

Progress-aware Dynamic Slack Exploitation in Mixed-critical Systems: Work-in-Progress

Angeliki Kritikakou, Stefanos Skalistis

▶ **To cite this version:**

Angeliki Kritikakou, Stefanos Skalistis. Progress-aware Dynamic Slack Exploitation in Mixed-critical Systems: Work-in-Progress. EMSOFT 2020 - International Conference on Embedded Software, Sep 2020, Hamburg / Virtual, Germany. pp.1-3. hal-03125812

HAL Id: hal-03125812
https://hal.science/hal-03125812

Submitted on 1 Feb 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Progress-aware Dynamic Slack Exploitation in Mixed-critical Systems: Work-in-Progress

Angeliki Kritikakou
Univ Rennes, Inria, CNRS, IRISA, France

Stefanos Skalistis
Raytheon Technologies Research Center, Ireland

***Abstract*—Mixed-critical systems consist of high criticality and low criticality applications. When a high criticality task exceeds its less pessimistic Worst Case Execution Time (WCET), the system switches mode and low criticality tasks are usually dropped. To postpone mode switch, existing approaches explore the slack, created dynamically due to an actual execution of a task that is faster than its WCET. However, the slack is visible only after the task has finished. To further enable dynamic slack exploitation, we propose a fine-grained approach that exposes the slack created due to the progress of tasks, during execution, and safely uses it to postpone mode switch.**

***Index Terms*—Run-time adaptation, Remaining Response Time, Mixed-critical systems,**

I. Introduction and Motivation

Mixed-critical systems [1] consist of high criticality and low criticality applications, with different properties and requirements. The Worst-Case Execution Time (WCET) of applications is typically used in order to provide timing guarantees. However, due to both application and hardware characteristics, pessimism is introduced during WCET estimations. As a result, safe, but over-approximated WCET estimations are obtained, that typically over-allocate resources to high criticality applications. To reduce this negative impact, mixed-critical systems use different WCET estimations for high criticality tasks: a safe, but pessimistic, upper bound (C^H), typically based on static analysis, and a less pessimistic, but less trust-worthy, bound (C^L), typically obtained by measurement-based approaches [2]. Low criticality tasks are bounded by a single, less pessimistic, WCET estimation. The system starts execution in low criticality mode (LO-mode). Usually, if a low criticality task exceeds its C^L, it is dropped. However, as soon as a high criticality task exceeds its C^L, the system switches from low to high criticality mode. Low criticality tasks are usually dropped to meet the timing constraints of the high criticality tasks [3], degrading system QoS.

To improve QoS, existing approaches work on two directions: i) explore other strategies than dropping low criticality tasks, and ii) explore static or dynamic ways to postpone the mode switch. This work focuses on the latter category for uniprocessor systems, which is orthogonal and can be combined with the former category. Static approaches determine task overrun values, which when added to the C^L of high criticality tasks, the system remains schedulable. These values extend the mode switch further than C^L. Such examples are methods inspired by sensitivity analysis [4] and zero-slack [5]. As static approaches are applied before execution, they explore only the existing slack due to system under-utilisation. On the contrast, dynamic approaches exploit the slack created during execution. When the actual execution time of a task is lower than its C^L, slack is created that can be used by subsequent tasks. However, existing approaches are able to observe and use the slack, only after the task has terminated, e.g., single budget [6], bailout protocol [7] and feedback control mechanisms [2].

This work extends the state-of-the-art by dynamically exploiting, not only the slack created due to the early termination of tasks, but also the slack created due to the actual progress of tasks, during execution. To achieve that, we propose a run-time controller that regularly, at instrumentation points, exposes the actual execution progress of high criticality tasks and recomputes their response times in LO-mode. The observed differences are used to compute the dynamically created slack, which is used in order to postpone, or even avoid, the system mode switch. As our run-time computation takes into account the actual progress execution, the online computed response time bounds are typically reduced, compared to the initial worst case estimations, computed offline.

