Performance Isolation for Real-time Systems with Xen Hypervisor on Multi-cores

Wei Jing[1], Nan Guan[2,3] and Wang Yi[3]

[1] *Hatteland Display AS, Norway*
[2] *Northeastern University, China*
[3] *Uppsala University, Sweden*

Abstract—**Virtualization techniques are gaining significant interests in embedded real-time system design. However, existing virtualization platforms lack strong performance isolation among virtual machines. In this work we propose a method to monitor and control the shared memory accesses of individual virtual machines on multi-core processors with Xen hypervisor, to enhance the performance isolation among virtual machines and improve the timing predictability of real-time applications. Experiments with the SPEC2006 benchmark programs are conducted to validate the proposed method.**

I. Introduction

Due to close interaction with physical world, embedded systems are typically subject to timing constraints. The correctness of these *real-time systems* not only depend on the logical results, but are also sensitive to the time when the results are produced. A occasional violation of the timing constraints may lead to serious consequences. The major design principle of real-time systems is to guarantee the system *timing predictability*.

Modern multi-core processors provide great computation capacity on a single chip and can execute different applications simultaneously. These different applications may have different criticality levels in terms of the requirements of timing predictability. For example, in automotive electronics, an ECU may execute both an engine control application with strong real-time requirements, and a multimedia application with weaker real-time requirements but more demanding on system throughput. In such an integrated system, it is a challenging problem of how to guarantee the timing predictability of the engine control application while maintaining the average performance of the multimedia application.

Virtualization techniques are gaining significant interests in embedded system design [5]. The *functional isolation* between different virtual machines on a virtualization platform provides strong fault-confinement and security guarantees, which are essential to safe-critical embedded systems. Virtualization techniques are promising in providing *performance isolation* to meet the predictability requirements of real-time systems, where the resource assignment is under the strict control of the hypervisor and the interference between different virtual machines is easier to be predicted. So recently there has been rapidly increasing interests to apply virtualization techniques to the design of embedded systems with real-time requirements.

In existing virtualization platforms, the hypervisors manage hardware resources such as CPU, memory, disks and I/O devices. We call them *visible* resources as the accesses to these resources are directly under the control of programmers. On modern multi-core processors, there are also many “invisible” resources, such as the shared on-chip bus, shared memory and shared cache. The accesses to these resources are not directly under the control of programmers, but are consequences of both software behaviors and hardware mechanisms. These *invisible* resources also plays important role to the timing behavior of the system. Our experiments show that the execution times of common programs may vary significantly (up to 10 times) under different contention situations on these shared invisible resources. But unfortunately, these invisible resources have not been taken into consideration of existing virtualization techniques, so the performance isolation provided by existing virtualization platforms is not strong enough to guarantee the timing predictability for real-time applications.

In this paper, we propose a method to mange one of the most important invisible resources, the shared memory subsystem, in Xen hypervisor. The idea is to use the built-in Performance Monitoring Unit (PMU) on common processors to monitor the accesses to the shared memory by each virtual machine, and enforce the accesses by each virtual machine to comply with the predefined budgets. This provides stronger performance isolation among different virtual machines and provide better predictability guarantee for real-time systems while maintaining good resource usage of the overall system. Experiments are conducted with SPEC2006 benchmark programs on a dual-core AMD Athlon64 X2 machine to evaluate the proposed methods.

A. Related Work

A rich body of studies have investigated the shared-resource contention by different performance isolating methods in recent years. Bak et al. [2] proposed a real-time I/O management system, which uses a centralized hardware reservation controller to schedule the transactions on peripherals and provide the temporal isolation on the bus.

Akesson and Goossens [1] designed a predictable memory controller in which the critical requester could be served under a predictably low latency and a high bandwidth via the predictable arbitration. Compared with the hardware solutions above, software based schemes have higher flexibility and less complexity than HW modification. Fedorova et al. [4] presented an operating system scheduler which ensures co-running cores running as efficiently as the one under cache-fair allocation, by compensating extra CPU times to the cache-interfered cores. In [9], the contentions causing performance degradation are first classified and analyzed, concluding the LLC cache miss as the identified factors for contentions. Then the thread-base scheduling algorithm DI/DIO are developed to minimize the total cache misses, with the purpose of mitigating the degradation by contentions.

