Performance Isolation for Mixed Criticality Real-time System on Multicore with Xen Hypervisor

Wei Jing
Abstract

**Performance Isolation for Mixed Criticality Real-time System on Multicore with Xen Hypervisor**

Wei Jing

Multicore processors have imported the powerful computing capacity to real-time systems, allowing the multi-task execution on the co-running cores. Meanwhile, the competition for the shared resources among the on-die cores brings the side-effects which highly degrades the performance of single cores and puts the real-time requirements in danger.

This thesis work is focused on addressing the memory access contentions on the real-time systems with mixed-criticality. A throttling algorithm is designed to control the memory access flows from the interfering side, securing the deadlines of critical tasks. We implemented the throttling framework on Xen Hypervisor and evaluated the overall isolating performance with a set of benchmarks. The results prove the effectiveness of our design.
Acknowledgements

I would like to thank my supervisor Nan Guan for providing me his guidance from time to time as needed and helping me to structure the work and my presentation. I am really thankful to him for providing me freedom to give my inputs as the thesis was carried forward.

I would also like to thank my reviewer Kai Lampka for providing his support and valuable comments which led to the betterment of this project.

Finally, I would like to thank everyone else who was a small but important part of my thesis who from time to time gave some valuable comments which in turn helped me develop a better project. Last but not the least, I wish to thank my parents and wife who supported me during the project and helped me reach my goal.
Contents

1 Introduction 1

2 Background 5
  2.1 Xen Hypervisor ........................................... 5
  2.2 Performance Monitoring ................................. 6

3 Memory Access Controller 9
  3.1 System Model ............................................. 9
  3.2 Response Time Analysis .................................. 11
  3.3 Throttling Algorithm .................................... 12
  3.4 Compensation Mechanism ............................... 15

4 Implementation Details 19
  4.1 Target Platform .......................................... 19
    4.1.1 AMD Athlon64 X2 .................................... 19
    4.1.2 Virtual Machines ................................... 20
  4.2 Performance Monitoring ................................. 21
    4.2.1 Performance Monitoring Counters on Athlon64 X2 . 21
    4.2.2 Set-up Memory Access Monitoring in Xen .......... 22
  4.3 Throttling and Compensation Extension ............... 25
    4.3.1 Scheduling in Xen ................................... 25
    4.3.2 Memory Access Throttling Framework ............... 26

5 Evaluation 31
  5.1 Memory Throttling Performance ....................... 32
  5.2 Response Time on Critical VM ........................ 35
  5.3 Compensation Mechanism ............................... 38

6 Related Works 43

7 Conclusion and Future Work 45
## List of Figures

1.1 The performance degradation of SPEC CPU2006 benchmarks co-running with others on a multicore processor . . . . . . . . 2
1.2 Average Memory Access Rate of SPEC CPU2006 benchmarks 2

2.1 Xen Architecture . . . . . . . . . . . . . . . . . . . . . . . . 6

3.1 System Model . . . . . . . . . . . . . . . . . . . . . . . . . . 9
3.2 Arrival Curve on Interfering Core . . . . . . . . . . . . . . . 11
3.3 Budget \( Q \) and Arrival curve . . . . . . . . . . . . . . . . 15
3.4 Compensation Mechanism . . . . . . . . . . . . . . . . . . . 17
3.5 2-Period Compensation Policy . . . . . . . . . . . . . . . . 17

4.1 Athlon64 X2 Architecture [1] . . . . . . . . . . . . . . . . . . 20
4.2 Flow Charts of PMC Initialization . . . . . . . . . . . . . . . 23
4.3 Memory Throttling Structure . . . . . . . . . . . . . . . . . . 27
4.4 Timing Graph of Throttling . . . . . . . . . . . . . . . . . . . 28
4.5 Throttling Process in Xen . . . . . . . . . . . . . . . . . . . . 29

5.1 403.gcc Phase Graph Comparison . . . . . . . . . . . . . . . 33
5.2 470.lbm Phase Graph Comparison . . . . . . . . . . . . . . . 34
5.3 429.mcf Execution Time . . . . . . . . . . . . . . . . . . . . . 37
5.4 470.lbm Execution Time . . . . . . . . . . . . . . . . . . . . . 37
5.5 459.GemsFDTD Execution Time . . . . . . . . . . . . . . . . 38
5.6 Execution Time of Interfering Tasks after Compensation . . . 39
5.7 Comparison of Critical 429.mcf after Compensated . . . . . 40
5.8 Comparison of Critical 470.lbm after Compensated . . . . . 40
5.9 Comparison of Critical 459.GemsFDTD after Compensated . . 41
Chapter 1

Introduction

The boost of multicore processors significantly upgrades the computing performance of single hardwares, allowing the real-time systems hosting multiple applications in individual cores. For example, a real-time vehicle navigation system need handle the GPS positioning and image recognition of traffic signs concurrently on a single multicore processor [2]. With the growing demands of computing workloads, more cores are being integrated into a single chip. Although the more powerful ability for data processing is available now, the shared resources, such as the inter-connecting buses and memory controllers, are still shared among the on-chip cores. The available bandwidth on these shared-resources is asymmetrically limited, compared with the intensive access requests from on-chip cores. So the uncontrollable contentions for shared resource accessing generate the extra latency, which apparently degrades real-time systems’ running performance and attenuates the benefits of multicore architecture. For the safety-critical applications on real-time systems, the worst-case execution time (WCET) has the danger to exceed the deadline, due to the unpredictable interferes from co-running cores.

Figure 1.1 highlights the impacts to co-running tasks by the memory access contentions. The results are concluded from our experimental platform where two selected benchmarks from SPEC CPU2006 executing simultaneously on separate CPU cores. They issue memory accesses directly from core to inter-connection bus and memory controller without any impedance from a last-level shared cache. Figure 1.1 shows the worst case degradations when benchmark 470.lbm co-runs with others on a different core. The maximal slow-down reaches over 70% while it is running with self copy. The most friendly benchmark is 403.gcc, with which applications on the other core perform close to solo-run. In addition, Figure 1.2 shows average memory access rates of these four benchmarks profiled in solo runs. 470.lbm produces the largest memory access flows in a unit time period, whereas 403.gcc generates the smallest. They are approximately in the difference of
Figure 1.1: The performance degradation of SPEC CPU2006 benchmarks co-running with others on a multicore processor

Figure 1.2: Average Memory Access Rate of SPEC CPU2006 benchmarks
6 times. This indicates the impact from memory access contentions is indirectly relative to the amounts of memory visiting flows on the co-running cores. The challenge is presented that how to provide an isolated running environment for tasks, where interferes on shared resource by others on the co-running cores could be minimized or controllable.

Performance isolation [3] provides the resource management schemes which guarantee the shared resource allocation physically or logically to individual cores. So that, the cores are assigned to run on the specific resource entities rather than do the random accesses. It mitigates the contentions to the maximum extent.

Virtualization is a long-term utilized technology in the server field, and recently has been imported into real-time systems [4, 5]. By having the capability of partitioning hardware resources and cores, the specific memory ranges and I/O addresses could be pinned to the sub virtual machines(VMs) via several configurable means. This ensures the separate VMs having isolated resource divisions. The resource accesses from the tasks on dedicated VMs are constrained by a management layer, named hypervisor, between the virtual machines and the physical hardware. However virtualization does logical partitioning on the shared resources and provides the structural isolation among VMs, the bottlenecks on physically sharing the HW components stay the same as bare-metal.

