf_event* infrastructure to install the counter overflow handler at each period. The logic of both handlers is shown in Figure 4.

When an overflow interrupt occurs (budget depletion), the overflow handler may dequeue all runnable tasks in that core

[1]We used http://www.eece.maine.edu/~vweaver/projects/prefetch-disable/

[2]MemGuard is available at https://github.com/heechul/memguard.

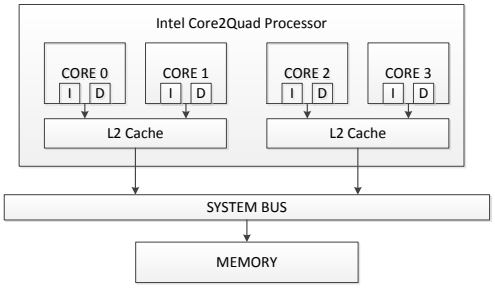


Fig. 6: Hardware architecture of our evaluation platform.

(in case reclaiming fails). The PMC programming overhead is negligible as it only requires a write to a core local register. The overhead of dequeuing/enqueueing tasks is small (around two microseconds) when the number of runnable tasks is small. The overhead, however, grows linearly as the number of runnable tasks increase. For example, with 10 runnable tasks, we observe around 10us overhead (i.e., 1% of the 1ms period). Although we argue that this is not a big problem in embedded systems where the number of running tasks is typically small, we plan to investigate ways to further minimize the overhead.

MemGuard supports several memory bandwidth reservation modes, namely per-core bandwidth assignment and per-task assignment mode. In per-core mode, system designers can assign absolute bandwidth (e.g., 200MB/s) or relative weight expressing 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). In per-task mode, the task priority is used as weight for the core the task runs on. Notice that in this mode, a task can migrate to a different core with its own memory bandwidth reservation. In Section V, we use per-core assignment with both absolute bandwidth mode and weight mode depending on experiments.

V. Evaluation Results and Analysis

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

A. Guaranteed Memory Bandwidth

We first experimentally measure the *guaranteed bandwidth*, which is introduced in Section III, and the peak bandwidth with two synthetic benchmarks: *latency* and *bandwidth*.

The latency benchmark is designed to access memory in a very inefficiently way. It iterates one linked list that is permuted randomly over a chunk of memory whose size is much bigger than the LLC size. Pointer chasing is inefficient because data must be fetched to move on to the next location, effectively enforcing serial accesses to the DRAM. Furthermore, as the linked list is randomly permuted, the next location is likely to hit different row in the DRAM, further reducing the access rate. Finally, caches do not help much here because the working set size is much larger than the LLC size. Therefore, we consider that this benchmark effectively represents the worst case memory access pattern.

	aggregated bandwidth (MB/s)		
benchmark	1 core	2 cores	4cores
latency	683	1185	1198
bandwidth	6190	8478	8490

TABLE I: Aggregated memory bandwidth.

On the other hand, the bandwidth benchmark accesses memory in sequence with no data dependency between consecutive accesses. This allows the CPU generate multiple memory requests in parallel, maximizing memory level parallelism (MLP) available in the memory system; a typical DRAM module consists of 4-8 DRAM chips each of which also contains 4-8 banks that can be accessed in parallel.

In order to stress the DRAM controller as much as possible, we assign one benchmark instance to each core and increase the number of cores from 1 to 4. Table I shows the aggregated memory bandwidth of all cores. First, note that the memory access pattern significantly affects achieved bandwidth. The latency benchmark running on a single core achieved only 0.6GB/s, while the bandwidth benchmark achieved 6GB/s. As we increase the number of cores, the aggregated bandwidths are increased in both benchmarks due to increased MLP. If all requests from all cores are directed to the same DRAM chip and the same DRAM bank, achieved bandwidth can be close to 0.6GB/s as MLP would be totally eliminated. On a typical multi-programmed workload, however, such possibility would be very low. We found 1.2GB/s is a practical minimum service rate for our platform and used it as the r_{min} for configuring MemGuard in the rest of the evaluation unless otherwise noted.