II. Overview of the Approach

We consider a dual-core processor and two applications, with each application executing on a fixed core. The critical application is subject to real-time constraints. We need to estimate an upper bound of the worst-case execution time (WCET) of its executed task. There are two methods of WCET analysis: *static analysis* and *measurement*. In both methods, the estimated WCET of a critical task heavily depends on the contention on the shared memory. The other application does not have real-time requirements, and we call it the interfering application. The interfering application consists of several tasks with different memory access characteristics.

In virtualization-based system design, the subsystems encapsulated in different virtual machines are usually provided by different developer. The system designer of the critical application has no information about the memory access behavior of the other virtual machine, so they are not able to perform meaningful WCET estimation.

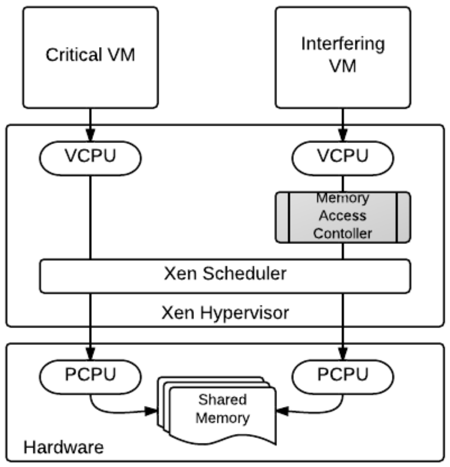

Figure 1. System structure

In order to solve the above problem, we propose to enhance the Xen hypervisor with memory access monitoring and control. The system architecture is shown in Figure 1. In the resource allocation phase, we assign a budget of the memory access to the interfering application. The budget is expressed by a tuple $\{P, Q\}$, where P is the period and Q is the maximal number of allowed memory accesses by the interfering application. We add a memory access control module in the Xen kernel, which at runtime monitors the actual memory access issued by the virtual machine of the interfering application. As soon as the actual memory access issued by the interfering application reaches Q in a period of length P, the memory access control module will notify the Xen scheduler. We modify the Xen scheduler such that it suspends the interfering application exceeding the budget until the start of the next period. In this way, we can guarantee that at runtime the number of memory accesses issued by the interfering application is at most Q in a period of length P, and thus perform precise WCET estimation for tasks in the critical application, by either static analysis or measurement with artificial interfering memory access generation.

Modern processors all equip with the Performance Monitoring Units (PMU), providing processor-level information, facilitating software-based sensing and analyzing the processors' or other devices' online performance. The PMU are implemented as privileged registers which are distributed on CPU cores, aside the CPU die or even I/O devices. A wealth of performance events, such as CPU cycles, cache misses and branch mis-predictions, could be programmed in certain performance counters. The numbers of event are accessible by reading instructions either from user-space or kernel. An interrupt may be triggered if the number of events reaches a pre-defined threshold.

III. Implementation

As the main purpose of this design is to address the contention on the shared memory among cores in multicore systems, we use AMD Athlon 64 X2 which does not have the last-level cache shared among cores, to avoid the effect of shared cache contentions. We choose Xen 4.0.1 and Linux 2.6.32 to build the virtualization software platform. Xen 4.0.1 is one of the stable versions in Xen Hypervisor, which has been tested in [8], [7]. Linux 2.6 is the latest kernel isolated from Xen components, which offers us the flexibility of modifying the Xen components.

A. Performance Counters in Athlon64 X2

AMD Athlon64 X2 3800+, our targeting platform, is a dual-core processor positioned by AMD as Hammer K8 "the 8th generation processor". Two cores featured with a private 64KB L1 data cache and a 64KB L1 instruction cache are incorporated on a single-die. Each core also exclusively owns a 512KB L2 victim cache, which is separated from a total 1MB L2 cache. The two independent cores are connected by a high-speed internal bus with the system request interfaces and the crossbar switch.

