Improving Real-Time Performance on Multicore Platforms Using MemGuard

Heechul Yun
University of Kansas
2335 Irving hill Rd, Lawrence, KS
heechul@ittc.ku.edu

Abstract

In this paper, we present a case-study that shows the challenges of using multicore based platforms for real-time applications, and demonstrate how our kernel based memory bandwidth management system, called MemGuard, can improve real-time performance efficiently.

We show that isolating cores does not necessarily provide desired performance isolation due to contention in shared hardware resources, DRAM in particular. More concretely, the periodic real-time task used in the case-study suffers 28% deadline violations due to memory accesses originated from a non-real-time task (X-server), even though each task is running on a different, isolated core. Using MemGuard, we can configure the system in a way that eliminates all deadline violations at the cost of small throughput reduction of the non-real-time task.

1 Introduction

DRAM is an important shared resource in modern multicore systems. When multiple tasks concurrently run on different cores, they compete for the shared DRAM bandwidth and their performance can vary widely due to contention in DRAM accesses. Such performance variability is highly undesirable in real-time embedded systems.

In our previous work [3], we presented a kernel level approach, called MemGuard, which can minimize such performance variations. MemGuard uses hardware performance counters to periodically monitor and regulate the memory bandwidth usage of each core. Each core can reserve a minimum memory bandwidth, which is guaranteed irrespective of memory access activities of the other cores. This property can greatly simplify archieving desired real-time performance on multicore platforms. Furthermore, each core can receive additional bandwidth depending on other cores memory bandwidth usage, efficiently utilizing bandwidth.

To demonstrating the effectiveness of MemGuard in [3], however, we mainly used the SPEC2006 benchmark suite [2] and the instruction-per-cycle (IPC) metric that do not capture important aspects of real-time systems such as deadline miss ratios and WCETs.

In this work, we present a case study of a real-time system that demonstrates how MemGuard can be configured to deliver the required real-time performance. The real-time system in the case-study consists of a periodic real-time task, which has a given deadline, and an X-server, which constantly updates the screen. The two desired goals of the system are (1) to meet the deadline of the real-time task and (2) to provide a fast screen updates for the X-server; the former goal is more important than the latter.

When we use an unmodified Linux kernel on an Intel Xeon 3553 based multicore platform, we found that contention in shared DRAM causes up to 41% WCET increases of the real-time task resulting in 28% deadline violations. Using MemGuard, however, we can configure the system to eliminate all deadline violations while only causing as small as 17% performance reduction for the X-server. The results suggest that MemGuard can efficiently provide desired real-time performance in a controllable manner.

The remaining sections are organized as follows: Section 2 reviews related background and the MemGuard system. Section 3 presents the case-study results and discuss benefits and limitations. We conclude in Section 4.

2 Background

In this section, we describe DRAM basics and review the MemGuard system.

2.1 DRAM Basics

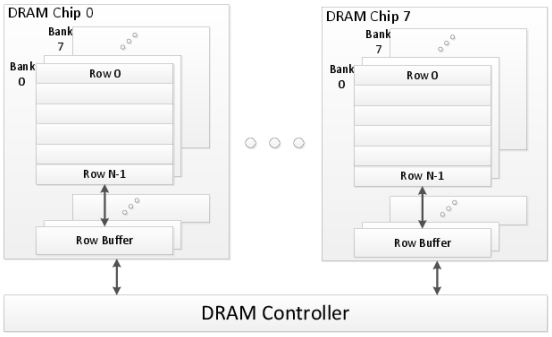

FIGURE 1: *DRAM organization*

Figure 1 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 (typically 64bits in PC). 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 entire data in the selected row into the row buffer of the bank. The required latency for this operation is denoted as $tRCD$, often called as the RAS latency. 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, often called as the CAS latency. If the requested data is in a different row, however, it must save the content of the row buffer back to the originating 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 must be 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, called "row hit", or not, called "row miss". The worst-case latency occurs when successive requests are targeting different rows in the same bank, which is determined by the tRC parameter.

