Combining Performance Monitoring and Resource Budgeting on Multi-core for Real-Time Guarantees

Jonas Flodin
Department of Information Technology
Uppsala University
jonas.flodin@it.uu.se

Kai Lampka
Department of Information Technology
Uppsala University
kai.lampka@it.uu.se

Wang Yi
Department of Information Technology
Uppsala University
yi@it.uu.se

ABSTRACT

Sharing of infrastructure is a common characteristic for modern multi-core architectures. However, this sharing, e.g. in the case of parts of the memory hierarchy, may yield unexpected waiting times which will significantly contribute to the execution time of software. This can become problematic, when executing applications with hard timing constraints in parallel to some soft real-time applications, mapped to different cores, but accessing the same main memory. As timing violations of system-critical applications need to be ruled out, use of a shared resource by soft real-time applications needs to be controlled. Run-time Budgeting and Monitoring are means of guaranteeing and controlling timing correctness at run-time. Whilst always satisfying the hard real-time constraints of the critical applications, worst-case based resource budgeting commonly leads to performance degradations of the soft real-time applications. In this paper we present a novel dynamic budgeting mechanism for achieving timing predictability in multi-core architectures with shared main memory. The proposed technique exploits a run-time monitoring mechanism for dynamically setting budget lines, s. t. hard real time guarantees hold, but average case performance of soft real-time application running on the same platform is not utterly destroyed.

1. INTRODUCTION

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 when it comes to the design and implementation of embedded control systems with hard timing constraints. The reason for this is as follows: integration of multiple (hard and soft time) applications into a single multi-core device brings the sharing of hardware infrastructure among the logically independent applications. The sharing might yield hidden dependencies, respectively interference between the applications, not only logical independent from each other, but possibly also running on different cores. As a result, one may experience unwanted side effects, which are not only extremely difficult to detect, but have the potential to corrupt to the system's timing behaviour.

As a concrete example, one may think of a dual core system with a shared L2-cache. The concurrently executing software will mutually over-write each others cache entries. This in turn will significantly add to their execution times as code segments and data items must be re-fetched from the main memory. However, this interference, which can be bounded by well known analysis techniques [10], is not the only source of interference. Each time when fetching an item from the main memory, task execution is suspended until the actual request is served. The resulting waiting time does not only depend on the number of pending memory access requests from the tasks executing on the other cores. It also depends on the complex memory arbitration scheme implemented by the Dynamic Random-Access Memory (DRAM) controller. The challenge inherent to such 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 shared resource and thereby reduce their responsiveness.

Resource servers are a standard mechanism to coordinate the access to a 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. The basic mechanism works as follows: the budget of the server is a function over the timeline, 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. One concrete example of a resource server is the Constant Bandwidth Server [1].

The cores in common multi-core architectures typically share parts of the memory hierarchy. This include caches, memory controller (MC) and DRAM modules. When a core requests access to memory, the completion time for the request is dependant on the state of the memory. To be able to give precise timing information for an access one would need to have information about the state of the memory hierarchy at the time of request. Such information could be for example: if the requested data is in one of the caches or not, how many requests are currently buffered in the MC and if the requested memory location resides in an open row in DRAM. Also, since the MC may reorder accesses [9] for higher throughput of the memory system, it is important to know the arbitration scheme in use and how many other interfering requests might be issued while the request is being served.

The state of the memory hierarchy is manipulated by all concurrent threads, so it is impossible to give precise timing information for a thread's memory accesses by analyzing it in isolation. At the same time, looking at all concurrent threads and their possi-

ble interleavings to try and figure out precise memory access times is practically infeasible. Therefore, in systems with real-time requirements, over-approximation of access times is used which often leads to over-provisioned systems.

There are some techniques which improve predictability of the performance of the memory hierarchy. As an example of such a technique, one may consider cache isolation through page-coloring [5]. Page-coloring is a technique where the caches are partitioned between tasks or cores, such that they cannot evict each others cache content, which makes cache hits more predictable. Coloring is done by controlling the virtual to physical address mappings.