In this paper, we present a novel software-based performance isolation scheme for mixed criticality real-time systems, which addresses the memory access contentions on the inter-connection buses and shared memory controllers. Compared with present solutions, virtualization is imported into our design for strengthening the logical management on hardware resources. As a logical image of the independent task set, each VM is pinned with a certain amount of physical cores. For the co-running VMs attributed with different criticality levels, the interfering extent on high criticality VMs is positively relative to the amount of memory accesses on the lower criticality ones [6]. We design a memory access controller mechanism to regulate the memory bandwidth on the VMs with low criticality. This controller behaves like a periodic server which could bound the target memory accesses within safe budgets. The interfering access streams from the low criticality VMs are throttled immediately when the monitored budgets are consumed up, in order to secure the throughputs on critical VMs. In this way, the interference from the low criticality VMs to higher ones is under the control of our framework.

Our first goal is to figure out the appropriate throttling periodic budgets for VMs with interfering tasks. By taking consideration of the worst-case arbitrating behaviors on the shared memory controller, memory access requests from interfering VMs are sacrificed by assigned safety bounds which isolates critical VMs. We describe the algorithm for analyzing WCET of critical tasks and computing the safe budgets for interfering ones.
The second goal is to implement a throttling mechanism with virtualization. We experimentally build up a test-bench platform with open-source hypervisor Xen and Linux. The memory access controller is attached with the native scheduler in Xen hypervisor, which does the real-time regulation on the memory accesses from the interfering VMs. In order to improve the under-utilization of abundant bandwidths on the interfering side, an improved mechanism is added. It compensates the throttled interfering VMs with extra budgets, according to the online monitored the running performance of the critical ones.

From the evaluation experiments with typical benchmarks in SPEC CPU2006, our solution is well-behaved in the memory bandwidth throttling and the deadline securities for critical VMs are guaranteed. Meanwhile, the performance of interfering VMs is lifted up by the compensating algorithm.

The rest of this paper is organized as follows. Chapter 2 introduces the background on Xen Hypervisor and Performance monitoring units. Chapter 3 proposes the system model and our memory bandwidth controlling algorithm. Chapter 4 describes an detailed implementation on this virtualization-based platform. Chapter 5 shows the evaluation of proposed system and comments our experiment results. Chapter 6 compares our solution with related works and finally chapter 7 concludes with future work.
Chapter 2

Background

2.1 Xen Hypervisor

Xen Hypervisor, started by P.Barham el. [7] in 2003, is a virtual machine monitor (VMM) platform and now is maintained in the open-source community. From Linux kernel v3.0, part of the Xen modules have been integrated into the source code. As a long-established virtualization solution, Xen has been deployed on x86, IA64 and ARM architectures.

Xen adopts the paravirtualization techniques in which the hypervisor layer abstracts a similar but modified system to underlying the physical hardware [8]. By bypassing complex hardware and instruction accommodation [9,10], the abstracted system makes the operating system on each virtual machine perform more natively by minimizing the intervention from the hypervisor. Also the united hardware framework gives the extra flexibilities for the VM distribution. One side-effect of paravirtualization is that operating systems need necessary modifications to keep accordance with the abstract hardware interfaces. Later released Hardware Virtual Machine (HVM) technique eliminates the need for extra OS modification by using the built-in extension on common x86 processors.

Figure 2.1 shows the architecture of Xen. As the heart of Xen virtualization platform, Xen Hypervisor lies between the physical hardwares and all guest machines, with the jobs of allocating resources and virtualizing the interface framework. Domain is a specific Xen instance representing one running virtual machine above the hypervisor. Assisting the hypervisor, Domain 0 is the virtual machine behaving part of the functionalities as the manager in Xen. Domain 0 is the first domain brought up together with Xen framework. Built-in modules such as the guest domain management and physical device handlers, are integrated with domain 0 in form of tools and daemons in user-space. That facilitates the configurations to the hypervisor and the other guest domains if necessary.

\footnote{Xen.org \texttt{www.xen.org}}
Other domains which are under the supervision of Domain 0 are named Domain U. In Domain U, the start-up configurations, such as allocated memories, network settings and graphic supports, are all processed in Domain 0. As an unprivileged domain, Domain U are not authorized to direct hardware accesses. Instead, Xen provides a split model for the hardware drivers, where Domain U owns the front-end drivers and Domain 0 possesses the back-end drivers with full functionalities. This is a major difference between Domain 0 and Domain U.

Due to the most privileged role of the hypervisor, the OS kernels in VMs are evicted from Ring 0 to avoid any possible violations against the operations by the hypervisor. Because of this degradation in privileged levels, the privileged instructions, such as HLT and CLTS, are replaced with either the system calls or the indirect instructions operated by the hypervisor layer. To address the extra overhead generated by indirecting the system calls to the hypervisor, Xen allows the guest VMs accepting 80h interrupt, bypassing the handlers in the hypervisor.

### 2.2 Performance Monitoring

Modern processors all support performance monitoring with a set of performance monitoring counters (PMC). These special counters provide the processor-level information. This technique facilitates the software-based sensing and analyzing the processor or other device's online performance.
The PMCs 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 mispredictions, could be programmed and monitored in the certain PMCs. The Result which is incremented by the numbers of event occurrence are accessible by the reading instructions from either user-space or kernel. In x86 [11] and PowerPC architectures, the instruction RDPMC is used to read the PMC values of the general purpose registers. Besides accepting the reading requests, PMCs could be configured to generate the interrupts while getting overflowed. This allows the event-based sampling in which the threshold is set to profile based on the overflow interrupts from PMC.

Although the direct PMC readings exhibit the low-level performance intuitively, the further performance needs to be explored by the different performance measurements. For example, Instructions per cycle(IPC), a symbolic figure of computational efficiency, needs two events: Non-halted CPU cycles and Finished instructions. In our case, the memory access bandwidth is potentially a major performance metric. [12] [13] indicate the measurement of profiling the front-side bus’s full cache line requests and elapsed cpu cycles. On an Intel Nehalem processor, the memory bandwidth is computed in the following formula:

$$\text{Bandwidth} = \frac{64 \times \text{BUS\_TRANS\_Burst} \times \text{Freq}}{\text{Elapsed\_Cycles}}$$  \hspace{1cm} (2.1)

where BUS\_TRANS\_Burst and Elapsed\_Cycles are the values of corresponding hardware events, 64 is the length of full cache line and Freq is equal to the standard bus frequency.

To serve the system-wide profiling work, several performance monitoring frameworks have been developed. These frameworks utilize the built-in drivers in kernel space and acknowledge the PMC configurations, accessing requests from the user-space interfaces. These events could be counted globally or in the thread-based mode. Extended from the basic reading and counting functions, the frameworks like perfctr and oprofile offer their pre-defined performance measurements and statistic methods. Recently, the performance monitoring solutions are introduced to the virtualization environment. R.Nikolaev [14] developed a direct performance counter profiling infrastructure with the combination of Xen and perfctr. XenoProf extends Oprofile to support the per-domain performance counter profiling in Xen.
Chapter 3

Memory Access Controller

This chapter describes a system model of multicore processor possibly suffered memory contentions. Then it discusses [15]'s algorithm of estimating the worst-case response time on critical cores and later proposes a memory throttling algorithm executed on the interfering cores which secure critical cores’ safety requirements. Lastly, a compensation mechanism is raised with the purpose of mitigating the serious attenuations on the interfering side, based on the previous memory access control algorithm.

