Dynamic Budgeting for Settling DRAM Contention of Co-running Hard and Soft Real-time Tasks

Jonas Flodin, Kai Lampka, Wang Yi
Department of Information Technology, Uppsala University
Email: {jonas.flodin, kai.lampka, yi}@it.uu.se

***Abstract*—In modern non-customized multicore architectures, computing cores commonly share large parts of the memory hierarchy. This paper presents a scheme for controlling the sharing of main memory among cores, respectively the concurrently executing real-time tasks. This is important for the following: concurrent memory accesses are served sequentially by the memory controller. As task execution stalls until memory fetches are served, the latter significantly contributes to the execution time of the tasks. With multiple real-time tasks concurrently competing for the access to the memory, the main memory can easily become the Achilles heel for the timing correctness of the tasks. To provide hard timing guarantees, release of access requests issued to the main memory has therefore to be controlled. Run-time budgeting is a well accepted technique for controlling and coordinating the use of a shared resource, particularly when the underlying hardware cannot be altered. Whilst guaranteeing timing correctness of the hard real-time applications, worst-case based resource budgeting commonly leads to performance degradations of the co-running (so called soft real-time) applications. In this paper we propose to combine worst-case based resource budgeting with run-time monitoring for dynamically reconfiguring the budget schemes. Thereby we aim at increasing the responsiveness of the soft real-time applications, while satisfying the strict timing constraints of the co-running hard real-time tasks. We have implemented the proposed scheme in a microkernel and present its empirical evaluation for which an industrial benchmark suite has been employed.**

I. Introduction

A. Motivation

Advances in chip- and networking technology driven by the high volume and high performance consumer electronics industry, opens up the road for building cost-effective embedded control systems. However, the use of "Commercial Off-The-Shelf" (COTS) components, originally designed for the consumer-electronic market, is still beyond today's engineering practice, at least when it comes to the design and implementation of integrated embedded systems which have to meet real-time requirements. The reason for this is as follows: integration of multiple (hard and soft time) applications into a single multicore device brings the sharing of hardware infrastructure among the logically independent applications. The sharing might yield hidden dependencies, respectively interferences between the applications, even with applications executing on different cores. As a result, one may experience unwanted timing effects, which are extremely difficult to detect, and have the potential to corrupt the timing correctness of the system.

As a concrete example, one may think of a dual core system with a shared L2 data-cache. The concurrently executing software will mutually evict each other's cache entries. This in turn will significantly add to their execution times as data items must be re-fetched from the main memory. However, this interference, which can only be pessimistically bounded by today's analysis techniques [16], is not the only source of trouble. In a synchronous setting, each time when fetching an item from the main memory, task execution is suspended until the actual request is served. The resulting waiting time depends on the number of pending memory access requests from the tasks executing on the other cores and the complex memory access arbitration scheme implemented by the DRAM controller. The challenge inherent to such a setting is to find a strategy which on one hand guarantees that real-time tasks do not miss their deadlines due to excessive waiting for the main memory. On the other hand, such a strategy must not drastically block non-real-time tasks from accessing the memory and thereby eliminate their responsiveness. This paper introduces a scheme which aims to achieve both timing correctness and good responsiveness of soft real-time tasks. The approach is based on budgeting and slack reclamation for coordinating the access to the main memory of concurrently executing hard and soft real-time tasks.

B. Contribution: memory bandwidth control with slack shifting from hard to soft-real-time tasks

The cores in common multicore architectures typically share parts of the memory hierarchy. This include caches, memory controller and DRAM modules. In this paper we consider sets of hard and soft real-time tasks executing concurrently on different cores and sharing parts of the memory hierarchy. When a core access memory, respectively fetches data from the main memory, the bounding of the completion time of the access requires the following questions to be answered:

1) Level-n Caching: does the requested data reside in one of the caches or not?
2) Main memory utilization: how many requests are currently buffered in the memory controller or arrive before the request under consideration is served?
3) Caching in the main memory: does the fetch operation address a memory location which resides in an open row in the DRAM?

This paper focuses on item 2 and presents a technique which controls the number of main memory access requests emitted by the soft real-time tasks. This is needed as the memory controller may reorder accesses for higher throughput of the memory system and the re-ordering can lead to unexpected waiting times experienced by access requests sent from a co-running hard real-time task [13]. To bound the interference

of co-running soft real-time tasks, past works have proposed server-based resource reservation mechanisms, not only for organizing the sharing of computing resources, but also for addressing the shared use of DRAM [2], [17], [18]. However, the computed budgets are extremely pessimistic, and reflect the worst-case rather than the normal resource use. Hence, tasks under memory access budgeting experience a severe degradation of their average response time. For the hard real-time tasks, commonly implementing system control functions, this degradation is irrelevant, what matters is the guarantee that all deadlines are met. However, for user-centric soft-real time applications performance degradation should be reduced which is the aim of the proposed approach.

We assume that the tasks with hard real-time constraints are mapped to a specific core(s) and that their scheduling ensures timing correctness under a fixed number of memory accesses to the memory performed by the remaining cores. Access to the main memory of the soft real-time tasks is dynamically throttled by a budgeting mechanism, put together with a memory access reclamation scheme to utilize unused memory access bandwidth. Our approach shifts unused memory access bandwidth from hard real-time tasks to the co-running soft real-time tasks in the form of lifting memory bandwidth restrictions. Whilst the budgeting ensures timing correctness of the co-running hard real-time tasks, slack reclamation of unused memory access bandwidth intends to increase the throughput of the co-running soft real-time tasks.