In [6] it is shown how the memory contention affects performance and how the effects can be mitigated by throttling memory accesses. Memory accesses are measured using performance monitor counters (PMC). When it is observed that a core accesses memory too frequently, the core is put to sleep until the next period starts.

The contributions of this paper are a budgeting mechanism and a slack reclamation scheme that together make interference in the MC and DRAM more predictable while increasing throughput for interfering workloads.

The remainder of the paper is organized as follows: in Sec. 2 we present our system model. In Sec. 3 we describe our proposed budgeting mechanism. In Sec. 4 we detail our implementation and our experimental setup and provide some data on how well our method works. In Sec. 5 we give a short overview of related work and Sec. 6 concludes the paper.

2. SYSTEM MODEL

We consider a system deployed on a typical COTS multicore architecture. There are M CPU-cores, one of which is executing highly critical real-time software which is modeled as a set of N sporadic tasks $T = \{\tau_1, \tau_2, ..., \tau_N\}$, $\tau_i = (C_i, P_i, D_i)$ where C_i is the worst case execution time (WCET) for the task when running alone on one critical core, P_i is the minimum interarrival time of the task and $D_i \leq P_i$ is the task's deadline $\forall i = 1..N$. Tasks are ordered by their priority such that τ_j has higher priority than τ_i if $j < i$. The other cores we collectively call soft-cores and they execute soft real-time or best-effort 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 timing behaviour, we overapproximate the time it takes to serve a single request in the worst case as a constant L. A request r to main memory may be delayed by at most $\#requests \cdot L$ when $\#requests$ arrive before or after the arrival time of r due to reordering of requests in the memory controller.

Suppose we know the worst case number of accesses to main memory that the soft-cores may issue during one task instance. We denote this number as B_i, then one instance of task τ_i may be delayed by at most $B_i \cdot L$ due to main memory contention. This leads to a new WCET for each task when executing concurrently with software running on the soft-cores. We denote this as $A_i = C_i + B_i \cdot L$ (assuming that the initial WCET analysis for deriving C_i doesn't rely on being able to predict row buffer hits/misses in the DRAM).

T must pass a scheduling test in order for the system to be feasible. We use the classical test [7] of making sure that the response times of all tasks are smaller than or equal to their deadlines,

$$R_i \leq D_i, \tag{1}$$

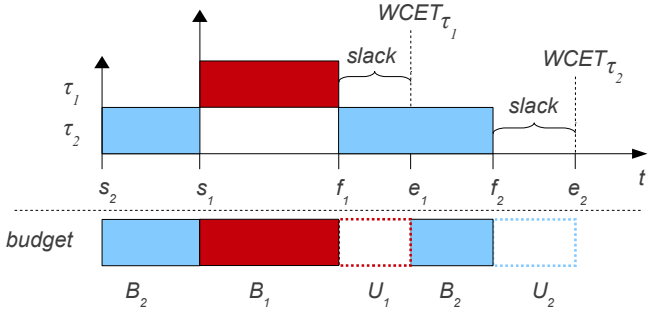

Figure 1: Budgeting example with two tasks. Arrows pointing up denote job releases and dashed vertical lines denote the point in time when a job would have finished if it needed the entirety of its WCET.

where R_i is the smallest possible solution to the recurrence relation

$$R_i = A_i + \sum_{j<i} \left\lceil \frac{R_i}{P_j} \right\rceil \cdot A_j. \tag{2}$$

3. RESOURCE BUDGETING MECHANISM

Based on formal modeling and analysis of multi-core architectures with shared resources [8, 4], one can compute an upper bound on B_i such that the system is feasible, i.e., the critical tasks meet their timing requirements. However, such techniques may yield too high values for interference when considering worst case interleavings of concurrently running tasks. Instead of analyzing memory accesses, we impose sensible values for B_i, such that (1) still holds. Then we enforce that the soft-cores do not exceed this value through the use of a budgeting mechanism.

