Design of a Mixed-Criticality Scheduler with improved QoS on a Multicore Platform

IYER RAJKUMAR VARSHA (BITS Pilani, KK Birla Goa Campus) – 29 November 2020.

Report for CS F376: Design Project.

ABSTRACT

The shift from a single level of criticality to multiple levels of criticality in the design of real-time systems and embedded systems has gained traction and has become a progressively significant trend. The question of how to incorporate multiple criticalities and assure against failure, in both the case of single-core and the more difficult case of multi-core systems, has been increasingly considered. However, with multiple criticalities, comes a decrease in the Quality of Service (QoS) when lower-criticality jobs are discarded from the system in order to assure against the failure of higher-criticality jobs missing their deadlines in an overly pessimistic manner. This report details an engineering design project focusing on the design of a scheduler for multiple criticality levels, with improved QoS, extended for multiple cores. A multi-criticality, multi-core, extended version of an algorithm called EDF-VD (Earliest Deadline First with Virtual Deadlines) is presented, which has been extended to include lower-criticality discarded jobs when possible in the event of job departures and system criticality changes. Finally, one implementation of this algorithm is presented, which uses threads to simulate different cores.

INTRODUCTION

Introduction

Unlike earlier real-time and embedded systems which only had one criticality level in most cases, mixed-criticality systems have the ability to accommodate tasks of different criticalities. A mixed-criticality system is one which has at least two or more well-defined and recognisable levels of criticality. Each task of the system has a worst-case execution time (WCET) for each distinct criticality level that the system can potentially go to – the WCETs of tasks are dependent on their criticality level.

Quality of Service (QoS), meaning the overall performance of the system, is the term used to judge the overall system on CPU utilization, resource utilization, scheduler efficiency, number of jobs that met deadlines etc. For the purpose of reducing pessimism in real-time scheduling, while also guaranteeing the finishing of critical tasks before deadline in the event of an unexpected overrun, mixed-criticality systems are an accepted and popularly used model. However, quite often, no QoS is considered in the treatment of non-critical (lower criticality) tasks (Hikmet et al. [1],). Even a single high-criticality job overrun can cause all high-criticality tasks to switch to higher criticality modes, with an increasingly pessimistic WCET, which consequently leads to inefficient resource utilization (Huang et al. [2],). In the event of a criticality change, all lower-criticality tasks are often dropped, even if accommodating them was actually possible. This causes a heavy degradation of QoS. Also, in mixed-criticality systems, improving schedulability may lead to the reduction of QoS as well (Chai et al. [3],).

Therefore, this paper introduces a scheduler with an improved QoS, by attempting to fit in/accommodate discarded jobs whenever it will not cause a higher-criticality job to overrun and miss its deadline.

Aim and motivation

The aim to undertake this project was to get more insight into the field of mixed criticality systems, and to try to implement a scheduler with improved quality of service, so that the scheduler could try to utilise as much of the processor(s) as possible.

This project was undertaken with the realisation that the processor(s) could be severely under-utilized in the event of criticality changes, due to the fact that the overly pessimistic WCETs being considered during such an event as well as the discarding of lower-criticality jobs severely brings down the QoS. Considering a mixed-criticality system in a multicore platform, while also trying to accommodate discarded jobs with minimal context switching but including minimal job migration when needed, seemed like a challenging and rewarding task.

RELEVANT RELATED WORKS

The relevant works listed here, have used an EDF-VD uniprocessor multi-level algorithm (Baruah et al. [4],), a slack-scheduling dual-criticality multicore algorithm (Niz et al. [5],), and a Criticality-based EDF uniprocessor dual-criticality algorithm (Park et al. [6],) respectively.