3.1 System Model

In order to address the memory contention problem among on-die mul-
ticore processors, we assume a system model as shown in Figure 3.1. All individual cores own their private instruction and data caches, but there is no last level caches shared among them. The cores are connected by an inter-connection bus with the memory controller. The single memory controller, which handles the memory reading access requests, behaves as a shared channel between the on-die cores and the system memory. There is no limitation of the memory transaction times for any single core while the global memory bandwidth is constrained by the pre-defined architecture of this memory controller. By considering the system model above, the memory access contentions are exposed as the major factors interfering the on-core running performance, because all executing stages are isolated physically except data fetching from the off-core memories. Despite these model features allocate the target issues on shared-memory multicores, the memory contention is difficult to be analyzed due to the invisible and confidential architecture inside the memory controllers. Although [16], [17] proposed their hardware implementations on the memory controller modules, for COTS processors, predicting the memory access contentions quantitatively is time-consuming and complex. In our work, additional assumptions are set to predigest the analyzing process: each memory access latency is capped by a constant worst-case value, the Round-robin arbitration is adopted on the memory access path and the running cores. It produces no other than the memory access traffics to the inter-connection bus.

Under the safety-criticality levels, the memory-sharing cores in processor are classified with the critical cores, which require the performance guarantees, and the interfering cores, which need to be constrained to minimize the contentions against the critical ones. On the critical cores, the task set $T_n$ with periodic tasks is scheduled preemptively under the deadline based scheduling algorithm. Each task is characterized by $\langle C_i, T_i, D_i \rangle$, $(D_i < T_i)$. A single task’s worst case execution time $C_i$ is in relation with the preempted time by the tasks with high priorities, and the external memory access interferes by other cores. We also assume the critical tasks running independently on the cores without any contention for private caches. That ensures each task $\tau_i$ generating the constant worst case memory accesses $N_i$ with a fixed access latency $L$. These accesses behave independent of the co-running tasks.

On the interfering cores, the memory access request flows are framed by a periodic server, in which every single memory access is considered as a virtual “task”. With the period $P$ and the budget $Q$, the memory access “task” is scheduled in a constant rate, that leads the interfering cores generated the memory access with an upper bounded bandwidth. No matter how the tasks on the interfering cores are scheduled, externally their worst case behaviors of memory accessing is predictable by a cumulative function as shown in Figure 3.2. Considering the edge-effect by the previous tasks, an extra $Q$ is accumulated to the beginning of the new task session. After
setting this controllable model, the next sessions will work around finding a suitable budget $Q$ with which the interfering cores are throttled while the critical cores are suffering reasonable amounts of interferes.

### 3.2 Response Time Analysis

This section describes the worst case response time estimation of the tasks on the critical cores, against the memory contentions by the interfering cores proposed in [?].

As in our model, the inter-connection bus processes the memory access requests based on the round-robin arbitration and the FCFS memory controller. It follows the rule that each access will be suspended by at most one other request from the other cores. In this case, the penalties of memory contentions is calculable by the memory accesses from both the critical and interfering side. Considering a critical core generates $N$ memory access requests in the worst case, meanwhile there are $k$ interfering cores with $·M_i$ times individually, the predictable interferes based on our assumption is $\sum_k \min(N, M_k)$. Then the global penalties suffered by the critical core are $\sum_k \min(N, M_k) · L$, where $L$ is the measured worst-case latency of a single memory access.

The arrival curve on the interfering core $k$ is constrained by a similar Leaky Bucket Controller [18], where the request flow is regulated into an upper-bound flow $S_k(t)$. $S_k(t)$ represents the worst case accepted requests from core $k$ in the interval $t$. For the purpose of simplifying our analysis, $S_k(t)$ is monotonically non-decreasing. Assuming a critical task $\tau_i$ attributed with $A_i$ memory accesses and the worst case execution time $C_i$. Within the
execution time \( C_i \), the interfering core \( k \) could generate \( S_k(C_i) \) interfering stream at most. This causes an extended delay \( S_k(C_i) \cdot L \) and increases \( \tau_i \)'s WCET, when more requests stream out of core \( k \):

\[
\Delta_k^{n+1} = S_k(C_i + \Delta_k^n \cdot L); \quad \Delta_0^k = S_k(C_i)
\]  

(3.1)

This iterative procedure will continue till the value \( \Delta \) stabilizes and full bounded by the arrival curve \( S(t) \). Hence, taking consideration of the predictable interferes, WCET \( \widehat{C_i} \) of \( \tau_i \) is expressed as a cumulative equation:

\[
\widehat{C_i}^{n+1} = C_i + \sum_{k} \min(A_i, S_k(\widehat{C_i}^n)) \cdot L
\]  

(3.2)

The sum of all delays caused by all the interfering cores together with the original execution time \( C_i \) are counted as one iteration value. As Equation (3.2), after finite times of iterations, it is finally bounded and returns WCET under the multicore memory contention scenario.

In addition to the WCET of a single task \( \tau_i \) which interfered by the co-running interfering cores, other tasks on the same core is increasing WCET interiorly. Refer to the general worst-case response analysis by Joseph and Pandya [19], the WCET for task \( \tau_i \) is expressed as the following recursive equation:

\[
\widehat{R_i}^{n+1} = C_i + \sum_{\forall j \in \text{hp}(i)} \left\lceil \frac{\widehat{R_i}^n}{T_j} \right\rceil \cdot C_j
\]  

(3.3)

The process starts with \( \widehat{R_i}^0 = C_i \) and is iteratively added with the preempted time by the higher priority tasks. It finally terminates until reaching a stablized value where \( \widehat{R_i}^{n+1} = \widehat{R_i}^n \). As known the deadline \( D_i \) of task \( \tau_i \), any transitional iteration \( \widehat{R_i}^m \) outriding \( D_i \) implies the failure of task \( \tau_i \).

The overall algorithm for computing WCET of task \( \tau_i \) on the critical cores is presented in Algorithm 1. It introduces an integrated worst case response time analysis, which covers the increased time from both the memory access interferences and the task preemptions. Line 3 updates the worst case memory access requests from the critical cores based on the previous iteration of response time. The total request number is equal to the self requests \( A_i \) plus the ones from all other high priority tasks within the interval of \( \widehat{R_i}^n \). Line 4 shows the iterative step of WCET. Here the inter-task delays, calculated along with the general algorithm as Equation (4), are superimposed onto the inter-core interferes from the other interfering cores. It is resulted from Equation (3), obtaining the overall WCET \( R_i \). The algorithm exits under the condition of either \( R_i \) is bounded or task \( \tau_i \) misses its deadline.

