Slot-Level Time-Triggered Scheduling on COTS Multicore Platform with Resource Contentions

Ankit Agrawal, Gerhard Fohler
Chair of Real-Time Systems
TU Kaiserslautern, Germany
{agrawal,fohler}@eit.uni-kl.de

Jan Nowotsch, Sascha Uhrig
Airbus Group Innovations
Munich, Germany
{jan.nowotsch,sascha.uhrig}@airbus.com

Michael Paulitsch
Thales Austria GmbH
Vienna, Austria
michael.paulitsch@thalesgroup.com

I. Introduction and Motivation

A number of safety-critical domains, such as avionics, use time-triggered (TT) architectures for reasons of reliability, ease of certification, reduced integration and maintenance costs, system-wide determinism, etc. [1].

The move to multicore platforms poses a number of fundamental problems for real-time scheduling in particular, even in idealized scenarios without consideration of overheads or platform characteristics. COTS multicore platforms generally share various hardware resources such as on-chip network, memory sub-system etc. amongst cores, introducing resource contentions and inter-core interferences. This results in large variability[1] in the execution time of a task depending on the latency and number of inter-core interferences from co-executing tasks on the other cores. Further, the execution of each new task, in turn, introduces additional inter-core interferences, affecting the variability in execution time of already co-executing tasks. E.g., it is shown in [2] that the single store request latency increases by 25.82 times when the number of active cores are increased from 1 to 8.

These challenges effect TT systems even more, as schedules have to be determined offline. The extension of offline scheduling to multicore platforms raises great concern in safety-critical application domains using single core platforms. E.g., in the avionics domain, in which certification and long product life are essential, only very limited steps are currently considered: The position paper from EASA and FAA proposes, as next step, at most 2 active processing cores [3]. Even with only one core active, certification is a challenge.

The problem of scheduling for TT systems on COTS multicore platform considering inter-core interferences is difficult because of three primary reasons: Firstly, we need to provide guarantees that offline computed bounds on variability in execution time of each task will hold at runtime, warranting runtime regulation of inter-core interferences during task execution. Secondly, we need to bound the variability in execution time of a task in the offline phase considering possible runtime inter-core interferences and task's deadline, as reserving hardware resources for each task considering worst-case inter-core interferences would be very pessimistic. Finally, we need to estimate at design time the maximum runtime inter-core interferences for each task in each slot, as we cannot obtain this information using traditional static WCET analysis tools due to unavailability of architecture models for the complex COTS multicore processors.

Related Work In [4], Yun et al. propose controlling memory accesses from all but one cores to limit inter-core interferences experienced by hard real-time tasks executing on just 1 core. Yun et al. extended the work in [5] allowing all cores to execute hard real-time tasks by regulating memory accesses using a memory server on each core. However, they assume that the given memory server budget reservations for each core are constant for each server period. Yao et al. [6] present a method to bound variability in execution time of each task on all cores considering round-robin arbitration between cores for memory accesses. However, the work does not consider additional arbitration and contention delay introduced by shared on-chip network in the analysis. In [2], [7], Nowotsch et al. consider a timeline divided into unequal length process frames and provide a method that bounds variability in execution time of each task in a given process frame by considering maximum inter-core interferences in the offline phase. The runtime mechanism enforces the offline computed bound in each process frame. However, they impose restrictions such as a new task is only allowed to execute on the completion of all tasks in a process frame.

A common way of operation in TT systems is to assume a minimum temporal granularity of operation, called slots [1], at runtime. Offline, a schedule table is created which assigns parts of task executions to these slots. Slots can be seen as units of resource reservation, i.e., reserving chunks of CPU time to the assigned tasks. In this paper, we propose to extend these reservations to several resources. We propose a two-part slot-level based resource-control method using a TT scheduling approach that enables the use of multicore platforms for executing hard real-time tasks in TT systems. We propose a runtime mechanism consisting of two servers running on each core - processing time server and memory access server - each having a fixed server period equal to the slot length. We guarantee the offline computed bound on variability in execution time of each task by enforcing offline computed slot-level server budget reservations on each core, thereby limiting

[1]By variability in execution time of a task, we mean the variability introduced beyond the traditional single-core WCET due to the inter-core interferences in a multicore system.

inter-core interferences from co-executing tasks, as well as, additional inter-core interferences introduced by the task under consideration. In the offline phase, we propose to generate schedule table containing mapping, scheduling and server budget reservations, that bounds the variability in execution time of each task due to inter-core interferences, such that all tasks meet their deadlines. The computation of the bound on variability in execution time of each task involves estimating maximum inter-core interferences from already scheduled co-executing tasks in each slot, as well as, limiting the slot-level inter-core interferences that may be introduced by the task at hand, at runtime.

