Memory Access Control in Multiprocessor for Real-time Systems with Mixed Criticality

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

***Abstract*—Shared resource access interference, particularly memory and system bus, is a big challenge in designing predictable real-time systems because its worst case behavior can significantly differ. In this paper, we propose a software based memory throttling mechanism to explicitly control the memory interference. We developed analytic solutions to compute proper throttling parameters that satisfy schedulability of critical tasks while minimize performance impact caused by throttling. We implemented the mechanism in Linux kernel and evaluated isolation guarantee and overall performance impact using a set of synthetic and real applications.**

I. Introduction

Data intensive workloads, which require frequent memory accesses, are increasingly more pervasive in modern embedded computing systems including critical real-time systems. For example, an aircraft now processes massive vision data in real-time to track objects in flight [1]. Processing such massive data requires more computing power. Therefore, there is a growing need for powerful multiprocessor to consolidate such workloads.

Consolidating data intensive tasks together with critical real-time tasks, however, poses a significant challenge due to interference on shared resources such as system bus and memory. It becomes more apparent as core count and memory intensity of tasks increase. The authors of [2] shows that a task can suffer 300% WCET increase due to memory interference even when tasks spend only 10% of their time on fetching memory in an eight core system.

One solution is to use specialized hardware which has capabilities to control such interferences. For example, a predictable DRAM controller [3] can provide guaranteed bandwidth and latency on accessing memory. Similarly a TDMA based system bus [4] can provide timing guarantee on accessing the shared bus. Hardware based approaches, however, prevent us from using cost effective commercial off-the shelf (COTS) components.

In this paper, we propose to use a software based memory throttling mechanism to explicitly control the memory interference. The basic idea of memory throttling is periodically limit the amount of memory accesses (i.e., last level cache-misses or LLC misses) similar to aperiodic servers for CPU bandwidth reservation [5]. In our implementation, OS scheduler directly monitors cache-misses of each core and dequeue/enqueue tasks based on the specified cache-miss budget and the period.

Using this mechanism, we are interested in protecting critical tasks from non-critical tasks where tasks are partitioned based on their criticality. As a first step, we consider a scenario where critical tasks run on a single core, we call *critical core*, and non-critical tasks run on the rest of the cores, we call *interfering cores*, as shown Figure 1. The interfering cores are, however, throttled using our memory throttling mechanism. Our goal is to find the throttling parameters, namely budgets for a given period value, on the interfering cores that satisfies schedulability of tasks on the critical core while minimizing performance impact of tasks on the interfering cores.

On throttling multiple interfering cores, we consider a static and a dynamic throttling strategies which differs in how budget is allocated in each period. We describe algorithms to get analytic solution on computing throttling parameters. We implemented the throttling mechanism on Linux kernel by extending standard group scheduling interface. We experimentally validate how the computed throttling parameters affect execution time of tasks on the critical core. We also compare the effect of static or dynamic throttling strategies in terms of slowdown experienced due to throttling on the interfering cores.

Although using throttling neither increases memory bandwidth of the hardware, nor reduces the memory access requests from tasks, it provides isolation for critical core among multiple cores. Our software based approach allows us to use COTS components and does not require any modification on the existing applications.

Remainder sections are organized as follows. Section II introduces memory throttling mechanism. Section III formally defines system model and the problem. Section IV describes solution in the case of single interfering core. Section V extends the results to multiple interfering cores. Section VI shows experiment results. We discuss the limitation and future work in Section VII and conclude in Section IX.

II. Memory Throttling

In this section, we introduce our software based memory throttling controller. In a typical shared memory multi-core system, there is a shared system bus and memory controller that arbitrates memory read/write requests among cores. The arbitration scheme usually tries to maximize the bus bandwidth and it is unaware of priority between memory requests. As a consequence, individual request can be delayed by requests from the other cores. Memory throttling is a technique to limit the memory request rate of each core. For example, if memory requests from a core exceeds a predefined value for a given duration of time, a memory

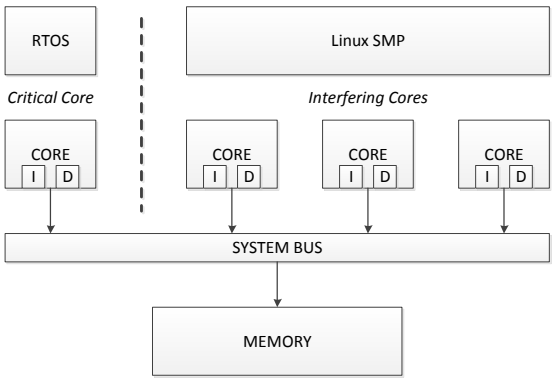


Figure 1: System model.

controller can delay the requests in order to maintain the specified rate for the core. In this way, we can limit the level of allowed interference of each core. While it can be implemented in hardware [6], we choose a software based approach for flexibility.

Fig. 2 shows how the throttling is performed in software. While executing tasks on a core, the scheduler monitors memory traffic by accounting the number of last level cache (LLC) misses. If the memory traffic exceeds the pre-defined budget, Q, the scheduler throttles tasks by temporarily dequeuing all the tasks in the run queue for the core so that no further requests can be performed in that period, P. At the beginning of the next period, the scheduler replenishes the budget in full and re-enqueues the throttled tasks, if any. To monitor LLC misses, we configured a hardware performance counter (PMC) in each core.

The behavior of throttling mechanism is similar to deferrable server [5] in aperiodic task scheduling in the sense that it allows quota to be used in full at any time during the interval.

We implemented the throttling mechanism by extending the standard Linux group scheduling interface called *cgroup*. The cgroup interface is originally designed to specify fraction of system resources such as CPU cores and memory capacity to a group of tasks. We extended the cgroup interface so that we can specify memory bandwidth with a pair of period P and budget Q.

This architecture allows us to enforce memory throttling

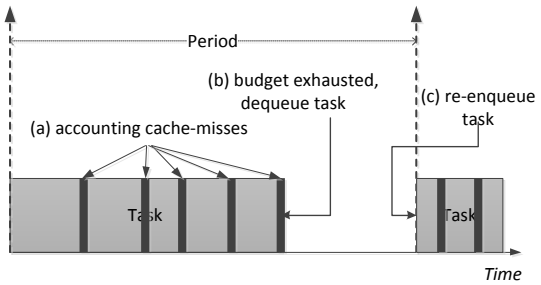


Figure 2: Basic throttling mechanism.

