UNIVERSITY *of York*

This is a repository copy of *Reducing Loss of Service for Mixed-Criticality Systems through Cache-and Stress-Aware Scheduling*.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/200274/

Version: Accepted Version

Proceedings Paper:

Lesage, Benjamin Michael Jean-Rene, Dai, Xiaotian orcid.org/0000-0002-6669-5234, Zhao, Shuai et al. (1 more author) (2023) Reducing Loss of Service for Mixed-Criticality Systems through Cache-and Stress-Aware Scheduling. In: Proceedings of the 31st International Conference on Real-Time Networks and Systems. , pp. 188-199.

https://doi.org/10.1145/3575757.3593654

Reuse

Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown

If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

White Rose
university consortium
Universities of Leeds, Sheffield & York

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

Reducing Loss of Service for Mixed-Criticality Systems through Cache- and Stress-Aware Scheduling

Benjamin Lesage
Onera, France
benjamin.lesage@onera.fr

Xiaotian Dai
University of York, UK
xiaotian.dai@york.ac.uk

Shuai Zhao
Sun Yat-sen University, China
zhaosh56@mail.sysu.edu.cn

Iain Bate
University of York, UK
iain.bate@york.ac.uk

ABSTRACT

Hardware resources found in modern processor architecture, such as the memory hierarchy, can improve the performance of a task by anticipating its needs based on its execution history and behaviour. Interleaved jobs, belonging to other tasks with different behaviours, can cause stress on those resources by disrupting the execution history thus slowing down more sensitive tasks. Schedulability analyses and policies tend to ignore such behaviours, in favour of conservative assumptions, as their effects are difficult to assess. When they are included, the analysis can be very complex and the measures needed are hard to obtain.

In this paper, we propose abstract timing models that capture stress and sensitivity with respect to the memory hierarchy. The advantage of an abstract timing model is that it can be derived through measurements without the need for a detailed understanding of the precise cache hierarchy and how it affects software execution. The disadvantage of course is that there is no *hard* timing guarantee especially if timing anomalies may exist. The contribution of this paper is to build on existing priority assignment schemes using the timing model to discriminate between tasks within sharing a priority levels and improve the system's timing behaviours in overload. More specifically, we show that for task sets scheduled with mixed-criticality scheduling the number of jobs not executed is often reduced.

CCS CONCEPTS

• **Computer systems organization → Real-time system architecture**; **Processors and memory architectures**.

KEYWORDS

Execution time model, abstract timing model, cache analysis, stress and sensitivity, mixed-criticality, scheduling

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
RTNS '23, June 7–8, 2023, Dortmund, Germany
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/3575757.3593654

ACM Reference Format:
Benjamin Lesage, Xiaotian Dai, Shuai Zhao, and Iain Bate. 2023. Reducing Loss of Service for Mixed-Criticality Systems through Cache- and Stress-Aware Scheduling. In *The 31st International Conference on Real-Time Networks and Systems (RTNS 2023), June 7–8, 2023, Dortmund, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3575757.3593654

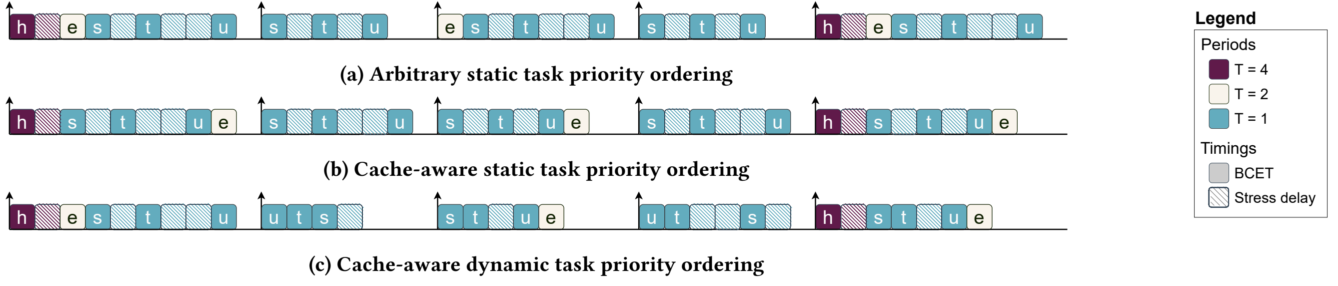
1 INTRODUCTION

Modern processor architectures introduce a number of resources which aim at improving the performance of prolonged or repeated executions of the same task. Those resources aim to anticipate the needs of a task based on its execution history and patterns. As such, interleaved jobs, belonging to tasks with different behaviours, might contend for resources disrupting their state and creating stress for subsequent jobs.

The memory hierarchy is an example of such a resource. It aims at improving the performance of tasks' memory accesses by exploiting the spatial and temporal locality principles. Those principles state that a memory access, be it for instructions or data, is very likely to be followed by accesses to the same address (temporal) or neighbouring ones (spatial). Whole blocks of data are thus brought from the memory into fast caches, close to the processor, to speed up subsequent accesses. Due to the limited size of caches, excess blocks are evicted to make room for more recent ones.

A memory-*sensitive* task might thus leave residual memory blocks in the memory hierarchy that subsequent jobs of the same task can exploit. A *stressing* task, contending for the memory hierarchy, would result in the eviction of those residual blocks in favour of its own. A scheduler informed by a stress and sensitivity model at run time could interleave the jobs of these two tasks to (1) maximise the benefits from residual blocks for sensitive tasks; and (2) minimise the impact of stress on the memory hierarchy.

Example. Consider the motivating example in Figure 1 composed of 5 tasks, `h`, `e`, `s`, `t`, and `u`, where the upward arrow indicates a tick of period $T = 1$. Tasks `t` and `s` have a high sensitivity to cache effects and cause low stress on others. Task `e` has low sensitivity but causes high stress, and task `u` has low sensitivity and low stress. The basic assumption underlying this work is that the fewer jobs execute between two consecutive instances of a task like `s`, the less it will be impacted by stress on the memory hierarchy. Stress in turn results in missing residual blocks and thus longer execution times for tasks, identified through the hashed blocks. As an example, the first instance of any task is subject to very high stress; the task executes against a *cold* memory hierarchy, without residual



(a) Arbitrary static task priority ordering

(b) Cache-aware static task priority ordering

(c) Cache-aware dynamic task priority ordering

Figure 1: Impact of static and dynamic task priority ordering on the schedule of the same task set.

memory blocks. In contrast, it doesn't matter as much how many or which jobs execute between instances of task u from the perspective of cache stress and sensitivity.

The higher priority, larger period task h should always execute first before other tasks. The order of the other low-priority tasks does not impact the schedulability of the system and has less impact on the response time of the tasks in the system. Following our assumption, however, it may help improve the service offered to tasks in a mixed-criticality scheduling scheme. Figures 1a and 1b presents a static fixed priority schedule, where the impact of the cache is not considered (Figure 1a) or improved to reduce the impact of high-stress tasks (Figure 1b). The dynamic fixed priority schedule (Figure 1c) allows the online reordering of jobs within their priority level.

Under the arbitrary ordering depicted in Figure 1a, the sensitive task t is always subject to high stress from task e when they are released together. This translates to longer execution times (hashed blocks) for t. Figure 1b has a different ordering where task t is only subject to interferences from tasks u, h, and s in the last period. This results in a small speed-up over the previous scenario, when all tasks are released. By alternating the execution of s, t, and u, the dynamic schedule in Figure 1c further reduces the stress the task inflicts upon each other mainly by reducing the impact of e.

Contributions: We introduce the concept of the *Cache Recency Profile* (*CRP*), an execution time model based on how recently a particular task has been executed – referred to as the *Recency Distance* (*RD*). The profile represents a simplified model of the speed-up of a task, built on measurements without a detailed understanding of the cache architecture or the actual execution order of tasks. The *CRP* model takes no part in the schedulability analysis of the system. Instead, it is used in conjunction with existing methods, such as Audsley's Optimal Priority Assignment (OPA) [5], acting as a tie-breaker when priority levels are shared. Based on the *CRP* and its inherent properties, we introduce a static priority allocation, and a family of scheduling approaches relying on predicting individual job's execution time at runtime.