Athlon64 X2 3800+ is equipped with an integrated memory controller. Processor cores are able to deliver memory access requests from their self-owned caches to the memory controller without any additional interfere by an intermediate layer such as last-level shared caches. It provides the convenience of addressing the problem of memory access interfering specially, after peeling off extra side-effects from the hardware features.

Athlon64 X2 3800+ provides four 48-bit performance counters supporting 87 events per physical core. Each counter is programmed with a specific performance event. The value is incremented once an occurrence of the event is detected. There are four Performance Event-Select Registers (EvtSel) corresponding to the counters. The bits in EvtSel are used to configure the working mode of the Performance Monitoring Counters (PMC). To start/stop its PMC, the EN bit is set enable/disable. By turning up the INT bit of EvtSel, the PMC is switched into overflow mode in which the overflow of PMC at bit 47 triggers an internal interrupt to the local core. The monitoring event is to be written into the lowest 8 bits and the unit mask which specifies detailed event is filled in bit 15-7.

Both PMCs and EvtSels are accessible regardless whether the performance counters is running or pausing. To use the overflow interrupt, an performance counter overflow interrupt vector must be registered and initialized with an entry to the exception handler. When an counter overflow interrupt signaled, the interrupt handler completes a context switch and hands over to an interrupt service routine. Here, PMCs could be re-initialized or assigned a new value.

B. Memory Access Throttling in Xen

Xen Hypervisor is installed in the paravirtualized mode with two virtual machine domains: Dom0 and DomU. As there are only two individual cores, each domain is pinned to a single processor core. The memories is also distributed evenly, 1.5GB per domain. Dom0 is appointed as an interfering VM and core 0 represents an interfering core. DomU is a critical VM and core 1 is treated as a critical core.

We developed a throttling structure integrated with the internal scheduling framework of Xen. An extended *Memory Access Controller* is located between the interfering VM and the physical core, to monitor the memory access flows. It timely notifies the The overflow interrupts triggered by the performance counters are used to monitor memory access budgets and notify once they are used up. After the interrupt signals the scheduler, further operations are taken to throttle interfering core. As a VCPU is bound to a physical core, the VCPU generating memory accesses should be immediately paused. The throttled VCPU is waken up at the start of the next period. The timing graph is shown in Figure 2.

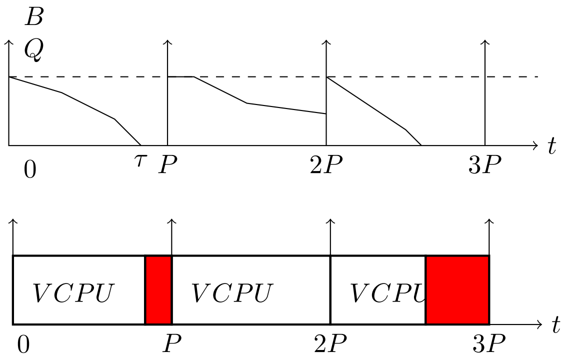

Figure 2. Timing Graph of Throttling

In the first period, the interfering core consumes up the memory access budget Q at the time point τ. The pinned VCPU is then removed from RUNQ. Because there is no runnable VCPU, IDLE VCPU is scheduled in and also the physical core turns in idle. After that, there is no interfering memory access request generated. Till the end of this period, the sleeping VCPU is brought back and the budget Q is replenished. And the interfering tasks run conservatively in the second period, thus budget Q has not been consumed up at point $2P$. The remaining budgets are discarded and the server starts with the budget of Q again. The periodic server above is integrated as an extension on Xen scheduler framework.

When the PMC monitoring off-core memory accesses gets overflowed, the hooked interrupt NMI orientates code into the PMC overflow interrupt service routine. Before further steps to deal with this overflow, we should confirm if Q has actually been consumed up. Then Budget Q for next period is written in disabled PMC beforehand. Till now, the system is still running in NMI context, where it is forbidden to interact with system tasks. In the next step we have to sleep VCPU which is judged as a single thread in system-wide, it is necessary to degrade interrupt to a lower level. So we called a customized soft irq handling the pausing VCPU. This soft irq callback starts with computing timing distance from the current point to next period's beginning, and set it as a timer to wake up VCPU on time. Immediately after, VCPU is paused by an inherent function $vcpu\_pause$ in the *schedule.c* file. Finally the schedule soft irq is signaled in order to inform the system bringing in the IDLE VCPU.