### 3.3 Throttling Algorithm

As mentioned in Section 3.1, the memory access requests on the interfering cores are shaped by an on-core bandwidth-preserved servers, which are
Input: $\tau_i$
Output: $R_i$ or Failure
1 $\hat{R}^0_i = C_i$
2 repeat
3 $A^{n+1} = A_i + \sum_{\forall j \in hp(i)} \lceil \hat{R}_j^n \rceil \cdot A_j$
4 $\hat{R}^{n+1}_i = C_i + \sum_{\forall j \in hp(i)} \lceil \hat{R}_j^n \rceil \cdot C_j + \sum_k min(A^{n+1}, S_k(R_i^n)) \cdot L$
5 until $(\hat{R}^{n+1}_i = \hat{R}^n_i)$ or $(\hat{R}^{n+1}_i > D_i)$
6 return $R_i$;

Algorithm 1: WCET Computation

assigned with a replenished period $P$ and a globally consumable budget $Q_i$, in the constant rate. So that the contentions for memory accesses between the critical and interfering cores are controllable, under the configurable arrival curves. To guarantee the critical cores well isolated, the budget $Q$ is required to restrict the interfering streams, securing the deadlines of tasks on the critical cores. In this section, the algorithm for determining $Q$ is introduced.

We assume every single task $\tau_i$ from the schedulable fixed-priority task set $\tau_1..n$ running in the period $T_i$. To guarantee the safety in this critical core, all tasks should have be kept away from failure. Therefore, each task $\tau_i$ on the critical core is considered to have a throttling value of budget $Q_i$ on all other interfering cores. The minimum value $minQ_i$ computed based on all the tasks is concluded as the feasible budget $Q$ which could be assigned to the interfering cores.

The next part is illustrating the process of calculating $Q_i$ for each task $\tau_i$. For the tasks modeled as $\langle C_i, T_i, D_i \rangle$ ($D_i < T_i$), one of the safety standards is that WCET is addressed no later than the deadline $D_i$. In the general analysis of WCET on a single core, the restriction is expressed as: $D_i >= C_i + \sum_{\forall j \in hp(i)} C_j$. The deadline should include both $\tau_i$’s self execution time and the preemption time by the higher priority tasks. In the multicore scenario, taking the inter-core interferes into consideration, the available stalled time left to interferes is $Res_i$, which is equal to the redundant time after execution and preemption and ahead of the deadline:

$$Res_i = D_i - C_i - \sum_{\forall j \in hp(i)} \lceil \frac{D_i}{T_j} \rceil \cdot C_j$$  

(3.4)

As determined in Equation (3.3), the available stalled time $Res_i$ is prepared for the predicted interfered time computed from $min(A_i, S(D_i)) \cdot L$. Here
$A_i$ is the worst case memory access requests from a critical core within $D_i$, and $S(D_i)$ is the shaped flow by the global periodic server function from the interfering cores. If $A_i$ is smaller than $S(D_i)$, the predicted number of interferes is determined by $A_i$, whereas $S(D_i)$ under the mutually exclusive condition. Then the later situation requires $S(D_i)$ throttling interfering flows in a safe way.

According to the definition in the system model, the interfering cores are served under a periodic server with the replenished period $P$ and the consumable access budget $Q$. The overall cumulative arrival curve is shown in Figure 3.2. Taking the legacy effects from previous periods into account, the starting period is budgeted with double of $Q$. Assuming the interfering flows behaving super-bursting when burning up the budget $Q$ at a speed of $L$ (worst-cast access latency) after one period replenished, the arrival curve is described in the stair-shape and the expression is:

$$S(t, Q) = Q \cdot L + \left\lfloor \frac{t - Q \cdot L}{P} \right\rfloor \cdot Q \cdot L + U((t - Q \cdot L) \mod P, Q) \quad (3.5)$$

where $U(t, Q)$ represents the unit equation for periodic behavior in $S(t)$:

$$U(t, Q) = \left\{ \begin{array}{ll} t & t \leq Q \cdot L \\ Q \cdot L & t < Q \cdot L \end{array} \right. \quad (3.6)$$

In order to guarantee the deadline of critical task $\tau_i$, the arrival curve should be bounded by the available stalled time $Res_i$ corresponding to the assigned deadline $D_i$. Then we have

$$S(t, Q) = \left\lfloor \frac{D_i - \Delta}{P} \right\rfloor \cdot \Delta + U((D_i - \Delta) \mod P, Q) \leq Res_i \quad (3.7)$$

where $\Delta = Q \cdot L \in [0, P]$. Assuming $P << D_i$, a feasible maximum budget $Q_{\max}$ could be figured out in this condition.

**Proof.** From Equation (3.5), $U(t, Q)$ is determined increasing along with $Q$ and bounded at $\Delta$. Because of the condition $D_i >> P \geq \Delta$, the element $\left\lfloor \frac{D_i - \Delta}{P} \right\rfloor \cdot \Delta$ is far above $U((D_i - \Delta) \mod P, Q)$ here.

In this case, the function $S(D_i, Q)$ is located in the range as

$$\left\lfloor \frac{D_i - \Delta}{P} \right\rfloor \cdot \Delta \leq S(D_i, Q) \leq (\left\lfloor \frac{D_i - \Delta}{P} \right\rfloor + 1) \cdot \Delta \quad (3.8)$$

Then it is easy to approve that $S(D_i, Q)$ is monotonically non-decreasing when $\Delta \in [0, P]$.

When $Q = 0, S(D_i, Q)_{\text{min}} = 0$, and when $Q = P, S(D_i, Q)_{\text{max}} = D_i + P$. Refer to Equation (3.4), the relation $S(D_i, 0) < Res_i < S(D_i, P)$, shown in
Figure 3.3, is established. Therefore, by Intermediate value theorem, there exists a $Q_{\text{final}}$ in the range of $[0, P]$ where $S(D_i, Q_{\text{final}})$ is closest to $\text{Res}_i$.

Based on the proved property that $S(D_i, Q)$ is monotonically non-decreasing in $\Delta \in [0, P]$. We formulated the process of computing $Q_{\text{max}}$ as Algorithm 2, adopting the Bisection method in root-finding. The variation range for $Q$ is being narrowed while $S(D_i, Q)$ is approaching $\text{Res}_i$. The process terminates when $S(D_i, Q)$ reaches closest to $\text{Res}_i$ and the maximum $Q_{\text{max}}$ is found there.

After solving the deadline guarantee in task $\tau_i$, we can follow this procedure to calculating the budgets for the other tasks on the critical cores.

### 3.4 Compensation Mechanism

Till now, the interferes between the critical and the interfering cores are restricted by the periodic throttling mechanism parameterized with a Period $P$ and a budget $Q$, throttling the memory access flows from the interfering cores. Although it guarantees the safety of critical tasks and avoids them suffering extra penalties, the performance on the interfering cores are severely cut off. In reality, not all the execution periods of the critical cores will ex-
perience the interference as much as $Q$, leading the under-utilization of the memory bandwidth given out by the interfering cores. This section proposes a compensation mechanism, which reserves the not-in-effect budgets for the future use, for the throttling mechanism above.

As it is assumed, in a unit period $P$, the critical cores will suffer at most $Q$ interferes, when its self memory accesses $A$ is larger than $Q$ correspondingly. If $A < Q$, then there are $Q - A$ times judged as under-utilized. The basis of this compensation algorithm is to accumulate the unused budgets from the previous period to next on the interfering cores. So that, 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 $Q$. The best-case for the interfering cores shown in Figure 3.4 is that, the critical tasks are running conservatively, while the interfering ones consume budget at a constant rate $Q$. The unused budgets are accumulated losslessly. If the interfering cores utilize these abundant budgets, their running performance could be apparently mitigated due to the increasing budgets to consume.

