CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Conference Paper

Towards realistic core-failure-resilient scheduling and analysis

Borislav Nikolic
Konstantinos Bletsas

CISTER-TR-151203

Towards realistic core-failure-resilient scheduling and analysis

Borislav Nikolic, Konstantinos Bletsas

CISTER Research Center

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail:

http://www.cister.isep.ipp.pt

Abstract

Towards realistic core-failure-resilient scheduling and analysis.

Borislav Nikolić and Konstantinos Bletsas
CISTER/INESC-TEC Research Centre, ISEP/IPP, Porto, Portugal

I. Introduction and existing work

On uniprocessors, a failure of the single core means unavoidable system failure. However, on multicores, when a core fails, it is conceivable that the computation could continue on remaining cores in a degraded system mode indefinitely, until orderly shutdown and servicing can take place. This would be very desirable for critical applications but, apart from hardware and software support, it would require (i) a scheduling approach designed for providing such resilience and (ii) accompanying schedulability analysis, that derives offline the guarantees about the system meeting its deadlines at run-time, even if one core fails.

We studied this problem in [5], for constrained deadlines[1] and global fixed task prorities, and assuming independent tasks sharing no resources. We focused on the provision of scheduling guarantees for the case of at most one core failing. Under our assumptions there, such a core failure is detected immediately. Of all task instances present in the system at the time, only the job currently executing on the failed core is killed; all other jobs are unaffected, because they run on intact processors or else because their state is assumed to reside entirely in memory, which is trusted not to fail. However, all job deadlines must be met, *including* the one of the job killed by the core fault; this necessitates some form of redundancy. Additionally, if the failure is permanent, this leaves one core less in degraded mode, but the system need not survive additional core failures because we assume that it will be shut down for servicing at the first opportunity. Although schedulability under multiple failed cores might appear as a requirement for the certifiability of a critical system, in practice the certification is done based on probabilities of failure. Thus, if the worst-case delay until servicing is at most Δh hours and the probability of a second core failing within those Δh hours is sufficiently small, then the system could still be certifiable.

Faced with this scheduling problem, two simple ways of dealing with it first came to mind, albeit both with obvious shortcomings:

Simple approach 1: Run *two* copies of each job instead of one, so that at least one can always complete even if one core fails; and if no core fails, upon completion of one job copy, the other one is terminated early. This is inefficient, as it doubles the processing capacity requirements, so no task set with utilisation above 50% would be schedulable.

Simple approach 2: When a core fails, launch from scratch the killed job. This only increases processing requirements marginally and transitively, when the core fails, but it cannot provide resilience in the general case, e.g., if the WCET of the killed and restarted job exceeds the time left until the deadline.

Inspired by both of the above approaches and elaborating on them, we therefore proposed [5] a mechanism that generalises both of them, trading off their shortcomings in a tunable manner.

In particular, our **proposed solution** involves launching a copy of every job by a task τ_i, not immediately but instead after a fixed, designer-set and task-specific offset O_i from its arrival (see Figure 1). Both jobs have the priority of the parent task, but, for tie-breaking purposes, the main job released immediately has higher priority that its copy job released at an offset. The selection of the offset for the copy job is a trade-off. A small value for O_i increases the amount of redundant execution for the task but decreases the amount by which the effective *relative* deadline of the copy job ($D_i - O_i$) is shortened, giving it more time within which to complete, in case of core failure. On the other hand, if too big a value for O_i is chosen, it might be impossible to provide guarantees for timely completion of the copy job in case the main job fails too close to the deadlne. Therefore, the optimal value for O_i is the biggest value in the range $[0,\ R_i^{\varnothing}]$ for which schedulability of the copy job is ensured. The term $R_i^{\varnothing}$ denotes an upper bound on the WCRT of the main job by τ_i, assuming that no core has failed by the time that it completes. And, for any $O_i \in [0,\ R_i^{\varnothing}]$, the amount of additional task execution per job pair in case of no failure (compared to the conventional single-job scheduling), is upper-bounded by $C_i' \stackrel{\text{def}}{=} \min(C_i, R_i^{\varnothing} - O_i)$. We term this quantity "overlap".