An Earliest-Deadline First – Virtual Deadlines (EDF-VD) algorithm which can schedule systems with any number of distinct criticality levels, was considered by Baruah et al. [4], for a single pre-emptive processor. The algorithm EDF-VD has also been analysed in the paper based on utilization bounds and other metrics, for up to thirteen criticality levels. However, in this approach, the same reduction/scaling factor “x-factor” is used for all tasks to determine virtual deadlines. Results are derived in the paper both for implicit-deadline systems (where each task τ_i satisfies $d_i = p_i$, i.e. deadline is equal to period) as well as the general arbitrary-deadline systems.

Niz et al. [5], was the first one to introduce an approach for dual-criticality systems where a slack scheduling scheme is used, where low-criticality jobs are run in the slack produced by high-criticality jobs. The execution budget of low-criticality jobs was used for this. However, in this paper, the execution of a low-criticality (higher priority) task beyond its deadline could cause the missed deadline of a higher criticality (lower priority) task, which raises questions about the certification of the system and assurance against failure (demonstrated by Huang et al. [7],). This was a huge flaw, since a system should always guarantee that high-criticality tasks do not miss their deadlines. For that reason, Niz et al. [5], modified their model to remove this issue in subsequent papers.

A scheduling algorithm called CBEDF (Criticality Based Earliest Deadline First) was introduced by Park et al. [6], for certifiable uniprocessor systems with multiple levels of criticality. There are two possible types of slack for low-criticality jobs in the event where the worst-case execution time units ($C_i(HI)$) is reserved by all high-criticality jobs in their window of scheduling – the remaining slack (if a high criticality job finishes earlier than its deadline), and empty slack (any available units of time after all high-criticality jobs reserve $C_i(HI)$). Both of these slacks are attempted to be used for low-criticality jobs in an effort to improve utilization, and as a result, QoS. A sufficient schedulability test is also proposed for this algorithm.

PROPOSED WORK

Basic Idea

The proposed work is a mixed criticality (*n levels*, up to fourteen levels), multicore scheduler with slack stealing using discarded job accommodation. (The implementation given in the appendix is for an implicit-deadline system, with phase of zero units for all tasks.) It uses the single scaling factor concept from Baruah et al. [4]. If there is a change in criticality level, it runs low criticality level jobs (jobs with criticality $<=k$), which have been discarded due to criticality change, in the slack generated by high criticality level jobs (jobs with criticality $>k$). It includes a global queue of discarded jobs, and tries to fit in as many discarded jobs as possible in all the cores (giving preference to the core which the job was discarded from, to reduce the possibility of context-switching as much as possible), and only removes the discarded job permanently from the queue if slack for the job in all the cores is zero.

Model Used

A multicore system with N cores $(N \in \mathbb{N}+)$ and K-levels $(K \in \mathbb{N}+)$, has been used, with a sporadic taskset of different criticalities. The taskset consists of a collection of tasks τ_{ni} $(i \in \mathbb{N}+, n \in [N])$, where n is the core the task is statically bound to. Each core n has the tasks τ_{n1}, τ_{n2} , … τ_{n_m}, where there are n_m tasks for core n, and the jobs of these tasks arrive in the core n itself. The tasks are characterised by different parameters – a criticality level $\chi_{ni} \in [K]$, and a tuple $(c_{ni}, d_{ni}, p_{ni})) \in Q\chi_{ni}+ \times Q+ \times Q+$ (a tuple of WCETs, relative deadlines and period), where:

- $c_{ni} = (c_{ni}(1), c_{ni}(2) \; .... c_{ni}(\chi_{ni}))$ is a vector of WCETs, one for each criticality level from 1 to χ_{ni}. The assumption is that $c_{ni}(1) \leq c_{ni}(2) \leq ..... \leq c_{ni}(\chi_{ni})$.
- d_{ni} is the relative deadline of the task τ_{ni}.
- p_{ni} is period (more accurately, minimum interarrival time) between two jobs of task τ_{ni}.