Finding optimal mappings of hard real-time tasks to cores together with the computation of budgets such that some objective is maximized, e. g., memory utilization, is extremely challenging and an open research field. But, finding answers to this problem, is not the target in this paper. Here, we focus on the reclamation of unused memory access bandwidth. Shifting access capacity from hard-real time tasks to off-core co-running soft real-time applications will increase reactivity of the latter. However, reclamation must be done in such a way, that the timing correctness of the hard real-time tasks executing on the bandwidth donating core are no corrupted. To the best of our knowledge, such a scheme has not been proposed yet; see the next section for the body of work in this direction.

C. Organization

The remainder of the paper is organized as follows: In the next section we review related work. Sec. III presents the system model. Sec. IV describes our proposed budgeting mechanism. In Sec. V we detail our implementation and our experimental setup and provide some data on how well our method works. Sec. VI concludes the paper.

II. Related Work

The proposed work is based on the concept of resource servers. Resource servers are a standard mechanism to coordinate the access to a shared resource. The basic mechanism works as follows: the budget of the server is a function over the time-line, where a resource access decreases the budget of the server accordingly and the budget is replenished at fixed points in time. Whenever the budget is used up, the server is not eligible to access the resource. Typically each resource server is equipped with some budget, where the dynamic assignments of priorities determine which server is given the priority to access the resource.
Example of resource servers are the Constant Bandwidth Server [1], Deferrable Server (DS) [15], Constant Utilization Server (CUS) [3], Hierarchical Resource Reservation, e. g., for bounded delay resource partitioning [5] and Online Reconfigurable CBS [12], to name only few of them.

In case of hard real-time tasks, it is assumed that its WCRT assumed in the scheduling analysis reflects the worst-case interference of DRAM accesses of co-running hard real-time tasks. The interference resulting from the co-running soft real-time tasks is bounded by the currently active budget of their resource server. We assume that the budgets for the soft real-time tasks are set in such a way that the resulting WCRT of the co-running hard real-time tasks meets the bounds employed in the scheduling analysis.

In such a setting, it appears now, that slack reclamation in case of DRAM access is, contrary to the standard setting of resource servers, not side effect free. Shifting unused bandwidth to co-runners located on other cores may inject additional delays into the execution time of tasks of the core, the additional bandwidth was donated by.
In case of lower priority tasks of the donating core, injection of unaccounted delays is safe if the availability of the donated bandwidth is limited to the worst-case execution time window of the donating task. With higher priority tasks this is different. In case execution of a task with a higher priority than the donor task is scheduled into the worst-case execution time window of the already terminated task, donation of bandwidth has to be cancelled. Otherwise, the unaccounted memory accesses may inject additional delays which have not been considered upon scheduling analysis time (i.e., a high priority task is delayed by the reclaimed slack from low priority task) and can therefore corrupt the timing correctness of the system. It is this observation which distinguishes this work from the existing body on resource servers and the works on resource servers for controlling memory access bandwidths as reported next.

The authors of [2] propose a 2-dimensional budgeting scheme, were budgeting of CPU time and access numbers to the main memory is controlled. The paper does, however, not consider reclamation, respectively donation of bandwidth.

In [17], Yun et al. present a way of guaranteeing memory bus bandwidth to one hard real-time core through memory access throttling on soft real-time interfering cores while minimizing the performance impact of throttling on the soft real-time cores. Periodically replenished memory bus budgets are given to interfering soft real-time cores. The bus usage, in the form of last level cache (LLC) misses, is measured every 1ms or every task switch, whichever comes first. If the budget is depleted, all ready soft real-time tasks on that core are moved away from the ready-queue, until the next replenishment point. The authors do also not consider slack reclamation of unused memory bandwidth. Implementation of a similar technique was presented in [9], the proposed slack reclamation was, however, not safe w. r. t. timing correctness.
In [18] Yun et al. present a bandwidth reservation and reclaiming scheme denoted as MemGuard. MemGuard utilizes a predicted bandwidth usage to assign budgets each period. The difference between a statically assigned budget and prediction

is added to a global budget, which tasks may then reclaim from if they have depleted their own budget. This makes it possible to distribute bandwidth to tasks which currently need more bandwidth than they are assigned. One drawback of MemGuard is that budget reclaiming only works for soft real-time task sets, since it gives no guaranteed bandwidth each period.

The above works are based on static analysis for determining the budgets of tasks. The worst case path w. r. t. memory accesses of tasks may be traversed rarely in practice. As a consequence, the memory access bandwidths assigned to the soft real-time tasks are unnecessarily low, yielding low response times of the soft real-time tasks and severely under utilized cores. Particularly as none of the aforementioned works considered re-distribution of un-needed bandwidth from hard to soft real-time task. Bandwidth shifting among soft real-time tasks as proposed in [18], will help to increase the average response time of a specific task. However, it does not solve performance degradation of all soft real-time tasks, as [18] kept the overall budget distributed among the soft real-time tasks constant with a periodic replenishment. Consequently, the average case response of all soft real-time tasks does not change.

In our work we specifically address this problem by dynamically changing sizes of budgets or simply ignoring them once a hard real-time task has terminated before its set WCRT and there is no job release of some other hard real-time task.