The budgeting mechanism we use works as follows:

- When a task τ_i starts executing at time s_i, a message is sent to the soft-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.
- The number of memory accesses on the soft-cores are monitored using appropriate PMC events (e.g. cache misses).
- If the budget on a soft-core is depleted, it may no longer continue executing until the expiration time.
- If τ_i finishes early at time f_i, another message is sent to the soft-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.
- In the case of a higher priority task τ_j starting to execute at time $s_j > s_i$, the soft-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.
- When a higher priority budget expires, the soft-cores fall back to using the budget of the second highest priority budget. If there are no more budgets, the soft-cores continue executing with unrestricted access to memory.

Consider an example execution as shown in Fig. 1, where the upper part depicts the interleavings of two tasks on the critical core

and the lower part shows which budget is in effect on the soft-cores. The critical core starts executing τ_2 and signals the soft-cores to use budget B_2 at time s_2. The critical core continues executing τ_2 until time s_1, when it is preempted by the arrival of τ_1, which also triggers the soft-cores to switch budget to B_1.

When τ_1 finishes early at f_1, the soft-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 high criticality core switches to executing τ_2. When U_1 expires at time e_1 the soft-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.

We claim that slack reclaiming using U_i as budget instead of B_i when task τ_i finishes early does not increase the worst-case response times and therefore (1) 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 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.

4. IMPLEMENTATION AND EVALUATION

For experimental evaluation we implemented our budgeting mechanism in the Fiasco.OC microkernel [3]. We argue that it is a good choice of operating system as it has a small code-base, is open-source, runs on common architectures and due to its microkernel nature, already has good 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 feature rich and hard-to-analyze operating systems and applications in isolation.

When a task starts executing on the critical core, this is signaled by an inter-processor interrupt (IPI) broadcast to all soft-cores. The message associated with the IPI tells the soft-cores which task is starting execution. The soft-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 a priority-ordered list of budgets.

Whenever the list of budgets is non-empty, the highest priority 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 budget expires. If the PMC overflow interrupt is triggered, the core is set to idle, using the HLT instruction, until the expiration timeout fires. When the expiration timeout fires, the budget is removed from the list of budgets and the next budget in the list is activated.

After a task is done with its execution another IPI broadcast signals the soft-cores that the budgets for this task instance are no longer needed. The soft-cores then, upon receiving the IPI, disable the PMC overflow interrupt for the budgets in question.

Only minor changes to the runtime environment are required to use our budgeting mechanism. The changes we made are only additions to enable communication of budget policies to the kernel and signaling task starts and finishes through IPIs.

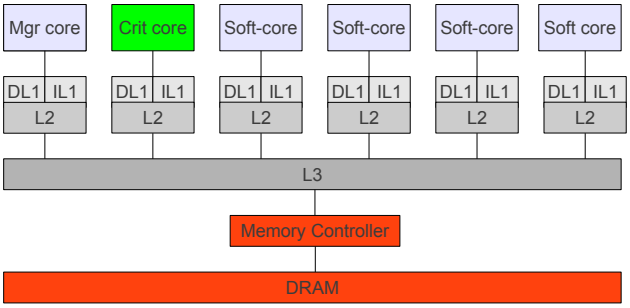

Figure 2: Hardware architecture for our experimental evaluation. Each core has a dedicated 32kB data L1 cache, 32kB instruction L1 cache and a 256kB unified cache. All cores share a 12MB unified L3 cache as well as a main memory populated with 4GB of DRAM.

Fig. 2 shows the architecture of our experimental setup. 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. On the second core we run critical tasks. The remaining 4 cores run non-critical tasks with limited access to the main memory through our PMC budgeting mechanism.

We use benchmark suite $Autobench$ from EEMBC [2] 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 [11], we construct scenarios where a critical periodic task would miss its deadline due interference on the memory bus. When running periodic tasks, 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 corunning with all other benchmarks. We then compute the slowdown for all pairings and

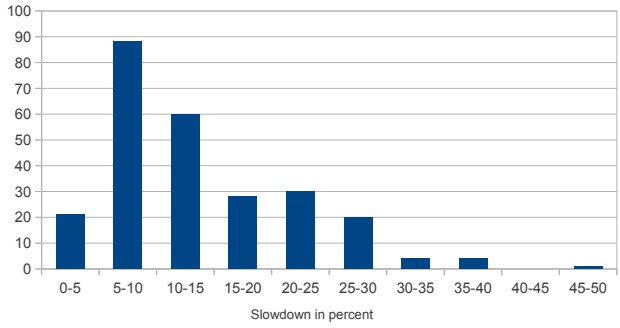

Figure 3: Slowdown distribution when corunning benchmarks.