Meanwhile, this continuous accumulation possibly results unlimitedly large values of $Q$, which could make the interferes uncontrollable in the next periods. So it is necessary to reset the budgets in a certain amount of periods to constrain the expansion of $Q$. Assuming there is a 2-period reset policy as in Figure 3.5, the budget $Q$ will be accumulated at most $2Q$, in the scenario that in the 1st period critical task is silent while in the 2nd and the last 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 be benefit $3Q$ budgets in every 2 periods, leading 50% lift-up of performance. The reset periods could also be extended to any other number than 2.
Figure 3.4: Compensation Mechanism

Figure 3.5: 2-Period Compensation Policy
An additional assumption should be raised here that all the accumulated budgets are flushed once the task switches happen on the critical side, avoiding the extra back-logged interferes by this compensation mechanism. We demonstrate the performance of this compensation mechanism later in Chapter 5.
Chapter 4

Implementation Details

In this section, we discuss the design and implementation of the proposed performance isolation mechanism on Xen hypervisor. We firstly introduce our testing hardware platform and the utilization of built-in Performance monitoring units. Then, based on the throttling algorithm and the compensation mechanism in the previous section, we design a corresponding throttling extension attached with Xen’s original scheduler.

4.1 Target Platform

Our testing platform is composed with a multicore processor, the Xen hypervisor and the virtual machines running with Linux, architecturally from bottom to top. As the main purpose of this design is to address the memory contentions among the cores in multicore systems, we sort out an old-version of AMD Athlon 64 X2 in which there is no last-level cache shared among cores, avoiding the extra shared cache contentions. We also choose Xen 4.0.1 and Linux 2.6.32 as a pair for virtualization. Xen 4.0.1 is one of the stable versions in Xen Hypervisor which has been tested in [?][?]. And Linux 2.6 is the latest kernel isolated from Xen components, while in Linux 3.0 several functional modules are integrated. This offers the flexibility of modifying the Xen components.

4.1.1 AMD Athlon64 X2

AMD Athlon64 X2 3800+, our targeting platform, is a dual-core processor positioned by AMD as Hammer K8 ”8th generation processor”. Two cores featured with a private 64KB L1 data cache and a 64KB L1 instruction cache are incorporated onto a single-die. Each core also exclusively owns a 512KB sized L2 victim cache, which is separated from a total 1MB L2 cache defined by K8. These two mutually independent cores are cooperated by a high-speed internal bus with the help of the system request interfaces and the
crossbar switch. Athlon64 X2 3800+ is equipped with an integrated memory controller, providing the intermediating service between processor cores and DRAMs. The integrated memory controller handles the read/write requests in queues and regulate their visits to next stages according to the specific arbitration. Figure 4.1 shows the internal architecture of Athlon64 X2.

![Athlon64 X2 Architecture](image)

**Figure 4.1: Athlon64 X2 Architecture [1]**

Processor cores are able to deliver the 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 the extra side-effects from the hardware features.

Additionally, K8 processors [20] provide four 48-bit performance counters with 87 events’ support. All counters are classified in Machine Specific Registers and could be accessed by the instructions `WRMSR`, `RDMSR` or `RDPMC`.

### 4.1.2 Virtual Machines

In this system, Xen Hypervisor is installed in a paravirtualized mode with two running virtual machine domains: Dom0 and DomU. As there are only two individual cores, each domain monopolizes a single processor core and operates tasks on Ubuntu 10.04 with Linux kernel 2.6.32. Dom0 is set as a privileged domain and starts up on core 0. DomU is the guest domain.
brought up on core 1 by \textit{xend} commands in Dom0. The domains are pinned to their assigned physical processor cores. The migration among cores is forbidden. The memories is also distributed impartially, 1.5GB per domain. These two domains have the same hardware features and running independently on the whole. There is a definition of criticalities of these two domains. 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.

4.2 Performance Monitoring

In this section, we propose how to utilize the built-in performance monitoring counters of our target processor to profile the off-core memory accesses in Xen hypervisor.

4.2.1 Performance Monitoring Counters on Athlon64 X2

Processors in AMD K8 series are featured with four performance counters per physical core. They are distributed in the address range of MSRs, from 0xC001_0004 to 0007. 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, controlling their operations. The bits in EvtSel are used to configure PMC’s working modes. 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 the 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 the detailed event is filled in bit 15-7.

K8 processors supports up to 87 events covering all execution stages of an instruction could reach. The detecting points are distributed over the components on-die, such as execution units, caches, memory controllers and interrupt controllers.

The following steps finish the simple run-and-collect operations on PMC0 for monitoring L1 data cache misses:

| BEGIN: |
|-----------------|-----------------|
| WRMSR Perfctr0 , 0 ; Reset PMC value |
| RDMRSR edx, EvtSel0 ; Read present setting of EvtSel |
| OR edx, 0x41 ; Monitor L1$ miss event |
| OR edx, 0x20000 ; Start counting |

Listing 4.1: Run-and-Collect of PMC
Both PMCs and EvtSels are accessible, no matter whether the performance counters is running or pausing.

For the overflow interrupt use, 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 a interrupt service routine. Here, PMCs could be re-initialized or assigned a new value.

### 4.2.2 Set-up Memory Access Monitoring in Xen

In this part, we are first describing how to settle down the initialization work for the performance monitoring counters, prior to using them in Xen Hypervisor.

```
Listing 4.2: _start_xen

void _init _start_xen(multiboot_info_t *mbi)
{
   ....
   cmdline_parse(cmdline); // Parsing cmdline in config
   ....
   console_init_preirq(); // Initialize serial console
   ....
   BOOT_ALLOCATOR // Memory allocating
   ....
   init_IRQ(); // Initialize IRQs on APIC
   trap_init(); // Initialize IDT
   ....
   smp_cpus_done(max_cpus); // Finishing cpu bring-ups
   do_initcalls; // Wake up system
   ....
   perf_start(); // Initialize PMCs
   ....
   dom0 = domain_create // Create Dom0
   domain_unpause_by_systemcontroller(dom0); // Schedule Dom0
}
```
In order to make the PMCs workable together with Xen hypervisor and VMs, the initialization should be done at the very beginning of system boot-up. Listing 2 illustrates the key logics in the first C function called by the early-stage assembly files. A feasible gap among these initializing steps need to be explored for allocating the PMC initialization function. Refer to [21], we get the conclusion that the steps before BOOT_ALLOCATOR are in the pre-initializing stage. Afterwards, the interrupt controller and the interrupt description table (IDT) are initialized. All CPU cores are formally brought up until smp_cpus_done. Then the system is finally "waken up" at do_initcalls. Up to this point, the appropriate condition is setup to start PMCs: running the CPU cores and the available interrupts. So the PMC initializing function perf_start() is called between do_initcalls and Dom0's creation.

The function perf_start() is declared as the initialization function for PMCs. It is required to implement the EvtSels configuration and the interrupt hook-up before PMC is working. A conceptual running flow of perf_start is shown in Figure 4.2.

