PAStime: Progress-Aware Scheduling for Time-Critical Computing

Soham Sinha
Department of Computer Science, Boston University, MA, USA
soham1@bu.edu

Richard West
Department of Computer Science, Boston University, MA, USA
richwest@bu.edu

Ahmad Golchin
Department of Computer Science, Boston University, MA, USA
golchin@bu.edu

Abstract

Over-estimation of worst-case execution times (WCETs) of real-time tasks leads to poor resource utilization. In a mixed-criticality system (MCS), the over-provisioning of CPU time to accommodate the WCETs of highly critical tasks may lead to degraded service for less critical tasks. In this paper we present PAStime, a novel approach to monitor and adapt the runtime progress of highly time-critical applications, to allow for improved service to lower criticality tasks. In PAStime, CPU time is allocated to time-critical tasks according to the delays they experience as they progress through their control flow graphs. This ensures that as much time as possible is made available to improve the Quality-of-Service of less critical tasks, while high-criticality tasks are compensated after their delays.

This paper describes the integration of PAStime with Adaptive Mixed-criticality (AMC) scheduling. The LO-mode budget of a high-criticality task is adjusted according to the delay observed at execution checkpoints. This is the first implementation of AMC in the scheduling framework of LITMUS^{RT}, which is extended with our PAStime runtime policy and tested with real-time Linux applications such as object classification and detection. We observe in our experimental evaluation that AMC-PAStime significantly improves the utilization of the low-criticality tasks while guaranteeing service to high-criticality tasks.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Software and its engineering → Real-time systems software

Keywords and phrases progress-aware scheduling, code instrumentation, timing annotation

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.3

Funding This work is supported in part by the National Science Foundation (NSF) under Grant #1527050.

Acknowledgements Special thanks to Dr. Ramesh Peri and Intel for generous support and feedback. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

1 Introduction

In real-time systems, computing resources are typically allocated according to each task's worst-case execution time (WCET), to ensure timing constraints are met. However, worst-case conditions for an application are rather rare, resulting in poor resource utilization. Previous research work [54] shows that the worst-cases lie at the tiny tail-end of the probability distribution curve of the execution times for many programs. Instead, average-case execution times are significantly more likely, taking a fraction of the worst-case times.

Mixed-criticality systems (MCSs) provide a way to avoid over-estimation of resource needs, by considering the schedulability of tasks according to different estimates of their execution times at different criticality or assurance levels [52]. Higher criticality tasks are afforded more execution time at the cost of less time for lower criticality tasks, when it is impossible to meet all task timing constraints. There have been multiple proposals [6,7,12,15] since Vestal's work on MCSs [52]. Most prior work focuses on meeting task deadlines and ignores other Quality-of-Service (QoS) metrics [51] or average utilization. Although timely completion is most important for high-criticality applications, QoS is a significant metric for lower criticality tasks [11,36,48]. This has motivated our work on PAStime (**P**rogress-**A**ware **S**cheduling for *time*-critical computing), to maximize the QoS for low-criticality tasks.

In PAStime, we first identify a checkpoint in a high-criticality application at an intermediate stage in its source code. The application is then profiled offline to measure the time to reach the marked checkpoint. Using this timing data, the application evaluates its progress at the checkpoint during runtime. Based on the delay at the checkpoint, PAStime predicts the expected execution time of a high-criticality application. We consequently adjust the runtime of the application, given that the change does not hamper schedulability of co-running tasks. If at runtime a highly critical program is deemed to be making insufficient progress, it is given greater CPU time.

We combine PAStime with Adaptive Mixed-criticality (AMC) scheduling [7], to improve the performance of low-criticality tasks. In AMC, the system is started in *LO-mode*, where all tasks are scheduled with their LO-mode budgets. When a high-criticality task runs for more than its LO-mode budget, the system is switched to *HI-mode*. In HI-mode, all low-criticality tasks are stopped, and the high-criticality tasks are given their increased HI-mode budgets. However, switching to HI-mode should be avoided as much as possible [5], because it affects the performance of low-criticality tasks, which are not executed in HI-mode.