B. Isolation Effect of Reservation

In this experiment, we illustrate the effect of memory bandwidth reservation on performance isolation guarantee. We pair the most memory intensive benchmark, 470.lbm as the background task, with a foreground task selected from SPEC2006 benchmarks. Each foreground task runs on Core 0 with 1.0GB/s memory bandwidth reservation while the background task runs on Core 2 with reservation varying from 0.2GB/s to 2.0GB/s. Note that assigning more than 0.2GB/s on Core 2 makes the total bandwidth exceeds the estimated minimum DRAM service rate of 1.2GB/s. We disable both reclaim and spare bandwidth sharing mode so that each task can not use more than its assigned budget, we call this reservation only mode.

Figure 7 shows the IPC of each foreground task, normalized to the IPC measured in isolation (i.e., no background task) with the same 1.0GB/s reservation. First, notice that when we assign 0.2GB/s to Core 2 (denoted "w/ lbm:0.2G"), the IPC of each task is very close to the ideal value 1.0—i.e., no negative performance impact from the co-running background task. However, as we increase the memory bandwidth of Core 2, the IPC of the foreground task gradually decrease below 1.0—i.e., performance isolation is violated due to increased memory contention. For example, 462.libquantum on Core0 shows 30% IPC reduction when the background task is running on Core2 with 2.0GB/s reservation (denoted "w/ lbm:2.0G").

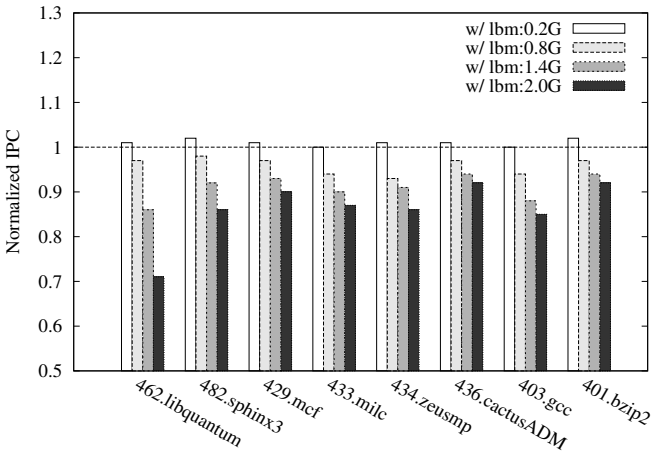


Fig. 7: Normalized IPC of a subset of SPEC2006 (Core 0), co-scheduled with 470.lbm (Core 2)

This results demonstrate that performance isolation can be achieved by regulating the aggregated total request rate. Specifically, limiting the rate to be smaller than r_{min} achieves performance isolation for the SPEC benchmarks shown in this figure. The rest of SPEC benchmarks also show consistent behavior but we omit them for space limitation.

C. Results with SPEC2006 Benchmarks on Two Cores

We now evaluate MemGuard using the entire SPEC 2006 benchmarks. We first profile the benchmarks to better understand of their characteristics. We run each benchmark for 10 seconds with the reference input and measure the instruction counts and LLC miss counts, using *perf* tool included in the Linux kernel source tree, to calculate the average IPC and the memory bandwidth usage. We multiply the LLC miss count with the cache-line size (64 bytes in our testbed) to get the total memory bandwidth usage.

Table II shows the results in decreasing order of average memory bandwidth usage, when each benchmark runs alone in 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).

1) Effects of Reclaiming and Spare Bandwidth Sharing: In this experiment, we evaluate the effect of reclaim and spare bandwidth sharing modes to the performance (measured in IPC) of co-scheduled foreground and background tasks (Core 0 and 2 respectively). We configure r_{min} as 1.2GB/s and assign a memory bandwidth weight of five on Core 0 and one on Core 2 (i.e., 1000MB/s for Core 0 and 200MB/s for Core 2).