![Figure 4.2: Flow Charts of PMC Initialization](image)

The process of initialization starts with the reservation operations on counter 0 of PMCs. Since PMC0 has been captured as a associated timer
for the NMI watchdog interrupt on the local APIC. PMC0 periodically signal the NMI interrupt to verify the system state, ensuring the system is running safely. To vacate PMCs and NMI for further use, this NMI watchdog needs to be disabled first. In the next step, EvtSels are configured according to the defined settings of PMC. The different functional bit ranges are be configured separately. The initial values are written in PMC afterwards. Here we register NMI as the vector of PMC overflow interrupt, since NMI is the non-blocked interrupt with high priority in APIC, satisfying the requirement of continuous monitoring in this design. Finally PMCs are started by EN bit enabled. Additionally, for the convenience of observing the PMC values, a sampling timer could be optionally turned on.

After PMCs are activated in Xen Environment, the problem turns back to how to choose the right metrics for the off-core memory access. Along with the designed throttling algorithm and the additional compensations, the numbers of memory accesses in a certain period is the key parameter to determine the extent of memory contentions. Refer to [20], K8 processor provides several memory access related events:

**E0h DRAM Accesses** indicates the number of total memory accesses conducted by the on-chip DRAM controller. Each access represents a full cacheline size 64byte transferred. There is no indication of the source(CPUs, I/O) or the access type(Read, Write).

**E9h CPU/IO Requests to Memory/IO** reflects the number of cache-line sized block requests from the request source(CPU, I/O) to the target(Memory, I/O). The Mask Unit for local CPU to Memory transfer is A8h.

**6Ch System Read Responses by Coherency State** represents the number of the responses from the system back to a single core for cacheline refills. Each response is same as one cacheline transferred.

**EAh Cache Block Commands** shows the total number of requests to system for cacheline transfer.

The first two events include all memory read and write operations happened through the memory controller, while the last event EAh only counts cacheline_ins. As the memory accesses by the reading operations are the target figures declared in the system model and algorithms, the event CacheBlockCommands is likely to measure the global off-chip memory accesses in our model. Although the event SystemReadResponsesbyCoherencyState is a good indicator for the core-to-system cacheline requests, it takes the inter-core cache coherency into account where the cacheline exchanges among the caches on different cores are possibly counted as memory accesses. According to the assumption that the VMs running on different cores own their private memory ranges, there is no data overlapped and shared between these 2 physical
cores. So that the event 6Ch counts only the responses from memory rather than the neighboring cores. We also verifies with several test cases that, the sum of event 6Ch values from two cores are equal to the global counting from EAh.

4.3 Throttling and Compensation Extension

This section is proposing how we extended the built-in scheduler of Xen hypervisor to throttle the memory accesses on the interfering VMs. It represents the extension framework and some details in implementation.

4.3.1 Scheduling in Xen

One of the key features of Xen hypervisor is to assign CPU cycles to the running VMs, ensuring each VM receiving the reasonable CPU resources. The CPU resource is abstracted with a conceptual scheduling unit VCPU. Similar to the common operating systems, a VCPU is scheduled as a thread, which is allocated on the physical CPU by the scheduler in Xen. From the guest OS side, there is no difference between VCPUs and the physical ones. The kernel threads are mapped to VCPUs.

As an abstraction of the physical CPU, VCPU keeps track of the general CPU infos such as states, register contents, interrupts and timers. Additionally, VCPU is defined in 4 run states: running, runnable, blocked and offline. The scheduler switches VCPUs among the pinned CPUs. When a VCPU is brought in, an context-switch operation is operated by the scheduler: the previous VCPU is saved and switched out, meanwhile a new VCPU is loaded and switched in.

Xen provides a universal framework of the scheduler, in schedule.c, containing all scheduling codes with the core functionalities. Extended from schedule.c, an abstract interface is present, as shown in Listing, for scheduling the algorithms’ implementation.

Listing 4.3: Xen Scheduler Interface

```c
struct scheduler {
    char *name;
    char *opt_name;
    unsigned int sched_id;

    void (*init)(void);
    . . . .
    int (*init_vcpu) (struct vcpu *);
    void (*destroy_vcpu) (struct vcpu *);
    void (*sleep) (struct vcpu *);
};
```
The key function \texttt{do\_schedule} is a sub-running of \texttt{schedule} in schedule.c. It determines the the next-to-run VCPU by the designed scheduling algorithm. Commonly, there is a queue storing all the available VCPUs called RUNQ in the scheduler. \texttt{do\_schedule} pick up the first element from RUNQ and pass it back to \texttt{schedule} for the further context-switch. If there is no runnable VCPU in RUNQ, an IDLE VCPU will be scheduled in and the physical core is set to idle as well.

The functions \texttt{wake} and \texttt{sleep}, just as the names imply, are a pair to resume and pause the VCPUs. \texttt{wake} is a sub-call of \texttt{unblock} and \texttt{wake} in schedule.c. It inserts the target VCPU back to RUNQ and raise an scheduling softirq signaling the scheduler to do schedule again. Reversely, \texttt{sleep} is called in \texttt{block} and \texttt{sleep} in schedule.c. It removes the VCPU from RUNQ.

Xen 4.0.1 has already been equipped with two schedulers: SEDF and Credit. The SEDF scheduler works as a periodic server, in which VCPU are leveled by their deadlines. The VCPU with high priority will be scheduled ahead. In Credit scheduler, each VCPU is attributed with a budget. The scheduler continues burning this budget and periodically replenish it. Only the VCPUs with positive budgets will be chosen in a high priority, while the burned-up VCPUs will be degraded into another queue. Both of them follows the interface structure and are integrated with their own scheduling algorithms.

4.3.2 Memory Access Throttling Framework

In this part, we are proposing the implementation of the memory access throttling mechanism based on Xen scheduler. As mentioned in Section 3.1, the interfering cores are required to behave controllable in the aspect of memory access requests, in order to guarantee the critical cores suffering interferes within a safety bound. In Xen Environment, we developed a throttling structure integrated with the internal scheduling framework of Xen, shown in Figure 4.3. Distinct with the original Xen, an extended Memory Access Controller is located between the interfering VM and the physical core, with the functionality of monitoring memory access flows and regulating the bandwidths. It will timely notify the scheduler taking further operations of cutting off the requests from VM to the hardware CPU, once the excessive memory accesses occur within a certain period.

The periodic server is an efficient but convenient way to provide or restrict a constant resource bandwidth. On a single interfering core, assuming
we authorize it consuming the budget $Q$ within a period $P$, it should be throttled once the budget burned up. So that, from system view, its memory access bandwidth is limited at $Q/P$ MB/S. It is precisely fitting our model and algorithm.

Section 4.2 has introduced how to monitor the memory accesses with PMCs on this platform. The functionality of overflow interrupts provides an appropriate way to monitor the memory access budgets and notify once they burned up. After the interrupt signals the scheduler, the further operations are taken to throttling the interfering cores. In Xen environment, as a VCPU abstracts a physical core, the VCPU, which generates memory access, should be immediately paused. Also in order to maintain the assigned bandwidth, this throttled VCPU should be waken up finally. The timing graph is shown in Figure 4.4.

In the first period, this interfering core consumes up the memory access budget $Q$ at the time point $\tau$. 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
The periodic server above is integrated as an extension on the Xen scheduler framework. Avoiding the modifications in the individual schedulers, we utilize the universal scheduler-related functions in schedule.c. Figure 4.5 illustrates the process in Xen.

When the PMC which is monitoring the off-core memory accesses get overflowed, the hooked interrupt NMI orientates code into the PMC overflow interrupt service routine. Before the 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 the disabled PMC beforehand. Till this step, code is still running in NMI context, where it is forbidden to interact with system tasks. As 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, VCPU is paused by an inherent function `vcpu_pause` in schedule.c. Finally the schedule soft irq is signaled in order to inform the system bringing in the IDLE VCPU.