Several works extend the mixed-criticality task model to improve the performance of low-criticality tasks, such as providing an offline extra budget allowance to the high-criticality tasks [45], and estimating multiple budgets [35,43] and periods [46,48] for low-criticality tasks. However, these approaches do not utilize runtime information.

We extend AMC with PAStime to avoid mode switches, by dynamically adjusting the LO-mode budget for a high-criticality task, based on progress to execution checkpoints. Then, we predict the expected execution time of a high-criticality task based on the observed delay until a checkpoint. We carry out an efficient online schedulability test to determine whether we can increase the LO-mode budget of the delayed high-criticality task to our predicted execution time. In case the taskset is still schedulable with the increased budget, we extend the LO-mode budget of the high-criticality task to the predicted execution time. When a high-criticality task finishes within its extended budget, we keep the system in LO-mode and avoid a mode switch that would otherwise happen in AMC. Thus, PAStime improves the QoS of low-criticality tasks by keeping the system in LO-mode for a longer time.

Factors such as I/O events and hardware microarchitectural delays lead to actual execution times exceeding those predicted by PAStime. Any high criticality task running at the end of its predicted execution time causes a timer interrupt to switch the system into HI-mode, as is the case with AMC scheduling. Thus, a high-criticality task never misses its deadline.

1.1 Contributions

The central idea of PAStime is to help the OS make scheduling decisions based on a program's runtime progress. This work is the first implementation of AMC in the scheduling framework of LITMUS^{RT} [10,14]. Such an implementation helps in testing AMC scheduling with a wide range of real-time Linux applications. We also implement an extension to AMC scheduling with our PAStime runtime policy. We test our implementation with real-world applications: an object classification application from the Darknet neural network framework [44], an object tracking application from the dlib machine learning toolkit [16], and an MPEG video decoder [20]. We show that PAStime increases the average utilization of low-criticality tasks by 1.5 to 9 times for 2 to 20 tasks. We also demonstrate that our implementation of AMC-PAStime has minimal and bounded additional overhead in LITMUS^{RT}.

We provide a C library for PAStime to instrument checkpoints in high-criticality programs. In addition, we modify the LLVM compiler [29] to *automatically* identify potential locations of checkpoints during profiling for simple time-critical applications, and also instrument the checkpoints in the final binary executable file.

The next section describes our approach to PAStime. Section 3 details the theoretical background behind AMC and its extension with PAStime. Section 4 describes the design and implementation of PAStime, which is then evaluated in Section 5. Finally, we describe related work, followed by conclusions and future work in Sections 6 and 7, respectively.

2 Approach

Compiler infrastructures such as LLVM are capable of producing a program's control flow graph (CFG). A CFG represents the interconnection between multiple *Basic Blocks (BBs)*, where a block is a sequence of straight-line code without any internal branches. However, CFGs are not typically utilized by an OS to manage time for different computing resources, in spite of being a rich source of analytical information about a program. Consequently, current OSs are oblivious to a program's computing requirements (e.g., CPU utilization) at different points in its execution. A developer of an application can, instead, help the OS make decisions related to resource management, by providing runtime information about a program at certain points in its source code.

PAStime dynamically decides a program's execution budget based on its runtime progress and theoretical analysis of the allowable delay at specific checkpoint locations. At runtime, PAStime measures the time (i.e., CPU time) to reach a checkpoint from the start of a task, and then compares that time to a pre-profiled reference value. The task's execution budget which was previously set based on profiling, is then adjusted according to actual progress.

Figure 1 shows the CFG for a program with two loops, starting at BB2 and BB6. In this example, PAStime inserts a checkpoint between the two loops at the end of BB5. BB5 is a potentially good location for a checkpoint because there is one loop before and after this BB, providing an opportunity to increase the budget to compensate for the delay until BB5.

Suppose that we derive the LO-mode budget of the whole program to be 2000 ms by profiling, and the LO-mode time to reach the checkpoint at BB5 is 500 ms. The program is then executed in the presence of other tasks. The execution budget at the checkpoint (in BB5) is adjusted, to account for the program's actual runtime progress. For example, suppose the program experiences a delay of 100 ms to reach the checkpoint in BB5, thereby arriving at 600 ms instead of the expected 500 ms. Therefore, the program is delayed by $(\frac{100}{500} \times 100 =)$ 20% from its LO-mode progress.