Paper organisation: We first introduce the task model and system assumptions (Section 2). This leads to the definition of our stress and sensitivity model, i.e. the *CRP*, its semantics and construction from observations on a target platform in Section 3. Section 4 then introduces the intuition behind the relationship between stress and the properties of tasks, to define new fixed priority allocation and scheduling approaches. The evaluation of both the proposed execution time model and its application are presented in Section 5. We finally discuss related work and conclude respectively in Sections 6 and 7.

2 TASK MODEL, NOTATION, AND SCHEDULABILITY

We present in the following an execution time model, the *CRP*, for mixed-criticality systems (MCS) under Non-Preemptive Fixed Priority Scheduling (NP-FPS) on a single-core processor. The *CRP* does not intend to produce or replace WCET estimates, nor is it aiming to contribute to the schedulability of a system. Our approach exploits shared priority levels from existing analyses to refine scheduling decisions and improve the service to tasks.

A system comprises a set of N tasks (τ_1, τ_2, ..., τ_N). Each task τ_i is defined by its period, execution time budget, criticality level, priority, and *CRP* model: $\tau_i := (T_i, C_i, L_i, P_i, CRP_i)$. Task deadlines are implicit, constrained to their periods such that $D_i = T_i$. Tasks may share the same priority, $P_i = P_j$, thus belonging to the same priority group. The scheduler always ensures the next task is picked amongst the ready tasks in the highest priority group, $ready^{hp}$. The selection of the highest priority ready task within the said group is decided at run time based on the recent execution order of tasks.

We consider the standard dual-criticality task model (*LO* and *HI*). Each task τ_i is categorised in one of the two classes according to its criticality L_i. *LO*-criticality tasks have a single timing estimate $C_i = C_i^{LO}$, and *HI*-criticality tasks have an additional more conservative estimate $C_i = (C_i^{LO}, C_i^{HI})$, such that $C_i^{LO} \leq C_i^{HI}$. C_i^{HI} is a WCET estimate obtained through conservative timing analyses, and guaranteed to hold for a *HI*-criticality task under all conditions.

The scheduler recognises different execution modes. In the *LO*-criticality mode, all tasks execute within their C^{LO} bound and all jobs are completed within their deadline. In the *HI*-criticality mode, only *HI*-criticality tasks are guaranteed to run and assumed to do so within their C^{HI} bound. A watchdog monitors tasks for timing overruns, events where a task's execution exceeds its C^{LO}. *LO*-criticality jobs are terminated by the watchdog on overruns, and their contribution to cache stress is accounted as full. An overrun from a *HI*-criticality task will result in a mode change, to the *HI*-criticality mode. The job is left to complete up to its C^{HI}, and ready *LO*-criticality jobs are cancelled. The system returns to the *LO*-criticality mode on an idle tick.

The system is known to be *schedulable* if it is schedulable in both LO and HI modes, and upon a mode change from the LO to the HI mode [7]. This is solved by assessing the response time R_i^L of each task τ_i against its deadline D_i, in all its criticality modes $L \leq L_i$:

$$R_i^L = B_i^L + C_i^L + \sum_{j \in hep^{\geq L}(i)} \left(\left\lceil \frac{R_i^L}{T_j} \right\rceil \times C_j^L \right) \quad (1)$$

where $hep^{\geq L}(i)$ is the set of higher or equal priority tasks τ_j of criticality $L_j \geq L$, and B_i^L is the blocking term, i.e. the maximum time spent by the task waiting for a lower priority task $lp^{\geq L}(i)$ of criticality $L_k \geq L$ to complete:

$$B_i^L = \max_{k \in lp^{\geq L}(i)} (C_k^L - 1) \quad (2)$$

A transition to the HI mode occurring during blocking, as a result of a HI criticality task overrun, would fall under the case covered by R_i^{HI} in Equation (1). The response time R_i^* of task τ_i during a mode change thus assumes some tasks execute to their C^{LO} before a transition caused by a higher or equal priority task:

$$\begin{aligned} R_i^* =& B_i^{LO} + C_i^{HI} \\ &+ \sum_{j \in hep^{=LO}(i)} \left(\left\lceil \frac{R_i^{LO}}{T_j} \right\rceil \times C_j^{LO} \right) \\ &+ \sum_{j \in hep^{=HI}(i)} \left(\left\lceil \frac{R_i^{*}}{T_j} \right\rceil \times C_j^{HI} \right) \end{aligned} \quad (3)$$

3 SINGLE-CORE STRESS AND SENSITIVITY

A stress and sensitivity model [14, 16] captures the effect on the execution time of a task of other tasks contending for shared resources; the *sensitivity* of a task thus outlines the variations of its execution time in response to the *stress* induced by other tasks. While originally proposed in the context of multi-core architectures, the concepts of stress and sensitivity apply just as well to single-core architectures, where contention occurs not as a result of concurrent accesses to a resource, but as a result of preemptions, or the interleaving of jobs from different tasks.

Stress and sensitivity models are rarely considered as part of the schedulability analysis of a system. The conservative WCET timing estimates used for verification remain valid for all execution contexts generated by the system. A sensitivity model however can inform the scheduler on the expected behaviour of tasks at runtime, and arbitrate conflicts between equivalent tasks w.r.t. to the schedulability of the system. It is important to note that the models do not need to accurately model cache behaviours but any inaccuracies may lead to sub-optimal run-time orderings; a complete and correct model is impractical given the number of different paths that can be taken through each task or the number of different run-time orderings of jobs.

We introduce the stress and sensitivity components of our model respectively in Sections 3.1 and 3.2. This includes the process required to collect observations to build the model, and, in Section 3.3, the process of building a model usable at runtime.

3.1 RD – Measuring stress on the memory hierarchy

We consider the following properties as desirable in a good stress metric:

- *Significant*: Variations captured by the metric imply substantial variations in timing;
- *Observable*: Contributions to the stress levels are easy to monitor at runtime;
- *Controllable*: Contributions that cannot easily be influenced or controlled represent an unnecessary level of detail;
- *Intuitive*: The link between the metric(s) and variations of the execution time can be explained;
- *Positively correlated*: Higher stress levels, as captured by the metric, imply higher contention and thus execution times;
- *Profilable*: The metric can be extracted from tasks through a one-time profiling phase.

We propose a stress metric to assess the impact of a task on others, w.r.t. the memory hierarchy and in particular residual cache blocks. Residual cache blocks are the memory blocks that may have been loaded into cache when a task was last executed and that may not have been evicted before it is accessed again. As interleaving jobs from contending tasks perform memory accesses, they might contribute to stress on the memory hierarchy by evicting residual cache blocks of the modelled task. The likelihood a residual cache block is evicted increases with each access to a new, distinct cache block. We thus introduce the *Recency Distance* metric of a task (RD_j) as its contribution to stress. RD_j captures the maximum number of unique cache blocks accessed by τ_j during its execution, a proxy of the evictions it incurs in the memory hierarchy (*Intuitive*). An increase in the recency distance between instances of a modelled task τ_i should increase the likelihood its residual blocks are evicted, resulting in longer execution times (*Positively correlated*). The RD is akin to the concept of *cache reuse distance* [30].

The RD_j metric used in this paper abstracts itself from the actual memory accesses and evictions caused by a task τ_j. The RD also ignores the location of the blocks in memory, i.e. whether contending and modelled tasks occupy the same cache space. That is, the cache effects of a single large task or combined smaller distinct tasks may be similar, in terms of the model, and evenly distributed across the cache space. These abstractions help reduce the required knowledge on other tasks in the system and their memory mapping, and a single number captures the contribution of a task to stress.