Overall, our proposed method considers a real COTS multicore platform - Freescale P4080 - and accounts for the delay introduced by arbitration and contention in the on-chip network and the memory sub-system. Further, we did a preliminary bare-metal implementation of our proposed runtime mechanism on the P4080 platform. Moreover, our method adheres to the TT architecture model [1] preserving system-wide properties like slot-level determinism, clock synchronization, etc., enabling integration of COTS multicores in safety-critical systems using a TT approach.

II. System Model, Task Model, and Job Model

A. System Model

We consider an abstract multicore hardware architecture inspired by the readily available real COTS multicore system - Freescale QorIQ P4080 platform [8]. We focus only on the hardware resources essential for task scheduling on the P4080 platform. These are 8 e500mc cores, the memory sub-system, and the crossbar CoreNet on-chip network. Along similar lines, we assume that the abstract multicore hardware comprises only two types of hardware resources: *N* homogeneous processing cores (including private caches) from *1,..,n,...,N*, and 1 shared resource consisting of a on-chip network with a memory sub-system. As this work is a first step, we only consider 1 memory controller in the memory sub-system, even though the P4080 platform has 2 memory controllers. Further, we assume that the hardware mechanisms like pre-fetchers, cache-coherency etc., that may implicitly introduce unaccounted inter-core interferences are disabled.

The inter-core interference latency, includes the time taken by a load/store request issued from a core to access the on-chip network and the memory sub-system, considering contentions. We consider the same latencies for our abstract multicore architecture as used in [2], [7]. The measurement-based approach as described in [9], using which these latencies are obtained, tries its best to create worst-case inter-core interference scenario, but is not guaranteed to do so, as the hardware model is not provided by Freescale. However, this is not a potential limitation of our proposed method in Section III as, when available, it will also work with inter-core interference latencies obtained through static analysis.

As shown in Table I, the latency of inter-core interference varies with the number of active processing cores partly due to varying arbitration delay from shared hardware resources. Each inter-core interference latency δ_j depends on the j number of active cores. E.g., as shown in Table I, if (say) $j = 3$ cores are active from time $[t, t + 1)$, we consider all inter-core interferences that occur during this time interval to have latency δ_3 (worst case) as listed in column 2 in Table I.

We consider a time-triggered (TT) scheduling approach and assume the timeline is divided into fixed equal length slices called slots [1]. A slot $S_{t,n}$ represents a time interval $[t, t+1)$, (where t is an integer multiple of slot length $|S|$) on core n. We also assume the system is preemptive at each slot boundary.

TABLE I: Inter-core interference latency and corresponding memory access server budget reservation for different number of active cores

No. of active cores (j)	Inter-core interference latency (δ_j in clock cycles)	Memory access server budget reservation in 1ms (Acc_j)
1	41	29385
2	164	7346
3	245	4917
4	463	2602
5	517	2330
6	737	1634
7	784	1536
8	1007	1196

B. Task Model and Job Model

The set Γ represents V hard real-time periodic tasks with arbitrary deadlines. Each task τ_i is characterized by the tuple $\langle C_i^s, MA_i, T_i, D_i \rangle$, where, C_i^s is the single core WCET excluding the time taken by memory accesses, MA_i is the maximum number of memory accesses to the shared resource, T_i is the period, and D_i is the relative deadline. C_i^s and MA_i are obtainable using a combination of static timing analysis tool like aiT and measurements [7]. Tasks may have precedence and communication constraints specified in graph $\mathcal{G}$.