A potentially unbounded sequence of jobs $(J_{ni1}, J_{ni2} ....)$, is generated by the task τ_{ni}, where the parameters of the jobs are related to parameters of the tasks. They are statically bounded to the same core n as the task, and so arrive in that core. A job J_{nij} of task τ_{ni} has the parameters $J_{nij} = (a_{nij}, \gamma_{nij})$, where:

$a_{nij} \in \mathbb{R}+$ is the job's time of arrival, and $\gamma_{nij} \in (0, c_{ni}(\chi_{ni})]$ is the job's execution requirement.

The absolute deadline of the job J_{nij} will be $a_{nij} + d_{ni}$.

Related Semantics

The utilization of task τ_{ni} at level k is: $u_{ni}(k) = \frac{c_{ni}(k)}{p_{ni}(k)}$

The total utilization at level k of tasks that are of criticality level l is $U_l(k) = \sum_{i \in n_m, \chi_{n_i}=l} u_{ni}(k)$.

Algorithm

The algorithm has two parts – offline schedulable pre-processing, and the scheduling at runtime. The offline pre-processing calculates whether the system is schedulable, and if it is, the x-factor and the value of k needed for the virtual deadlines and criticality change. The runtime scheduling includes a criticality change algorithm as well as a discarded-job accommodation algorithm in addition to the runtime process.

Offline Schedulable Pre-processing

The offline schedulable pre-processing in this paper is an extended multi-core version of Baruah et al. [4], in which only a uniprocessor was considered.

```
1. The input taskset is taken in for $n$ cores parallelly (, with the tasks statically allotted to a core. Each core has a taskset $\tau_n = (\tau_{n_1}, \tau_{n_2}, \ldots, \tau_{n_m})$, where $n_m$ is the number of tasks statically bounded to that core $n$, $n \in [N]$ where $N$ is the total number of cores.
2. For each core $n$, the steps from 3 to 21 is done parallelly.
3. if $\sum_{l=1}^{K} U_l(l) \leq 1, k \leftarrow K$, then:
4. for $i=1, 2 \ldots. n_m$ do
5. Virtual deadline of task $\tau_{n_i}$, $\widehat{d_{n_i}} \leftarrow d_{n_i}$ (actual deadline)
6. end for
7. else
8. Let $k_n$ $(1 \leq k_n \leq K)$ be such that the condition $1 - \sum_{l=1}^{k_n} U_l(l) > 0$ and $\frac{\sum_{l=k_n+1}^{K} U_l(k_n)}{1 - \sum_{l=1}^{k_n} U_l(l)} \leq \frac{1 - \sum_{l=k_n+1}^{K} U_l(l)}{\sum_{l=1}^{k_n} U_l(l)}$ holds.
9. if no such $k_n$ exists, then
10. return entire system as unschedulable according to the input
11. else
12. Let $x_n \in [\frac{\sum_{l=k_n+1}^{K} U_l(k_n)}{1 - \sum_{l=1}^{k_n} U_l(l)}, \frac{1 - \sum_{l=k_n+1}^{K} U_l(l)}{\sum_{l=1}^{k_n} U_l(l)}$(this is the reduction/scaling factor) be the x-factor of each core $n$
13. for $i=1,2 \ldots. n_m$ do
14. if $\chi_{ni} \leq k_n$ then
15. Virtual deadline of task $\tau_{n_i}$, $\widehat{d_{n_i}} \leftarrow d_{n_i}$ (actual deadline)
16. else
17. Virtual deadline of task $\tau_{n_i}$, $\widehat{d_{n_i}} \leftarrow x_n * d_{n_i}$ (actual deadline)
18. end if
```

```
19. end for
20. end if
21. end if
22. If no core returns as unschedulable on step 10, then return (schedulable, vector of $k_n$ where $n \in [N]$, and updated virtual deadlines $\widehat{d_{n_i}}$ from $i$=1 to $n_m$ and for all $n \in [N]$).
23. Find the hyper-period of all the tasks together.
```

Runtime Scheduling

The function current_level() returns the level of the core by the scenario exhibited so far.