The RD of a task can be profiled through a number of existing tools [8, 29, 30] (*Profilable*), as an example by counting the number of cold cache misses (due to a block having never been accessed) generated by a run of the task. Profiling, and the specifics of these tools, might result in noise, variations, or precision loss in the measurements. However provided the relative RD values collected across different tasks are representative of their relative impact, the metric should be adequate for modelling purposes.

Upon execution of a task τ_j, all prior tasks might suffer from the evictions it causes. The contribution of τ_j to stress on the memory hierarchy, captured as the single value RD_j, needs to be accounted. For any two consecutive jobs of τ_i which interleave with an execution of τ_j, RD_j is added to the stress suffered by τ_i to capture the contribution of τ_j. To assess the stress suffered by a job of τ_i

at runtime, we simply need to which other tasks executed since the last instance of τi, and their respective *RD* (*Observable*). We provide more details on the computation of the stress suffered by a job in the next section.

3.2 CRP – Profiling the sensitivity to cache behaviour

The *RD* denotes the contribution of tasks to stress on the memory hierarchy. To predict how a task might behave at runtime, we need to model its sensitivity, that is how its execution time might be impacted by said stress. We propose here a *CRP* model to capture the potential benefits of residual cache blocks to the execution of a task. The *CRP* outputs a scaling factor relative to a baseline execution time, in our case the task's WCET. The execution time (ET) estimation of task τ_i under a given stress level r is thus predicted using the *CRP* as such:

$$\hat{ET}_i(r) = CRP_i(r) \times C_i^{L_i} \tag{4}$$

where CRP_i is the *CRP* for task τ_i, $C_i^{L_i}$ is its WCET estimate according to its criticality. The use of a scaling factor to model sensitivity, as opposed to an additive one, offers several benefits. With the use of the WCET as the baseline for timing estimates, the model intuitively defines the *CRP* as the expected speed up for the modelled task thanks to residual cache blocks left by a prior job. The speed-up can be constrained such that no predictions exceed the task's WCET. The model output is independent of the execution time measurement unit. The *CRP* could accommodate prediction errors caused by external influences if their range is known. For example, the task may be further slowed down by 10% due to cache effects introduced by an RTOS.

To build the *CRP* of task τ_i, we need to understand how the stress generated by contending tasks, captured by their *RD* contribution, impacts the execution time of consecutive jobs of τ_i. The stress suffered by the kth instance of task τ_i, $J_{i,k}$, since its last instance is defined as:

$$r_{i,k} = \sum_{\{j \mid \exists J_{j,m}, J_{i,k-1} < J_{j,m} < J_{i,k}\}} RD_j \tag{5}$$

where $r_{i,0} = \infty$, RD_j is the contribution of task τ_j to stress on the memory hierarchy (profiled as discussed in §3.1), and $J_{j,m} < J_{i,k}$ denotes that the mth instance of task τ_j executes prior to the kth job of task τ_i. We only account once for the blocks loaded by a task τ_j. The intuition behind this abstraction is that repeated executions of τ_j will either find the blocks in cache, causing no further evictions, or having been evicted themselves as well as the other task's blocks[1].

The level and impact of stress on a job can be monitored at run time, from the actual system, by collecting jobs' execution times and their scheduling history. At first, the models may not be as accurate as those generated using contender-based approaches, e.g. [21]. However with time, they have the advantage of addressing concerns, principally representativity, highlighted in [19]. Refinements of the model based on observations from the actual system have been considered in [13]. We focus in the following on bootstrapping the model, in the absence of the actual system.

[1]We have observed only negligible effects due to repetitions on our target platform, but we acknowledge that the abstraction of the evictions caused by task τ_j may vary e.g. based on the cache replacement policies.

Algorithm 1 Collecting CRP observations through profiling

```
1: function RUN_CONFIG(task, contenders, repetitions)
                                  /* Collect ET for task under contention */
2:     timings ← []
3:     for i in range(repetitions) do
4:         run(task)                          ▷ Warm up memory hierarchy
5:         for c in contenders do             ▷ Run all contenders
6:             run(c)
7:         end for
8:         t ← instrumented_run(task)         ▷ Measure execution time
9:         timings ← timings + [t]
10:    end for
11:    return timings
12: end function

13: function PROFILE_TASK(task, rd_range, reps)
                          /* Profile task sensitivity at selected range */
14:    observations ← []
15:    explored_rd ← []
16:    for r in rd_range do
17:        explored_rd ← explored_rd + [r] × reps.stress
18:    end for
19:    shuffle(explored_rd)
20:    for target_rd in explored_rd do
                               /* Pick contenders to match target RD */
21:        cntd ← pick_contenders(target_rd)
22:        for t in run_config(task, cntd, reps.config) do
                               /* Collect (RD, ET) for configuration */
23:            observations ← observations + [(rd(contenders), t)]
24:        end for
25:    end for
26:    return observations
27: end function
```

We consider the introduction of a dedicated profiling step, to build the initial *CRP* of a task. The profiling approach, outlined in Algorithm 1, provides control over the exercised contention levels, allowing for a more systematic exploration of the *RD* value range (*Controllable*). The resulting observations contribute to the characterisation of the task's sensitivity, like *RD* for stress, on a given hardware platform, in a variety of scenarios, and irrespective of some of its context. The core profiling loop (lines 1-12) instruments the execution of the modelled task to capture its execution time. The task is executed twice, first to warm up the memory hierarchy (line 4), and the selected contenders (whose *RD* is known) are executed in between to generate controlled stress levels (lines 5-7). The process is repeated across various *RD* levels in a random order (lines 15-19), each time picking a set of contenders matching the target *RD* (line 21). Each configuration of contenders and target *RD* contention can be repeated (resp. lines 3 and 17) to assess the variability of execution times under the same stress level. This allows for estimating some of the noise inherent to the *CRP* modelling approach and *RD* abstraction. Each collected execution time is matched against the exercised *RD* (line 23).

Algorithm 2 Predict CRP with PWLF model

```
1: function CRP_PREDICT(r, crp)
2:     c = min({b | r > crp.breaks[b]})          ▷ Find segment for r
3:     p ← crp.slopes[c] × r + crp.inters[c]      ▷ Fit r to segment
4:     return max(0, min(p, 1.0))                 ▷ Clamp output
5: end function
```

3.3 Model fitting the CRP

The *CRP* observations from the profiling step, described in the previous sections, provides a raw map of the sensitivity of a task (ET) to contention in the memory hierarchy (RD). We now consider how to fit a suitable model to our observations, to capture the main trends in the task's behaviour, and provide a concise representation to be embedded at runtime to inform the scheduler. Our approach relies on piece-wise linear regression (PWLF).

PWLF [24] partitions the model into M linear segments. Each segment models the sensitivity of the task over a continuous, non-overlapping partition of the *RD* value range. A single model only stores information about the slope and intercept of each segment, and the breakpoints that partition the *RD* value range. Algorithm 2 outlines the prediction of the *CRP* at a given *RD* level, as discussed earlier with a clamped output, i.e. restricted to the interval $[0, 1]$. PWLF provides for models with low memory ($O(M)$) and computation ($O(M)$) overheads. The segments capture the diminishing returns of residual cache blocks in the memory hierarchy. As the distance between jobs of the same task increases, measured by *RD*, those blocks are evicted from the higher level caches, closer to the processor, and only reside in larger, slower levels which access latency tends towards that of the main memory.

The focus is therefore on capturing trends in the task sensitivity, when and how much the benefits of residual cache blocks wear off, more than an exact estimate of these benefits. We adjust the trained model to ensure (1) monotonicity (2) and continuity of the model predictions with increasing *RD*. The process starts from the tail of the last segment and adjusts the parameters of each segment such that its tail meets the head of its successor at their shared breakpoint (continuity), and the slope of the segment is null or positive (monotonicity). The adjusted model is by construction more pessimistic than the trained one, predicting the same or lesser speedups.