II. System Model

The system is defined as a finite set of tasks T executed on a single processor. For simplification reasons, and without loss of generality, a dual-criticality system is assumed, where each task has a level of criticality equal to either high (H) or low (L), with $H > L$. Each task, τ_i, is defined by its period (minimum arrival interval) T_i, deadline D_i, criticality level CL_i and the WCET C_i^{CL} for each criticality level, i.e., C_i^H and C_i^L. It is reasonable to assume that C_i^{CL} is monotonically non-decreasing with increasing CL_i [8]. Tasks give rise to a potentially unbounded sequence of jobs. Jobs can be either dependent, or independent, and can be periodically executed in a preemptive manner on the processor. We use the basic mixed criticality model, where the system has two modes of execution: i) LO-mode, where both high criticality tasks and low criticality tasks are executed on the processor, and ii) HI-mode, where only the high criticality tasks are executed on the processor.

III. Proposed Methodology

The proposed approach, during execution, regularly exposes the progress of high criticality tasks and reflects that information globally to the system. This allows a more accurate re-computation of the remaining response time of each high

criticality task in LO-mode, as the execution progresses. Having such information, it is now possible to compute in a fine-grained way the dynamic slack created during execution. To achieve that, each high criticality task is instrumented by set of points. During execution, the proposed controller is enabled at a point p of a high criticality task i, when the system is in LO-mode. It applies the following steps:

1) *Remaining WCET computation* (RC_i^L), i.e., the run-time estimation of the WCET of the code that remains to be executed, from point p, until the end of execution in LO-mode (inspired by [9]).
2) *Remaining Response Time computation* (RR_i^L), i.e., run-time computation of the new response time of i in LO-mode, taking into account the actual time and the remaining preemption delays of both low and high criticality, higher priority, tasks ($RP_{hp(i)}^L$).
3) *Dynamic slack computation* in LO-mode, obtained by the difference of the newly computed response time compared to the previously computed value.
4) *Safety condition*: if the slack is higher or equal to the additional time required to reach the next instrumentation point, in HI-mode, high and low criticality tasks continue execution. Otherwise, the slack is depleted and mode switch must occur.
5) *Remaining preemption delay computation* ($RP_{hclp(i)}^L$), i.e. update the preemption delay that task i may cause to other high criticality, but lower priority, tasks, according to its updated Remaining WCET.

IV. Evaluation

In this preliminary experimental set-up, we consider the PULP RISC-V processor of GAP8 platform [10] as our platform model. Table I depicts the characteristics of the task set, consisting of two high criticality tasks (`AES-CTR` and `CannyEdge`) and two low criticality tasks (`matmul` and `FFT`). The C^L of the benchmarks is obtained by the observed execution cycles, when the code is executed on a single PULP core [10]. To obtain the C^H, we apply safe margins of 20%. Fig. 1 shows the schedule in the low criticality mode.

TABLE I
Task set characteristics.

Benchmark	CL	C^L	C^H	Priority	Arrival
`CannyEdge`	H	99,500	119,400	3	0
`matmul`	L	41,900	-	1	50,000
`AES-CTR`	H	15,300	18,360	0	52,000
`FFT`	L	28,200	-	2	53,600

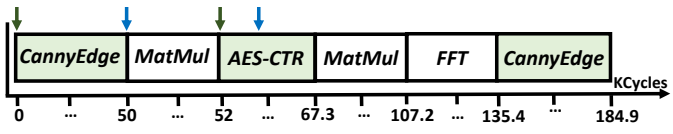
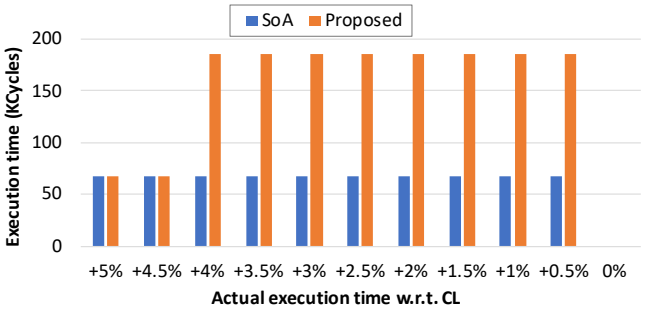


Fig. 1. Execution in low criticality mode; arrows depict the arrival of each task.