The extension from the present framework to the compensation mechanism is that we have to compute the abundant budgets which actually not generate any interfere after each period. The remaining budget, calculated by the difference between the memory accesses on the critical core and the budgeting access on the interfering core, is then cumulated to the next period. The cumulative process ends up when the compensation cycles have accomplished.
Figure 4.5: Throttling Process in Xen
Chapter 5

Evaluation

In the previous chapters, we have illustrated a throttling algorithm on the interfering cores and its implemented framework on the basis of Xen Hypervisor. The next sections will detailly evaluate our implementation and summarize the performance of the designed algorithm on addressing the memory access contentions.

There are three major properties we would like to examine on our implemented platform:

1. Memory Throttling Performance: The interfering memory access flows ideally should be throttled timely once the monitoring PMCs get overflowed under this framework. And the flows could be regulated under the constraints of the configurable bandwidth budgets. The best phenomenon we expect is that the arrival curves of the evaluated tasks show the similar shape as the one in Figure 3.3.

2. Response Time on Critical VM: After verifying the functionalities of our testing platform, the next step is to examine whether the throttling algorithm is able to mitigate the present contentions on the memory accesses. The criterion is that the assigned deadline for a critical task need be secured when it co-runs with the other throttled interfering VMs.

3. Compensation Mechanism: As we proposed a compensation mechanism on the throttled cores, it is interesting to see how much the performance of interfering tasks could be lifted-up. Beforehand we need to check whether it will affect the running performance of the critical tasks.

The entire experimental evaluation is operated with SPEC CPU2006 Benchmark Suite. We select four benchmarks which have typical behaviors of memory access on this platform: 403.gcc, 429.mcf, 470.lbm and 459.GemsFDTD. As shown in Figure 1.2, 403.gcc behaves mild-mannered on
the aspect of memory access, while 470.lbm is the most aggressive one. The other two benchmarks generate the moderate memory access flows. Each one of them is executed critically on one VM, co-running with any of these four on the other throttled VM. The execution time on these 16 pairs of benchmarks are recorded for the further analysis.

5.1 Memory Throttling Performance

In this section we evaluate the performance of our implemented throttling framework, obtained by profiling the memory accesses on the throttled cores. An ideal throttling framework is desired to be equipped with the following two features:

1. Able to do the online monitoring of memory accesses and cut off the request flows immediately once the budgets consumed in the current period. Globally the memory access controller behaves as a periodic server of which the observed arrival curve is a staircase function.

2. The memory Access flows outcoming from the throttled VM are reshaped by the assigned throttling budget in the memory access controller. The consumed budget in a period should be strictly constrained within the setting value.

As introduced in Chapter 4, the memory access controller is embedded in Xen Environment, throttling the VM labeled as interfering. In the following experiments, 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 the system activities across the Xen Hypervisor layer and the 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 the up-running VMs. In user-space, Xentrace daemon reads and takes the buffer down to the log files. In this case, we could customize the trace event ”Mem_Acc 0x28011” tracking the assigned memory access counter PMC. This event is called every 1ms, so from user-space we get the phase graph of running task in the resolution of 1ms.

Figure 5.1 reveals the throttling performance of the implemented framework working on benchmark 403.gcc. The top graph exhibits the phase graph when 403.gcc running without any memory bandwidth limitation.
Figure 5.1: 403.gcc Phase Graph Comparison
Figure 5.2: 470.lbm Phase Graph Comparison
The transitional memory access within 1ms could reach 1.3 MB in maximum, while it is around 0.5 MB in average. A complete execution takes approximately 3.3s. The middle graph is the profiled phase graph after being throttled. The memory access controller is configured with the budget in 10ms, equivalent to the bandwidth of 91MB/s. Obviously the impulses in the middle phase graph is trimly scraped at the level of 0.9e+06. And the total execution time is delayed to 7s. Compared with these two zoom-in windows, the memory accesses behave dense before being throttled, while it is regularly sparse in a rate of 10ms. The bottom graph contrasts the arrival curves when throttled and non-throttled. On a macro level, the throttled arrival curve is linearly increasing at a constant rate, while the non-throttled one randomly follows the original phases. The zoom-in window displays a staircase curve with the step length of 10ms, similar to Figure 3.3.

Distinct with 403.gcc, benchmark 470.lbm generates more aggressive memory access flows, 6 times in the rate as shown in Figure 1.2. We also experiment it under the same configurations, shown in Figure 5.2. 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. Then these intensive access flows are regulated into a smooth rectangle shape with the height of 0.9e+06. The total execution time is extended from 13s to 200s.

Refer to the previous two expecting features, the memory throttling framework is verified performing properly: the memory access flows are cut off timely and the overall arrival curve is successfully shaped by the configured budget.

5.2 Response Time on Critical VM

Section 3.3 introduced an algorithm of computing the suitable throttling budget, to secure the deadlines of the critical tasks. The following section is examining whether the computed budgets could intervene in the memory access contentions between VMs. Here we have an obvious criterion: no deadline offense is allowed for the critical tasks.

As discussed in Section 4.1.2, two VMs attributed with different criticalities are installed on this multicore platform. Each of them monopolizes a single CPU core. The interfering VM has already been equipped with a memory access controller, throttling the memory access request from the running tasks in guest OS. We utilize any two of three benchmarks, which generate the considerable memory accesses, as a co-running couple on the critical and the interfering VMs. The execution time of the benchmark on the critical VM is used to compare with the assigned deadline.

Next part, we start illustrating the algorithm verification with the test case of 429.mcf as a critical task, co-running with one of 429.mcf, 470.lbm, 459.GemsFDTD on interfering VM. 403.gcc is not considered because of
the low interfering ratio shown in Figure 1.1.

Table 5.1: Worst-case Execution time of 429.mcf

<table>
<thead>
<tr>
<th>Co-runner</th>
<th>Solo</th>
<th>429.mcf</th>
<th>470.lbm</th>
<th>459.GemsFDTD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exe-time/(s)</td>
<td>10.38</td>
<td>11.73</td>
<td>17.57</td>
<td>13.10</td>
</tr>
</tbody>
</table>

Before computing the throttling budget, we first measured an average execution time of 429.mcf when it is solo-running: 10.38s. Table 5.1 represents the worst case execution times when 429.mcf co-run with the other benchmarks. Reviewing the throttling algorithm mentioned in Section 3.3, the budget computation requires two assigned prior conditions: Deadline $D_i$ and replenish period $P$, as the parameters in Inequation:

$$S(D_i, Q) = \left\lfloor \frac{D_i - \Delta}{P} \right\rfloor \cdot \Delta + U((D_i - \Delta) \mod P, Q) \leq \text{Res}_i$$ (5.1)

where $\Delta = Q \cdot L$. $\text{Res}_i$ is the available stalled time of the critical task. Because there is the only one single task running on the critical VM, $\text{Res}_i = D_i - C_i$. Refer to the worse-case execution times co-running with the interfering VM, the deadline of task 429.mcf is set at 11.38s, in a 1s distance to solo-running, indicating $\text{Res}_i$ is equal to 1s. The replenish period $P$ is configured at 10ms. [22] tells us the worst case latency for a single cacheline size access to memory is 64ns.