4 CRP-BASED SCHEDULING FOR MIXED-CRITICALITY

OPA [5] provides an optimal solution to the priority assignment problem. That is the priority assignment produced for a given task set, if any, is at least as good as any other assignment. Three necessary and sufficient conditions have been identified for the optimality of the solution [15].

In condition 1 (respectively condition 2), the schedulability of a task may depend on any independent properties of higher (resp. lower) priority tasks, but not on their relative priority ordering. Condition 3 states that when swapping the priorities of two tasks, the now higher priority one cannot become unschedulable if it was previously schedulable. These conditions hold in our task model for the schedulability test presented in Section 2. Neither response time Equations (1) nor (3) depends on task ordering, and the additional blocking by swapping the order of two tasks with the same priority level is balanced by the reduced interferences from higher priority tasks.

4.1 Conflict resolution in shared priority levels

Our approach relies on a variant of OPA which minimises the number of priorities required [6]. This variant of OPA provides a partial ordering between tasks, if any, which ensures the system is schedulable. As long as this partial order is preserved, the system is schedulable. If no higher priority task is ready, two tasks assigned the same priority level can be executed in any order. Under the *CRP* model, however, different task schedules and interleaving may result in different stress levels and execution times for each task. We thus propose a static priority assignment which aims to maximise the benefits of residual cache blocks in the memory hierarchy. We focus in particular on the optimisation of the critical instant where all tasks are simultaneously released. The proposed approach relies on three intuitions regarding the impact of tasks on the memory hierarchy. Those informal precepts should hold in the general case, but we aim to provide neither a formal proof, nor a counter-example.

Intuition 1. Tasks with a smaller period contribute to the stress of higher period ones upon simultaneous release.

Let us consider a periodic tick T_H where two jobs $J_{i,k}$ and $J_{j,m}$ are released. As such if $T_i < T_j$, the last release of task τ_i occurred at time $T_H - T_i$, and the last release of τ_j at $T_H - T_j < T_H - T_i$. Both releases must complete before T_H. The execution of τ_i is thus likely to have completed more recently. There is at least one job of τ_i contributing to stress in the memory hierarchy since the last instance of τ_j. When both are released simultaneously, there is no analytical impact for $J_{j,m}$ in running after $J_{i,k}$ (priority ordering allowing) as τ_i's contribution has already been accounted for in the stress $r_{j,m}$ suffered by $J_{j,m}$ (see Equation (5)).

Intuition 2. Higher-priority jobs contribute to the stress of lower-priority ones upon simultaneous release.

When two jobs from distinct priority classes are released at the same time, the higher priority job should be executed first thus contributing to stress for the lower priority one. Note that if the two tasks have the same period, they continuously conflict with each other: the lower priority task k^{th} instance, $J_{j,k}$, will be executed after the instance of the higher priority task, $J_{i,k}$, but before the next one, $J_{i,k+1}$.

Intuition 3. All fixed orders of tasks within a same priority and same period group are equivalent w.r.t. how the tasks conflict with each other.

Following from intuition 2, any task τ_h set to run first in the group will contribute to the others' conflicts upon simultaneous release. And conversely, all other tasks in the group, executed afterwards, will contribute to conflicts for the next release of τ_h.

We thus present the *Cache-based Priority Assignment* (CPA). The intuition behind the CPA is to refine the priorities assigned through OPA, from the highest priority group to the lowest, to maintain the schedulable partial order. Tasks within a priority group are ordered offline such that small period tasks are executed first, as they do

not add to preexisting stress (Intuition 1). Within a priority and period group, tasks are left in their arbitrary definition order, as no solution is deemed more favourable (Intuition 3). CPA does reduce to OPA with Deadline-Monotonic Priority Ordering applied within each priority level, irrespective of the current implicit deadline constraint where $T_i = D_i$ (Section 2).

4.2 Dynamic priority refinements using CRP

The static CPA priority assignment is informed by the *CRP* model properties but itself makes no call or execution time prediction through the model. CPA instead relies on reducing the stress on tasks outside of preexisting effects, to provide a total ordering between tasks. Under OPA, the scheduler must pick the next job amongst the valid candidates $ready^{hp}$ (see Section 2) in the same priority group. Using the *CRP*, the scheduler could identify the tasks that are the most likely to benefit from residual cache blocks and, where possible, schedule them first.

Intuition 4. The initial execution of a task, its first job, will see no benefit from the memory hierarchy.

Barring scenarios where different tasks may share data or instructions, a task that was never executed could not have allocated cache blocks. Such tasks could thus be scheduled last, while maintaining the relative priorities defined by OPA. If all ready tasks have passed their first job, lesser stress levels, i.e. smaller *RD* contributions, on the hierarchy should be favoured.

Intuition 5. Any task already present in another task's history will contribute no further stress to the latter.

By construction (see Eq. (5)), we only account once for the contribution of a task to another one's stress. The scheduler could thus prioritise jobs of tasks whose contribution is already accounted for in other ready tasks.

We propose four heuristics to (1) reduce the contribution to the stress suffered by following jobs ($[RD <]$, Eq. (6)), (2) favour jobs under the least stress ($[r <]$, Eq. (7)), (3) those with the highest speedup ($[CRP <]$, Eq. (8)), (4) or those with the highest timing improvements ($[\Delta\hat{ET} >]$, Eq. (9)). Situations where multiple jobs have the same value output from the heuristics are resolved by picking the one with the least contribution to stress (RD_i) on the memory hierarchy.

$$[RD <] = \min_{J_{i,k} \in ready^{hp}} (RD_i) \tag{6}$$

$$[r <] = \min_{J_{i,k} \in ready^{hp}} (r_{i,k}) \tag{7}$$

$$[CRP <] = \min_{J_{i,k} \in ready^{hp}} (CRP_i(r_{i,k})) \tag{8}$$

$$\begin{aligned} \Delta ET_i(r) &= C_i^{L_i} - \hat{ET}_i(r) \\ [\Delta\hat{ET} >] &= \max_{J_{i,k} \in ready^{hp}} (\Delta ET_i(r_{i,k})) \end{aligned} \tag{9}$$

We order tasks for their initial release, considering that the initial release of tasks might result in a pattern maintained by the heuristic as observed in Figure 1c. Each heuristic uses the pre-defined ordering to break ties upon the first job of a task. The initial ordering is computed through a *greedy algorithm* which aims to maximise the benefits of scheduling a task ($\Delta ET_i(r)$) first over the slowdown it causes on others ($\Delta ET_j(r) - \Delta ET_j(r + RD_i)$). The greedy approach starts by considering all tasks in decreasing priority, picking the best one and appending it first to the static order. Once a task has been picked its contribution is added to the current order *RD*, and the process repeats until all tasks have been ordered.

Note that all heuristics, with the exception of $[RD <]$, make use of the stress or *CRP* model at runtime with predictions feeding into their scheduling decision. This requires additional monitoring on the system's behalf, in particular a sufficient representation of the job history. For each task executed τ_i, the scheduler needs to capture its contribution RD_i to stress, and whether any task τ_j executed since its last job. This can be modelled with a bit matrix ($O(N^2)$ where N is the number of tasks in the system), such that $h[i][j] = 1$ if τ_j executed since the last job of τ_i. Upon execution of τ_j, its line $h[j][\cdot]$ is reset and its column $h[\cdot][j]$ is set. The extra monitoring is not considered prohibitive given the benefits attained through the use of the heuristics that depends on it.

5 EVALUATION

The *CRP* aims to provide a high-level model of the behaviour of tasks, and especially their interleaving, in response to stress on the memory hierarchy. It could thus provide guidance on how to schedule tasks to minimise said stress. In this section, we evaluate two key aspects of our approach:

- §5.2: *How accurately do the RD and CRP models represent stress and sensitivity?* This question focuses on the quality of the *CRP* as a model of the execution time of a task, whether the *RD* is a *Significant* indicator of execution time variability, and potential gaps in the model.
- §5.3: *How do the proposed policies affect the loss of service for systems?* The proposed scheduling policies aim to exploit properties of the *CRP*, or deploy the model at runtime to perform decisions. We need to assess whether those benefit the system as a whole, and the underlying trade-offs.