With respect to the above works we make the following contribution:

1) In contrast to [17] we allow more than one core with hard real-time tasks to execute.
2) We shift slack from hard real-time tasks to soft real-time tasks. Slack reclamation is absent in the work [17], [2]. Yun et al. [18] propose slack shifting among soft real-time tasks only.
3) Budget replenishment in the above works takes place periodically, yielding presumably low access rates to the main memory by soft real-time tasks. In our approach, we have hard real-time co-runner specific budgets. This should already yield better response times of co-running soft real-time tasks, not to mention the slack reclamation which is implemented in addition to that.

Adding structure to the run-time system for guaranteeing deadlines under main memory contention is one way of coping with the unpredictability inherent to the behaviour of modern multicore architectures. Another way for achieving this, is enforcement of TDMA (time division multiple access) based resource arbitration, where such schemes are often already implemented in hardware.

TDMA provides more predictable access times for memory requests, but is by its design inflexible and cannot adapt to information available at runtime, which leads to underutilization of bandwidth. Another downside with TDMA is that it nullifies an *open-page policy* if time slots are small while worst case latencies increase if time slots are large.

Goosens et al. [8] present a *conservative open-page policy* that can exploit some locality among memory requests without sacrificing predictability of worst case response times. It is shown that for parallel applications some speedup can be achieved, if for each application two consecutive time slots in the memory access are schedule.

Rosén et al. present a way of combining WCET-analysis and system scheduling for bus scheduling optimization [14]. Four different bus access policies are evaluated and it is shown how the length of a task schedule can be reduced as a trade-off with how much memory is needed to store the bus schedule.

An alternative approach to analysing the duration of accesses to a TDMA-arbitrated resource is presented by Kelter et al. in [10]. Based on abstract interpretation and integer linear programming (ILP) to analyse statically all possible offsets of a new access request within the TDMA arbitration cycle, the proposed method is claimed to achieve high precision in reasonable analysis time. However, the analyzed scenarios refer to dual-core systems and the proposed TDMA scheme does not reflect the capabilities of contemporary COTS memory components.

Typically, strict ordering (e. g., TDMA) versus non-strict ordering (e. g., prioritizing accesses that hit in an open row) of memory accesses can be seen as a trade-off between maximizing worst case performance and average case performance. Our method is a compromise between these by making use of the average case improvements by non-strict memory access scheduling as long as safe execution of hard real-time tasks can be guaranteed. Additionally, our method can be implemented on top of COTS components where TDMA memory arbitration might not be available and synchronizing memory access phases between cores requires considerable overhead.

The authors of [11] present techniques to bound delay caused by interference on the DRAM level for COTS-based multicore architectures with First-Ready First-Come-First-Served (FR-FCFS) memory access arbitration. Considering the different instructions executed by the memory controller of a typical COTS platform when fetching a data item from the memory, inter- and intra-bank accesses times are bounded. With this precise information, it is possible to get tighter bounds on interference through the use of bank partitioning, where cores gets exclusive access to a memory bank or share it with a subset of the other cores. Memory partitioning is orthogonal to the proposed budgeting scheme. As a result it could be integrated into our scheme. The upper bounding of memory access is of great value for us, as it allows us to bound memory access times and safely over-approximate the budgets of the soft real-time tasks such that hard real-time tasks meet their deadlines. Analysing or finding suitable budgets is sensed by us as highly relevant, however, for now it is out of the scope of the research presented in this paper.

III. System Model

We consider a system deployed on a typical COTS multicore architecture.

- There are M CPU-cores, K of which are executing hard real-time software and $M - K$ are executing set of soft real-time tasks.
- There are N sporadic hard real-time tasks $T = \{\tau_1, \tau_2, ..., \tau_N\}$, each defined by the quadruple $\tau_i =$

(C_i, P_i, D_i, H_i), with

- C_i as the WCET for the task when running alone on one hard real-time core,
- P_i as the minimum inter arrival time of the task,
- $D_i \leq P_i$ as the task's relative deadline and with
- H_i as the largest number of memory access requests produced by τ_i during one task instance.

- Each core has its own fixed priority scheduler and each task τ_i is mapped to one specific core out of the K hard real-time cores.
- Hard real-time tasks are ordered by their priority such that τ_j has higher priority than τ_i if $j < i$.
- $hi(\tau_i)$ denotes the set of tasks mapped to the same core and having a higher priority than τ_i.
- $cr(\tau_i)$ is the set of hard real-time tasks which are assigned to other hard real-time cores and potentially co-run, i. e., execute in parallel, with task τ_i.
- The other cores we collectively call soft real-time cores and they execute soft real-time or best-effort tasks, we do not make any assumptions about the soft real-time tasks.

All cores share a single memory controller which acts as an arbiter for serving requests to DRAM. Since memory controllers and DRAM are complex and have a hard-to-analyze temporal behaviour, we over-approximate the time it takes to serve a single request in the worst case as a constant[1] L, which can be computed with methods described in [11]. This constant also bounds the worst case delay a memory access request might suffer from a single interfering request by another core.