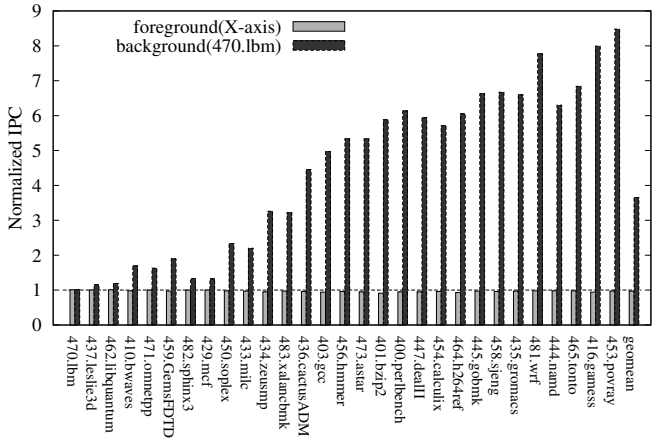
Figure 8 shows the normalized IPCs of each pair of foreground (X-axis) and background (470.lbm) tasks, w.r.t. IPCs measured in isolation with the same memory reservations, for (a) reclaim, (b) spare bandwidth sharing, and (c) reclaim+spare bandwidth sharing. The X-axis shows foreground tasks, sorted in decreasing order of memory intensity. Note that Core 2, which runs the background task, is severely under-reserved; only 200MB/s is reserved while the task's average bandwidth is above 2GB/s in Table II.

Benchmark	Avg. IPC	Avg. B/W(MB/s)	Memory Intensity
470.lbm	0.52	2121	
437.leslie3d	0.51	1581	
462.libquantum	0.60	1543	
410.bwaves	0.62	1485	
471.omnetpp	0.83	1373	High
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	
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	Medium
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	
458.sjeng	1.10	74	
435.gromacs	0.86	60	
481.wrf	1.73	38	Low
444.namd	1.47	18	
465.tonto	1.38	2	
416.gamess	1.34	1	
453.povray	1.17	1	

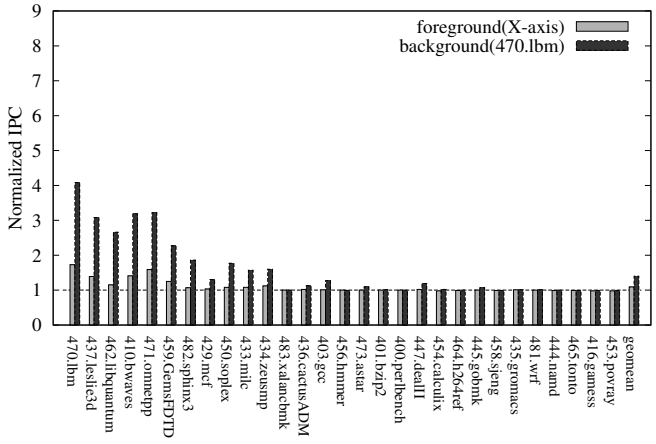
TABLE II: SPEC2006 characteristics

In this configuration, we are particular interested to see (1) how the background task would improve performance and (2) how the foreground task would be affected (ideally should not be affected by the co-running background task).

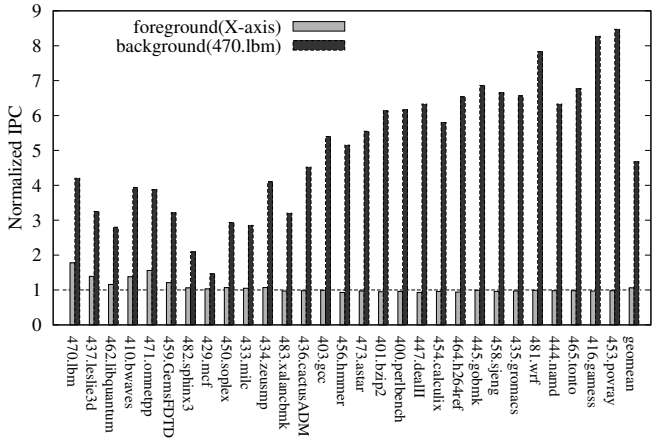
Figure 8(a) shows the effect of bandwidth reclaiming. For most pairs, the background task achieves a higher IPC w.r.t. running alone under the same reservation without reclaiming. This can be explained as follows: if the foreground task does not use the assigned budget, the background task can effectively reclaim the unused budget and make more progress. In particular, the background tasks in right side of the figure show significant performance improvements (from 433.milc on the X-axis). This is because the corresponding foreground tasks use considerably smaller average bandwidth than the assigned budget. Consequently, the background tasks can reclaim more budget and achieve higher performance. The average IPC of each background task is improved by 3.8x, compared to the case when it runs alone under 0.2 GB/s bandwidth reservation. Note that the slowdown of foreground task, due to reclaiming of background task, is 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 which can result in reclaim underrun error as described in Section III-C. To better understand this, Figure 9 shows the reclaim underrun error rate (error periods / total periods) for Figure 8(a). On average, the error rate is 4% and the worst case error rate is 16% for 483.xalancbmk. Although 483.xalancbmk suffers higher reclaim underrun error rate, it