(1) for each individual core by creating a cgroup for each core, or (2) for a group of cores by creating a single cgroup and assigning the cores to the cgroup. We call the former scheme *static budget assignment* and the latter *dynamic budget assignment*, which we will detail in Section V.

In this paper, we use the described throttling mechanism as a tool for providing performance isolation for a *critical core* running critical tasks, from other interfering cores running potentially unpredictable workload. Yet another goal is to minimize delay for the tasks running on the *interfering cores* while satisfying the schedulability tasks on the critical core.

In the next section, we first formally define the system model and the problem.

III. System Model

We consider a multiprocessor architecture as shown in Fig. 1 where system bus and memory are shared among cores and each core has its private cache. We assume that cache miss is synchronous in the sense that whenever there is a miss, the core is stalling until the cacheline is fetched from the memory. There is only one DRAM controller connected to the system bus which is a common system configuration in practice. Furthermore, we assume that a single core can fully utilize memory bandwidth and the bus arbitration scheme is based on round-robin algorithm. These assumptions are reasonable for many existing platforms including the Intel platform we used in this paper.

We categorize cores into two groups, a core under analysis which we call a *critical core* and *interfering cores*. For the core under analysis, we assume that a fixed priority preemptive scheduler is used to schedule tasks. We assume WCET and the worst case number of cache misses for each task are given a priori. These parameters can be obtained from static analysis or from measurement by running in isolation. We assume preemption does not affect the number of cache-misses of a task, for example by partitioning cache to each task.

On the core under analysis, i.e., *critical core*, a set $\mathcal{T} = \tau_1, \ldots, \tau_n$ of n periodic real-time tasks are scheduled with a deadline monotonic scheduling algorithm; the period and the relative deadline of τ_i is denoted by T_i and D_i ($D_i < T_i$). The WCET of a task is denoted by C_i and the number of worst case cache misses is given by CM_i. Note that the CM_i does not necessarily coincide the worst case execution path. We denote the subset of tasks with priority higher/lower than task τ_i with $hp(i)/lp(i)$, and further more, let $hep(i) = hp(i) \cup \tau_i$.

On the *interfering cores*, the number of cache-misses can occur is throttled with period, P, and cache miss budget, Q as described in Section II. The budget is distributed among the interfering cores either (1) statically at the beginning of each period, or (2) dynamically at runtime based on the demand of each core. We made no assumption about the scheduling policies on the interfering cores. In other words, any scheduling algorithm can be used (e.g., CFS in Linux).

The goal is to find the throttling configuration, budget Q values for a given period P, for throttled cores such

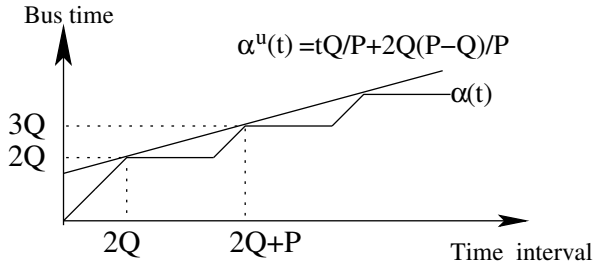


Figure 3: Arrival curve and its upper bound for one throttled core with P as period and Q as the cache-miss budget.

that satisfies schedulability of critical tasks on the critical core while minimizing slowdown of tasks running on the interfering cores.

IV. Single Interfering Core

In this section we consider a simple case where there are only two cores: one is the core under analysis with critical tasks assigned to it and the other is the interfering core. We later extend our analysis to consider multiple interfering cores in the next section.

We first describe how memory contention from the interfering core can be estimated based on the throttling parameters as presented in [7]. We then extend the response time analysis by taking into account the task stalling caused by contention from interfering cores. Finally, we formulate a problem of finding a throttling budget Q for a given period P of the interfering core such that the schedulability of the tasks assigned on the critical core is satisfied.

A. Flow "reshaping" by throttling

In order to account the task stalling due to the contention on the shared memory, we need to know the memory access pattern of the task. However, this turns to be rather complicated since it heavily depends on the dynamic task execution as well as the scheduling algorithm applied on the system. Prior work either assumes a specific memory access pattern [8] or a static cyclic scheduler [2], [4], however, these approaches suffer limitations when applying to the real applications.

The throttling mechanism, described in Section II, provides another alternative to account the memory accesses from the interfering core. The throttling controller works independently of the specific scheduler and task set on the throttled core, it simply stalls the task on the throttled core when the budget is consumed within the period.

The arrival curve of the flow can be derived similar to the method as commonly found in [9], i.e., the maximum possible traffic amount $\alpha(t)$ for a given time interval t. In the worst case the core can generate up to $2Q$ continuous cache misses in a time window with $2Q$ length due to the back logged one Q from the previous period, and then another Q every period. The derived flow arrival curve is shown in Figure 3. Notice that this curve is a step function and the upper bound of the arrival curve, $\alpha^u(t)$, is also depicted in the figure.

B. Stall time calculation

As we assumed that the inter-connection network to memory is bus and its arbitration is based on round robin, each memory access could be delayed by one memory access from the other core. The stall that one task can suffer depends on both the number of memory accesses this task needs to perform, as well as the number of memory accesses generated on the other core during this task's execution. To this end, we show how these two factors and the task stall can be accounted.

Let us consider a task which generates CM memory accesses in worst case. Since each access can be delayed by interfering core's access, the maximum stall time this task can suffer is upper bounded by $CM \cdot L$ where L is the time needed to perform one memory access. It represents the task interior requirement on the memory resource.