Because banks can be accessed in parallel, the achievable memory bandwidth also varies significantly depending on how many banks are utilized concurrently. If, for example, two successive memory requests are targeting to bank0 and bank1 respectively, they can be requested in parallel, achieving higher bandwidth. In the best case, the peak bandwidth is only limited by the DRAM I/O bus speed. For example, A DDR2 memory system with 400MHz clock can feed up to 6400MB/s (400M x 2 x 64bit). In the worst case, when there is no bank level parallelism and each memory access would cause a row-switch, the bandwidth would be limited by tRC parameter. For example, a DDR2 memory with a 50ns tRC value can serve up to 1280MB/s in the worst-case. Because this bandwidth can be guaranteed even in the worst-case, we call it *guaranteed bandwidth*. As we will describe in the next subsection, MemGuard uses the guaranteed bandwidth as the basis for its bandwidth reservation.

2.2 MemGuard

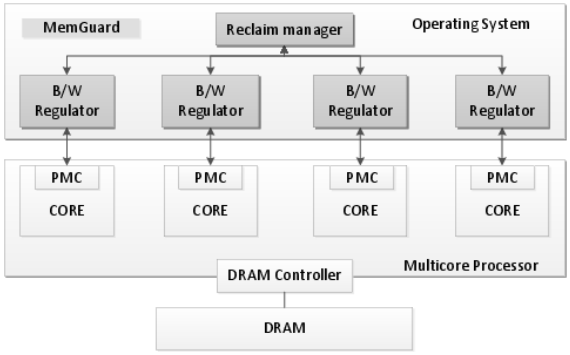

FIGURE 2: *MemGuard system architecture*

In this subsection, we briefly review the MemGuard system [3]. MemGuard is a software system with a goal of providing 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 *minimum* 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.

Figure 2 shows the overall system architecture of MemGuard. MemGuard uses hardware performance monitoring counters (PMCs) to account memory bandwidth usage of each core. Each core can reserve a fraction

of memory bandwidth and the MemGuard system guarantees the reserved bandwidth to be usable to the core. To guarantee the reserved bandwidth, the total reserved memory bandwidth should be limited up to the guaranteed bandwidth—the worst-case memory bandwidth. It operates periodically and assign a memory usage budget for a given time period, which is currently set as 1ms. At the beginning of each period, the budget is recharged according to the core's reserved bandwidth. When the core uses the budget within the period, it stops executing—by scheduling a high priority idle-task with SCEHD_FIFO—until a new period begins. When all cores use their budgets before the next period begins, the additional bandwidth is considered as best-effort bandwidth and shared by all cores. It currently provides two sharing modes: "spare sharing" and "proportional sharing" (See [4] for details). In this paper, we use the latter sharing mode, denoted as "PS".

3 Case Study

In this section, we present a multicore based real-time system which shows how real-time performance—WCET and deadline miss rate—can be impacted by memory interference. We then present how MemGuard can be used to improve real-time performance of the system with minimal throughput reduction of non-real-time part of the system.

3.1 Target Application

The target system in this case-study models a real-time data acquisition system, which is composed of a periodic real-time task and a non-real-time task. The real-time task periodically read a memory region and performs basic processing. The computation must be finished within a given deadline in order to avoid data corruption as the hardware can overwrite the buffer with incomding data. A non-real-time task is responsible for analyzing and post-processing the data acquired by the real-time task. This is a typical real-time data-acquisition application found in a variety of real-time systems. For example, an Unmanned Aerial Vehicle (UAV) with a real-time video recording and analysis capability can be designed in this way.

We simulated the system on a PC platform as follows: First, we create a synthetic real-time task *HRT* that periodically reads a chunk of main memory (the size is equal to 32bpp raw HD video frame data). Each job should be finished within the deadline of 13ms and the period is 20ms (e.g., 50Hz HD video processing with per-frame deadline of 13ms). Hence, we schedule the task with a real-time scheduling policy (SCHED_FIFO) in Linux. Note that HRT is insensitive to the shared L3 cache as all memory accesses are causing cache-misses. Second, for non-real-time data processing workload, we use the standard X-window server, which is responsible for updating the screen. When the X-server updates screen, it generates memory traffic which can be contended with the HRT task concurrently. In order to make the X-server update the screen, we simply use a gnome-terminal to output text continuously scrolling the screen (in fact the standard output of the HRT program). We run the X-server with the standard scheduling policy (SCHED_OTHER) in Linux. When the real-time task and the X-server run together on different cores on the platform, they compete for the shared main memory, causing contention. We create a *cgroup* partition for all non-real-time tasks, including the X-server, and assign a dedicated core. We assign another core for the real-time task using *taskset*.