In our proposed method (Section III), as we allow each instance of a task to have a different bound on variability in execution time, we convert the given task set to jobs, where each instance of a task is a job. The set $\mathcal{J}$ represents all jobs W of all tasks in a task set Γ in time $[0, H)$, where H is the hyperperiod of the task set Γ. Each job $\tau_{i,k}$ is characterized by the tuple $\langle C_{i,k}^s, MA_{i,k}, C_{i,k}^m, r_{i,k}, d_{i,k} \rangle$. $C_{i,k}^s$ and $MA_{i,k}$ are same across all jobs of task τ_i. $C_{i,k}^m$ is the multicore execution time i.e. the bound on variability in execution time of job $\tau_{i,k}$, computed offline, considering possible runtime inter-core interferences. $C_{i,k}^m$ may differ between different jobs of the same task τ_i. $r_{i,k}$ is the absolute release time and $d_{i,k}$ is the absolute deadline of job $J_{i,k}$, which are computed based on the related parameters of the corresponding task.

III. Proposed Slot-Level Based Method

In this section, we present our slot-level based method using a TT scheduling approach. The runtime mechanism is described in Section III-A and the offline phase in Section III-B

A. Runtime mechanism

We propose two server types - processing time server and memory access server, implemented using built-in hardware monitors. Each server type runs on each core and controls only one type of resource.

1) Processing time server: On each core n, a processing time server τ_{sp_n} regulates the execution time in each server period based on the slot-level server budget reservations computed in the offline phase. During runtime, an executing job at time t consumes the server budget reservation $Q_{sp_n,t}$ for the computation time on core n and stall time due to cache misses and/or memory accesses, resulting in a corresponding decrease of the server budget.

2) Memory access server: The memory access server τ_{sm_n} regulates the total number of memory accesses allowed from each processing core n in each slot S_t based on slot-level offline computed server budget reservations $Q_{sm_n,t}$, thereby controlling the inter-core interferences. At runtime, an executing job uses the server budget reservation only for memory accesses, resulting in a decrease of server budget by 1 for each memory access issued.

3) Runtime behaviour: During runtime, each core-level scheduler, at the start of each slot, assigns a job to the respective core and sets the corresponding server budget reservations for each server based on the schedule table obtained in the offline phase. On each core, if the budget of any server reaches 0, the corresponding core-level scheduler suspends the executing job, irrespective of the remaining budget of the other server. Jointly, the two servers on each core guarantee that the server budget reservations provided for each slot in the offline schedule table hold at runtime, thereby enabling bounding of variability in execution time for each job in the offline phase considering possible runtime inter-core interferences.

4) Inter-relationship amongst two servers, slots, and inter-core interference latencies: We consider the server period of each server is equal to the slot length $|S|$. For each processing time server instance, we allow only two mutually exclusive server budget reservation values: Either the budget reservation equals to zero which means an idle slot (no task is allowed to execute), or it equals to some fixed positive value X chosen by the system designer, such that $X \leq |S|$. For each memory access server instance, we allow $N+1$ mutually exclusive server budget reservation values. The N different budget reservation values directly associate with the different number of active cores. An additional budget reservation value of 0 relates to the idle slot, resulting in a total of $N+1$ possible budget reservation values. Based on the description of each server, the relationship between the server budget reservations of the two servers and the inter-core interference latency δ_j, for each active core n, is given by the formula $Q_{sm_n,t} = \left\lfloor \frac{Q_{sp_n,t}}{\delta_j(t)} \right\rfloor, \forall t$, where j represents the number of active cores at time t. E.g., we consider a slot length $|S|$ of 1ms and processing time server budget of 1ms. Table I then, shows the memory access server budget reservation values Acc_j (column 3) for j active processing cores (column 1).

5) Preliminary implementation: We did a preliminary bare-metal implementation of our proposed runtime mechanism running on all cores of the P4080 COTS multicore platform. We implemented the processing time server using the multicore programmable interrupt controller (MPIC) timer that enables slot-level synchronization amongst all cores. The MPIC timer also allows to set multiple processing cores as interrupt recipients, and provides each recipient core a unique interrupt copy [8]. We implemented the memory access server using a core-level hardware performance monitor that counts requests to the on-chip network [10]. We implemented our proposed suspension rules in interrupt service routines of the MPIC timer and hardware performance monitor on each core.