The maximum increased delay for hard real-time tasks as a consequence of interfering requests by a set G of hard real-time tasks mapped to other cores during a time interval t is upper bounded by the function

$$F(G,t) = \sum_{\tau_i \in G} \left\lceil \frac{t}{P_i} + 1 \right\rceil \cdot H_i \cdot L$$

Let B_i be the worst case number of requests sent by all soft real-time cores to the memory controller upon the execution of τ_i. It is important to note that in contrast to other approaches, we map B_i to the execution of τ_i and not to a time interval. This allows us to adjust the WCET of task τ_i by the delay caused by interfering requests from the soft real-time cores as follows:

$$A_i = C_i + B_i \cdot L$$

Based on this, the worst case response time (WCRT) of a hard real-time task τ_i is a solution to the recurrence relation

$$R_i = A_i + \sum_{\tau_j \in hi(\tau_i)} \left\lceil \frac{R_i}{P_j} \right\rceil \cdot A_j + F(cr(\tau_i), R_i), \qquad (1)$$

and a system is timing correct, respectively feasible if

$$\forall \tau_i \in T : \; R_i \leq D_i \qquad (2)$$

holds.

The value selected for B_i is the budget that is assigned to the soft real-time cores during the execution of τ_i. Selecting values for B_i can be seen as tuning the WCET of individual tasks by sacrificing performance on soft real-time cores. Computation of optimal values and their distribution among soft real-time cores is out of this scope for this paper, so we assume safe values for all budgets B_i are known. An adaptation of the framework presented in [7] could be exploited to assure timing correctness of hard real-time tasks for given budgets. We enforce that the soft real-time cores do not exceed the budgets through the use of a budgeting mechanism introduced next.

IV. Resource Budgeting Mechanism

For simplicity we begin with the case when all hard real-time tasks are mapped to a single core which is the setting presented in [17]. The extension of our basic scheme to the case of multiple cores with hard real-time tasks assigned to them, denoted as hard real-time cores, follows thereafter.

A. Hard real-time tasks mapped to a single core

The proposed budgeting mechanism and reclamation scheme works as described below.

- When a task τ_i starts executing at time s_i, a message is sent to the soft real-time cores that they must activate memory access budgeting using B_i as a budget. The budget expires at time $e_i = s_i + A_i$. Each core receives a precomputed fraction of the budget such that the total sum is no more than B_i. **(lines 3, 10-14 in Algo. 1)**[2]
- The number of memory accesses on the soft real-time cores are monitored using appropriate performance monitor counter (PMC) events (e.g. cache misses). **(line 37 in Algo. 1)**
- If the budget on a soft real-time core is depleted, it may no longer continue executing until the expiration time. **(lines 19, 35 in Algo. 1)**
- If τ_i finishes early at time f_i, another message is sent to the soft real-time cores that the budget B_i may be exchanged for a budget U_i, which has the same priority and expiration time as B_i, but has unlimited accesses to main memory. **(lines 5, 16, 32 Algo. 1)**
- In the case of a higher priority task τ_j starting to execute at time $s_j > s_i$, the soft real-time cores switch budgets such that they use the budget of the highest priority task. During the time when the higher priority budget is in use, we do not count the time towards expiration for lower priority budgets. **(lines 11, 24 in Algo. 1)**
- When a higher priority budget expires, the soft real-time cores fall back to using the budget of the second highest priority budget. If there are no more budgets, the soft real-time cores continue executing with unrestricted access to memory. **(lines 22, 24 in Algo. 1)**

[1] Since we consider COTS hardware without modifications, i.e., FR-FCFS scheduling of memory requests (see Sec. II), we cannot accurately predict memory access times and must use a worst case constant.

[2] The MemGuard approach of [18] can be incorporated, a long as each B_i is an upper bound on all memory access requests of the soft real-time cores, timing correctness is preserved regardless of budget distribution among the soft real-time cores.

Algorithm 1 Budgeting and reclamation scheme

```
1: activeBudgets := ∅ // Priority sorted list of budgets.
2: procedure StartTask(task τ_i) // Hard RT task is started.
3:     SignalTaskStart(τ_i)
4:     DoWork(τ_i)
5:     SignalTaskFinished(τ_i)
6: end procedure
7: procedure Budgeting[3](event e)
8: // Event is received on one of the soft real-time cores.
9:     B := LookupBudget(e)
10:    if e = task_started then
11:        Insert(activeBudgets, B)
12:        MarkFinished(B, FALSE)
13:        Replenish(B)
14:    end if
15:    if e = task_finished then
16:        MarkFinished(B, TRUE)
17:    end if
18:    if e = budget_depleted then
19:        SleepUntilNextEvent()
20:    end if
21:    if e = budget_expired then
22:        Remove(activeBudgets, B)
23:    end if
24:    ActivateBudget(Head(activeBudgets))
25: end procedure
26: procedure ActivateBudget(budget B)
27:    if B = NULL then
28:        DisablePMC()
29:    else
30:        SetExpirationTimeout(B)
31:        if HasFinished(B) then
32:            DisablePMC()
33:        else
34:            if IsDepleted(B) then
35:                SleepUntilNextEvent()
36:            else
37:                EnablePMC(B)
38:            end if
39:        end if
40:    end if
41: end procedure
```

Consider an example execution as shown in Fig. 1. The upper part depicts the interleavings of two tasks on the hard real-time core and the lower part shows which budget is in effect on the soft real-time cores. The hard real-time core starts executing τ_2 and signals the soft-real time cores to use budget B_2 at time s_2. The hard real-time core continues executing τ_2 until time s_1, when it is preempted by the arrival of τ_1, which also triggers the soft real-time cores to switch budget to B_1. When τ_1 finishes early at f_1, the soft real-time cores are signaled to exchange the budget B_1 for U_1, which means that they have unlimited access to main memory until e_1. At the same time, the hard real-time core switches to executing τ_2. When U_1 expires at time e_1 the soft real-time cores fall back to use budget B_2 until τ_2 finishes at f_2. The budget B_2 is then switched for U_2 until it expires at e_2.