The two desired goals of the system are (1) to meet the deadline of the real-time application and (2) to provide a high frame-rate for the X-server; the former goal is more important than the latter. We use the deadline miss ratio and the execution time distribution to gauge the first goal. We use the CPU utilization of the X-server to measure the latter goal.

3.2 Hardware Platform

For evaluation, we use an Intel Xeon W3530 processor (Nehalem architecture) based desktop computer. The processor has four cores, runs at a 2.8GHz frequency, and has a 8MB 16-way shared L3 cache and 256KB private L2 caches. The memory controller is integrated in the processor. It has one 4GB PC10666 DDR3 DIMM module with the maximum clock speed of 1333MHz; the maximum transfer rate is 10.6GB/s. For evaluation, we only use only two cores—one for real-time and the other for non-real-time tasks—and disable others. We also disable the Intel hyperboost feature and hardware prefetchers in order to minimize CPU performance unpredictability and to reduce memory bandwidth usage (some prefetched data may not be used).

3.3 Real-Time Performance

In this section, we first present measured real-time performance and throughput of the model system under the vanilla Linux 3.6.0. We then show how we can improve the real-time performance using MemGuard. Throughout the experiments, we use only two cores: Core0 (for HRT) and Core1 (for X-server).

3.3.1 Vanilla Linux Kernel

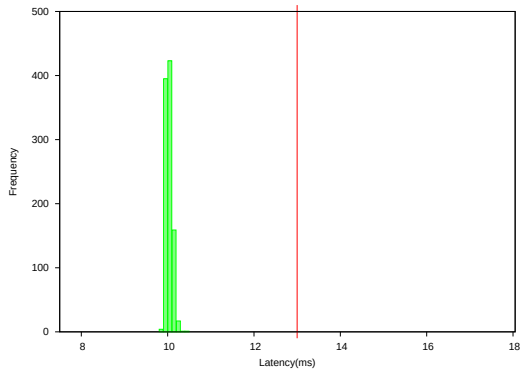

FIGURE 3: *Runtime distribution of HRT, running alone (solo), under original Linux 3.6.0.*

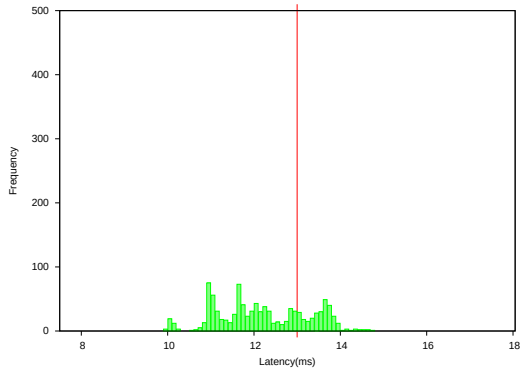

FIGURE 4: *Runtime distribution of HRT, running w/ X-server (corun), under original Linux 3.6.0.*

Figure 3 shows the runtime distribution of the real-time task running alone ("solo") on Core0 for the duration of 1000 executions with the period of 20 ms, under the unmodified Linux 3.6.0 kernel; X-axis shows execution times (in "ms" unit) and the Y-axis shows the corresponding frequencies. Because the HRT task runs alone using the FIFO real-time scheduling policy, its execution time variations are small. The 99 percentile execution time is 10.22ms and all meet the deadline of 13ms (the vertical red line).

However, when we make X-server, running on Core1, update the screen continuously with the output of the HRT task, running on Core0, the runtime distribution varies significantly as shown in Figure 4; the 99 percentile execution time is now 14.33ms (41% increase) and the deadline miss ratio is 28%. Because each of the two tasks is running on a dedicated core and the shared L3 is not utilized by HRT by design, we can infer that the observed performance variations mainly come from the contention in the shared memory hierarchy.