C. The Compensation Mechanism

Till now, the interference between the critical and the interfering cores are restricted by the periodic throttling mechanism parameterized with a Period P and a budget Q to the memory access flows from interfering cores. Although it guarantees the safety of critical tasks and avoids them suffering extra penalties, the performance on the interfering core degrades severely. In reality, not all the execution periods of the critical cores will experience the interference as much as Q, leading the under-utilization of memory bandwidth of the interfering cores. This section presents a compensation mechanism, which reserves the not-in-effect budgets for the future use, to improve the performance of the interfering application.

We use A to denote the memory accesses issued by the critical application itself in a period of length P. In one period P, the critical cores will suffer at most Q interferes if A is larger than Q. If $A < Q$, then there are $Q - A$ times considered as under-utilized. The basis of this compensation algorithm is to accumulate unused budgets for the interfering application from previous period. In the next period, the interfering cores gain $2Q - A$ budgets, releasing more running time for the throttled tasks. The budget is increasing cumulatively only if the critical tasks continue running under the estimation of pre-designed budget Q.

Meanwhile, this continuous accumulation possibly results in an unlimitedly large values of Q, which could make the interferes uncontrollable in the following periods. So it is necessary to reset the budget in a certain amount of periods to limit the expansion of Q. If we reset the budget every two periods, the budget Q will be accumulated at most $2Q$, in the extreme case that in the first period critical task is silent while in the second and period running in full speed. No matter how the critical tasks behave, the maximum interferes are controlled at $2Q$ constantly as before being compensated. In the meantime, the interfering cores could get $2Q$ budgets in every 2 periods, leading 50% lift-up of performance. The designer may choose any number of periods for budget resetting to balance the performance of interfering application and the predictability of the critical application.

IV. Evaluations

We conduct experiments to evaluate our proposed approach in the following three aspects:

1) Worst-Case Execution Time of Critical Tasks: We will examine whether the throttling algorithm is able to mitigate the present contentions on memory accesses. The criterion is that the assigned deadline for a critical task should always be respected when it runs simultaneously with different programs in the throttled interfering VM.
2) Memory Throttling Performance: The interfering memory access flows should be throttled immediately once the monitoring PMCs gets overflowed. The flows could be regulated under the constraints of the configurable bandwidth budgets.
3) Compensation Mechanism: We will evaluate how much the performance of interfering tasks could be lifted-up. We also need to check whether it will affect the running performance of the critical tasks.

Experimental evaluation are conducted with the SPEC CPU2006 Benchmark Suite. We select four benchmark programs: $403.gcc$, $429.mcf$, $470.lbm$ and $459.GemsFDTD$. Figure 3 shows the memory access characteristics of these programs, where $403.gcc$ has mild memory access, while $470.lbm$ is the most aggressive one. The other two benchmarks generate the moderate memory access flows.

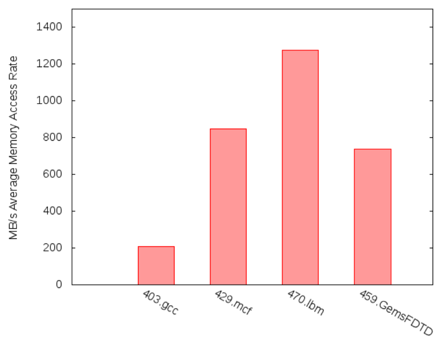

Figure 3. Average Memory Access Rate of SPEC CPU2006 benchmarks

A. Execution Time on Critical VM

We assume the critical VM executes program $429.mcf$, co-running with one of $429.mcf$, $470.lbm$, $459.GemsFDTD$ on the interfering VM. $403.gcc$ is not considered since its memory access ratio is too low.