Runtime

There are n cores where tasks statically allotted to a core. Each core has a taskset $\tau_n = (\tau_{n_1}, \tau_{n_2}, \dots, \tau_{n_m})$, where n_m is the number of tasks statically bounded to that core n, $n \in [N]$ where N is the total number of cores.

```
1. loop
2. loop: while time is lesser than hyper-period:
3. for each core n, $n \in [N]$ , do the steps 4 to __ parallelly.
4. if it is time for decision point in that core (the three decision points in a core are job arrival, job departure and criticality change):
5.  if core n's criticality is lesser than other cores, then cause forced criticality change by causing it to be the time for a criticality change (step 11 will cause the actual change)
6. end if
7. if it is time for a job arrival, add to the runtime queue of core n, and find out next closest arrival time in that core, and also update departure time of currently running job in the core
8. end if
9. if it is time for a job departure, cause the job departure, and update next closest departure time. Try to accommodate a discarded job in the core n by calling *discarded_job_accomodation().*
10. end if
11. If it is time for a criticality change, increase the core's level by one, and cause criticality change by calling *criticality_change_function().* Update next closest departure time. Try to accommodate a job by calling *discarded_job_accomodation().*
12. end if
13. print the running job in core $n$ (it is the job with minimum virtual deadline in the core's ready queue)
14. end if
15. Increase time by one time granularity
16. end loop
```

criticality_change_function():

This function is called by a core in step 11 of runtime, when it is time for a criticality change. A forced criticality change in step 5 of runtime also leads to this.

```
1. if the core's level is (1…$k_n$), then:
```

```
2. discard all jobs in core n's runtime queue of criticality < core's criticality level. The discarded jobs are added to the global discarded job queue.
3. future arrivals of the jobs from the tasks with criticality < core's level can also be added to the discarded job queue.
4. The rest of the jobs and tasks run with no change in virtual deadlines.
5. else, the jobs in core n's runtime queue with criticality <=k are discarded and added to the global discarded job queue
6. future arrivals of jobs from the tasks with criticality < core's level can also be added to the discarded job queue
7. Original deadlines of all other tasks and jobs are restored. Virtual deadlines now equal to the original actual deadlines.
8. end if
9. exit the function
```

discarded_job_accomodation():

This function is used to attempt to accommodate as many discarded jobs as possible from the global discarded queue. It is called either by a core which has changed criticalities, or by one which just underwent a job departure. The fitting in of a discarded job in a core is done by calculating the slack and checking if it is greater than the WCET of the discarded job, done by the function *slack_calculation()*.

```
1. Loop: while the global discarded queue is not empty:
2. Choose discarded job as the one with the highest criticality (if equal, the tie is broken by the one with the lowest deadline)
3. Try to fit it in the core which it is statically bound to, in order to reduce context switching. This core is the preferred core for this job.
4. if yes (was able to fit):
5. add it to ready queue of the preferred core, remove it from the global discarded queue. Continue the loop by going to step 1.
6. else (was unable to fit it in the preferred core):
7. attempt to fit it in the core which called the function discarded_job_accomodation().
8. if yes (was able to fit in the core which called the function):
9. add it to ready queue of the core which called the function, remove it from the global discarded queue. Continue the loop by going to step 1
10. else
11. try all the remaining cores while keeping a track of how much slack was available for the discarded job in each core
12. if any core was able to fit the discarded job:
13. add it to ready queue of the core which was able to fit it, and remove it from the global discarded queue. Continue the loop by going to step 1
14. else:
15. if all cores have zero slack for the discarded job:
16. it is removed from the global discarded job queue forever. Continue the while loop by going to step 1.
17. else:
```

```
18. it can potentially be accommodated in the future. Add it to temporary discarded queue, and remove it from the global discarded queue. Continue while loop by going to step 1
19. end if
20. end if
21. end if
22. end if
23. end loop
24. Add jobs from the temporary discarded queue in step 18, if any, to the global discarded queue.
25. exit the function
```

slack_calculation():