Though the idea to regulate memory accesses from each core using memory access server may seem similar to MemGuard [5], there are three key differences. Firstly, MemGuard considers minimum guaranteed memory bandwidth as constant, whereas our proposed method considers it as variable depending on the number of active cores (see Table I). Secondly, MemGuard assumes the memory server budget reservations for each core as given and constant across all server periods, whereas we do not make such an assumption. Thirdly, MemGuard does not consider if the given server budgets meet task deadlines, whereas our proposed method (introduced later in the Section III-B) gives offline guarantees.

B. Offline phase: Bounding variability in execution time

In the offline phase, we bound the variability in execution time of each job by computing server budget reservations for each slot, such that all tasks meet their deadlines. This limits, at runtime, the maximum inter-core interferences from co-executing jobs as well as the additional inter-core interferences introduced by the job under consideration.

In the offline phase, let's consider at slot S_t on core n, the offline scheduler tries to schedule job $\tau_{i,k}$, such that the job $\tau_{i,k}$ meets its deadline without affecting the already scheduled jobs. In order to compute the multicore execution time $C^m_{i,k}$, the offline scheduler first tries the simple case, where it considers a job $\tau_{i,k}$ executes in each slot on some core n with constant memory access server budget reservation Acc_j. In this simple case, we compute the multicore execution time of job $\tau_{i,k}$ using the formula $C^m_{i,k} = \left\lceil C^s_{i,k} + \frac{MA_{i,k}}{Acc_j} \right\rceil$ based on some memory access server budget reservation Acc_j chosen by the offline scheduler (slot length S of 1ms).

However, it is possible that the offline scheduler is unable to find and reserve enough slots for the job $\tau_{i,k}$ that fulfill chosen memory access server budget reservations Acc_j on core n until its deadline. In that case, we propose a different way to compute the multicore execution time.

Let's consider on core n, in the time $[t, t+z+1)$, due to already scheduled jobs on remaining cores, the offline scheduler only finds slots with different memory access server budget reservations in time $[t, t+z+1)$. Then (say) from slot S_{t+z+1}, the scheduler finds slots with constant memory access server budget reservations $Acc_{j'}$. In this case, we propose to first determine the minimum progress the job $\tau_{i,k}$ can make in time $[t, t+z+1)$ (both computation time and memory accesses). Then, we subtract the same while computing the remaining multicore execution time $c^m_{i,k}(t+z+1)$ based on new budget reservation $Acc_{j'}$ from time $t+z+1$ onwards. The offline scheduler then reserves the slots in time $[t, t+z+1)$

with the available server budget reservations, and from time $t+z+1$ to $t+z+c^m_{i,k}(t+z+1)$ with Acc_j. If the job $\tau_{i,k}$ still cannot meet its deadline, the offline scheduler will try to reschedule some already scheduled jobs, e.g., by backtracking.

IV. Example

Figure 1 shows an example of our proposed method considering only 2 cores due to the space constraints. Each core n has two servers: processing time server τ_{sp_n} and memory access server τ_{sm_n}, with server period equal to the slot length $|S|$ of 1ms. For each server, the dotted horizontal lines depict the possible server budget reservation values. During runtime, at the start of each slot, each core-level scheduler assigns a job to the respective core and sets the corresponding server budgets for each server, based on the offline schedule table.

At time $t = 0$, as both the cores are active, each core-level scheduler sets the corresponding processing time server budget reservation to 1ms and memory access server budget reservation to $Acc_2 = 7346$ accesses (based on Table I). At time $t =$ 1ms, only job $\tau_{0,0}$ is active resulting in memory access server budget $Q_{sm_1,1} = Acc_1 = 29385$ accesses (based on Table I) and processing time server budget of 1ms. In the time interval $[1, 1.33)$ms, the job $\tau_{0,0}$ issues memory access as shown by corresponding decrease in memory access server budget. Then in the time interval $[1.33, 2)$ms, it does not perform any memory accesses as shown by memory access server budget being constant. Later, it again briefly issues memory accesses for the next 100μs. In the time interval $[1.6, 2)$ms, since, only the processing time server budget decreases and not the memory access server budget, the job $\tau_{0,0}$ only performs computations. At time $t =$ 3ms, the job $\tau_{1,0}$ completes execution and the scheduler of core 2 discards the unused memory access server budget from the previous server instance $\tau_{sm_2,2}$. Further, at time $t =$ 3ms, as only job $\tau_{0,0}$ is active, the memory access server budget $Q_{sm_1,3}$ equals Acc_1. The job $\tau_{0,0}$ issues memory accesses in the first half of the slot as shown by decrease in memory access server budget. Then, for the next 200μs, it does not issue any memory accesses as the memory access server budget does not decrease and at time $t = 3.7$ms completes execution, resulting in discarding of unused server budgets by the core-level scheduler.