In each experiment, the critical task $429.mcf$ runs for 200 times accompany the interfering throttle tasks. Figure 4-(a) summarizes the execution times of $429.mcf$ running together with three benchmarks. We set an artificial deadline for $429.mcf$ as 11.4s. The average degradation of $429.mcf$ is controlled within 9% and deadline is successfully secured, even with the most aggressive co-runner $470.lbm$. The same experiments are repeated with $470.lbm$ and $459.GemsFDTD$ as the critical task, the results of which are shown in Figure 4-(b) and (c).

In all the experiments above the execution time of the critical task is kept low and stable, regardless the memory access behavior of its co-runner. Meanwhile, we have noticed that the memory interferes by co-running tasks are not so intensive as it was expected in assumption. Due to the different micro-activities of tasks, memory accesses are not generated continuously nor intensively. Factors of processor architecture, such as inter-connection bus and memory controller, also affects the extent of memory contentions in reality. Concluded from the experimental results, the budget is considered pessimistically and has potential to find a still safe but more loose bound for the interfering VMs/cores.

B. Memory Throttling Performance

The memory access controller is embedded in Xen hypervisor, throttling the interfering VM. In the following

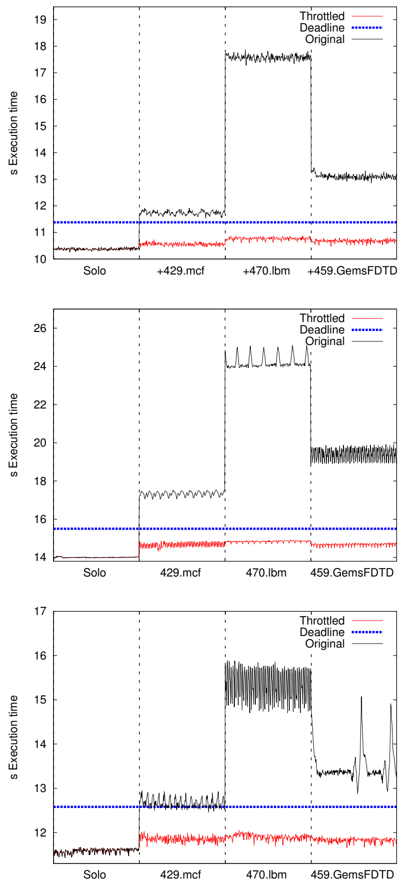

Figure 4. Execution Time on Critical VM

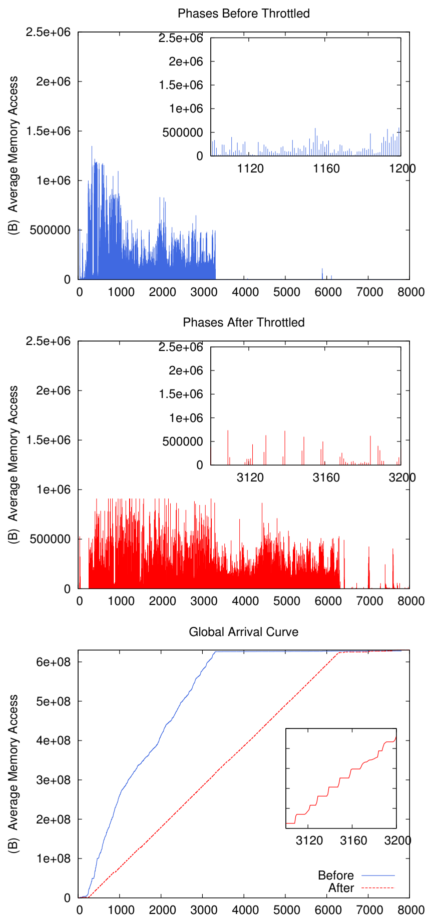

Figure 5. Memory Throttling Performance

experiments, the SPEC CPU2006 benchmark suite is installed on the throttled VM where the bandwidth budget is configurable. The single benchmark will be executed under two budget settings: no throttling (unlimited) and low bandwidth throttling. We are profiling the phase records of the memory access requests on all solo-running benchmarks with the built-in tracing tool $Xentrace$ in Xen.