(a) Reclaim



(b) Spare Share



(c) Reclaim+Spare Share

Fig. 8: Effect of Reclaim, Spare Sharing, and Reclaim+Spare Share

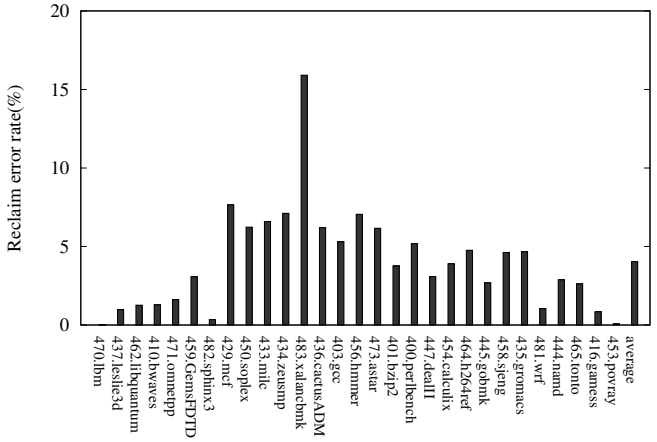


Fig. 9: Reclaim underrun error rate

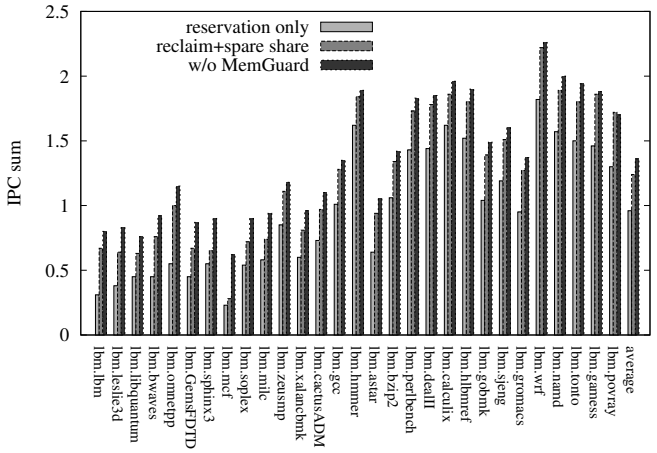
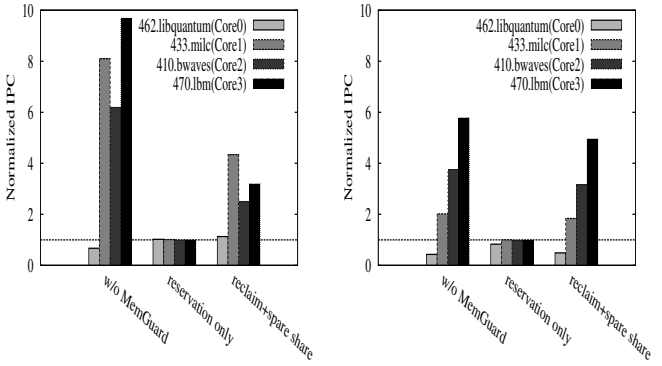


Fig. 10: Throughput comparison

does not suffer noticeable performance degradation, because the absolute difference between the reserved bandwidth and the achieved bandwidth is relatively small in most of the reclaim underrun error periods.