The task stall due to the memory accesses from the other core can be estimated by accounting the memory access traffic during this task's execution. However, unlike the CM value which does not change depending on its stall time, there is a circular dependency between the task stall time and the number of memory accesses from the other core. Figure 4[1] illustrates the stall for one task with worst case execution time C, measured in isolation, and a throttled core with arrival curve $\alpha(t)$. The amount of traffic $d^1 = \alpha(C)$ during the task's execution could interfere this task and cause an increase of d^1 in its execution time. However, the increased execution time, $C + d^1$, would possibly suffer a higher level of memory traffic interference, which is equal to $d^2 = \alpha(C + d^1)$. This process continues until it converges. As clearly showed in the figure, this can be formulated as an iterative procedure, it terminates and returns the stall for this task when the procedure is converged at Δ, that is

$$d^{(k+1)} = \alpha(C + d^{(k)}) \tag{1}$$

Finally, the task stall cannot exceed one of these two factors—cache misses of the task under analysis and the interfering flow of the other core, hence, the stalled execution time $\widehat{C}$ for one task can be expressed as the solution of this iterative procedure:

$$\widehat{C^{(k+1)}} = C + \min\{CM \cdot L, \alpha(\widehat{C^{(k)}})\} \tag{2}$$

C. Extended response time analysis

With the stalled execution time for each task ,we can perform the classical response time analysis [10]. However, this turns out to be quite pessimistic as each task under analysis is assumed to possibly suffer the $2Q$ delay. In reality, this cannot happen since for a continuous time interval, there could be only one back-logged Q at the beginning. We still use the iterative response time analysis; refine it by adding another term to account the delay due to the memory contention with other core. The main intuition is that, instead of applying the delay upon each single task,

[1]The time length in the figure is selected to better demonstrate the iterative procedure of task stall calculation. i.e., $\Delta \leq C$ and $\alpha(t) \leq t$ in Figure 4.

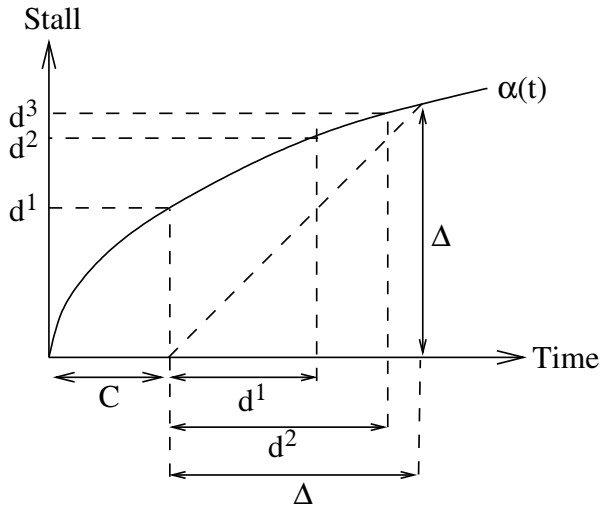


Figure 4: The circular dependency between the traffic from throttled core and the stall of task on the critical core, d represents the amount of traffic for a given time interval while Δ represents the overall stall for this task.

we directly compute the delay for the response time at each iteration.

The iterative response time calculation is expressed as follows:

$$R_i^{(k+1)}=C_i+\sum_{\tau_j\in hp(i)}\left\lceil\frac{R_i^{(k)}}{T_j}\right\rceil\cdot C_j+\min\{\mathcal{N}(R^{(k)})\cdot L,\alpha(R^{(k)})\} \tag{3}$$

where

$$\mathcal{N}(t)=CM_i+\sum_{\tau_j\in hp(i)}\left\lceil\frac{t}{T_j}\right\rceil\cdot CM_j. \tag{4}$$

When ignoring the third term on the RHS of the equation, this is exactly the same as the classical response time analysis. The new introduced third term represents the maximum possible stall upon all tasks executing (include τ_i itself and the preempting tasks) during the time interval R^k. Specifically, number $\mathcal{N}(R^k)$ captures the total number of cache misses among all tasks executing within this time interval R^k, whereas $\alpha(R^k)$ is the total memory access traffic from the throttled core during R^k. Following the stall analysis for one task in previous subsection, we know that the third term (the min function) is the total stall caused by memory contention between all the executing tasks and the traffic from the throttled core within this R_i time interval. The response time of τ_i is obtained when R^k converges as the classical response time analysis.

D. Calculation of throttling budget

Given a throttling period P on the throttled core, we now consider how to calculate the maximum budget value Q such that the tasks assigned on the critical core can have their deadlines guaranteed. We first assume that tasks are schedulable when they run in isolation. We also assume the tasks on the critical core are scheduled according to fixed priority assignment with τ_1 being the highest priority task, the budget Q for the throttled core can be calculated by considering the feasibility of individual task on the critical core, in decreasing priority order. Let Q_i denote the maximum budget value such that the subset $hep(i)$ on critical core have their deadline satisfied. The following properties can be derived.

Property 1. The Q_i sequence is monotonically non increasing.

Property 2. The maximum budget that can be assigned to the throttled core is Q_n where n is the number of tasks on the critical core.

Proof: Property 1 can be easily proved by contradiction. Suppose for a task τ_i, there exists one task $\tau_j \in lp(i)$ with Q_j value larger than Q_i. Since Q_j is the value such that the task set $hep(j)$ is feasible, hence task subset $hep(i) \in hp(j)$ is also feasible. This contradicts the assumption that Q_i is largest value such that $hep(i)$ is feasible.

Property 2 follows directly from Property 1. ■

The overall algorithm to compute the budget Q is presented in Algorithm 1. Line 2 initializes the Q value to the period P; this serves as the upper bound of Q. Then the tasks on the critical are checked in decreasing priority order: if the task is verified to be feasible by computing its response time, then the algorithm moves on to the next task; otherwise, the value Q_i is calculated and Q is updated with Q_i. Notice that since the algorithm follows the priority order, $hp(i)$ is guaranteed to be feasible when checking task τ_i, hence Line 5 only needs to deal with task τ_i alone. Finally, when the algorithm finishes all the tasks, it returns the value Q as the maximum budget for the throttled core.

Input: The throttling period P and the taskset parameters on the critical core.
Output: The maximum budget value Q such that the critical core is feasible

```
1 begin
2     Q = P ;
3     for i ← 1 to n do
4         Calculate the response time R_i according to
          Equation (3) ;
5         if R_i > D_i then
6             Calculate Q_i such that τ_i is feasible ;
7             update Q with Q = Q_i;
8     return Q ;
```

Algorithm 1: Calculate budget Q such that the critical core is feasible.