V. Conclusion and Future Work

In this work, we presented an initial step towards enabling time-triggered (TT) scheduling on a real COTS multicore platform P4080. It takes into account inter-core interferences in the on-chip network and the memory sub-system. Our proposed method comprises a runtime mechanism and an offline phase. For the runtime mechanism, we proposed a processing time server and a memory access server for each core. Jointly, the two servers on each core, enforce slot-level offline computed server budget reservations, thereby limiting the maximum inter-core interferences introduced and experienced by each task considering different inter-core interference latencies. In the offline phase, we proposed a procedure for the offline scheduler to compute the bound on variability in execution

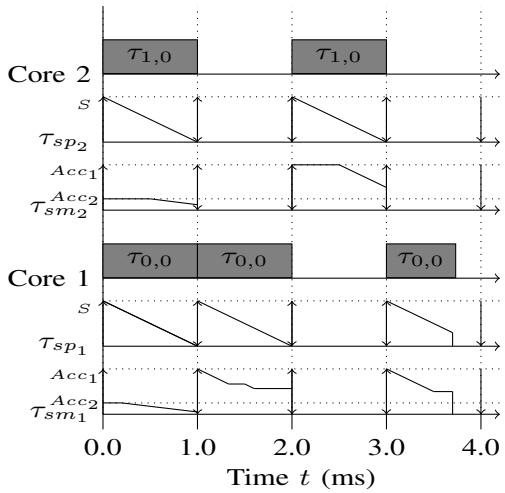

Fig. 1: Example of our proposed slot-level based method

time of each task while allowing different slot-level memory access server budget reservations. Overall, our proposed method facilitates integration of COTS multicore platforms in TT systems, while maintaining features of TT architecture like slot-level determinism, clock synchronization, etc.

We did a preliminary bare-metal implementation of our proposed runtime mechanism on a real COTS multicore platform P4080. In future work, we aim to provide safe bounds for the variability in execution time and will integrate the procedure in our existing offline scheduler to generate schedule tables containing mapping, schedule and server budget reservations.

Acknowledgment

The work was supported by ARTEMIS project 621429 EMC2. We thank the referees for several useful comments.

References

[1] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, 2nd ed. Springer-Verlag, 2011.

[2] J. Nowotsch, M. Paulitsch, D. Buhler, H. Theiling, S. Wegener, and M. Schmidt, "Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement," in *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, July 2014, pp. 109–118.

[3] *CAST-32 Multi-core Processors*. Certification Authorities Software Team, May 2014.

[4] 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*, July 2012, pp. 299–308.

[5] ——, "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*, April 2013, pp. 55–64.

[6] G. Yao, H. Yun, Z. P. Wu, R. Pellizzoni, M. Caccamo, and L. Sha, "Schedulability analysis for memory bandwidth regulated multicore real-time systems," *Computers, IEEE Transactions on*, vol. 65, no. 2, pp. 601–614, Feb 2016.

[7] J. Nowotsch and M. Paulitsch, "Quality of service capabilities for hard real-time applications on multi-core processors," in *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, ser. RTNS '13. New York, NY, USA: ACM, 2013, pp. 151–160.

[8] *P4080 QorIQ Integrated Multicore Communication Processor Family Reference Manual Rev. 2*, Freescale Semiconductor, 2014.

[9] J. Nowotsch and M. Paulitsch, "Leveraging multi-core computing architectures in avionics," in *Dependable Computing Conference (EDCC), 2012 Ninth European*, May 2012, pp. 132–143.

[10] *e500mc Core Reference Manual Rev. 3*, Freescale Semiconductor, 2013.