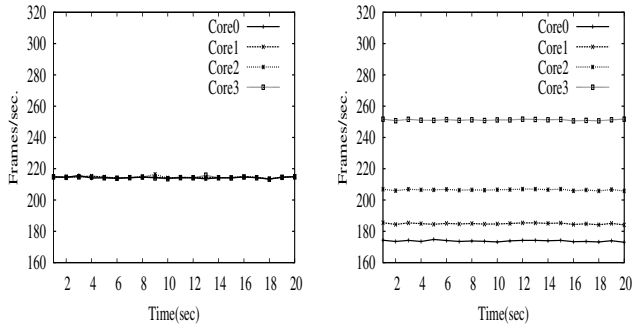
Figure 8(b) shows the effect of spare bandwidth sharing. Notice that the background tasks on the left side of the figure show more significant improvements. This is because when both tasks are highly memory intensive, the reserved bandwidth of each core is consumed more quickly. After using the total reserved bandwidth, r_{min}, the spare bandwidth sharing mode allows both tasks make more progress by allowing them to continue to execute. Note that the spare bandwidth sharing mode is also beneficial to the foreground tasks, especially the memory intensive ones, as they can receive more budget while they still exclusively use their reserved budgets. On average, the performance is improved by 40% for background tasks and by 9% for foreground tasks.

Figure 8(c) shows the effect of using both reclaim and spare bandwidth sharing. It can be considered roughly as the combination of the two previous results: the background task gets speedup either from reclaiming or spare bandwidth sharing. The performance of background tasks is improved



(a) r_{min}=1.2GB/s (b) r_{min}=2.4GB/s

Fig. 11: Isolation and Throughput Impact of r_{min}



(a) w/o MemGuard (b) with MemGuard

Fig. 12: Frame-rate comparison

by 368% (i.e., 4.68x speedup) on average. This shows the effectiveness of our approach.

Finally, Figure 10 compares throughput with and without using MemGuard (both reclaim and spare bandwidth sharing modes are enabled). The Y-axis shows the IPC sum of each pair of foreground and background tasks that represents the system throughput of the pair. Although the system without MemGuard achieves higher throughput in general, it does not provide performance isolation. MemGuard provides performance isolation at a reasonable throughput cost.

D. Results with SPEC2006 on Four Cores

In this experiment, we evaluate MemGuard on four cores using 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 11(a) shows the normalized IPC of each task when all four tasks are co-scheduled (1) without using MemGuard, (2) MemGuard with reservation only, and (3) MemGuard with both reclaiming and spare bandwidth sharing. The Y-axis is normalized to the IPC measured in isolation under MemGuard with reservation only mode. The r_{min} is 1.2GB/s and the weight assignment of each core is 9:1:1:1 for Core 0-3 (i.e., 900MB/s for Core0, 100MB/s for Core1-3).

The first group of bars, without MemGuard, shows that 462.libquantum on Core 0 is 33% slower than the baseline reservation only case while the other three tasks have much higher IPCs. Although it is clear that overall throughput is better without using MemGuard, it can not provide isolated performance guarantee for one specific task, in this case 462.libquantum. The second group, MemGuard with reservation, shows that each task achieves its performance goal set by its reserved memory bandwidth, as the IPC of each benchmark is nearly identical to the one of running alone. The final group, MemGuard with reclaim and spare bandwidth sharing, shows that performance of all three tasks at Core 1,2 and 3 are considerably improved without hurting the performance of 462.libquantum.

Figure 11(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. The tendency is that the performance of each task on Core 1, 2, and 3 improves at the cost of reduced performance of 462.libquantum at Core 0. Under MemGuard with reservation only, 462.libquantum is slowed by 17% compared to running alone under the same reservation. In other words, reservation does not provide performance isolation anymore due to memory contention. This is consistent with our finding in Section V-B. Under MemGuard with reclaim and spare bandwidth sharing mode, the IPC of 462.libquantum is further reduced, because other cores can generate more interference using reclaimed bandwidth that 462.libquantum donated. This shows the trade off between throughput and performance isolation when using MemGuard.

E. Effect on Soft Real-time Applications

In this experiment, we illustrate the effect of MemGuard for soft real-time applications. We run four instances a synthetic soft real-time image processing benchmark *fps*, one for each core, and compare the average frame-rate of each instance with and without using MemGuard. The benchmark processes an array of two HD images (each is 32bpp HD image data: 1920x1080x4 bytes = 7.9MB) in sequence. It is greedy in the sense that it attempts to process as quickly as possible. With MemGuard, we assign different weights of 1, 2, 4, and 8 for each core respectively, set r_{min} = 1.2GB/s, and enable spare bandwidth sharing mode to utilize spare bandwidth effectively.

