Adaptive Resource Sharing in Multicores

Kai Lampka Jonas Flodin Wang Yi
Department of Information Technology, Uppsala University

Adam Lackorzynski
Technische Universität Dresden

***Abstract*—This short paper presents an adaptive, operating system (OS) anchored budgeting mechanisms for controlling the access to a shared resource. Temporarily blocking accesses from a core reduces the waiting times of other applications executing in parallel on other cores. This helps to guarantee the assumed worst case execution time bounds at run-time. In addition to our previous work [1], the presented scheme considers shifting of unused access bandwidth among applications and takes advantage from a time-triggered scheduling policy for executing real-time applications at core-level.**

I. Introduction

a) Motivation: Sharing of hardware as found in COTS multicores brings in hidden dependencies when consolidating hard and soft real-time applications on a single processor. These dependencies can provoke timing faults that are difficult to foresee and can corrupt the functionality of the system.

The challenge inherent to the design of the run-time environment to support the timing correct execution of mixed critical workloads is three-fold.

Firstly, hard real-time tasks need to be isolated, such that their assumed upper bound on their execution time always holds. In addition, standard real-time analysis builds on task sets with known bounds on their execution times. The feasibility of a scheduling strategy, shown at design time, is guaranteed to hold at run-time if the upper bounds on the execution times (and activation frequencies) are not violated. As unaccounted waiting at a resource prolong execution times, it can become a threat to a systems timing correctness.

Secondly, resource sharing needs to be considerably dynamic, to avoid over-provisioning and thereby achieve good utilization of the used equipment.

Thirdly, the mechanism to coordinate the access to a shared resource must not be too complex to limit the computational overhead experienced at run-time.

b) Technical problem description: As an example to resource sharing, this short paper considers the sharing of the dynamic random access memory (DRAM).
When carrying out a worst-case response time analysis (WCRT) for quantifying the computation time consumption of an application, one has to assume that a memory access from a core can be delayed by all other memory accesses occurring while the respective access is waiting at the DRAM-controller. With n access requests from other cores, this yields a delay of $(n + 1)$ times the worst case service time until a request is served. This assumption is conservative as it overapproximates the actual behaviour of the system at run-time. However, it is safe as long as less than n competing access requests occur. It is therefore of uttermost importance to ensure at run-time that the number of competing memory access requests is bounded by a pre-defined number and one does not experience unaccounted waiting times due to unaccounted memory requests.
In this short paper, we summarize our effort to do this efficiently and effectively and point out directions for improvements left to the future.

This work is partly funded by DFG-SPPEXA's project FFMK.

c) Related Work: For dealing with memory access contention effects in the setting of multicore architectures, several strategies have been proposed.
Time deterministic memory designs avoid interference by physically separating relevant parts of the memory hierarchy and exclusively assigning parts to cores. This ranges from the use of scratchpad memories [5] to the partitioning of main memory [3]. However, these techniques all rely on the layout of the memory hierarchy.
Another way to feature timing predictability is provision of isolation mechanisms as part of the run-time environment. At the level of OS, this can be done by controlling the virtual to physical address mappings [6] or by restricting access frequencies of the main memory for each core [7], [8].

d) Own Contribution: Advancing over the work of Pellizzoni et al. [7], [8], this short paper propose the following innovations when it comes to resource access budgeting schemes: (a) we enable lifting of budgets, namely once all real-time tasks are pre-maturely completed. (b) we also feature donation of budgets. But, donation is only allowed, if the donating real-time task has already terminated.
Both features can be considered safe, the safeness of budget lifting is demonstrated in [1]. The safeness of budget donation comes from the fact that we avoid premature shifting of resource accesses. This is important and this way we avoid starvation of real-time applications which could provoke timing faults.
In addition to our own work [1], this short paper presents a budgeting scheme which takes advantage of a time-triggered scheduling strategy of real-time applications at the levels of cores. This way, we not only lift unneeded budgets more often. We also hope to shift unused access budgets more often as this can take place every time all real-time applications of a time-frame have processed their workload.