Now it remains to explain how to calculate Q_i to make τ_i feasible, as the Line 5 in the algorithm. Notice the response time calculation depends on both the arrival times of high priority tasks and the delay caused by the throttled core during the response time interval, which makes the Q calculation not straightforward. Since for a given traffic delay function $\alpha(t)$, the response time function for τ_i has discontinuous points only at the arrival time of high priority task. We denote this set of time instants as the testing set as

in the following equation.

$$\mathcal{TS}(\tau_i) \doteq \{t | t \in [C_i, D_i] \cap t = k \cdot T_j \ \forall \tau_j \in hp(i),\ k \in \mathbb{N}\} \tag{5}$$

For a certain time point from $\mathcal{TS}(\tau_i)$, the number of preemptions on τ_i is a fixed value independently of the traffic function, which allows us to solve the $\alpha(t)$ function in terms of budget Q. The Q can be calculated by the following equation:

$$Q_i(t) = \begin{cases} \emptyset; & \text{if } \mathcal{S}(t) \leq 0 \\ \infty; & \text{if } \mathcal{S}(t) \geq \mathcal{N}(t) \cdot L \\ \frac{2P+t}{4} - \frac{((2P+t)^2 - 8\mathcal{S}(t)P)^{1/2}}{4} & \\ & \text{if } 0 < \mathcal{S}(t) < \mathcal{N}(t) \cdot L \end{cases} \tag{6}$$

where $\mathcal{S}(t)$ is the maximum allowed stall time for τ_i at time t that still satisfies the schdulability constraints and can be expressed

$$\mathcal{S}(t) \doteq t - C_i - \sum_{\tau_j \in hp(i)} \left\lceil \frac{t}{T_j} \right\rceil \cdot C_j. \tag{7}$$

Proof: The first case is when the task execution plus the interference due to the preemptions from high priority tasks already exceeds the time interval t, hence, there is no solution at this point. On the other hand, the second line shows the situation when the cache misses from the tasks executing on this core is small or the task slack is big enough, such that the delay bound from the task itself is enough, regardless of the traffic flow on the other core.

The throttling mechanism plays an important role in the third case: we have to limit the memory access traffic from the other core by controlling the budget Q so that the delay on this task would not cause deadline miss. With the upper bound of traffic function as shown in Fig. 3, we have

$$\alpha(t) \leq \alpha^u(t) = t\frac{Q}{P} + \frac{2Q(P-Q)}{P}.$$

hence it is enough to guarantee that

$$t\frac{Q}{P} + \frac{2Q(P-Q)}{P} \leq \mathcal{S}(t). \tag{8}$$

Solve this equation and discard the unfeasible solution we get

$$Q \leq \frac{2P+t}{4} - \frac{((2P+t)^2 - 8\mathcal{S}(t)P)^{1/2}}{4}.$$

This proves the Equation (6). ∎

Notice that τ_i is feasible if there exists one point from $\mathcal{TS}(\tau_i)$ that satisfies Equation (3), therefore, the budget value Q_i that guarantees the feasibility of τ_i can be expressed as:

$$Q_i = \max_{t \in \mathcal{TS}(\tau_i)} Q_i(t) \tag{9}$$

where $Q_i(t)$ is computed according to Equation (6).

With this calculated Q_i for each task, now we can follow Algorithm 1 to calculate the budget Q for the throttled core with the schedulability of the tasks on the critical core guaranteed.

V. Multiple Interfering Cores

Having shown the interference from one single core to the critical core, now we extend the result to the case when the system contains several throttled cores, which both contend for the access to the main memory with the critical core. This is the situation shown in Figure 1. Depends on how the budget is distributed among all the throttled cores, we propose two different throttling schemes: *static budget distribution*, which assigns fixed amount of budget to each core, and *dynamic budget distribution*, which assigns dynamically assign budget on-demand basis. For each scheme, our goal is to find a budget assignment for throttled cores that maximizes utilization of throttled cores that satisfy schedulability of the critical core.

A. Static budget distribution

In this scheme, we consider each core owns its budget which is statically distributed from a global budget and all cores share the same period. We assume the static distribution proportion for each throttled core is given in priori by, for example, analyzing the characteristics of tasks on each throttled core.

First, we describe method to compute the stall of one task on the critical core. Assuming each throttled core has an individual budget, it is easy to see the upper bound of its memory access can be computed by the corresponding period and budget. Similar to the analysis presented in the previous section, we denote the arrival curve for each throttled core by $\alpha_c(t)$, where $c = 1, \ldots, M$ and M is the number of throttled cores. The increased execution time of one single task τ_l on the critical core, denoted by $\widehat{C_l}$, can be computed by the following iterative way:

$$\widehat{C_l^{(k+1)}} = C_l + \sum_{1 \leq c \leq M} \min\{CM_l \cdot L,\ \alpha_c(\widehat{C_l^{(k)}})\} \tag{10}$$

The stalled execution time is calculated by summing up all the delays caused by each throttled core and its original execution time. The delay factor from each core is determined similar to Equation (2) as analyzed in the section before: the cache misses due to the task itself or the memory traffic flow from the throttled cores. When the iterative procedure converges, it returns the increased execution time, which includes the original execution requirement plus maximum delay from all throttled cores during this increased execution time.

The response time analysis can be extended to the multi throttled core case with the similar technique used for the single throttled core. The main intuition is that, at each iteration we should consider delay from each throttled core and sum them up, as showed in the next equation.

$$R_i^{(k+1)} = C_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil C_j + \sum_{1 \leq c \leq M} \min\{\mathcal{N}(R^{(k)})L, \alpha_c(R^{(k)})\} \tag{11}$$

, where $\mathcal{N}(t)$ is defined in Equation (4) and c is the index of throttled core.

We assume throttled core with index $c \in [1, M]$ is assigned a fixed ratio r_c of the global budget, i.e., the budget for this core is $r_c Q_i$. Furthermore, let the throttled cores are indexed by increasing ratio order. Given the distribution proportion of global budget, how to calculate the budget still turns out to be a problem not easy to solve. The tricky part is that when summing up the delay factor from each throttled core, it is obtained either from the task cache miss (as the first item in the $\min$ function) or the throttled core flow (as the second item in the $\min$ function), depending on the specific arrival curve of this throttled core and the time instant.