In [5], we detailed how to perform schedulability testing (and also compute the optimal O_i in the process) for each task, in order of decreasing priority. The schedulability must be proven for three cases: (i) no core failing, (ii) a core failing and killing a higher-priority job and (iii) a core failing and killing a job by the task in consideration. The schedulability test that we use is the state-of-the-art one by Guan et al. [4], for conventional global fixed-priority scheduling. We apply it to a transformed task set instance, with each task τ_i modelled as two separate tasks with execution times of C_i and C_i'; we also discount the number of cores by one, for irrecoverable core failures and add a term which bounds the additional transitive workload upon the core failure. These are all pessimistic

[1]Each task τ_i has a worst-case execution time (WCET) of C_i, a relative deadline D_i and an interarrival time T_i, with $D_i \leq T_i$.

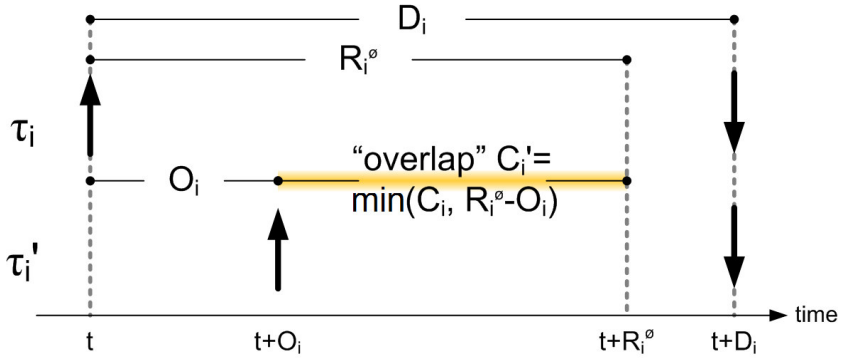

Fig. 1: Illustration of our proposed approach for scheduling with resilience to core failures. For each job, a copy job is released at an offset O_i from each arrival. $R_i^{\varnothing}$ is an upper bound on the WCRT of the main job under consideration under the assumption that no core has failed by the time that it completes.

transformations.

However, overall, the system model is characterised by several unrealistic assumptions which limit its potential practicality. This work focuses on remedying this situation. Next, we will identify and separately discuss each of those aspects and how they can be potentially dealt with.

II. Unrealistic assumptions to be addressed

In this section we list the most important practical limitations of our approach, and how we intend to address them.

A. Resource sharing by different tasks

Every non-trivial real-world system with more than one computing task has shared resources, which need to be managed with care, in order to avoid synchronisation hazards. Therefore, developers place such resources inside *critical sections*, guarded by *semaphores* or *locks* and implement one of the many existing (uniprocessor or multiprocessor) resource management protocols, to be found in the literature.

In the absence of core failures, even with the job-pair arrangement proposed earlier in place, it would not be difficult, for any of the established resource management protocols, to introduce worst-case blocking terms to our WCRT equations. After all, our equations are largely derived from those by Guan et al. [4]) for standard global scheduling. However, special considerations need to be made for the case of a core failure.