[3]The procedure *Budgeting* is called on soft real-time cores whenever one of the following events occur: *SignalTaskStart* or *SignalTaskFinish* is called on a hard real-time core, a budget times out or a budget is depleted.

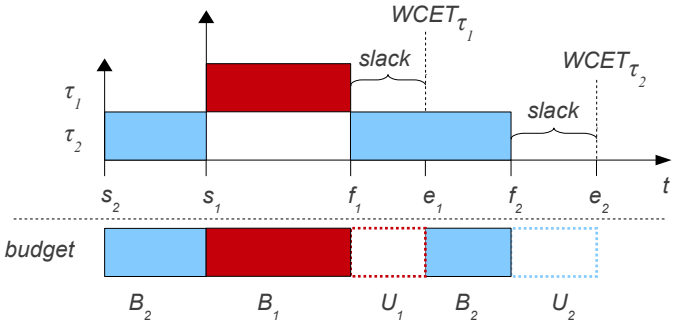

Fig. 1. Budgeting example with two tasks. Arrows pointing up denote job releases and dashed vertical lines denote the point in time when a job would have finished if it needed the entirety of its WCET.

Correctness of scheme: Slack reclaiming using U_i as budget instead of B_i once task τ_i finishes early does not increase the worst-case response times and therefore (2) still holds.

To justify our claim, we offer the following argumentation. One instance of a task τ_i may cause a delay of at most the WCET A_i for lower priority tasks when we do not employ slack reclaiming techniques. We show that this amount of delay, and therefore response time, is not increased by introducing our slack reclaiming scheme.

Suppose a task starts executing at time s_i, finishes at f_i and could have ended at latest possible time $e_i = s_i + A_i$, not counting the time it is delayed by preemption of higher priority tasks. This results in the use of unlimited budget U_i during the time interval $[f_i, e_i]$. A safe upper bound of the delay introduced by all the main memory contention during this interval in time is $d = e_i - f_i$, which would be the case when all lower priority hard real-time tasks simply stall due to memory contention. The total delay would then be the sum of the actual execution time of τ_i and d,

$$exec\_time + d = f_i - s_i + e_i - f_i = -s_i + s_i + A_i = A_i,$$

which is what we want to show.

In case the time window associated with the WCRT of task τ_i is filled with the execution of a higher priority task $\tau_j \in hi(\tau_i)$, budget U_i is replaced with budget B_j. This eliminates all side effects of budget lifting w. r. t. preemptive higher priority tasks. Moreover, as task preemption has already been considered upon scheduling analysis time, the system remains timing correct.

Lastly, switching from B_i to U_i does not increase response time of τ_i as it has already finished.

B. Hard real-time tasks mapped to multiple cores

With multiple cores and their hard real-time tasks being executed at the same time, the base scheme is extended with the following:

- The scheme is extended by letting the soft real-time cores have several lists of active budgets as the one

used in Algo. 1, where each of these lists are mapped to a single hard real-time core.

- To account for additional delay caused by interference from hard real-time tasks running on the other cores, calculation of expiration time of a budget is updated to $e_i = s_i + A_i + F(cr(\tau_i), R_i)$.
- When a soft real-time core is to pick a budget to program the performance monitor counter (PMC), it always picks smallest remaining budget among the heads. Similarly, when the timer is to be programmed, the smallest remaining time until timeout among the heads of the budget lists is used.
- Whenever budget is consumed, all heads decreases their remaining budget value.
- Time progression towards budget timeout is counted for all budget list heads.
- If one or more head budgets are depleted, the core may no longer execute.
- For soft real-time cores to set an unlimited budget, all budget lists must either have no active budgets or have a head set to unlimited by a finish signal.

As an example where hard real-time tasks are mapped to multiple cores, consider a system where we have two tasks $\tau_1 = (1, 2, 2, H_1), \tau_2 = (2, 3, 3, H_2)$. These tasks cannot execute on the same core with our method, as they require too much computation time. We therefore allocate one core to each of the tasks. Suppose that H_1 and H_2 are small enough for the system to be feasible when executing the tasks concurrently. B_1 and B_2 are chosen such that $R_1 = D_1$ and $R_2 = D_2$ respectively.

An example execution of these tasks and the corresponding budgeting on a soft real-time core are shown in Fig. 2. The uppermost part in the figure shows task releases and execution times. In the middle part of the figure, we can see the remaining part of the budget that the tasks enforce on one of the soft real-time cores. These are set to infinity after the task finishes early. In the bottommost part of the figure we see which budget is in use on the same soft real-time core. The duration of the core stall due to budget depletion is shown with a box of diagonal lines.

Correctness of scheme: The behaviour regarding memory accesses of hard real-time cores does not change by using our reclamation scheme as budgets only apply to soft real-time cores. The soft real-time cores always pick the smallest budget allowed by any hard real-time task, which means that any budget set by a hard real-time core is a safe upper bound on the memory accesses by soft real-time cores. These two facts allow us to look at the interference from the perspective of a single hard real-time core in isolation. From there we follow the same argumentation as in the single hard real-time core case for each of the hard real-time cores to show the correctness of our scheme.