Then we utilize Algorithm 2 searching a safe budget for the critical task 429.mcf. After 29 iterations, $S(D_i, Q)$ is approximately equal to $\text{Res}_i$ at the point of 13718.17. It means that 13718 times of cacheline size memory accesses are allowed on the interfering VM. In this case, the worst case execution time of the critical task will be no later than the deadline $D$.

In this experiment, the critical 429.mcf runs 200 times accompany with the interfering throttle tasks. The execution time in each cycle is recorded. Figure 5.3 is the collected execution time of 429.mcf co-running with three benchmarks. The average degradation of 429.mcf is controlled within 9% and deadline 11.38s is successfully secured. Even with the most aggressive co-runner 470.lbm, the critical task resists the interferes from intensive memory accesses.

The same experiments are repeated on 470.lbm and 459.GemsFDTD as well. Figure 5.4 concludes 470.lbm’s co-running execution time after the interfering VM being throttled. The deadline is assigned at 15.5s in the distance of 1.5s from the solo-running execution time. Figure 5.5 is for 459.GemsFDTD, when the critical task is entitled to have 1s stalled time before the deadline.

These results prove that our algorithm for determining the throttling budget is capable to inhibit the excessive memory interferes from the co-
Figure 5.3: 429.mcf Execution Time

Figure 5.4: 470.lbm Execution Time
running VMs/cores. The expected deadlines for critical tasks are successfully secured. 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, the memory accesses are not generated continuously nor intensively. The factors of processor architecture, such as inter-connection bus and memory controller, also affect the extent of memory contentions in reality. Concluded from the experimental results, the budget is considered pessimistically and has potential to find a safe but more loose bound for the interfering VMs/cores.

Figure 5.5: 459.GemsFDTD Execution Time

5.3 Compensation Mechanism

The previous workable throttling algorithm has controlled the impacts from memory contentions among the on-die cores. Simultaneously it brings side-effects to the throttled interfering cores. The performance of the interfering tasks is sharply reduced by the tight budget on the memory access controller. As introduced in Section 3.4, the Compensation Mechanism is an exploring trial to mitigate the pain of performance on the interfering side.

In this experiment, we are verifying whether this compensation mechanism could lift up the performance on the throttled core, on the premise of no offense to the deadline of critical tasks. The 2-period reset compen-
sating policy is turned on in this throttling framework, where the abundant budget from the previous replenishing period will be accumulated to the next and reset at the end of next period, shown in Figure 3.5. The execution time of interfering tasks could be decreased up to 67% of the one before compensated. Continued with the last section, 429.mcf, 470.lbm and 459.GemsFDTD are coupled as the co-running tasks.

Figure 5.6: Execution Time of Interfering Tasks after Compensation

Figure 5.6 represents the measured execution time of interfering cores after compensation. The overall performance has been promoted up to 20% and the extent of speed-up vary with the different co-runners on the critical side. For the case of running with 470.lbm, all three benchmarks get the least promotion, whereas the best performance occurs with 459.GemsFDTD. Due to the intensive memory access flows by 470.lbm, the co-runners get limited abundant budgets from the previous period. For moderate critical tasks, the interfering core has more opportunities to get benefits from this compensation mechanism.

Although these experiments have proved the compensation mechanism is able to mitigate the pain on the interfering side, we still cannot conclude that the compensation is workable for the whole system. Whether it will affect the running performance on the critical side or even put the tasks in danger, is another key criterion for our evaluation. Figure 5.7 to 5.9 reflects the changes of the execution time on the critical side after utilizing this compensation mechanism. The overall execution time increases slightly, but still in distance with the assigned deadline. The difference in the critical task running times is related with our system models. As we assumed the Round-
<table>
<thead>
<tr>
<th>s</th>
<th>Execution time</th>
<th>Compensated</th>
<th>Normal</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>429.mcf</td>
<td>10</td>
<td>10.2</td>
<td>10.4</td>
<td>10.6</td>
</tr>
<tr>
<td>470.lbm</td>
<td>14</td>
<td>14.5</td>
<td>15</td>
<td>15.5</td>
</tr>
<tr>
<td>459.GemsFDTD</td>
<td>429.mcf</td>
<td>470.lbm</td>
<td>459.GemsFDTD</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.7: Comparison of Critical 429.mcf after Compensated

Figure 5.8: Comparison of Critical 470.lbm after Compensated
robin arbitration on the memory access path from cores, real processors own more complicated inner structures, leading unpredictable contentions. Generally, the compensation mechanism could actually lift-up the system performance, meanwhile the deadline for all the critical tasks are successfully secured.

Figure 5.9: Comparison of Critical GemsFDTD after Compensated
Chapter 6

Related Works

A rich body of studies have investigated and addressed the shared-resource contention by different performance isolating methods in recent years. S.Bak [23] 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 bus. B.Akesson [16] 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. A.Fedorova [24] 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 the work of [6], 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.

[25] presents a similar way as we implemented, also inspired by [15]’s algorithm. It proposed a software based memory throttling mechanism which guarantees the schedulability of critical tasks. The mechanism was built in Linux kernel. Different from their design, our algorithm is based on a more tight and precise arrival curve: staircase function, rather than linear line in [25]. Additionally the mitigation of pain on throttled interfering side is considered in our design and an effective compensation mechanism is proposed. Our Xen-based framework provides the better isolating performance and more flexible portability.
Chapter 7

Conclusion and Future Work

This thesis work is carried out to address the shared memory access contentsions on the multicore platform with real-time requirements. A throttling algorithm is designed for computing a suitable budget for the memory access controller on the interfering cores, which guarantees the safety of tasks on the critical cores. In addition to mitigate the pains by this throttling mechanism, a compensation mechanism for the throttled cores are introduced, in which the throttling budgets are dynamically adjusted according to the number of potential memory access contentsions on the critical side.

Virtualization solution is imported to enforce the performance isolation among the on-die cores. A pre-defined throttling framework has been integrated and implemented in Xen Hypervisor. We also built up a testing platform for experimentally demonstrating our mechanism in Chapter 4.

The evaluations in Chapter 5 shows the performance of our throttling framework. The phase graphs after throttled prove that the interfering memory access flows could be successfully regulated. The response time on the critical side is examined no offense to the security deadlines. At last, the compensation mechanism is able to lift-up the performance on the throttled side.

The overall results illustrates that both the designed algorithm and the platform implementation, perform as the pre-defined requirements: able to control the memory access flows from shared-memory cores, prevent the critical tasks from suffering the interferes on memory accessing.

Future work could be extended and improved in the following aspects:

1. Starting with the system model, we assume the inter-connection bus with Round-robin arbitration and the memory controller following FCFS. This model may be too pessimistic for the real multicore processors. The new models closer to reality, such as FR-FCFS and TDMA, could be taken into consideration for the system modeling.

2. With the development of modern multicore processors, the performance monitoring counters can handle more particular hardware events.
For example in Intel Nehalem processors, the memory access latency after a certain amount of requests could be profiled. Utilizing these more detailed measuring events, the precision of controlling memory access contentions will be optimized.

3. Due to the limitation of hardware, we did not import any real-time operating system in our implementation. In future work, working with RT-linux could help exploring more real-time related events in the multicore processors.
Bibliography


[22] Amd k8 (athlon 64).