To consider the schedulability of one task $\tau_i, i \in [1, N]$, on the critical core, we get the testing set $\mathcal{TS}(\tau_i)$ for τ_i as in Equation (5) and the slack $\mathcal{S}(t)$, as defined in Equation (7), at one specific time instant in the testing set. When considering the schedulability of the task τ_i, for a given time instant $t \in \mathcal{TS}(\tau_i)$ and one throttled core with index $c \in [1, M]$, we calculate the ranges of global budget so that we can determine the delay from this throttled core is obtained from which factor. Notice that because the throttled cores are indexed by the order of budget ratio, therefore the arrival function, we can now determine the delay factor for the remaining throttled cores. We solve the equation of the $Q_i^{c,t}$ and consider all the indexes of c, t, to get the value of Q_i to make task τ_i on the critical core schedulable. Finally, we merge all the solutions for each task τ_i on the critical core to get the final result of Q.

The computation process is explained in detail now. We need to distinguish three cases depends on the range of value $\mathcal{S}(t)$:

When $\mathcal{S}(t)$ is no larger than 0. In this case, the task on the critical core could not suffer any delay *at this time point*. Therefore, the budget would be zero.

When $\mathcal{S}(t)$ is no less than $M \cdot \mathcal{N}(t)L$. In this case, no matter how large the traffic flow would be, since the delay from each core is bounded by $\mathcal{N}(t)L$, the budget can be assigned arbitrarily.

We focus on the case when $\mathcal{S}(t)$ is within the range between the two previous values. The main idea is to divide the throttled cores by the return value of the $\min$ function in Equation (11). Since the throttled cores are indexed by increasing budget ratio, there may exist one core with index c such that, for a specific time instant t and a global budget Q_i, the delay from the throttled core with a index no less than c is obtained from the task cache miss term. These cores have even higher traffic flow and the $\min$ function would return the task cache miss item. Now consider the $\min$ function and replace $R^{(k)}$ by t we have:

$$\alpha_c(t) \geq \mathcal{N}(t)L$$

where $\mathcal{N}(t)$ is defined in Equation (4). Notice the $\alpha_c(t)$ function is a step function and its upper bound $\alpha_c^u(t)$ is used to simplify the computation.

$$\alpha_c^u(t) = t\frac{r_c Q_i}{P} + \frac{2r_c Q_i(P - r_c Q_i)}{P} \geq \mathcal{N}(t)L$$

Solving this equation, let $Q_i^L(c,t)$ denote the lower bound of Q value such that stall caused by core c at time t is obtained from the cache miss part, we have:

$$Q_i^{c,t} \geq Q_i^L(c,t) = \frac{r_c(2P+t)}{4r_c^2} - \frac{(r_c^2(2P+t)^2 - 8r_c^2 L\mathcal{N}(t)P)^{1/2}}{4r_c^2}. \tag{12}$$

Now that we know if $Q_i^{c,t}$ is larger than $Q_i^L(c,t)$, the core with index no less than c would cause delay equal to the cache miss part. This is actually the key idea to solve the problem, we group the cores into the two sets depending on their flow value. To be more specific, the first $c-1$ throttled core contribute stall from the traffic flow part, while the remaining cores (starting from index c inclusive) contribute $\mathcal{N}(t)L$ each. Now we are ready to tackle Equation (11). The third item of the RHS with $R^{(k)}$ replaced by t is:

$$\sum_{1 \leq c \leq M} \min\{\mathcal{N}(t)L, \alpha_c(t)\} = \sum_{1 \leq j < c} \alpha_j(t) + (M - c + 1)\mathcal{N}(t)L$$

Therefore, Equation (11) can be rewritten as

$$\sum_{1 \leq j < c} \alpha_j(t) + (M - c + 1)\mathcal{N}(t)L \leq \mathcal{S}(t)$$

where $\mathcal{S}(t)$ is defined as in Equation (7)

Now the solution becomes similar to the single throttled core case: we expand the summation part α_j for each throttled core with r_j and Q_i and sum up. Let $\widehat{\mathcal{S}(t)} = \mathcal{S}(t) - (M-c+1) * \mathcal{N}(t)L$, $\widehat{r_c} = \sum_{1 \leq j < c} r_j$ and $\widehat{r_c^*} = \sum_{1 \leq j < c} r_j^2$. Then we are ready to solve the following equation to get Q:

$$t\frac{\widehat{r_c} Q_i}{P} + \frac{2\widehat{r_c} Q_i P - 2\widehat{r_c^*} Q_i^2}{P} \leq \widehat{\mathcal{S}(t)}.$$

Let $Q_i^H(c,t)$ denote the upper bound of Q value this equation is satisfied. We get:

$$Q_i^{c,t} \leq Q^H(c,t) = \frac{\widehat{r_c}(2P+t)}{4\widehat{r_c^*}} - \frac{(\widehat{r_c}^2(2P+t)^2 - 8\widehat{r_c^*}\mathcal{S}(t)P)^{1/2}}{4\widehat{r_c^*}} \tag{13}$$

Put Equation (12) and Equation (13) together and consider all indexes of throttled cores and all time points in testing set. Since task τ_i is schedulable as long as there is one throttled core c and one time instant $t \in \mathcal{TS}(\tau_i)$ satisfied, we have:

$$Q_i = \bigcup_{1 \leq c \leq M} \bigcup_{t \in \mathcal{TS}(\tau_i)} \{Q_i^{c,t} \geq Q_i^L(c,t) \bigcap Q_i^{c,t} \leq Q_i^H(c,t)\} \tag{14}$$

where $Q_i^L(c,t)$ and $Q_i^H(c,t)$ are solved in Equation (12) and Equation (13) respectively.

Finally, considering the multiple tasks on the critical core, the final result on Q is the intersection of all Q_i ranges.

$$Q = \bigcap_{1 \leq i \leq N} Q_i$$

where each Q_i is calculated in Equation 14.

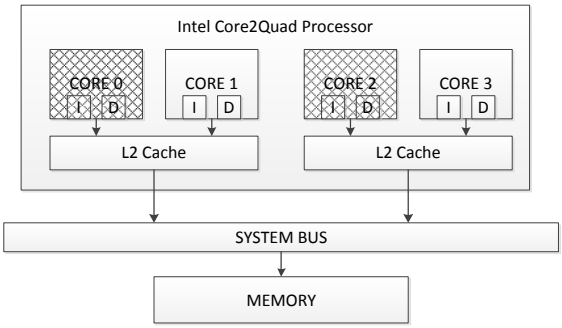


Figure 5: Architecture of our evaluation platform.