This function is used to calculate the slack for the accommodation of jobs in the *discarded_job_accomodation()* function. Basically, a window is used, which lies between current time and the maximum of all deadlines of jobs arriving between current time and the deadline of the discarded job being considered.

```
1. if the discarded job's deadline >= all the jobs arriving between current time and discarded job deadline and all ready queue jobs of the core:
2. The slack used will be the sum of all the WCETs (of the core level, or the highest WCET in the event where it is a discarded job's WCET being considered from the ready queue) of all the jobs considered in step 1
3. return slack
4. else: (there are jobs being considered with deadline higher than the discarded job):
5. loop: considering each job one by one:
6. if the job being considered has deadline <= discarded job deadline:
7. the WCET (of the core level, or the highest WCET in the event where it is a discarded job's WCET being considered from the ready queue) of the job being considered, is added to the slack
8. else the partial WCET will be considered and added to the slack
9. end if
10. end loop
11. return slack
12. end if
```

Implementation

The implementation is provided in the git profile. It uses four threads to simulate four different cores, and a global discarded queue for accommodating discarded jobs, and gives preference to the core in which the job is statically allocated/executed previously in order to reduce context switches as much as possible. It assumes that the input is an implicit-deadline system with phase zero. It also causes delayed forced criticality change in all the other cores when one core undergoes criticality change.

CONCLUSION

Vigorous testing and certification are needed by systems with multiple criticalities. Therefore, mixed-criticality systems demand efficient scheduling algorithms and as much QoS as possible. One step towards doing so is to fit as many lower-criticality jobs as possible while making absolutely sure that highest criticality jobs do not miss their deadline. For this purpose, this paper presents an EDF-based scheduling algorithm, which makes use of a single reduction-factor, and is suitable for a multicore, multi-level criticality system. The algorithm consists of two sub-algorithms, an offline pre-processing algorithm which checks the schedulability conditions and finds the necessary parameters for the next algorithm to run, which is the runtime scheduling. The runtime scheduling algorithm makes use of a global discarded queue, which enables job migration to an extent, if needed.

REFERENCES

[1] M. Hikmet, M. M. Kuo, P. S. Roop and P. Ranjitkar, "Mixed-Criticality Systems as a Service for Non-critical Tasks," *2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC),* York, 2016, pp. 221-228, doi: 10.1109/ISORC.2016.38.

[2] L. Huang, I. Hou, S. S. Sapatnekar and J. Hu, "Improving QoS for Global Dual-Criticality Scheduling on Multiprocessors,*" 2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA),* Hangzhou, China, 2019, pp. 1-11, doi: 10.1109/RTCSA.2019.8864597.

[3] H. Chai, G. Zhang, J. Sun, A. Vajdi, J. Hua, J. Zhou. "A Review of Recent Techniques in Mixed-Criticality Systems". *Journal of Circuits, Systems and Computers,* Vol.28, No. 07, 1930007(2019).

[4] S.K. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie. Preemptive uniprocessor scheduling of mixed-criticality sporadic task systems. *Journal of the ACM (JACM)*, 62(2):14, 2015. 16

[5] D.de Niz, K. Lakshmanan, and R. Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *Real-Time Systems Symposium*, pages 291–300. IEEE Computer Society, 2009. 13, 17, 20, 37

[6] T. Park and S Kim. Dynamic scheduling algorithm and its schedulability analysis for certifiable dual-criticality systems. In *Proc. ACM EMSOFT*, pages 253–262, 2011. 8, 15

[7] H-M. Huang, C. Gill, and C. Lu. Implementation and evaluation of mixed criticality scheduling approaches for periodic tasks. In *Proc. of the IEEE Real-Time Technology and Applications Symposium (RTAS),* pages 23–32, 2012. 13, 43