3.3.2 MemGuard with Insufficient Bandwidth Reservation

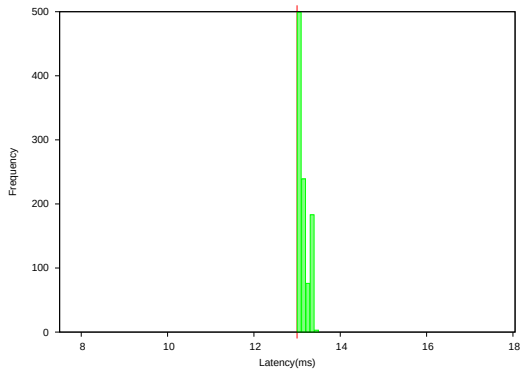

FIGURE 5: *Runtime distribution of HRT, running alone (solo) under MemGuard(600:600).*

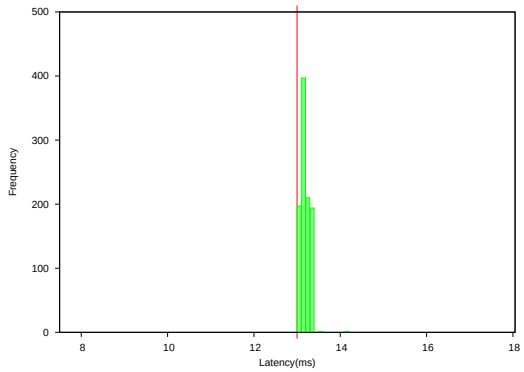

FIGURE 6: *Runtime distribution of HRT, running w/ X-server (corun) under MemGuard(600,600).*

Next, we use MemGuard and reserve a fraction of the memory bandwidth to improve the real-time performance of HRT. We assign 600MB/s bandwidth for each core without using best-effort sharing to see the effect of bandwidth reservation only; we denote this configuration as MemGuard(600,600).

Figure 5 shows the runtime distributions of the HRT task running alone on Core0 under MemGuard(600,600). First note that the runtimes are increased considerably—the 99 percentile is 13.35ms (31% increase)—compared to the previous results under the unmodified Linux kernel. This is because the reserved bandwidth 600MB/s is slightly less than the task needs—i.e., insufficient bandwidth reservation. Therefore, the HRT task is self-regulated by the MemGuard system due to the bandwidth limit.

However, an important benefit of bandwidth reservation can be seen in Figure 6 when we make the X-server

constantly update the screen concurrently on Core1. Note that the "corun" runtime distribution of HRT is almost identical to the 'solo' in spite of the X-server. In other words, real-time performance is not affected by the contending X-server. That is because the X-server is also regulated by the reserved bandwidth of 600MB/s, which significantly reduces the contention that the real-time task experiences. As a result, the performance variation due to contention is much smaller. Note, however, that much of the available memory bandwidth is wasted because cores cannot use more than their small reserved bandwidth.

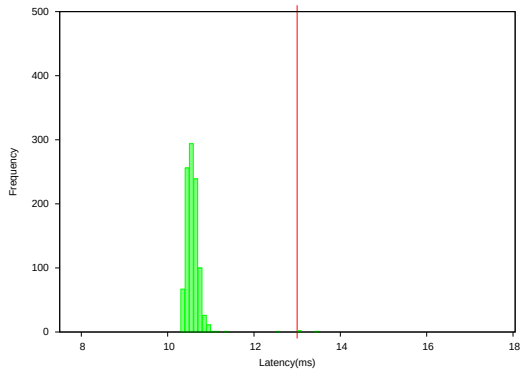

FIGURE 7: *Runtime distribution of HRT, running alone (solo), under MemGuard(600:600)+PS—reservation + best-effort sharing.*

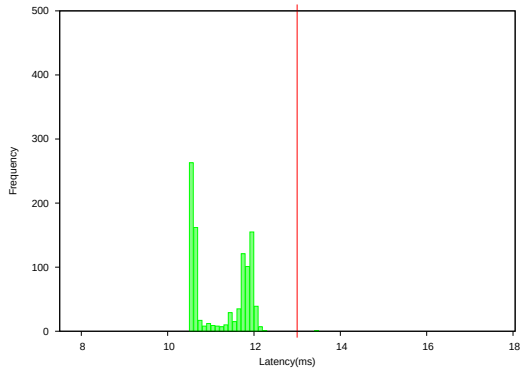

FIGURE 8: *Runtime distribution of HRT, running w/ X-server (corun), under MemGuard(600:600)+PS.*

To better utilize the available memory bandwidth, we now enable the best-effort sharing (see Section 5.3 in [4] for more details) of MemGuard. Figure 7 shows the runtime distribution of the HRT task running alone under the MemGuard with 600MB/s bandwidth reservation and the best-effort sharing, denoted as MemGuard(600,600)+PS. First, note that the runtime behavior is improved significantly to the point similar to that of the vanillar Linux kernel. This is because best-effort memory bandwidth is efficiently redistributed to the HRT task. Second, Figure 8 shows the real-time performance of "corun" experiment, which also show significant real-time performance improvements. Even in the presence of the contending X-server, the HRT task can meet the deadline in most cases (only 1 miss out of 1000) and the 99 percentile performance is 12.1ms, which is well below the deadline.