B. Dynamic budget distribution

In this scheme, all throttled cores share a single global budget and a period. When each core accesses memory, it consumes the global budget. When the global budget is exhausted, tasks on all throttled cores are suspended until the next period begins.

The dynamic budget distribution scheme reduces the possibility of throttling on the throttled cores because budget is consumed more efficiently on-demand basis. Therefore it improves responsiveness of applications on the throttled cores. It becomes more evident when tasks have high variance on memory access requests. On the other hand, however, it makes analysis more difficult, because budget distribution among cores keeps changing over time.

A safe, but pessimistic, upper bound of delay function is

$$\widehat{C^{(k+1)}} = C + \min\{M \cdot CM \cdot L, \alpha(\widehat{C^{(k)}})\} \qquad (15)$$

where M is the number of interfering cores. This is identical to Equation (2) except that M is multiplied in the first part of the min function. This is because the critical task on the critical core now can wait up to M arbitration.

Similarly, we can use Equation (3) for response time analysis just by multiplying M to the first part, $\mathcal{N}(R^k)$, of the min function.

Finally, computing the global budget Q can be obtained by using Equation (6), again multiplying M to all $\mathcal{N}(t)L$.

Note that this analysis is more pessimistic than the analysis in Section V-A, because we do not consider each individual flow, which could possibly produce a tighter bound.

VI. Evaluation

In this section, we experimentally evaluate our approach in terms of isolation guarantee of the critical core and performance impact of throttled cores (i.e., interfering cores).

The testbed contains Intel Core2Quad Q8400 processor running at 2.66GHz shown in Figure 5. Core0-1 share a 2MB L2 cache, and Core2-3 share another 2MB L2 cache. L2 caches are directly connected to the shared FSB running at 1333MHz. For this paper, we only use Core1 and Core3 for the experiment in order to eliminate the effect by sharing L2

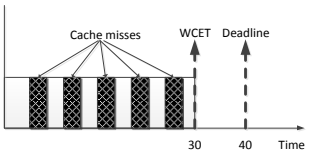


Figure 6: Task under analysis on the critical core.

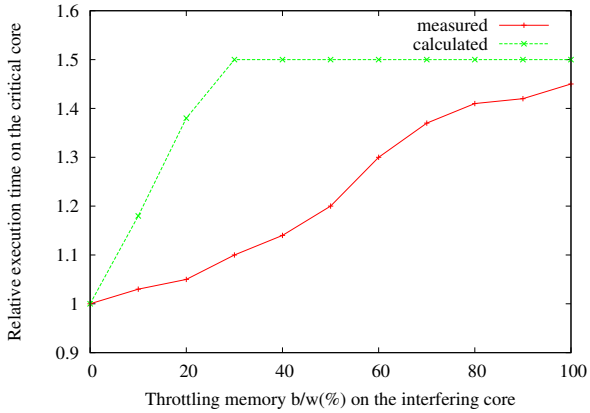


Figure 7: Impact of throttling bandwidth to the response time on the critical core.

cache [2]. We experimentally obtained the maximum memory bandwidth of 1.8GB/s and the average cost of 33ns for an single cache-miss delay (i.e., $L = 33ns$). In this section, we use the number of cache-misses to specify budget, instead of stall time. We convert the cache-miss budget into stall time value by multiplying 33ns. We use last level cache (LLC) miss count, reported from hardware performance counter of each core, as the cache-miss number. We use Linux 3.2.0 kernel and patched [3] the scheduler as described in Section II.

A. Response Time on the Critical Core

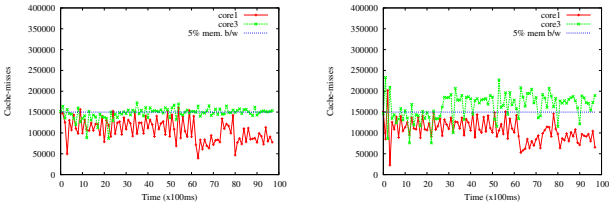
We present measured response time on the critical core while varying memory bandwidth on the interfering core. In this experiment, we use Core1 as a *critical core* and Core3 as an *interfering core*. The interfering core is regulated by the throttling mechanism with period P, and budget Q.

Fig. 6 shows the task under analysis,τ_{crit}, that runs on Core1. It is engineered to take 30ms to finish when run in isolation; it spends 50% of time stall on cache-misses. The cache-misses are placed in 10 equally spaced chunks; each chunk generates 1.5ms of continuous cache-misses; Note that we engineered each memory access to cause a cache-miss during the chunk. Then the task spends another 1.5ms for pure computation without any cache-miss.

On Core3 another engineered task, τ_{inter}, is running. It generates continuous cache-misses but throttled with P and Q values. We measured response time of τ_{crit} on the critical

[2]We disabled cpu cores because the processor we used does not support disabling L2 cache.

[3]The source code can be found in https://github.com/heechul/linux-sched-coreidle/tree/sched-3.2-throttle-v2



(a) Static budget distribution (b) Dynamic budget distribution

Figure 8: Cache-miss differences between static vs dynamic budget distribution scheme. Red line is Core1 and blue line is Core3

Scheme	Total throttled time
Static	9,689 ms
Dynamic	4,622 ms

Table I: Throttling time of interfering cores.

core, while varying the throttling memory bandwidth of the interfering core.

Figure 7 shows response time impact of throttling. X-axis shows the allocated memory bandwidth on the interfering core, Core3; The throttling period, P, is set to 10ms and budget, Q, is varied so that the bandwidth changes from 0 to 100%. Note that zero bandwidth is equivalent to the case when nothing is running on the interfering core. Y-axis shows the relative response time increase of T_{crit} on the critical core. The *calculated* curve shows the calculated response time of T_{crit} according to Eq 2. The *measured* curve shows the measured worst case response time among 1000 repeated invocations of T_{crit}; We randomly vary the interval time between two successive invocations from 0 to 30ms in order not to be affected by regularity of interfering flow on the interfering core.