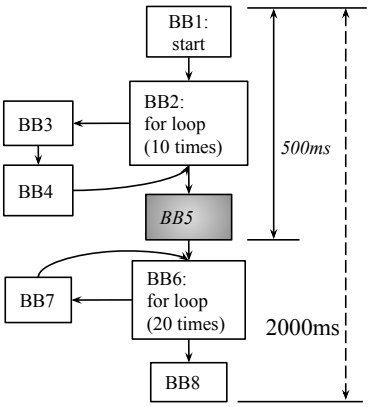


Figure 1 CFG and Average Time Estimates of a Program.

Depending on the relationship between the task's LO- and HI-mode progress to the checkpoint, the task's budget is adjusted according to the 20% observed delay at the checkpoint. One approach is to extrapolate a linear delay from the checkpoint to the end of the task's execution. Thus, the total execution time of 2000 ms is predicted to complete at $(2000 + 20\% \times 2000=)$ 2400 ms. PAStime uses the available information at a checkpoint to extend the LO-mode budget of a high-criticality application. In Section 5.8, we explore other possible execution time prediction models.

2.1 Benefits of Adaptive Mixed-criticality Scheduling

We see progress-aware scheduling as being beneficial in mixed-criticality systems. Adaptive Mixed-criticality scheduling [7] is the state-of-the-art fixed-priority scheduling policy for Mixed-criticality tasksets. In AMC, a system is first initialized to run in *LO-mode*. In LO-mode, all the tasks are executed with their LO-mode execution budgets. Whenever a high-criticality task overruns its LO-mode budget, the system is switched to *HI-mode*. In HI-mode, all low-criticality tasks are discarded (or given reduced execution time [5, 35]), and the high-criticality tasks are given increased HI-mode budgets. The system's switch to HI-mode therefore impacts the QoS for low-criticality tasks.

By combining PAStime with AMC (to yield AMC-PAStime), we extend the LO-mode budget of a high-criticality task to its predicted execution time at a checkpoint. Going back to our previous example in Figure 1, we try to extend the LO-mode budget of the task by 400 ms. We carry out an online schedulability test to determine if we can extend the task's LO-mode budget by 400 ms. If the whole taskset is still schedulable after an extension of the LO-mode budget of the delayed high-criticality task, the increment in the task's LO-mode budget is approved. We let the task run until the extended time and keep the system in LO-mode. In case the high-criticality task finishes within its extended time, we avoid an unnecessary switch to HI-mode. Thus, the low-criticality tasks run for an extended period of time and do not suffer from degraded CPU utilization and QoS, as occurs with AMC.

In case the task does not finish within the predicted time, then the system is changed to HI-mode, and low-criticality tasks are finally dropped. This behavior is identical to AMC, and every high-criticality task still finishes within its own deadline. Therefore, we improve the QoS of the low-criticality tasks when the high-criticality tasks finish within their extended LO-mode budgets, and otherwise, we fall back to AMC. When there is no delay at a checkpoint, we do not change a task's LO-mode budget.

3 Theoretical Background

In this section, we provide a response time analysis for AMC-PAStime by extending the analysis for AMC scheduling. We also describe details about the online schedulability test based on the response time values.

3.1 Task Model

We use the same AMC task model as described by Baruah et al [7]. Without loss of generality, we restrict ourselves to two criticality levels - LO and HI. Each task, τ_i, has five parameters: $C_i(LO)$ - LO-mode runtime budget, $C_i(HI)$ - HI-mode runtime budget, D_i - deadline, T_i - period, and L_i - criticality level of a task, which is either high (HC) or low (LC). We assume each task's deadline, D_i, is equal to its period, T_i. A HC task has two budgets: $C_i(LO)$ for LO-mode assurance and $C_i(HI)$ ($> C_i(LO)$) for HI-mode assurance. A LC task has only one budget of $C_i(LO)$ for LO-mode assurance.