V. Implementation and Evaluation

We implemented the proposed budgeting mechanism in the *Fiasco.OC* microkernel, part of the L4 kernel family [6]. This

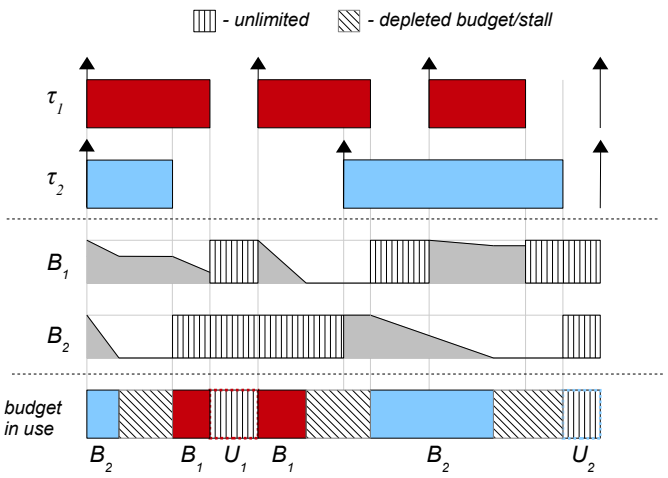

Fig. 2. Budgeting example with two task executing on separate cores. The upper part depicts the task execution, the middle part the budget level on one soft real-time core and the bottom part which budget is used to program the PMC on the same soft real-time core.

was motivated as this modern operating system has a small code-base, is open-source, runs on common architectures and due to its microkernel nature, already provides separation of tasks. Additionally, *Fiasco.OC* has support for virtualization, where a virtual machine executes as a special type of task. This means that the isolation properties of our budgeting mechanism naturally extend to virtual machines, which may run operating systems and applications the memory usage of which are impossible to analyze.

When a task starts executing on a hard real-time core, this is signaled by an inter-processor interrupt (IPI) broadcast to all soft real-time cores. The message associated with the IPI tells the soft real-time cores which task is starting execution. The soft real-time cores then do a look-up in a precomputed table of tasks to find out which budget they should activate, which priority it has and when it expires. The budget is added to the priority-ordered list of budgets associated with the core where the task is executing.

Whenever at least one list of budgets is non-empty, the highest priority budget with the least remaining budget is in use. When a budget is in use, a PMC is configured to fire an overflow interrupt when the budget is depleted and a timeout is set to fire when the one highest priority budget with least time remaining expires. If the PMC overflow interrupt is triggered, the core is halted, using a special instruction, until the expiration timeout fires. When the expiration timeout fires, the associated budget is removed from the list of active budgets and the next budget in the list is eligible to be picked as the budget in use.

After a task is done with its execution another IPI broadcast signals the soft real-time cores that the budget for this task instance is no longer needed. The soft real-time cores then, upon receiving the IPI, marks the budget as unlimited, without removing it from the list of active budgets. Then the next highest priority budget with least budget remaining is selected.

Integration of the proposed budgeting mechanism has only requested minor changes to the microkernel. We mainly had to enable communication of budget policies to the kernel and

Hard real-time task	Miss rate	Worst slowdown	Worst co-runner
a2time	1.408	32.3%	aifftr
aifftr	1.767	20.9%	bitmnp
aifirf	1.123	23.1%	canrdr
aiifft	1.405	25.6%	ttsprk
basefp	1.202	30.7%	aifirf
bitmnp	1.454	36.5%	aifirf
cacheb	1.179	17.0%	matrix
canrdr	1	25.5%	rspeed
idctrn	1.422	27.2%	cacheb
iirflt	1.488	22.7%	aiifft
matrix	1.981	30.9%	a2time
pntrch	2.306	47.6%	bitmnp
puwmod	1.62	28.6%	idctrn
rspeed	1.387	25.1%	idctrn
tblook	1.46	26.7%	idctrn
ttsprk	1.384	35.5%	bitmnp

TABLE I. NORMALIZED CACHE MISS RATE. WORST CASE SLOWDOWN IS IN PERCENT OF EXECUTION TIME WHEN HARD REAL-TIME TASKS RUN ALONE. WORST CO-RUNNER IS THE BENCHMARK RUNNING ON THE SOFT REAL-TIME CORES WHEN THE WORST SLOWDOWN WAS MEASURED.

had to incorporate the signalling for indicating the start and termination of tasks through IPIs.

Our experiments run on a Intel Xeon X5650 2.67GHz 6-core CPU. There is only one execution thread per core. We dedicate one of the cores to manage the other cores. This manager-core is responsible for configuring what to run on the other cores and setting up the PMC budgets. For the first experiment, the second core is dedicated to run hard real-time tasks. The remaining 4 cores run soft real-time tasks with limited access to the main memory through our PMC budgeting mechanism.

We use benchmark suite *AutoBench* from EEMBC [4] to evaluate our budgeting mechanism. During the first experiments with the benchmarks we noticed that there was almost no slowdown due to contention for the shared main memory. This was most probably due to the caches being so large that all code and data fit into them. To simulate more constrained system, we disable caching for data but not for code. To show the effects of our budgeting mechanism and reclamation scheme and to compare it to periodic budgeting without slack reclamation like the one presented in [17], we construct scenarios where a hard real-time periodic task would miss its deadline due to interference on the memory bus. When running a single periodic task with equal deadline, WCET and period, our budgeting mechanism behaves like a periodic resource server with replenishment points synchronized with task invocations.