$Xentrace$ is an event-based logger for recording system activities across the Xen Hypervisor layer and virtual machines. The pre-defined trace events cover scheduling, memory page and hvm. The trace functions embedded in the running code record the corresponding information into a log-buffer, which is accessible by up-running VMs. In user-space, $Xentrace$ daemon reads and takes the buffer down to log files.

Figure 5 shows the performance of two benchmark programs $403.gcc$ and $403.lbm$ under throttling. In the first two figures of both Figure 5-(a) and (b), the x-axis represents the execution time and the y-axis represents the actual memory access bandwidth usage in each period. When $403.gcc$ executes without any memory bandwidth limitation, the transitional memory access within 1ms could reach 1.3 MB in maximum and is around 0.5 MB in average. It takes about 3.3s to finish the program execution. Then we execute $403.gcc$ with the memory access budget with period $P = 10$ms and configure Q such that the effective bandwidth is 91MB/s. In this case, the actual memory access bandwidth usage in each period is limited by $0.9e+06$B and the total execution time of $403.gcc$ is 7s. Comparing the two zoom-in windows, the memory accesses is dense before being throttled, while it is regularly sparse in a rate of 10ms with throttling.

The last graph in Figure 5-(a) illustrates the *arrival curve* [3] of the memory access issued by $403.gcc$ with and without throttling. The arrival curve is a standard workload/interference representation in real-time analysis techniques. The obtained arrival curves enables the system designers to perform real-time performance analysis of the system by either analytical or simulation-based methods [3], [6]. A detailed explanation of the arrival curve is omitted due to the page limit, and we refer to [3] for details. In general, a smoother and less-bursty arrival curve of the interference is better for the real-time performance.

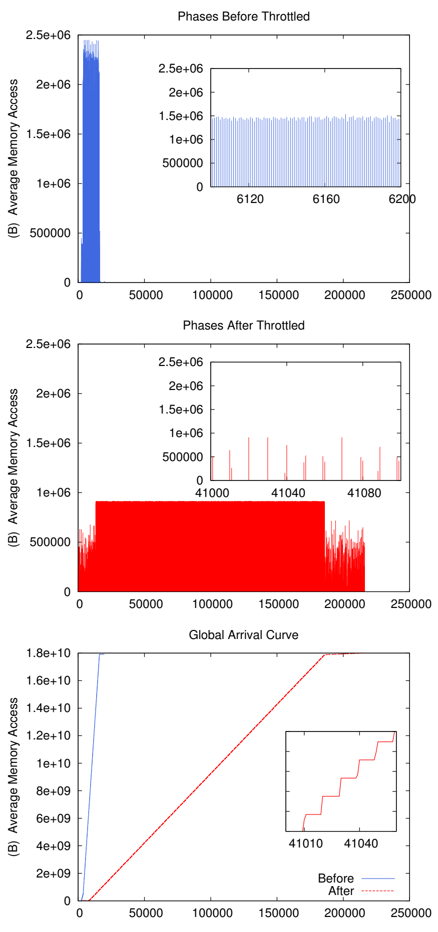

Figure 6. 403.lbm Phase Graph Comparison

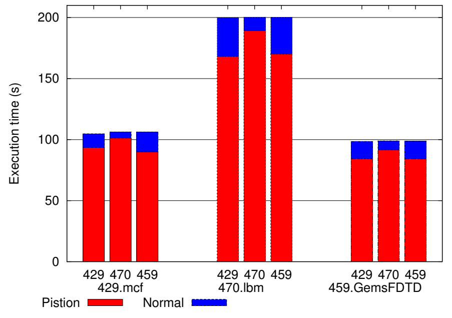

Figure 7. Execution times of interfering tasks with and without the compensation mechanism.

From the figure we can see that the throttled arrival curve is linearly increasing at a constant rate, while the non-throttled one is very bursty in the sense that it may increase very fast in short time intervals. The zoom-in window displays a staircase curve with the step length of 10ms.