3.3.3 MemGuard with Sufficient Bandwidth Reservation

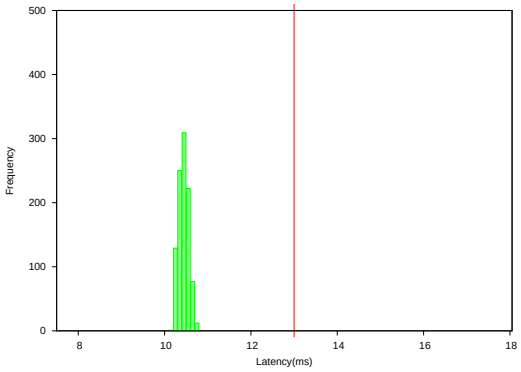

FIGURE 9: *Runtime distribution of HRT, running alone (solo), under MemGuard(900:300)—reservation only.*

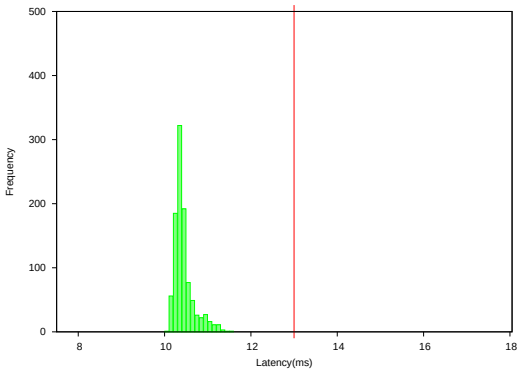

FIGURE 10: *Runtime distribution of HRT, running w/ X-server (corun), under MemGuard(900:300).*

In this experiment, we increase the reserved bandwidth for the HRT task at Core0 from 600MB/s to 900MB/s. At the same time, we reduce the reserved bandwidth for the X-server at Core1 from 600MB/s to 300MB/s, in order to make the sum of reservations is unchanged. We, however, again disable the best-effort bandwidth sharing. We denote this configuration as MemGuard(900,300).

Figure 9 shows the runtime distribution of the HRT task running alone (solo) at Core0 under MemGuard(900,300). Note that the HRT task's runtime distribution is very similar to that of the unmodified Linux in Section 3.3.1. This means that the reserved bandwidth is enough for the HRT task that it does not suffer performance penalties. In the "corun" case, shown in Figure 10, the runtime distribution is still very similar to the "solo" case, meaning that X-server's influence to the HRT task is very small. The 99 percentile execution times in "solo" and "corun" cases are 10.72ms and 11.23ms, respectively. As a result, the HRT task always meets the deadline.

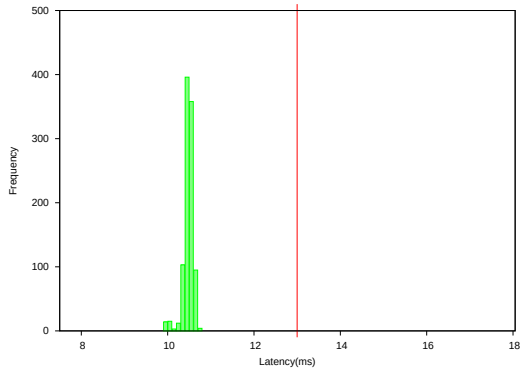

FIGURE 11: *Runtime distribution of HRT, running alone (solo), under MemGuard(900:300)+PS*

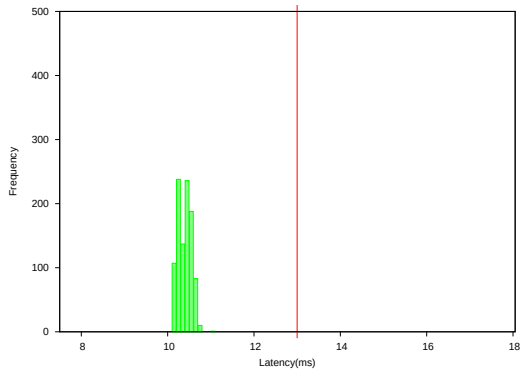

FIGURE 12: *Runtime distribution of HRT, running w/ X-server (corun), under MemGuard(900:300)+PS.*

When we enable the best-effort bandwidth sharing, denoted as MemGuard(900,300)+PS, the real-time performance further improves, albeit slightly, as shown in Figure 11 and Figure 12. The 99 percentile execution times are 10.68ms and 10.72ms in "solo" and "corun" cases, respectively.