We run all benchmarks in the *AutoBench* suite by themselves and measure their execution time. Then we measure the execution time of all benchmarks when co-running with all other benchmarks. We then compute the slowdown for all pairings and pick the pairing with the highest slowdown, which in this case is when $pntrch$ runs on the hard real-time core and $bitmnp$ runs on the soft real-time cores. The slowdown was measured to be 47.6%.

Table I shows the worst slowdown due to memory contention experienced by the different benchmarks. We can see in the table that different benchmarks are represented in the *worst co-runner* column, which leads to the conclusion that it is not only the miss rate that determines how much delay is introduced to the hard real-time core, it is also affected by how the benchmarks are laid out in memory. The reason for that is as follows: The bulk of the delay in DRAM is caused by waiting for the bank to store and load (precharge and activate) the row buffer. When memory accesses target different banks, the precharge and activate commands can be executed in parallel and interference is limited to waiting on the shared channel.

A. Single hard real-time core

We construct scenarios where $pntrch$ runs as a periodic task with $period$, $deadline$ and $WCET$ (for budget expiration purposes) all being 123.8% of the solo-run execution time and $bitmnp$ continuously runs on all soft real-time cores. To adjust slack in the system, we chose budgets so the hard real-time task has a safety margin towards its deadline. The safety margins can be seen as a simulation of different levels of pessimism in a WCET analysis, where a larger safety margin corresponds to a more conservative and therefore pessimistic analysis. The intuition behind is as follows: with shared caches among concurrently executing tasks, formal analysis will become extremely pessimistic, as worst-case co-runners need to be considered. This we model by adjusting the safety margin of our WCET estimate. The latter is in general to optimistic, as it was obtained by us by measurement, i. e., empirically, rather than bounding it by an exhaustive, formal WCET-analysis method.

The budget is distributed evenly to the soft real-time cores. In Fig. 3 we can see that for larger safety margins (or more pessimistic analysis), the slack reclamation technique yields a substantial performance improvement. Fig. 3 shows the performance improvement of soft real-time tasks when employing our method compared to the one presented in [17]. However, due to limitations of our implementation, the figures of the approach of [17] are too positive. The picture would be even more positive for our method, if we would have had compared our method to non-synchronized periodic budgeting. In this latter case, the budgets assigned under the approach of [17] need to be halved in order to give the same timing guarantees. This is because, if the budgets are not synchronized with task executions, there may be two full budgets available

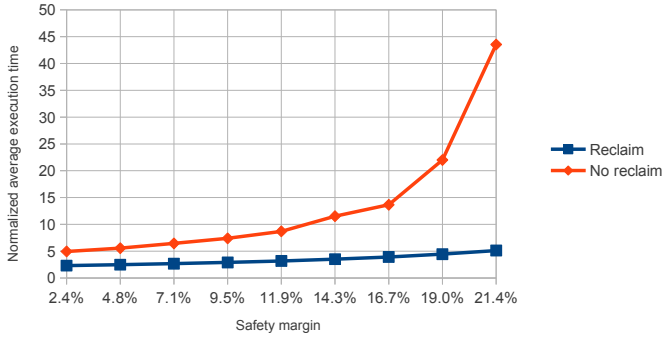

Fig. 3. Comparison of normalized average execution times of $bitmnp$ running on a soft real-time cores when running budgeting with our slack reclamation technique and without it. Safety margins are expressed as a percentage of solo-run execution time.

Task-set	C_1	P_1	D_1	$H_1 \cdot L$	C_2	P_2	D_2	$H_2 \cdot L$
T_1, T_6	1	2	2	0.37	1	2	2	0.47
T_2, T_7	1	2	2	0.57	1	3	3	0.46
T_3, T_8	1	2	2	0.75	1	4	4	0.41
T_4, T_9	1	3	3	0.48	2	4	4	0.65
T_5, T_{10}	1	3	3	0.46	2	5	5	0.95

TABLE II. TASK SETS USED FOR PERFORMANCE MEASUREMENTS. ALL TIMES ARE NORMALIZED.

for the soft real-time cores during the execution of a hard real-time task with the same period as the budget replenishment. In our experiments we have given the same budget to both methods for simplicity, even though this is less advantageous for our budgeting scheme.

B. Multiple hard real-time cores

To see the benefits from our method with multiple hard real-time cores, we perform an experiment with two periodic tasks behaving like the ones in the example in Sec. IV-B (T_2 in Table II). Workloads for the tasks are picked from the EEMBC benchmarks and we tune the number of iterations through the benchmarks to fit the task descriptions. We know the number of cache misses for both tasks and use these together with $L = L_{conf}$ to find the highest possible values for B_1 and B_2, where L_{conf} is taken from [11]. Finding budgets in this case is simple as there is only one budget affecting response time for each task. We run the tasks on separate cores and let the three remaining cores run best-effort tasks. To have something to compare with, we run the same tasks with unlimited budgets and add a dummy task that simulates periodic budgeting without doing any computations. The budget for the dummy task is chosen as high as possible and such that timing correctness for our two tasks is not violated. With this setup, normalized average execution time for a best-effort task running on a soft real-time core is **4.6** when using our method and **56.7** when using static periodic budgeting (T_2 in Fig. 4).