3.2 Scheduling Policy

Both AMC and AMC-PAStime use the same task priority ordering, based on Audsley's priority assignment algorithm [2]. If a task's response time for a given priority order is less than its period, then the task is deemed schedulable. The details of the priority assignment strategy are discussed in previous research work [3, 7]. We do not change the priority ordering of the tasks at runtime.

AMC-PAStime initializes a system in LO-mode with all tasks assigned their LO-mode budgets. We extend a high-criticality task's LO-mode budget at a checkpoint if the task is lagging behind its expected progress, as long as the extension does not hamper the schedulability of the delayed task and all the lower or equal priority tasks. An increase to the LO-mode budget of a task that violates its own or other task schedulability is not allowed. If a high-criticality task has not finished its execution even after its extended LO-mode budget, then the system is switched to HI-mode.

3.3 Response Time Analysis

The AMC response time recurrence equations for (1) all tasks in LO-mode, (2) HC tasks in HI-mode, and (3) HC tasks during mode-switches are shown in Equation 1, 2 and 3, respectively. $hp(i)$ is the set of tasks with priorities higher than or equal to that of τ_i. Likewise, $hpHC(i)$ and $hpLC(i)$ are the set of high- and low-criticality tasks, respectively, with priorities higher than or equal to the priority of τ_i.

AMC provides two analyses for mode switches: AMC-response-time-bound (AMC-rtb) and AMC-maximum. We use AMC-rtb for our analysis, as AMC-maximum is computationally more expensive. However, AMC-rtb does not allow a taskset which is not schedulable by AMC-maximum. Therefore, AMC-rtb is *sufficient* for schedulability.

$$R_i^{LO} = C_i(LO) + \sum_{\tau_j \in hp(i)} \lceil \frac{R_i^{LO}}{T_j} \rceil \times C_j(LO) \tag{1}$$

$$R_i^{HI} = C_i(HI) + \sum_{\tau_j \in hpHC(i)} \lceil \frac{R_i^{HI}}{T_j} \rceil \times C_j(HI) \tag{2}$$

$$R_i^* = C_i(HI) + \sum_{\tau_j \in hpHC(i)} \lceil \frac{R_i^*}{T_j} \rceil \times C_j(HI) + \sum_{\tau_j \in hpLC(i)} \lceil \frac{R_i^{LO}}{T_j} \rceil \times C_j(LO) \quad (3)$$

With AMC-PAStime, we not only measure $C_i(LO)$ and $C_i(HI)$, but also the LO-mode time to reach a checkpoint $C_i^{CP}(LO)$. If a high-criticality task, τ_i, is delayed by $X\%$ at a checkpoint, compared to its LO-mode progress, then τ_i takes $(C_i^{CP}(LO) + \frac{C_i^{CP}(LO) \times X}{100})$ time to reach the checkpoint. Hence, τ_i's budget is tentatively increased from $C_i(LO)$ to $C_i'(LO)$, where $C_i'(LO) = f(C_i(LO), X)$. Here, $f(C_i(LO), X)$ is a function to predict the delayed total execution time, given that the observed delay until the checkpoint is $X\%$. For example, by a linear extrapolation of the observed delay of $X\%$ at a checkpoint, the original budget $C_i(LO)$ changes to $f(C_i(LO), X) = \big(C_i(LO) + \frac{C_i(LO) \times X}{100}\big)$. It is possible for f to depend on task-specific and hardware microarchitectural factors such as cache and main memory accesses. We show more execution time prediction techniques in Section 5.8.

With the increased LO-mode budget $C_i'(LO)$, an online schedulability test then calculates the the extended LO-mode response time, $R_i^{LO\text{-ext}}$, for τ_i, using Equation 4. Similarly, $R_i^{*\text{-ext}}$ is calculated using Equation 5. Equations 4 and 5 are extensions of Equations 1 and 3. AMC-PAStime checks at runtime whether both $R_i^{LO\text{-ext}}$ and $R_i^{*\text{-ext}}$ are less than or equal to τ_i's period to determine its schedulability.