II. 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 $M - K$ cores are executing best-effort applications.

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

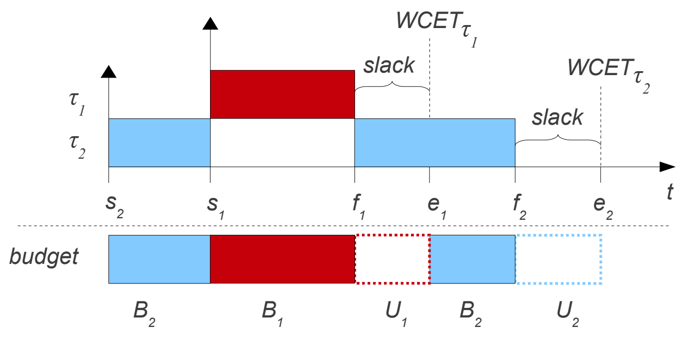

Fig. 1. Budgeting example with two tasks. Arrows pointing up denote job releases and dashed vertical lines denote the point in time when a job would have finished if it needed the entirety of its WCET.

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.
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. **It is these cores which we intend to control through the presented budgeting scheme and thereby ensure timing correctness of the hard real-time applications running in parallel.** All cores share a single memory controller which acts as an arbiter for serving requests to DRAM.

III. Dynamic Budgeting with Lifting

The initial scheme of budget enforcement and lifting is presented in [1]. We briefly recall its working principle by means of an example.

Fig. 1 illustrates the execution of two tasks. The upper part depicts their interleaved execution on the hard real-time core. 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.

IV. Combining Ressource access budgeting and Time triggered application scheduling

A. Time-triggered execution of tasks

Scheduling of hard real-time tasks is organized according to a standard time-triggered scheme, e. g., as defined in [2].

A time-triggered schedule at core i is a sequence of K_i slots $s_{i,j}$, where $s^{\Delta}_{i,j}$ refers to the time length of each slot.

While executing a slot $s_{i,j}$, we need to guard that all the cores running soft real-time applications do not issue more than $B^{eff}(s_{i,j})$ accesses to the main memory in total.

Algorithm 1 Enforcing budgets on a soft core

```
Requires: timer $T$, active budget $B$,
          set of active budgets $Budget$
Input: signal $e$ mapping to a slot and action
procedure BScheduler(signal $e$)
    $PREEMPTION = OFF$
    if action($e$) $\in \{depleted, expired\}$ then
        wait4Timer(T)
        goto line 28
    end if
    update($Budgets, B.b^{eff}$ − readPMC(), $B.t - T$)
    if action($e$) $== activate$ then
        insert($Budgets$, slot($e$))
    else if action($e$) $== deactivate$ then
        remove($Budgets$, slot($e$)))
    else if action($e$) $== donated$ then
        $C$ = peek($Budgets$, slot($e$))
        updateDonation($Budgets, B.d, C.t$)
    end if
    while $B$ = peek($Budgets$)) $\neq \emptyset \wedge B.t \leq 0$ do
        remove($Budgets, B$)
    end while
    if $B == \emptyset$ then
        stopTimer($T$)
    else
        setPMC($B.b^{eff}$)
        setTimer($T = B.t$)
    end if
28: $PREEMPTION = ON$
end procedure
```

Below we detail on the algorithm to implement this basic functionality. For simplicity, we ignore the distribution of budgets and donations over multiple cores executing a soft real-time workload,. For the presented algorithms, the distribution could be arranged transparently, through a dedicated administering core.

B. Budget enforcement for soft real-time workloads

The required functionality for guarding the number of memory accesses such that timing correctness of the hard real-time tasks is ensured, is provided by Algorithm 1.

The implementation details of Algorithm 1 are as follows: we assume that there is a queue $Budgets$ of active budgets, with at most one active budget per hard real-time core.

Within the queue, the active budgets are ordered by increasing budget sizes. The following functions are used to access items of the queue: function `replace` and `remove`, which work as expected. Function update($Budgets, a, b$) decreases all budgets of the queue by value a and decreases their lifetimes by value b. This is needed once the decisive budget has reached its lifetime or is replaced by a newly activated budget. Function `peek` gives the head of the queue, i. e., the active budget with the smallest number of allowable cache misses. The functions does not remove the item from the queue.

The algorithm itself works as follows: upon depletion of the decisive budget or at the end of its lifetime the core suspend execution for the remaining lifetime, which in case of the "*end of lifetime*" situation is 0 (line 5).

In case the decisive budget has reached the end of its life time or a new budget to be activated has arrived, we update all active budgets with respect to to the number of cache misses and the expired time occurred during the current budget has been made the decisive one.

In case of a premature deactivation the decisive budget, it is removed from the budget queue and the next active budget is fetched. This can either be the same, but updated budget, a new one, where budgets with invalid lifetime are discarded, or it is an empty budget (line 18-20).

In case of an empty budget all active budgets have been prematurely invalidated and the core has a non-restricted allowance to the main memory.