We construct four additional task sets (see Table II) with EEMBC benchmarks, calculate the largest tolerable budgets and see how performance for co-running best-effort tasks is affected. In Fig. 4 the average normalized execution times for a best-effort task is shown. T_6 through T_{10} use the same task parameters as T_1 through T_5, but are instantiated with different offsets and with task parameters and budgets scaled down by a factor of 10. This has a noticeable impact when running T_6 compared to T_1, where normalized execution times of a best-effort task is more than three times longer when using our method. This difference can be explained by there being more or less overlap of the time intervals when the hard real-time cores allow unlimited access to the main memory.

C. Measured overheads of budgeting

Our technique adds some additional overhead in the form of communicating task starts and finishes via IPIs. This was also included in our comparisons for both methods though the signalling is not necessary with periodic budgeting. We measured the additional overhead to be between $2.5\mu s$ and $7.0\mu s$ per task invocation while the task executions are in the order of hundreds of milliseconds, so the additional overhead is negligible for this particular setup. The signalling overhead is also experienced by the hard real-time cores. In the case of high frequency tasks, the overhead can be halved by simply removing the "task finished" signal. The budget will then be discarded when it expires, but no reclaim can be performed.

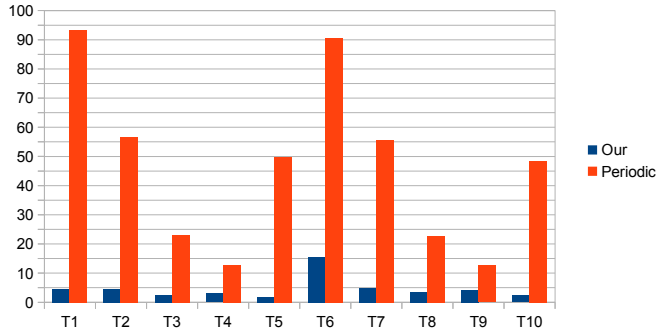

Fig. 4. Measurements of average execution times for best effort tasks under different budgeting strategies; our scheme with reclamation and periodic budgeting. Measurements are normalized to the average execution time of all benchmarks running solo.

VI. CONCLUSIONS

In this paper, we have introduced a dynamic budgeting mechanism to provide upper bounds on delay caused by contention on the memory system. The mechanism leverages the insight that when a hard real-time task finishes early, which it always will since execution times are often over-approximations, slack can be used to speed up soft real-time tasks by allowing them to access the memory in a non-restricted manner during the slack.

We have implemented the mechanism in the *Fiasco.OC* micro-kernel and evaluated it empirically. The dynamic budgeting mechanism significantly improves the performance for soft real-time tasks at the cost of a small signalling overhead compared to static periodic budgeting.

Our study demonstrates that this is an important step towards timing-predictable use of multicore architectures in real-time systems, without having to sacrifice too much of the average case performance that modern architectures offer.

In the future work, we plan to develop techniques for mapping both hard and soft real-time tasks onto processor cores and for calculating budgets such that some objective is maximized, e. g., memory utilization or processor utilization of soft and hard real-time cores.

REFERENCES

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 4–13, 1998.

[2] M. Behnam, R. Inam, T. Nolte, and M. Sjödin. Multi-core composability in the face of memory-bus contention. *SIGBED Rev.*, 10(3):35–42, Oct. 2013.

[3] Z. Deng, J. W. S. Liu, and J. Sun. A scheme for scheduling hard real-time applications in open system environment. In *Real-Time Systems, 1997. Proceedings., Ninth Euromicro Workshop on*, pages 191–199, 1997.

[4] EEMBC. http://www.eembc.org/.

[5] X. Feng and A. Mok. A model of hierarchical real-time virtual resources. In *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, pages 26–35, 2002.

[6] Fiasco.OC. http://os.inf.tu-dresden.de/fiasco/.

[7] G. Giannopoulou, K. Lampka, N. Stoimenov, and L. Thiele. Timed model checking with abstractions: Towards worst-case response time analysis in resource-sharing manycore systems. In *Proc. International Conference on Embedded Software (EMSOFT)*, pages 63–72, Tampere, Finland, Oct 2012. ACM.

[8] S. Goossens, B. Akesson, and K. Goossens. Conservative open-page policy for mixed time-criticality memory controllers. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 525–530, San Jose, CA, USA, 2013. EDA Consortium.

[9] W. Jing. Performance isolation for mixed criticality real-time system on multicore with xen hypervisor. Master's thesis, Uppsala University, Department of Information Technology, 2013.

[10] T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury. Bus-aware multicore wcet analysis through tdma offset bounds. In *23rd Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3 –12, 2011.

[11] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. R. Rajkumar. Bounding memory interference delay in cots-based multi-core systems. Technical report.

[12] P. Kumar, N. Stoimenov, and L. Thiele. An algorithm for online reconfiguration of resource reservations for hard real-time systems. In *Proc. of 24th Euromicro Conference on Real-Time Systems (ECRTS 2012)*, pages 245–254, Pisa, Italy, Jul 2012. IEEE.

[13] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 128–138, New York, NY, USA, 2000. ACM.

[14] J. Rosn, 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*, pages 49–60, 2007.

[15] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. Comput.*, 44(1):73–91, Jan. 1995.

[16] R. Wilhelm and B. Wachter. Abstract interpretation with applications to timing validation. In A. Gupta and S. Malik, editors, *CAV*, volume 5123 of *LNCS*, pages 22–36. Springer Verlag, 2008. Princeton, NJ, USA.

[17] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 299–308, 2012.

[18] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 55–64, 2013.