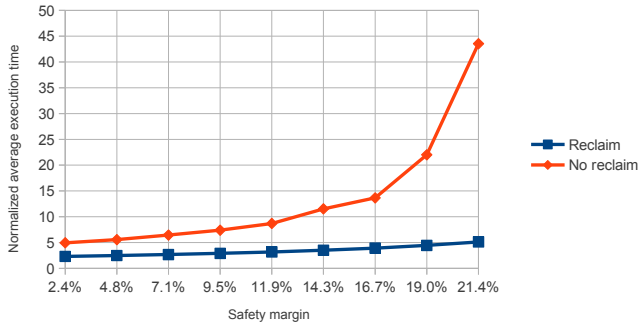

Figure 4: Comparison of normalized average execution times of $bitmnp$ running on a soft-core when running budgeting with our slack reclamation technique and without it. Safety margins are expressed as a percentage of solo-run execution time.

pick the pairing with highest slowdown, which in this case is when $pntrch$ runs on the critical core and $bitmnp$ runs on the soft-cores. The slowdown was measured to be 47.6%. In Fig. 3 show the distribution of the slowdown due to memory contention. 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-cores. To adjust slack in the system, we chose budgets so the critical task has a safety margin towards its deadline. The budget is distributed evenly to the soft-cores. In Fig. 4 we can see that for larger safety margins, the slack reclamation technique yields a performance improvement of a factor 8.5.

Our technique adds some additional overhead in the form of communicating task starts and finishes via IPCs. Due to limitations of our implementation, this was also included in our comparisons. 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. The additional overhead should in the most cases be insignificant but for some high frequency tasks, it may be too much to justify the use of our technique.

5. RELATED WORK

We are not the first to exploit budgeting of main memory accesses in order to gurantee deadlines of hard real-time tasks executing on top of multi-core architectures.

In [11], Yun et al. present a way of guaranteeing memory bus bandwidth to one critical core through memory access throttling on non-critical interfering cores while minimizing the performance impact of throttling on the non-critical cores. Periodically replenished memory bus budgets are given to interfering 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 tasks on that core are moved away from the ready-queue, until the next replenishment point.

In their forthcoming work [12], Yun et al. present a bandwidth reservation and reclaiming scheme they call 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.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have introduced a 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 over-approximations, slack can be used to speed up soft real-time tasks by allowing them unrestricted memory access during the slack.

We have verified our mechanism by implementing it in the Fiasco.OC micro-kernel and had it evaluated empirically. Our mechanism gives better performance for soft-realtime tasks at the cost of a small overhead on for hard-realtime tasks executing on a separate processing core.

We belive that this work 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.

Our method currently only supports one core as a critical hard real-time core. One natural extension that we would like to look at in the future is to have multiple such cores.

Some shared resources may have to be guarded with critical sections, but currently we have no scheme for sharing critical sections between soft and hard real-time tasks. We would like to look into how to extend our method with techniques from critical section handling in resource servers.

How to calculate budgets is still an open question. As a future work we would like to find some heuristic to guide assignment of budgets, such that feasability is maintained while soft real-time tasks have as high throughput as possible.

7. 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] EEMBC. http://www.eembc.org/.

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

[4] 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.

[5] N. Guan, M. Stigge, W. Yi, and G. Yu. Cache-aware scheduling and analysis for multicores. In *Proceedings of the seventh ACM international conference on Embedded software*, EMSOFT '09, pages 245–254, New York, NY, USA, 2009. ACM.

[6] 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.

[7] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[8] M. Lv, W. Yi, N. Guan, and G. Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 339–349, 2010.

[9] 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.

[10] 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.

[11] 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.

[12] 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.