5.1 Experimental setup

In this evaluation, we explored two complementary facets of the proposed *CRP* model, first in terms of the quality of the model for execution time predictions, then its benefit to a scheduling policy. To assess the quality of the model, we profiled a selection of 11 benchmarks from the TacleBench suite [18]. The selected benchmarks, described in Table 1, were picked to capture a variety of sizes and behaviours, and ensure all benchmarks' execution follows a single path. The benchmarks are sorted increasingly by the number of memory accesses they perform, be it instruction or data accesses. These are good metrics of the overall length of each benchmark and of their behaviour w.r.t. the memory hierarchy. Additional accesses imply additional instructions to execute, and they tend to target new unique cache blocks causing more stress as captured by RD_i.

We applied Algorithm 1 to profile the benchmarks with the configuration depicted in Table 2. The profiled *RD* range (`rd_range`) covers the interval where benchmarks exhibited most variations due to stress. For each target *RD* level, we generated 25 configurations of contending tasks (`reps.stress`) to capture a spectrum of

Benchmark	RD_i	# Mem. accesses
statemate	97	60632
ndes	194	126978
adpcm_dec	151	306214
adpcm_enc	148	307446
h264_dec	648	402850
g723_enc	182	1054418
gsm_dec	536	3714177
anagram	1215	7160559
gsm_enc	916	9990908
ammunition	970	261630684
mpeg2	4105	568118381

Table 1: Profiled benchmarks characteristics

stress sources, e.g. from data vs. instruction blocks, for each level. The configurations were randomly picked amongst the valid combinations of remaining benchmarks, such that their combined *RD* tends towards the target *RD* level. Each configuration was further executed 5 times (`reps.config`) to ensure consistent timings.

Parameter	Value	Description
`rd_range.min`	0	Minimum target *RD* level
`rd_range.max`	10000	Maximum target *RD* level
`rd_range.step`	100	Step between target *RD* level
`reps.stress`	25	Number of configurations per *RD* level
`reps.config`	5	Number of runs per configuration
`mb_size`	32B	Memory block size

Table 2: Parameters applied during profiling (Alg. 1)

The tasks were profiled on a quad-core Intel i5-6500 machine, with three layers of cache of respectively 64KB, 256KB and 6MB, under the Linux Debian operating system. The CPU core frequency is fixed to 800MHz, by deactivating Dynamic Voltage and Frequency Scaling, to reduce execution time variability. The process was mapped on an isolated core, such that no other user application was running alongside, but no effort was made to isolate the profiling process from system noise. The use of a complex off-the-shelf processor, whose specifics are unknown aims to assess how the high-level *CRP* model can still explain variations in the execution time of a task.

We evaluated the *CRP*-based scheduling policies by simulating the system. For each simulation, the behaviour of each task set under each policy was simulated for 100 million cycles while collecting the following metrics:

(1) *JNE*: the number of jobs not executed, due to a missed released, or criticality change (invalidating current and future releases); and
(2) *Effective Utilisation*: the effective utilisation of the system, the portion of time spent executing jobs over the total simulation length.

We generated 100 schedulable task sets under each combination of task generation parameters defined in Table 3. We applied the DRS-based (Dirichlet-Rescale) algorithm in [20] to generate our task sets, generating first the tasks' HI utilisation, then their LO utilisation. Periods are randomly picked in the set described in Table 3, determining a task's C^{LO} (and C^{HI} if applicable) by combining it with their utilisation. A task's BCET is obtained by multiplying its C^{LO} with a random factor picked in the $[0.5, 1.0]$ interval.

Shared Parameters	
Length (cycles)	100,000,000
Cache size (blocks)	10,000
Task periods (cycles)	$\{1x, 2x, 3x, 4x, 5x, 10x, 20x, 25x, 40x, 50x\}$
Task Generation Parameters	
Task count	{10, 20, 30, 50, 80, 100, 150, 200}
Target utilisation	{10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%}
Target cache utilisation	{25%, 50%, 100%, 200%, 400%, 1000%}
HI tasks ratio	{0.1, 0.2, 0.3, 0.4, 0.5}
BCET to C^{LO} ratio	$[0.5, 1.0]$

Table 3: Simulation and Task generation parameters

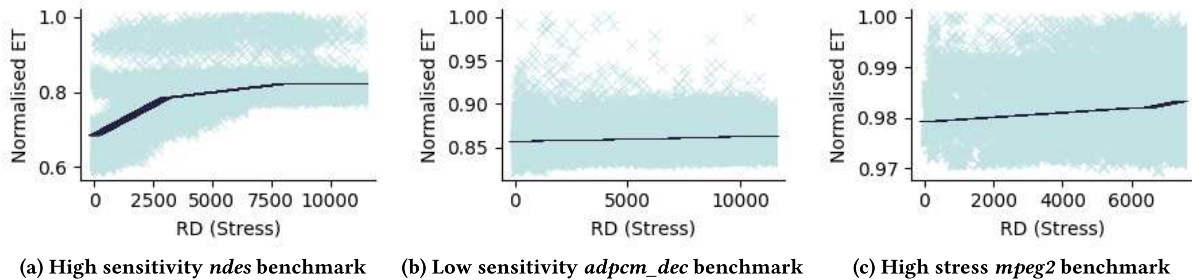
We generate the *CRP*-specific parameters using a similar approach. The *RD* contribution of tasks is picked uniformly such that their combined contribution amounts to the task set target cache utilisation. The *CRP* for each task is generated by picking two breakpoints, b_{LO} and b_{HI}, in the $[0; \textit{Cache size}]$ interval using DRS. Those delimit the segments of the task *CRP*, modelled using a piece-wise line function. At $r = 0$, the task executes to its BCET. Its execution time linearly increases with stress up to C^{LO} at $r = b_{LO}$. Similarly the task execution time increases from C^{LO} to C^{HI} between $r = b_{LO}$ and $r = b_{HI}$. The task sees no benefit from residual cache blocks if the exercised stress exceeds b_{HI}.

5.2 RD and CRP as a stress and sensitivity model

This section evaluates the *RD* and *CRP* as a stress and sensitivity model for a task, that is whether the stress on the memory hierarchy as measured by *RD* explains some of the execution time variability of a task, and how much variability is explained by the model. We first profile the selected benchmarks using the approach described in §3.2, and fit a *CRP* model to each as per §3.3. Outliers, observations with a z-score [26] (or distance from the mean) above 3, are removed from the data. Execution times for each benchmark are normalised to its highest observed one, barring outliers. Each *CRP* is composed of $M = 3$ segments, to capture the diminishing returns of the 3 cache layers in the profiled memory hierarchy.

The collected observations and the corresponding predictions through the fitted *CRP* are presented in Figure 2 (resp. using light and darker colours). We identified three classes of benchmarks, and only present results for a representative of each class:

- High sensitivity benchmarks in Figure 2a, represented by *ndes* (and *statemate*), exhibit clear benefits from residual cache blocks in the memory hierarchy. Those benchmarks are the shortest in terms of the number of memory accesses. The presence or absence of just a few residual cache blocks is thus enough to cause execution time variability.



(a) High sensitivity *ndes* benchmark **(b) Low sensitivity *adpcm_dec* benchmark** **(c) High stress *mpeg2* benchmark**

Figure 2: Observed and predicted execution times under the CRP model for selected benchmarks from the TacleBench suite.

- Low sensitivity (Figure 2b), represented *adpcm _dec* (*adpcm_enc*, *gsm_enc*, *gsm_dec*, *h264_dec*), exhibit some temporal variability but it is not explained through residual cache blocks. This can stem from a longer baseline execution time which eclipses the comparatively smaller benefits of residual cache blocks, or internal conflicts which prevent the reuse of those blocks.
- High stress (Figure 2c), represented by *ammunition* (*anagram*, *mpeg2*), see marginal benefits from the cache, and cause a high level of disruption for other tasks. Those tend to be the longest tasks in our sample, as more memory accesses tend to correlate with more unique blocks and higher stress levels.