Figure 12(a) shows the frame-rate of each fps instance on each core without using MemGuard. As they all access memory in a same way, they get the same fraction of memory bandwidth; hence resulting almost identical frame-rates. Figure 12(b) shows the frame-rates with MemGuard. As we assign different weights for each core, each instance shows different frame-rates, demonstrating the effect of MemGuard.

VI. Related Work

Resource reservation has been well studied especially in the context of CPU scheduling [17], [1] and has been applied to other resources such as GPU [14], [15]. 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 [10], [7], [6]. 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 reclaiming on memory bandwidth which is very different from CPU bandwidth in many ways.

DRAM bandwidth is different from CPU bandwidth in the sense that achieved bandwidth depends on the DRAM state and access history which makes it difficult to guarantee performance. To solve this problem, several DRAM controllers were proposed. Akesson et al. proposed Predator DRAM controller that uses combination of regulators and credit based scheduler to provide performance guarantee among multiple hardware components that access the DRAM [4], [3]. Reineke et al. proposed PRET DRAM controller that partitions the physical address space based on the internal structure of the DRAM chip in order to eliminate contention caused by sharing such internal resource [18]. Also in more general purpose computing systems, DRAM controller is studied in order to improve fairness and throughput. Nesbit et al. applied the network fair queuing theory in designing DRAM controller [16]; Ebrahimi et al. proposed a cache controller level throttling mechanism [9], which is similar to our method in the sense that it effectively changes request rates. While these DRAM controllers provide solutions to improve predictability and isolation in hardware level, we focus on a software level solution that can be applied to commodity hardware platforms.

OS level memory access control was first discussed in literature by Bellosa [5]. Similar to our work, his work also provide memory bandwidth reservation through an OS level mechanism (by inserting idle loops in the TLB miss handler). Unlike to our work, however, his work provide neither sound definition of maximum reservable bandwidth nor reclaiming of unused bandwidth, which may result in poor memory bandwidth utilization.

VII. Conclusion

We have presented MemGuard, a memory bandwidth reservation system, for supporting efficient memory performance isolation on multi-core 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 exploiting best effort bandwidth after each core satisfies its guaranteed bandwidth. It has been implemented in Linux kernel and evaluated on a real multi-core hardware platform.

Our evaluation with SPEC2006 benchmarks showed that MemGuard is able to provide memory performance isolation under heavy memory intensive workloads on multi-core platforms. It also showed that the proposed reclaiming algorithm improves overall throughput compared to a reservation only system under time-varying memory workloads.

Acknowledgements

This research is supported in part by ONR N00014-12-1-0046, Lockheed Martin 2009-00524, Rockwell Collins RPS#6 45038, NSERC, and NSF A17321.

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 *HW/SW codesign and system synthesis (CODES+ISSS)*, 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)*, 2008.

[5] 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.

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

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

[8] T. Cucinotta, L. Abeni, L. Palopoli, and G. Lipari. A robust mechanism for adaptive scheduling of multimedia applications. *ACM Trans. Embed. Comput. Syst.*, 10(4), November 2011.

[9] E. Ebrahimi, C.J. Lee, O. Mutlu, and Y.N. Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. *ACM Sigplan Notices*, 45(3), 2010.

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

[11] J.L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4), 2006.

[12] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. http://download.intel.com/products/processor/manual/325383.pdf.

[13] A. Jaleel. Memory characterization of workloads using instrumentation-driven simulation. *http://www. glue. umd. edu/ajaleel/workload*, 2010.

[14] 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.

[15] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *USENIX Annual Technical Conference*, 2011.

[16] K.J. Nesbit, N. Aggarwal, J. Laudon, and J.E. Smith. Fair queuing memory systems. In *International Symposium on Microarchitecture (MICRO)*, 2006.

[17] 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)*, 1998.

[18] J. Reineke, I. Liu, H.D. Patel, S. Kim, and E.A. Lee. Pret dram controller: Bank privatization for predictability and temporal isolation. In *HW/SW codesign and system synthesis (CODES+ISSS)*. ACM, 2011.

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

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

[21] 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.