(a) **Time when mode switch occurs (y-axis)**



(b) **Low criticality progress (y-axis)**

Fig. 2. Comparison of proposed and SoA approaches

We compare the proposed approach with typical existing approaches that decide to switch mode, when the actual progress reaches the C^L, and the task has not finished execution (SoA), e.g. [3]. We explore their behavior with respect to the possible actual execution time of benchmarks by allowing any part of the benchmarks, defined by two consecutive points p, p', to executed faster or slower than $C_{p,p'}^L$. In particular the actual execution time of each part is selected within the range $[80\% \times C_{p,p'}^L, 120\% \times C_{p,p'}^L]$, while maintaining the total actual execution time of the benchmarks within the range $[C^L, 105\% \times C^L]$, with a step of $0.5\% \times C^L$. Fig. 2(a) depicts the moment when each approach decides to switch modes. As long as the actual execution time is larger than C^L, the SoA decides always to switch mode when the first high criticality task exceeds its C^L, i.e., when `AES-CTR` actual progress reaches $15,300$ cycles, i.e., at $67,300$. On the contrary, the proposed approach exposes the dynamic slack created due to the actual progress of `CannyEdge`, before being preempted. As long as this slack is large enough, the mode switch is not required to be decided by `AES-CTR`, but later on, by `CannyEdge`. As Fig. 2(b) shows, the postponement of the mode switch had allowed the execution of all low criticality tasks, compared to SoA. Notice that, in this experimental set-up, dynamic approaches will behave as SoA, since `CannyEdge` has not finished execution, and thus, the potential slack is not yet observable.

REFERENCES

[1] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *RTSS*, 2007, pp. 239–243.

[2] A. Papadopoulos, L. Bini, S. Baruah, and A. Burns, "AdaptMC: A Control-Theoretic Approach for Achieving Resilience in Mixed-Criticality Systems," in *ECRTS*, vol. 106, 2018, pp. 14:1–14:22.

[3] S. Baruah, A. Burns, and R. I. Davis, "Response-time analysis for mixed criticality systems," in *RTSS*, 2011, pp. 34–43.

[4] F. Santy, L. George, P. Thierry, and J. Goossens, "Relaxing mixed-criticality scheduling strictness for task sets scheduled with fp," in *ECRTS*, 2012, pp. 155–165.

[5] D. de Niz and L. Phan, "Partitioned scheduling of multi-modal mixed-criticality real-time systems on multiprocessor platforms," in *RTAS*, 2014, pp. 111–122.

[6] B. Hu, K. Huang, P. Huang, L. Thiele, and A. Knoll, "On-the-fly fast overrun budgeting for mixed-criticality systems," in *International Conf. Embedded Software (EMSOFT)*, 10 2016, pp. 1–10.

[7] I. Bate, A. Burns, and R. I. Davis, "A bailout protocol for mixed criticality systems," in *ECRTS*, 2015, pp. 259–268.

[8] S. Baruah, L. Haohan, and L. Stougie, "Towards the design of certifiable mixed-criticality systems," in *RTAS*, 2010, pp. 13–22.

[9] A. Kritikakou, O. Baldellon, C. Pagetti, C. Rochange, and M. Roy, "Run-time control to increase task parallelism in mixed-critical systems," in *ECRTS*, 2014.

[10] E. Flamand, D. Rossi, F. Conti, I. Loi, A. Pullini, F. Rotenberg, and L. Benini, "Gap-8: A risc-v soc for ai at the edge of the iot," in *ASAP*, 2018, pp. 1–4.