All fitted *CRP* models exhibit the same trend in the observations with higher *RD* leading to higher execution times. The *RD* is thus an interesting metric to capture the distance between two jobs of a task while implicitly including the whole memory hierarchy, i.e. caches but also other control structures such as translation look-aside buffers, or write registers. However, it is insufficient to explain all execution time variability.

We further assess the quality of the *CRP* through two metrics comparing observed and predicted execution times (1) the root-mean-square error (RMSE [23]); and (2) the Spearman's rank correlation coefficient [32]. The RMSE is a scale-dependent measure of the accuracy of the model predictions with lower values indicating better predictions. The value captures the absolute, average distance between observations and predictions with a value of 0 being a perfect match. The correlation coefficient assesses whether the prediction and observation have a monotonic relationship, i.e. increases in one relate to an increase (1) or decrease (-1) in the other. A score closing on 0 indicates a lack of correlation between observations and predictions, or a lack of correlation between increases in *RD* and observed timings. We present the error and correlation metric for the *CRP* fitted to each benchmark in Table 4, computed from a sample of 10,000 observations. All correlation coefficients are noted as significant.

The prediction error across all benchmarks tends to be quite low, with the exception of *adpcm_enc*. It could be modelled with a simple error model considering a ±10% interval around predictions. *adpcm_enc* is a small benchmark, in terms of both memory footprint and execution time, which either does not benefit from residual cache blocks or such that small perturbations could cause comparatively large effects on the task. In either case, the *RD* stress metric is insufficient to explain the variability within or between instances of the benchmark. *adpcm_dec* and *anagram* also exhibit a poor correlation, however, a small error seems to indicate instead that the benchmarks show little variability w.r.t. *RD* in our experiments. Other benchmarks such as *h264_dec* and *mpeg2* show a small correlation between observations and predictions, indicating that the model does capture a trend in the data but it is insufficient to explain most execution time variability. It does however manage to capture a lot of variability for the smallest tasks *statemate* and *ndes*. Both exhibit a bi-modal distribution, as highlighted in Figure 2a. The figure shows a larger density of observations close to the baseline execution time irrespective of the observed *RD*.

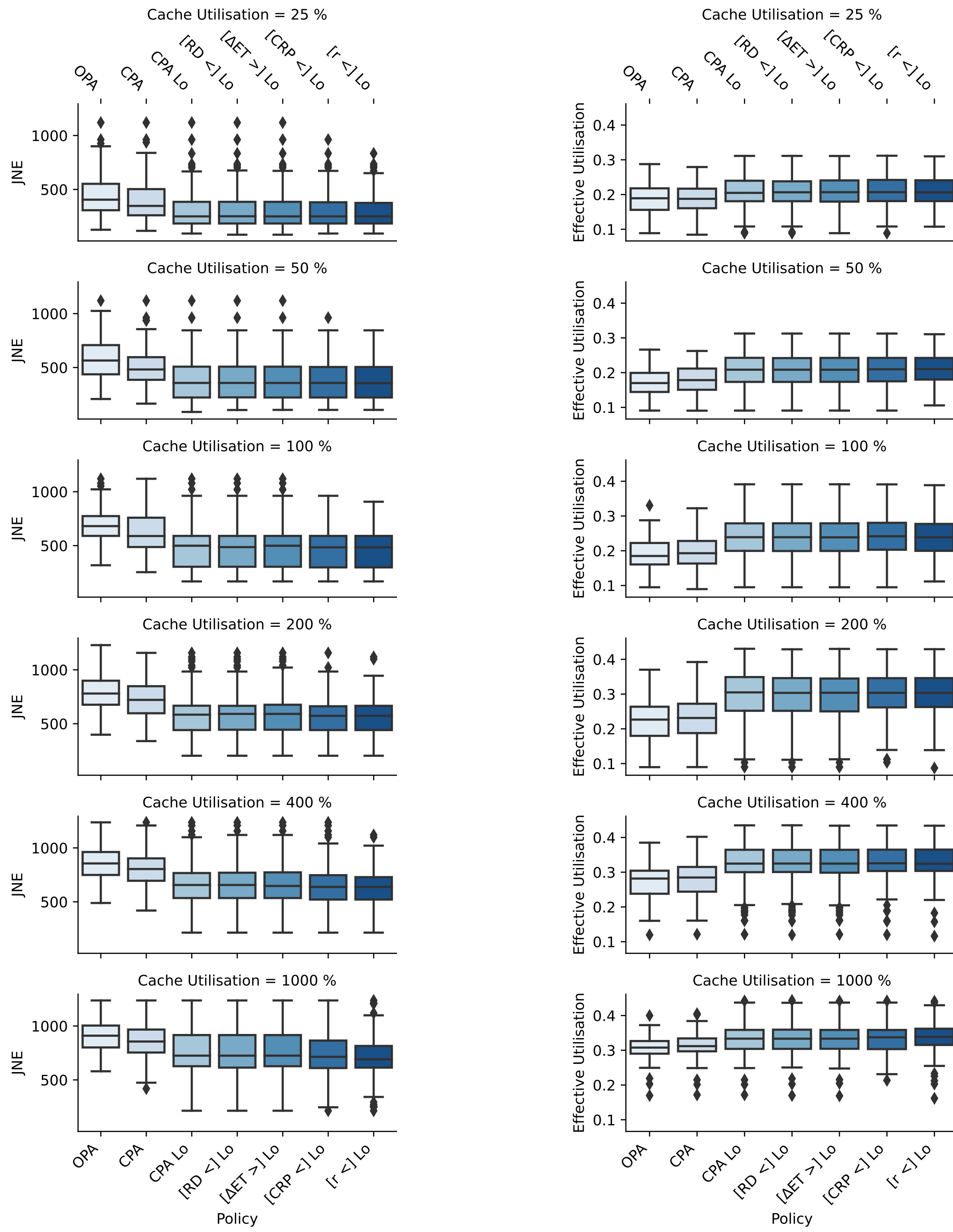
Benchmark	Error (RMSE)	Correlation (Pearson)
statemate	0.069	0.41
ndes	0.057	0.44
adpcm_dec	0.019	0.07
adpcm_enc	0.223	0.06
h264_dec	0.051	0.27
g723_enc	0.071	0.35
gsm_dec	0.012	0.17
anagram	0.036	0.05
gsm_enc	0.026	0.18
ammunition	0.008	0.15
mpeg2	0.005	0.12

Table 4: Metrics for the quality of the CRP model

The *CRP* model manages to provide some relatively accurate predictions across most benchmarks, with provision for a simple error model. If the *RD* metric captures some of the stress on the memory hierarchy, there is still room for improvement on our profiling platform, i.e. additional metrics to explain further variations in timings. As expected smaller benchmarks seem to benefit the most from close locality between jobs.

5.3 Scheduling impact of CRP-based policies

We evaluate the impact in this section of scheduling policies built considering the *CRP* model, based either on the locality principles underlying the model or calling the model at runtime to improve scheduling decisions. The evaluation focuses on the service granted



(a) JNE across varying cache utilisation

(b) Effective Utilisation across varying cache utilisation

Figure 3: Comparison of the scheduling policies with task sets composed of 100 tasks, 20% *HI* tasks, 30% utilisation.

to *LO* criticality tasks by measuring the number of jobs not executed (JNE), and the overall impact of the policies on the effective system utilisation. We also consider a *LO* variant of policies where applicable which first prioritises *LO* criticality tasks within a priority group.

The explored task generation parameters also cover a blend of configurations, from realistic, challenging task sets to more manageable ones. Due to space constraints, we can only present a limited number of them highlighting common trends, benefits, and limits

of the proposed policies. We simulated all proposed policy combinations and variants, but only present the ones providing interesting grounds for discussion notably if a variant strictly dominates others.