The benchmark program 470.lbm generates more aggressive memory access flows than 403.gcc, as shown in

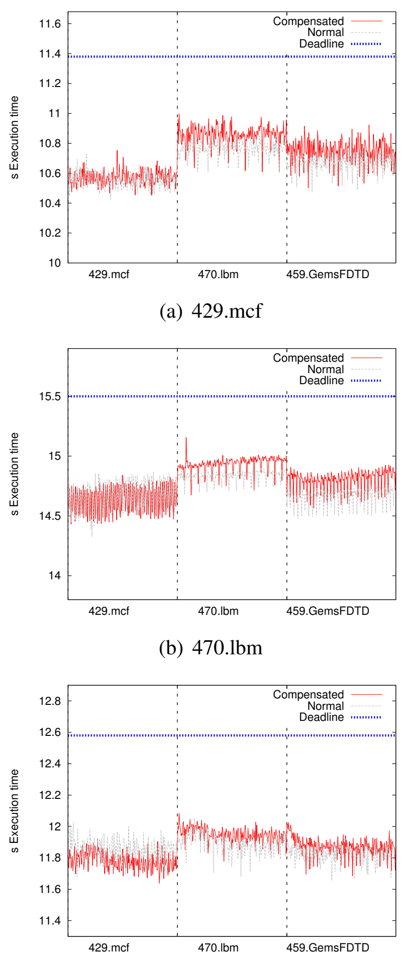

Figure 8. Execution time of critical task with the compensation mechanism.

Figure 3. We also conduct the experiment under the same configurations, and the results are shown in Figure 6. From the top graph, the maximum memory access rate is approximately 2.5 MB/ms, close to the reference system memory bandwidth 3 MB/ms. The total execution time of 470.lbm is extended from 13s to 200s.

C. Compensation Mechanism

The above experiments show that while the memory access throttling improves the timing predictability of the critical VM, the price we paid is the average performance of the interfering tasks. Now we evaluate the compensation mechanism, which aims at improving the average performance of the interfering tasks on the throttled core, while maintaining the good timing predictability of the critical VM. We assume that the budget is reset every two periods.

The first graph in Figure 8 shows the measured execution time of interfering task with and without compensation (with different co-runner benchmark program as the critical task).

The overall performance has been improved by up to 20% by the compensation mechanism.

On the other hand, the compensation mechanism may degrades the timing predictability of the critical task. The three graphs in Figure 8 show the execution time of the critical task with different co-runners with the compensation mechanism. From Figure 8 we can see that The overall execution time of the critical task increases slightly. If we use more aggressive compensation mechanism, e.g., resetting the budget in a larger number of periods. It is up to the system designer to choose proper compensation mechanisms to balance the timing predictability of the critical task and the average performance of the interfering applications.

V. Conclusion and Future Work

In this work we propose a method to monitor and control the shared memory access of virtual machines with Xen hypervisor on multi-core processors, to enhance the performance isolation among virtual machines and improve the timing predictability of real-time applications. The memory access monitoring is implemented using built-in hardware PMU of processors, which keep track of the number of target events such as cache misses and bus accesses. In order to control the contention on the shared memory, we assign a periodic memory access budget to the virtual machine. As soon as the budget is exceeded, the PMU triggers an interrupt and scheduler will suspend the corresponding virtual machine. Experiments with SPEC2006 benchmark suite are conducted to validate the proposed method.

References

[1] B. Akesson and K. Goossens. Architectures and modeling of predictable memory controllers for improved system integration. In *DATE*, 2011.

[2] Stanley Bak, Emiliano Betti, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Real-time control of i/o cots peripherals for embedded systems. In *RTSS*, 2009.

[3] S. Chakraborty, S. Kunzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *DATE*, 2003.

[4] A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *PACT*, 2007.

[5] G. Heiser. The role of virtualization in embedded systems. In *IIES*, 2008.

[6] S. Kunzli, F. Poletti, L. Benini, and L. Thiele. Combining simulation and formal methods for system-level performance analysis. In *DATE*, 2006.

[7] Ruslan Nikolaev and Godmar Back. Perfctr-xen: a framework for performance counter virtualization. In *VEE*, 2011.

[8] S. Xi, J. Wilson, C. Lu, and C.D. Gill. RT-Xen: Towards Real-time Hypervisor Scheduling in Xen. In *EMSOFT*, 2011.

[9] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. *SIGARCH Comput. Archit. News*, 2010.