In calculated curve, response time increases as the assigned bandwidth of the interfering core increases. When the assigned bandwidth increases above 30%, resulting response time saturates because from that point it is bounded by the number of cache-misses of the T_{crit} itself. In measured curve, response time also increases as bandwidth increases, but slower than the calculated curve. Notice that response time increase is sharper as assigned bandwidth is above 50% and approaches to the calculated bound. The difference between the calculated and measured curves show pessimism of our analysis. In the analysis, we considered the worst case scenario where every cache-miss from t_{crit} is delayed from cache-misses of the interfering core. However, the probability of such worst case scenario is low in a real situation as suggested by the measurement result.

B. Performance Impact on Throttled Cores

In this experiment, we compare performance impact of throttled cores under static and dynamic budget distribution schemes with realistic workload.

We use both Core1 and Core3 as interfering cores. In this experiment, we do not have a critical core, since we focus on the impact of budget distribution schemes on the throttled cores. We use two mpeg4 video streams, 720p and 1080p movie trailers, as workloads and played them on Core1 and Core3 respectively using *mplayer*.

For static scheme, the total budget is configured to be evenly divided between the two cores. We set the period equal to 10ms and the budget equal to approximately 5% of memory bandwidth (15000 cache-misses per 10ms per core). For dynamic configuration, we set the period equal to the same 10ms and the shared global budget equal to approximately 10% of memory bandwidth (30000 cache-misses per 10ms on both cores).

Figure 8 shows measured cache-misses behavior of static budget distribution scheme and dynamic budget distribution scheme. X-axis is time in 100ms unit and Y-axis is the number of measured cache-misses for each 100ms time interval. At any time instance, both schemes limit the total number of cache-misses less than the global budget (300,000 misses for every 100ms). In the case of static scheme, both Core1 and Core3 are limited to the specified 5% bandwidth (150,000 misses) all the time. Note that, ideally both Core1 and Core3 should not exceed 5% bandwidth line. However, due to the limitation of current implementation, detailed in Section VII, we are observing small fluctuation. In case of dynamic scheme, each core can generate much more cache-misses than 5% bandwidth line, but when combines the cache-misses for both cores, it does not exceed 10% bandwidth limit. This allows more efficient bandwidth utilization.

Table I compares total throttled time, the amount of suspended time by our kernel throttling controller, measured over ten seconds of the video playback experiment. In the static scheme, Core1 and Core3 are throttled total 9,689ms (Core1: 3019ms, Core3: 6670ms), while they are throttled only 4,622ms in the dynamic scheme. This is because of more efficient global budget distribution of the dynamic scheme.

C. Evaluation Summary

Evaluation result first demonstrate response time impact of memory throttling both analytically and experimentally on a real hardware platform. The results suggests that our technique can provide isolation guarantee with analytic support. We also investigate performance impact of tasks on the throttled cores under static and dynamic budget distribution schemes. The result shows that dynamic distribution scheme can provide better performance to the throttled cores for the same aggregated memory bandwidth budget. However, this can adversely affect to the critical core due to increased contention and poor analytic bound.

VII. Discussion

There are several assumptions we made on hardware that may affect the validity of our analysis results. We assume CPU synchronously waits fetch for cache-miss while it may concurrently execute out-of-order instructions in reality. Also, we assume that each single cache-miss take a constant time, $33ns$, but it can vary in reality (e.g., DRAM access cost significantly differ whether data is located in an opened row or a closed row). Taking into account of memory

access cost could be an interesting research topic. Finally, we assumed bus arbitration schemes follows round-robin. However, actual arbitration is not well known and may differ from vendors. From our observation on the tested platform, we believe our model reasonably follows the actual hardware.

Our current throttling implementation has a couple of limitations in providing guaranteed cache-miss bounds. Ideally, the scheduler should stop immediately after the budget is exhausted. However, the current implementation polls the hardware performance counters to account the budget consumption. The polling occurs at every 1ms or context switching time, whichever comes first. Therefore, there can be some amount of jitter from the time when the budget is expired to the time when scheduler actually performs the throttling operations. We are working on improving the implementation to use counter overflow interrupt. This will allow us to throttle at an exact point.

Another concern is scalability of our approach. Putting more interfering cores may result in more pessimistic throttling configuration for each of them. As future work, we plan to investigate adaptive schemes to mitigate pessimism in the analysis.

VIII. Related Works

Shared resource contention in modern multi-core/multiprocessor systems is a big challenge in real-time system design. Much effort has been spent to develop analysis frameworks for shared resource arbitration, in particular bus and memory, to compute worst case timing bounds. Thiele et al. presented Real-time calculus [11], [12] to model real-time tasks with request and service curves. Real-time calculus is extended to support multiprocessor systems by Leontyev et al [13]. Pellizzoni et al, also used real-time calculus to model CPU memory traffic and PCI IO traffic [7]. Schranzhofer el al. developed an analysis framework to compute the worst-case response time of real-time tasks under TDMA based bus arbitration and adaptive arbitration [14], [4]. Pellizzoni el al. also developed a delay analysis framework for round robin and fifo arbitration based multiprocessor systems [2]. They assumed a task consists of a set of superblocks and described the method to compute tight WCET bound. Our delay model use the main results from [2], [7] to compute maximum throttling parameters that still guarantees the schedulability of a critical core.

A line of research focuses on the scheduling technique that is aware of resource contention, shared memory and cache. Calandrino et al. [15], [16] proposed a shared cache aware scheduling method for improving performance by co-scheduling tasks preventing cache trashing. Their work focuses on global multi-core scheduling in the context of soft real-time applications. Our work considers partitioned scheduling and focuses on providing hard real-time guarantees for a chosen critical partition. Recently Dasari et al. [17] developed a response time analysis for COTS based multi-core systems. However, they do not consider specific bus arbitration scheme and task cache-miss behavior, hence more pessimistic than our analysis.

OS level memory throttling is first discussed in literature by Bellosa [18]. He investigated several implementation methods and demonstrated the possibility of using throttling mechanism to reserve memory bandwidth for soft real-time applications. More recently, hardware clock modulation based throttling methods have been proposed [19] to provide QoS guarantees for high-priority applications that co-exist with memory intensive low priority applications. Existing studies using throttling were not designed to provide core isolation and lack analysis needed to support hard real-time guarantee. In contrast, we focus on analytic method to provide strong isolation for safety-critical real-time applications.