In case a valid budget is fetched from the queue, the LLC-register and the lifetime clock counter are set accordingly (line 25 and 26).

Budget donation executed by a hard real-time core is considered before actually fetching a budget from the queue. Function `updateDonation`$(Budgets, a, b)$ adds value a to each budget, here parameter $B.b^{eff}$ and does so only for those budgets which have a residual lifetime smaller than b.

V. Implementation

For evaluation, we use the L4Re microkernel system that provides the environment to run existing applications and operating systems through virtualization as well as native microkernel-based applications. The L4Re gives us the flexibility to use virtualization as well as specific native applications in a very controlled environment.

Scheduling in the L4Re microkernel applies scheduling contexts (SCs), a thread-specific data structure that contains all information required for scheduling [4]. A special features of the SC mechanism is that a thread, or vCPU, can have multiple SCs, allowing to give a thread/vCPU multiple different scheduling parameters. This is especially useful in virtualization contexts where the guest OS can use multiple SCs to express the requirements of its internal tasks to the microkernel. In our work we use the SC mechanism to implement budgets based on performance counters.

1) Hardware Performance Counters: Modern processors have a performance monitor counter (PMC) unit that allows to count hardware-related events in the CPU core, such as cache misses. The core can also generate interrupts when a counter reaches a predefined threshold. Using the PMC it is possible to count the number of last-level cache misses which is equivalent to the number of main memory fetches. If the number of memory fetches reaches a certain threshold, the microkernel may suspend the execution of soft real-time applications to avoid an overload of the main memory with memory access requests. The challenge is to use the PMC in such a way, that is dynamically resetting the PMC and adjusting the threshold, that the maximum amount of memory accesses can be placed on the DRAM without affecting real-time applications.

2) PMC Pecularities: All Intel Core-i CPUs have a minimal standard set of performance counters that includes the last-level-cache-miss counter. The first experiment we did was checking whether our test program indeed uses all of the memory bandwidth available. By running it on a different number of cores in parallel we expect the runtime of each program to increase with the number of cores. That is, on 4 cores each program shall run 4 times longer compared when running alone in the system. We observed this behavior.

However, when we added delays to the memory access loop in the test program, with the goal to not fully use up all the memory bandwidth, the respective last-level-cache-miss counter shows significantly less events although the same amount of memory was accessed. This is likely because of the hardware memory prefetcher where memory accesses are not counted, as they are no cache misses. We tried to disable the prefetcher via the `IA32_MISC_ENABLE` MSR [8], however, this yields to a general protection fault when writing the MSR on the used i7-4770 CPU. Using non-cached memory is no choice either because those accesses do not causes cache-relevant events, such as misses. Using other counters available on the specific CPUs showed either the same behavior (significantly different values for with and without delay loops), or did not count at all.

Intermediate result is that Intel-based x86 desktop CPU, such as the i7-4770, can not be used to implement memory access budgeting based on performance counters. We need to look at other CPU lines, such as Xeon CPUs, or older Intel CPUs, whether they are better suited, for example, because they allow to disable the prefetcher. Alternatively, looking at ARM Cortex-A CPUs shows a counter called `MEM_ACCESS` which sounds promising as well.

References

[1] Jonas Flodin, Kai Lampka, and Wang Yi. Dynamic budgeting for settling DRAM contention of co-running hard and soft real-time tasks. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems, SIES 2014, Pisa, Italy, June 18-20, 2014*, pages 151–159, 2014.

[2] Gerhard Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 152–161, 1995.

[3] Min Kyu Jeong, Doe Hyun Yoon, Dam Sunwoo, M. Sullivan, Ikhwan Lee, and M. Erez. Balancing dram locality and parallelism in shared memory cmp systems. In *High Performance Computer Architecture (HPCA) 2012*, pages 1–12, Feb 2012.

[4] Adam Lackorzyński, Alexander Warg, Marcus Völp, and Hermann Härtig. Flattening hierarchical scheduling. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '12, pages 93–102, New York, NY, USA, 2012. ACM.

[5] I. Liu, J. Reineke, and E.A. Lee. A pret architecture supporting concurrent programs with composable timing properties. In *ASILOMAR 2010*, pages 2111–2115, Nov 2010.

[6] Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen, and Chengyong Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 367–376, New York, NY, USA, 2012. ACM.

[7] Heechul Yun, Gang Yao, R. Pellizzoni, M. Caccamo, and Lui 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.

[8] Heechul Yun, Gang Yao, R. Pellizzoni, M. Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *RTAS 2013*, pages 55–64, 2013.