The following policies have been selected:

- OPA: a baseline priority allocation, with fixed, arbitrary ordering of tasks within a priority group;
- CPA and CPA LO: static priority allocation, based on CRP intuitions, with no shared priority; and
- $[RD <]LO$, $[\Delta\hat{ET} >]LO$, $[CRP <]LO$, and $[r <]LO$: dynamic scheduling policies, reordering tasks within a priority group at runtime.

We first consider the impact of increased cache utilisation on the CRP-based scheduling policies in Figure 3. The cache utilisation is the ratio of unique cache blocks used by the task set over the overall cache size, with a lower utilisation implying less stress in the memory hierarchy and an increased likelihood to benefit from resilient cache blocks. Conversely, a high cache utilisation is more likely to result in a single task evicting all residual cache blocks, thus forcing interleaved jobs to their WCET. We focus in Figure 3 on task sets composed of 100 tasks, with a 30% target utilisation for both LO and HI modes, of which 20% are HI tasks.

As tasks' cache utilisation increases, generating more stress on the memory hierarchy, it takes only a few jobs to completely flush residual cache blocks out of the cache. Therefore tasks are less likely to benefit from the residual cache blocks and therefore the proposed policies are less beneficial. As a result, increased cache utilisation leads to an increased number of jobs not executed and a more uniform behaviour across all policies. The $[CRP <]LO$ and $[r <]LO$ outperform all others in our observations, especially at high cache utilisation. However, while the $[CRP <]LO$ and $[r <]LO$ manage to complete a similar number of LO criticality jobs, the latter tends to do so more efficiently resulting in a lower effective utilisation (see Figure 3b). This highlights a balance between quality of service and performance.

The initial ordering matters mostly for $[r <]LO$. The policy exploits existing stress contribution to prioritise tasks, favouring jobs whose contribution to stress has already been accounted. Any difference introduced during the initial release of tasks is overtaken by the execution time variance accounted for by the $[\Delta\hat{ET} >]$ and $[CRP <]$ policies. Under our constraints, prioritising LO criticality jobs is always beneficial. The additional stress for a HI criticality job is balanced by the opportunity to complete LO jobs, especially if the HI criticality job was to overrun its C^{LO} anyway and cause a criticality mode change.

Looking at the impact of increased system utilisation in Figure 4, we observe similar trends to those observed with increasing cache utilisation. Figure 4 focuses on task sets composed of 20 tasks, with a 40% portion of HI tasks, and a 50% cache utilisation. Increased system utilisation does result in less completed LO criticality jobs. Although cache utilisation is below 100%, stress on the memory hierarchy still impacts the execution of sensitive tasks. The model and task generation thus account for capacity conflicts where multiple cache blocks might compete for a same, limited portion of the memory hierarchy. The $[r <]LO$ policy still slightly outperforms the others, although with only marginal benefits at a high system utilisation (80%). The heuristic $[\Delta\hat{ET} >]LO$ which relies on the interpretation of the CRP model at runtime currently focuses on the short-term benefits of running a task first, ignoring its impact on the memory hierarchy and future jobs.

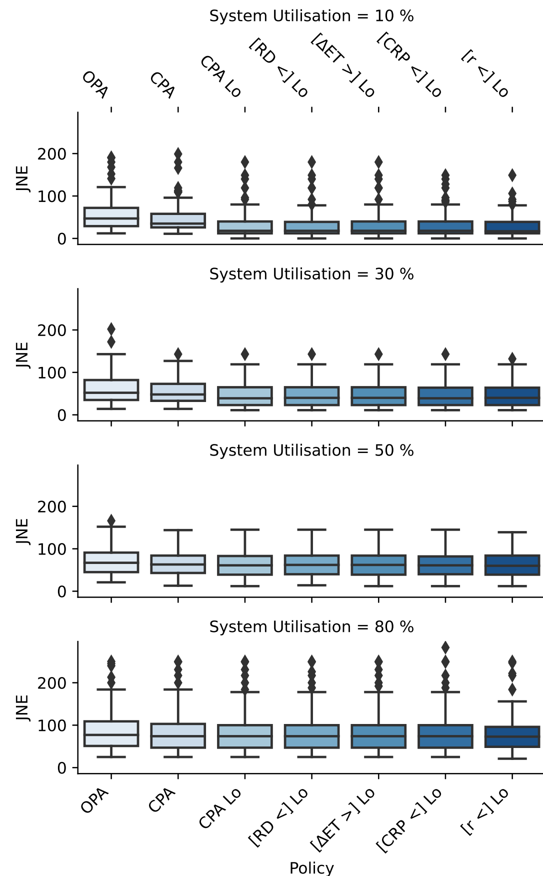


Figure 4: Comparison of the scheduling policies across varying target utilisation with task sets composed of 20 tasks, 40% *HI* tasks, and 50% cache utilisation.

6 RELATED WORK

The concept of a stress and sensitivity model to capture variations in execution time induced and suffered by a task was explored in [14, 16]. Our approach relies on those concepts with some important differences: (1) we focus on a single core architecture, using resources shared by jobs of different tasks; (2) our model produces scaling factors as opposed to additive timing variations; (3) the RD stress metric does not capture an execution time penalty on others; (4) our model is not aimed towards refining schedulability analyses but inform the scheduler; and (5) the proposed model encompasses multiple resources in the memory hierarchy at once.

A number of studies consider the definition of execution time [4, 22] and stress [12, 17, 21] models for shared resources through the abstraction of tasks or their contenders to observable factors. Hardware-level performance counters do characterise the low-level behaviour of a task [4, 17, 21, 22]. The number of performance counters is often limited, therefore even with careful selection, the

accuracy of any model is limited. We favour a high-level *RD* stress metric easily monitored at runtime and profiled offline.

The work by Courtaud et al. [12] in particular relies on an offline profiling phase to build a high-resolution, qualitative characterisation of tasks. The profile of a task provides the grounds for predicting the worst-case overheads if suffers as a result of memory contention. Our approach targets a less accurate but wider scope sensitivity model to inform online scheduling decisions, as opposed to worst-case estimates.

The *CRP* solves a question similar to the Cache Related Preemption Delay (CRPD) problem [1, 27, 28, 31], that is what is the impact on a task caused by interleaved jobs evicting useful cache blocks (UCBs) from the memory hierarchy. However, CRPD computation, assuming limited preemption regions or not, requires low-level hardware models and analyses to capture the intersection of UCBs and evicting cache blocks.

Early work on cache-aware scheduling for single core in [25] also relies on such a low-level model to compute the cache contents after each task, and its impact on the next job. More recent techniques do account for the sensitivity of tasks to co-schedule the ones which share common resources [2, 3], or favour tasks based on profiled [9] or monitored [10] stress metrics. However, none of these approaches relies on an execution time model deployed at runtime.

The same intuition that consecutive jobs benefit from cache reuse is considered in [11], where periodic schedules are optimised as part of a co-design problem. The cache model is however relatively simple such that timing improvements are purely based on the last executed job.

7 CONCLUSION

In this work, we introduce the concept of *CRP*, a stress and sensitivity model, to capture the impact of a task on the memory hierarchy and the variations of its execution time due to the impact of others. The *CRP* models the benefits of cache blocks in the memory hierarchy leftover from previous jobs. The model successfully captures some of the execution time variability, especially for smaller, cache-dependent tasks, in response to the high-level *RD* stress metric. Longer or streaming tasks however are more sensitive to internal conflicts and would require additional stress metrics to capture variability.

We further propose a number of cache-aware scheduling heuristics, building on the properties or the deployment at runtime of the *CRP* model, and assess how they benefit the overall quality of service of a mixed-criticality system. Our evaluation shows that more informed decisions can result in a better utilisation of the memory hierarchy, leaving more room for low criticality workloads. However, the deployment of the *CRP* at runtime requires some consideration to balance the expected benefits of a job with its impact on subsequent ones.