Better timing guarantees on accessing shared memory and bus can be achieved by adopting specially designed hardware architectures. For example, a priority based bus arbiter could easily solve the problem of protecting critical core presented in this paper. Predator [20] adapted priority based arbitration in the context of predictable DRAM controllers to provide bandwidth and latency guarantees for each requestor. Such arbitration scheme can be applied to a system bus as well. Similarily, TDMA based bus arbitration is a very attractive approach since it allows predictable performance analysis independent from other cores. Indeed, Rosen et al, designed a TDMA based system bus arbiter with algorithms that produce efficient TDMA bus schedules [21]. However, these schemes require extensive hardware modifications which are not readily available in modern COTS based systems. Our approach in contrast is entirely based on software with the help of hardware performance counters readily available in most COTS processors.

In real-time systems, various aperiodic real-time servers, such as deferrable server, polling server, and sporadic servers, are used execute aperiodic tasks together with other real-time tasks without messing schedulability analysis [5]. Deferrable server maintains the budget for the duration of each period. While it maximally utilizes the given budget, it lowers schedulability because the budget can be executed in sequence spanning two consecutive periods. Due to its simplicity in implementation, however, it is commonly used in practice as shown in CPU bandwidth controller in recent Linux 3.2 kernel [22]. Sporadic server is theoretically best but complex to implement and suffer significant overhead in high load [23]. Our memory throttling controller implementation follows the semantic of deferrable server.

IX. Conclusion

In this study, we presented a method to control shared resource contention on COTS based multiprocessor for use in critical real-time systems. As a first step, we considered a scenario where a dedicated core executes critical tasks while other cores execute tasks which cause significant accesses on the shared bus and memory. We presented a software based memory throttling mechanism and analytic solutions to compute throttling parameters that guarantee schedulability of critical tasks while minimizing performance impact

of tasks on the throttled cores. We implemented the mechanism in Linux kernel and experimentally demonstrated the viability of our approach. Future works include extending this approach for more general scenarios and improving the throttling implementation to support more fine-grained control.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for useful feedback that improved the quality of this paper. The material presented in this paper is based upon work supported by Lockheed Martin and This effort is funded in part by Lockheed Martin, NSERC, and NSF under Award No. CNS-1035736. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF or Lockheed Martin.

REFERENCES

[1] A. Kurdila, M. Nechyba, R. Prazenica, W. Dahmen, P. Binev, R. DeVore, and R. Sharpley, "Vision-based control of micro-air-vehicles: Progress and problems in estimation," in *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, vol. 2. IEEE, 2004, pp. 1635–1642.

[2] R. Pellizzoni, A. Schranzhofery, J. Cheny, M. Caccamo, and L. Thiele, "Worst case delay analysis for memory interference in multicore systems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*. IEEE, 2010, pp. 741–746.

[3] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: a predictable sdram memory controller," in *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. ACM, 2007, pp. 251–256.

[4] A. Schranzhofer, R. Pellizzoni, J. Chen, L. Thiele, and M. Caccamo, "Timing analysis for resource access interference on adaptive resource arbiters," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*. IEEE, 2011, pp. 213–222.

[5] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard-real-time systems," *Real-Time Systems*, vol. 1, no. 1, pp. 27–60, 1989.

[6] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: a predictable sdram memory controller," in *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. ACM, 2007, pp. 251–256.

[7] R. Pellizzoni and M. Caccamo, "Toward the predictable integration of real-time cots based systems," in *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*. IEEE, 2007, pp. 73–82.

[8] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*. IEEE, 2011, pp. 269–279.

[9] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*. IEEE, 2003, pp. 2–13.

[10] N.C.Audsley, A. Burns, M.Richardson, K.Tindell, and A.Wellings, "Applying new scheduling theory to static priority preemptive scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, 1993.

[11] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, vol. 4. IEEE, 2000, pp. 101–104.

[12] S. Chakraborty, S. Künzli, and L. Thiele, "A general framework for analysing system properties in platform-based embedded system designs," in *Proc. 6th Design, Automation and Test in Europe (DATE)*, 2003, pp. 190–195.

[13] H. Leontyev, S. Chakraborty, and J. Anderson, "Multiprocessor extensions to real-time calculus," in *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*. IEEE, 2009, pp. 410–421.

[14] A. Schranzhofer, J. Chen, and L. Thiele, "Timing analysis for tdma arbitration in resource sharing systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*. IEEE, 2010, pp. 215–224.

[15] J. Calandrino and J. Anderson, "Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study," in *Real-Time Systems, 2008. ECRTS'08. Euromicro Conference on*. Ieee, 2008, pp. 299–308.

[16] ——, "On the design and implementation of a cache-aware multicore real-time scheduler," in *Real-Time Systems, 2009. ECRTS'09. 21st Euromicro Conference on*. IEEE, 2009, pp. 194–204.

[17] D. Dasari, B. Andersson, V. Nelis, S. M. Petters, A. Easwaran, and J. Lee, "Response time analysis of cots-based multicores considering the contention on the shared memory bus," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, nov. 2011, pp. 1068 –1075.

[18] F. Bellosa, "Process cruise control: Throttling memory access in a soft real-time environment," in *Sixteenth Symposium on Operating Systems Principles (Poster Session)*, 1997.

[19] A. Herdrich, R. Illikkal, R. Iyer, D. Newell, V. Chadha, and J. Moses, "Rate-based qos techniques for cache/memory in cmp platforms," in *Proceedings of the 23rd international conference on Supercomputing*, ser. ICS '09. ACM, 2009.

[20] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens, "Real-time scheduling using credit-controlled static-priority arbitration," in *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA'08. 14th IEEE International Conference on*. IEEE, 2008, pp. 3–14.

[21] J. Rosen, A. Andrei, P. Eles, and Z. Peng, "Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip," in *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*. IEEE, 2007, pp. 49–60.

[22] P. Turner, B. Rao, and N. Rao, "Cpu bandwidth control for cfs," in *Ottawa Linux Symposium*, 2010.

[23] M. Stanovich, T. Baker, and A. Wang, "Experience with sporadic server scheduling in linux: Theory vs. practice," in *Real-Time Linux Workshop, 2011*, 2011.