First of all, multiprocessor resource management schemes that treat the available cores asymetrically, by designating one core as responsible for executing all the critical sections or which, more generally, assign the execution of one critical section to a particular core, would have to be excluded from consideration. This is because any core is equally likely to fail and such an arrangement would introduce single points of failure, which the system could not survive. Since it is a requirement to survive a single failure *by any one core*, our approach would therefore need to be coupled with a resource management scheme that uses processors interchangeably. Various protocols applicable to globally-scheduled systems have been formulated (e.g., [1], [2]). However, we are looking into adapting MrsP [3] (originally formulated for partitioned systems) to work with global scheduling and eliminating the distinction between local and global shared resources. The reason is that MrsP has the advantage of relatively simple blocking terms, that evoke those of uniprocessor scheduling. This would hopefully prevent our analysis from becoming too complex. Under MrsP, at run-time, tasks which would otherwise have been blocked may instead act as "servers" for other tasks, and this may lead to complex chains of task interactions. However, we are confident that this would not break the fundamental properties of our job-pair arrangement (nor vice versa), as long as cores were, somehow, guaranteed to never fail *during* the execution of a critical section.

However, in the general case, cores may fail (and jobs may die) even during the execution of a critical section. How the protocol can cope with this (and what this does to the corresponding worst-case blocking terms) is still an open question, which can only be answered if the semantics of failing while executing a critical section are defined.

In particular, to avoid ill-defined and inconsistent states, we believe that resilience to core failures under our approach effectively necessitates *transaction* semantics for shared resource accesses, with all changes in state being finalised with a *commit*, at the very end of each critical section. This would allow partial state updates to be rolled back, in case of core failure. We intend to look at current engineering practice in order to (i) propose appropriate implementation mechanisms for the provision of such semantics and (ii) include the overheads of these mechanisms into our schedulability analysis. Additionally, the developer would most likely need to be aware of those mechanisms and explicitly use the respective APIs.

Even so, this does not solve all challenges related to the sharing of resources by different tasks because the question arises: what are the implications of both the main job and the corresponding copy job of a given task both accessing (at different points in time) a given resource, shared with another task? Depending on the actual application, this might be benign (e.g., updating the stored value of a sensor reading)

or problematic (e.g., triggering twice a system event meant to be triggered only once per task arrival). Ultimately, such issues are best dealt on the application level, but once again the developer needs to be aware of our scheduling arrangement with job copies, at development time.

B. Race conditions and synchronisation hazards arising from the coexistence of jobs by the same task

In addition to explicitly shared resources, by *different tasks*, our proposed scheduling approach must handle another challenge: When two job copies by *the same task* coexist in the system, they may both access resources for which the programmer never expected any concurrent access by different processes, if the coding was oblivious to our scheduling arrangement that uses job copies for resilience. We propose various alternative approaches for dealing with this scenario, each suited to different circumstances:

First, an obvious approach would be to enforce, as a programming convention, to the extent possible, that each job should read all of its input upon release. Thenceforth, it should operate exclusively on its dedicated process variables and only write its output prior to completion. Depending on the semantics of the application, and whether or not the main job and the corresponding copy job of a given task would need to operate on the same input as each other or not, some buffering mechanism might be required so that the copy job reads the same input values as the main job did, even if their sources have changed in the meantime (e.g., as in the case of sensor readings). This approach is simple but it might be too restrictive to accommodate all applications; it would also require the developer to be aware of this convention and program the applications accordingly.

A second alternative, would be to enclose in critical sections those resources that may entail a synchronisation hazard, upon access by both the main job and the corresponding copy job of the same task. This would allow them to be handled analogously to explicitly shared resources (e.g., by different tasks); and our analysis already models the main and copy jobs of a given task as originating from different tasks anyway. This approach would piggyback on our solution for regular resource sharing (by different tasks) discussed in the earlier subsection. But it would also require the cognisance of the programmer at development time and might involve too many resources being encapsulated in critical sections.