We consider the *CRP* model in the context of multi-core architectures in [33], to inform the allocation of jobs to cores, to exploit locality in the memory hierarchy. Future work and refinements to the *CRP* model could help include the effects of other sources of interference on the memory hierarchy, especially layers shared concurrently across cores.

REFERENCES

[1] Sebastian Altmeyer and Claire Maiza Burguière. 2011. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *Journal of Systems Architecture* 57, 7 (2011), 707–719. https://doi.org/10.1016/j.sysarc.2010.08.006

[2] James H Anderson and John M Calandrino. 2006. Parallel real-time task scheduling on multicore platforms. In *27th International Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, USA, 89–100.

[3] James H. Anderson, John M. Calandrino, and UmaMaheswari C. Devi. 2006. Real-Time Scheduling on Multicore Platforms. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '06)*. IEEE Computer Society, USA, 179–190. https://doi.org/10.1109/RTAS.2006.35

[4] Tadeu Nogueira C. Andrade, George Lima, Veronica Maria Cadena Lima, Yasmina Abdeddaïm, and Liliana Cucu Grosjean. 2021. On the Selection of Relevant Hardware Events for Explaining Execution Time Behavior. In *SBESC 2021 - XI Brazilian Symposium on Computing Systems Engineering*. IEEE.

[5] N. C. Audsley. 1991. *Optimal Priority Assignment and Feasibility of Static Priority Tasks With Arbitrary Start Times.* Technical Report YCS-164. Department of Computer Science, University of York.

[6] N. C. Audsley. 2001. On Priority Assignment in Fixed Priority Scheduling. *Inform. Process. Lett.* 79, 1 (2001), 39–44.

[7] S.K. Baruah, A. Burns, and R.I. Davis. 2011. Response-Time Analysis for Mixed Criticality Systems. In *2011 IEEE 32nd Real-Time Systems Symposium*. IEEE Computer Society, USA, 34–43. https://doi.org/10.1109/RTSS.2011.12

[8] Derek L. Bruening and Saman Amarasinghe. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation.* Ph. D. Dissertation. Massachusetts Institute of Technology, USA.

[9] John M Calandrino and James H Anderson. 2008. Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study. In *20th Euromicro Conference on Real-Time Systems*. IEEE Computer Society, USA, 299–308.

[10] John M Calandrino and James H Anderson. 2009. On the design and implementation of a cache-aware multicore real-time scheduler. In *21st Euromicro Conference on Real-Time Systems (ECRTS '09)*. IEEE Computer Society, USA, 194–204.

[11] Wanli Chang, Debayan Roy, Xiaobo Sharon Hu, and Samarjit Chakraborty. 2018. Cache-aware task scheduling for maximizing control performance. In *Design, Automation & Test in Europe Conference & Exhibition*. European Design and Automation Association, Leuven, BEL, 694–699.

[12] Cédric Courtaud, Julien Sopena, Gilles Muller, and Daniel Gracia Pérez. 2019. Improving Prediction Accuracy of Memory Interferences for Multicore Platforms. In *IEEE Real-Time Systems Symposium (RTSS)*. IEEE.

[13] Xiaotian Dai, Shuai Zhao, Benjamin Lesage, and Iain Bate. 2022. Using Digital Twins in the Development of Complex Dependable Real-Time Embedded Systems. In *Leveraging Applications of Formal Methods, Verification and Validation. Practice - 11th International Symposium, ISoLA (Lecture Notes in Computer Science, Vol. 13704)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, 37–53.

[14] R. Davis, D. Griffin, and I. Bate. 2021. Schedulability Analysis for Multi-Core Systems Accounting for Resource Stress and Sensitivity. In *Euromicro Conference on Real-Time Systems (LIPIcs, Vol. 196)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 7:1–7:26.

[15] Robert I. Davis and Alan Burns. 2011. Improved Priority Assignment for Global Fixed Priority Pre-Emptive Scheduling in Multiprocessor Real-Time Systems. *Real-Time Systems* 47, 1 (Jan 2011), 1–40.

[16] Robert I Davis, David Griffin, and Iain Bate. 2022. A framework for multi-core schedulability analysis accounting for resource stress and sensitivity. *Real-Time Systems* 58 (2022), 53 pages.

[17] Enrique Díaz, Enrico Mezzetti, Leonidas Kosmidis, Jaume Abella, and Francisco J. Cazorla. 2018. Modelling Multicore Contention on the AURIX™ TC27x. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*.

[18] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. 2016. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Dagstuhl, Germany, 2.1–2.10.

[19] S. Gil, I. Bate, G. Lima, L. Santinelli, A. Gogonel, and L. Cucu-Grosjean. 2017. Open Challenges for Probabilistic Measurement-Based Worst-Case Execution Time. *Embedded Systems Letters* 9, 3 (2017), 69 – 72.

[20] David Griffin, Iain Bate, and Robert I. Davis. 2020. Generating Utilization Vectors for the Systematic Evaluation of Schedulability Tests. In *2020 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, IEEE Computer Society, USA, 76–88.

[21] David Griffin, Benjamin Lesage, Iain Bate, Frank Soboczenski, and Robert I. Davis. 2017. Forecast-Based Interference: Modelling Multicore Interference from Observable Factors. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems* (Grenoble, France) *(RTNS '17)*. Association for Computing Machinery, New York, NY, USA, 10 pages.

[22] Thomas Huybrechts, Siegfried Mercelis, and Peter Hellinckx. 2018. A New Hybrid Approach on WCET Analysis for Real-Time Systems Using Machine Learning. In *18th International Workshop on Worst-Case Execution Time Analysis (WCET)*.

[23] Rob J. Hyndman and Anne B. Koehler. 2006. Another look at measures of forecast accuracy. *International Journal of Forecasting* 22, 4 (2006), 679–688. https://doi.org/10.1016/j.ijforecast.2006.03.001

[24] Charles F. Jekel and Gerhard Venter. 2019. *pwlf: A Python Library for Fitting 1D Continuous Piecewise Linear Functions.* Jekel, Charles F. https://github.com/cjekel/piecewise_linear_fit_py

[25] Daniel Kästner and Stephan Thesing. 1999. Cache Aware Pre-Runtime Scheduling. *Real-Time Syst.* 17, 2–3 (dec 1999), 235–256.

[26] R.J. Larsen and M.L. Marx. 2012. *An Introduction to Mathematical Statistics and Its Applications.* Prentice Hall, USA.

[27] Will Lunniss, Robert I. Davis, Claire Maiza, and Sebastian Altmeyer. 2013. Integrating Cache Related Pre-Emption Delay Analysis into EDF Scheduling. In *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS '13).* IEEE Computer Society, USA.

[28] Filip Markovic, Jan Carlson, and Radu Dobrin. 2017. Tightening the Bounds on Cache-Related Preemption Delay in Fixed Preemption Point Scheduling. In *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, Vol. 57.

[29] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07).* 89–100. https://doi.org/10.1145/1250734.1250746

[30] Pavlos Petoumenos, Georgios Keramidas, Hakan Zeffer, Stefanos Kaxiras, and Erik Hagersten. 2006. Modeling Cache Sharing on Chip Multiprocessor Architectures. In *International Symposium on Workload Characterization.*

[31] Syed Aftab Rashid, Muhammad Ali Awan, Pedro F. Souto, Konstantinos Bletsas, and Eduardo Tovar. 2022. Cache-Aware Schedulability Analysis of PREM Compliant Tasks. In *Proceedings of the 2022 Conference & Exhibition on Design, Automation & Test in Europe* (Antwerp, Belgium) *(DATE '22).* European Design and Automation Association, Leuven, BEL, 1269–1274.

[32] C. Spearman. 1904. The Proof and Measurement of Association between Two Things. *The American Journal of Psychology* 15, 1 (1904), 72–101.

[33] Shuai Zhao, Xiaotian Dai, Benjamin Lesage, and Iain Bate. 2023. Cache-Aware Allocation of Parallel Jobs on Multi-cores based on Learned Recency. In *Proceedings of the 31th International Conference on Real-Time Networks and Systems.*

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009