3.4 Throughput Impact

It should be noted, however, that the improved real-time performance comes at the cost of reduced performance of the X-server. In this subsection, we compare the throughput impacts of bandwidth reservation and best-effort bandwidth sharing.

Config.	Util.(%)	Diff.(%)
Linux 3.6.0	78	N/A
MemGuard(600,600)	6	-72
MemGuard(600,600)+PS	61	-17
MemGuard(900,300)	4	-74
MemGuard(900,300)+PS	48	-30

TABLE 1: *CPU utilization of X-server at Core1.*

Config.	99 pct. (ms)	deadline miss(%)
Linux 3.6.0	14.33	28
MemGuard(600,600)	13.38	100
MemGuard(600,600)+PS	12.10	0
MemGuard(900,300)	11.23	0
MemGuard(900,300)+PS	10.72	0

TABLE 2: *HRT runtime characteristics at Core0.*

Table 1 shows the observed CPU utilization of the X-server at Core1. First, note that the X-server's CPU utilization is the highest at 78% under the unmodified Linux 3.6.0 kernel. When we use MemGuard(600,600), the CPU utilization is plummeted to mere 6%. This result suggests that the X-server needs significantly higher memory bandwidth when it is actively updating the screen and the reserved 600MB/s is not any close to the requirement. We can visibly notice the performance slowdown as screen updates (scrolling text) occurs much less frequently. When we enable the best-effort bandwidth sharing, MemGuard(600,600)+PS, however, the CPU utilization is significantly increased to 61% and we are not able to notice visible performance differences, although numerically there is 17% performance reduction. Remember that in this configuration, the HRT task is able to meet the deadline 100% of time as shown in Table 2. Similarly, MemGuard(900,300)+PS shows 30% performance reduction of the X-server while it provides even better real-time performance for the HRT task.

Clearly, there is a trade-off between the archived real-time performance of the HRT task and the archived throughput of the X-server. An important benefit of using MemGuard is that the system designers can control the trade-off depending on their system requirements.

3.5 Discussion

As shown in the previous section, MemGuard can efficiently improve real-time performance in a controllable manner. There are, however, several limitations which we will discuss in this section.

First, one problem is that the total reservable memory bandwidth—the guaranteed bandwidth—is small, compared to the peak memory bandwidth. Although this is needed in order to guarantee each core's reserved bandwidth in any circumstances, this also means that each core can only reserve a fraction of bandwidth. For example, in the case-study shown in the previous section, the total reservable memory bandwidth is only 1.2GB/s while the peak bandwidth is 10.6GB/s. The use of hardware prefetchers further exaggerates the problem of limited reservable memory bandwidth because it would increase overall memory bandwidth demand by prefetching potentially unnecessary data.

Second, MemGuard is implemented in a OS kernel and, therefore, the granularity of bandwidth regulation is currently 1ms in line of the OS tick timer. As a result, it cannot provide bandwidth guarantee for a task whose execution time is less than 1ms. Using more fine-grained timer, however, is not desirable considering interrupt handling overhead. We are investigating other ways to provide more fine-grain bandwidth management.

4 Conclusions

In this paper, we presented a case-study that shows the challenges of using multicore based platforms for real-time applications, and demonstrated how MemGuard can be used to improve real-time performance.

Under the standard Linux 3.6.0 kernel, the periodic real-time task in our case-study suffers up to 41% execution time increases and 28% deadline violations due to contention in shared memory from a non real-time task (X-server), even though each task is running on a dedicate core.

Using MemGuard, however, we can configure the system in a way that significantly reduces the performance variations and eliminates all deadline violations with minimal throuput impact on the non-real-time task.

MemGuard is publicly available at `http://github.com/heechul/memguard`.

References

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

[2] J.L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[3] 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.

[4] 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, September 2013.