Finally, we propose a third alternative, which does not rely on the awareness of the developer, but might not work in all cases. It involves reasoning offline about the location of resource accesses inside the code and the time intervals during which these might take place at run-time, in order to ensure that no synchronisation hazard may occur at run-time, given the offset O_i for the release of the copy job. For the general case, this approach would be computationally complex and would require knowledge of task structure, new assorted analysis and tool support. However in many cases, it could be easy to bypass all those steps by ensuring that the overlap C'_i of the task τ_i in consideration is zero, when possible to do so. This would ensure that the main job and corresponding copy job of a given task never co-exist in the system (i.e., the copy job would effectively only be launched if the main job is killed). The overlap is defined as $C'_i \stackrel{\text{def}}{=} \min(C_i, R_i^{\varnothing} - O_i)$. This means that it depends on the $R_i^{\varnothing}$ and the corresponding optimal O_i, both of which in turn depend on the priority of τ_i, which is set by the designer. Hence setting the corresponding task priority appropriately high may in many cases suffice to ensure zero overlap. This is akin to falling back to the second "simple approach" discussed earlier, but without any jitter in the release of the copy task. The drawback is that this will involve the priority demotion of some other task(s), all other things remaining equal, which may be detrimental to their schedulability.

C. Run-time support and incorporation of related overheads into the schedulability analysis

Our assumptions included that of instantaneous detection of core failures. In practice though, even with hardware support, detecting core failures will still require some degree of software support. Therefore, there will be some latency and associated scheduling overheads. These must be incorporated into the the schedulability analysis. In order to do so, one must have knowledge of exactly how the failure detection facility is implemented; and this would largely be platform-specific. We are therefore looking into how this facility could, in principle, be practically implemented upon popular multicore and manycore platforms for embedded systems.

Additionally, a separate software facility is required for (i) tracking the arrivals of the main jobs, (ii) launching the copy jobs at the appropriate offsets and (iii) keeping track of job completions in order to immediately also terminate (short of completion) the corresponding job copies. (Note that, e.g., in case of different control flows due to some source of non-determinism at run-time, an overlapping copy job might complete before its corresponding main job, and then it is the main job that should be terminated.). The corresponding system overheads of this mechanism must also be accounted for in the analysis. We plan to do this in a similar detailed manner as in our other recent work [6].

III. Concluding remarks

It is an established pattern, in the real-time systems literature, that initial simple but abstracted analytical models incrementally give way to more accurate, but also more complicated models, intended to capture the real-world effects. We are also going down that path with our line of work on providing hard real-time scheduling resilience in the case of failing processor cores. The list of simplified assumptions and aspects to be addressed is currently significant, especially with respect to resource sharing and the handling of potential synchronisation hazards. However, as explained, we already have specific approaches in mind, that we are working on. We would like to thank the reviewers of ECRTS 2015 and RTCSA 2015, for encouraging us to work on these aspects of our work and we hope to deliver useful results in the near future.

ACKNOWLEDGEMENTS

Work partially supported by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) and co-financed by ERDF (European Regional Development Fund) under the PT2020 Partnership, within project UID/CEC/04234/2013 (CISTER Research Centre); also by FCT/MEC and the EU ARTEMIS JU within projects ARTEMIS/0001/2013 - JU grant nr. 621429 (EMC2); by FCT/MEC and the ESF (European Social Fund) through POPH (Portuguese Human Potential Operational Program), under PhD grant SFRH/BD/81087/2011.

REFERENCES

[1] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proc. RTCSA*, page 4756, 2007.

[2] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Proc. RTSS*, page 4960, 2010.

[3] A. Burns and A. J. Wellings. A schedulability compatible multiprocessor resource sharing protocol - MrsP. In *Proc. ECRTS*, pages 282–291, 2013.

[4] N. Guan, M. Stigge, W. Yi, and G. Yu. New response time bounds for fixed priority multiprocessor scheduling. In *Proc. 30th RTSS*, 2009.

[5] B. Nikolić, K. Bletsas, and S. M. Petters. Hard real-time multiprocessor scheduling resilient to core failures. In *Proc. RTCSA*, 2015.

[6] P. F. Souto, P. B. Sousa, R. I. Davis, K. Bletsas, and E. Tovar. Overhead-aware schedulability evaluation of semi-partitioned real-time schedulers. In *Proc. RTCSA*, 2015.