The new response times are then calculated for all tasks in LO-mode with priorities less than or equal to τ_i, using Equation 4. Similarly, new response times are calculated for all HC tasks with lower or equal priority to τ_i during a mode switch, using Equation 5. If all newly calculated response times are less than or equal to the respective task periods, the system is schedulable. In this case, AMC-PAStime approves the LO-mode budget extension to τ_i. If the system is not schedulable, then τ_i's budget remains $C_i(LO)$.

$$R_i^{LO\text{-ext}} = C_i'(LO) + \sum_{\tau_j \in hp(i)} \lceil \frac{R_i^{LO\text{-ext}}}{T_j} \rceil \times C_j'(LO) \quad (4)$$

$$R_i^{*\text{-ext}} = C_i(HI) + \sum_{\tau_j \in hpHC(i)} \lceil \frac{R_i^{*\text{-ext}}}{T_j} \rceil \times C_j(HI) \quad + \sum_{\tau_j \in hpLC(i)} \lceil \frac{R_i^{LO\text{-ext}}}{T_j} \rceil \times C_j(LO) \quad (5)$$

AMC-PAStime only extends the LO-mode budget of a delayed HC task for its current job. When a new job for the same HC task is dispatched, it starts with its original LO-mode budget. If another request for an extension in LO-mode for the same task appears, AMC-PAStime tests the schedulability with the maximum among the requested extended budgets. The system keeps track of the maximum extended budget for a task and uses that value for online schedulability testing. We explain the AMC-PAStime scheduling scheme with an example taskset in Table 1.

Table 1 A Mixed-criticality Taskset Example.

Task	Type	C(LO)	C(HI)	T	Pr	R^{LO}	R^*
τ_1	HC	3	6	10	1	3	6
τ_2	LC	2	-	9	2	5	-
τ_3	HC	5	10	50	3	15	38

Suppose, task τ_1 is delayed by 66% at a checkpoint in the task's source code. PAStime will then try to extend the budget by $(3 \times \frac{66}{100}) \approx 2$ time units. Therefore, the potential extended budget $C_1'(LO)$ for τ_1 would be $(3 + 2) = 5$ time units. PAStime will calculate the

response times, $R_i^{LO\text{-ext}}$ and $R_i^{*\text{-ext}}$, for τ_1 and the lower priority tasks τ_2 and τ_3. $R_1^{LO\text{-ext}}$ would just be 5, and $R_1^{*\text{-ext}}$ would remain the same as $R_1^* = 6$, as τ_1 is the highest priority task. The new $R_2^{LO\text{-ext}}$ would be 7 (by Equation 4) which is smaller than its period of 9. Therefore, τ_2 would still be schedulable if we extend τ_1's LO-mode budget from 3 to 5.

For τ_3, the new $R_3^{LO\text{-ext}}$ and $R_3^{*\text{-ext}}$ would be, respectively, 26 (by Equation 4) and 40 (by Equation 5) which are also smaller than τ_3's period of 50. Therefore, the extended budget of 5 for τ_1 would be approved by AMC-PAStime. In conventional AMC scheduling, the system would be switched to HI-mode if τ_1 did not finish within 3 time units. However, AMC-PAStime will extend τ_1's LO-mode budget to 5 because of the observed delay at its checkpoint, so the system is kept in LO-mode. Consequently, LC task τ_2 is allowed to run by AMC-PAStime, if τ_1 finishes before 5 time units. If τ_1 does not finish even after 5 time units, the system would be switched to HI-mode.

In this example, 5 jobs of τ_1 are dispatched for every single job of τ_3, as τ_3's period of 50 is 5 times the period of τ_1. Suppose τ_1 asks for the 66% increment in its LO-mode budget for the first job, as we have explained above. In its second job, τ_1 asks for an increment of 33% (1 time unit) in its LO-mode budget. In this case, we again need to calculate the response times for all tasks. If we calculate the online response times for τ_2 and τ_3 assuming $(3 + 1) = 4$ time units for $C_1'(LO)$, then we would not account for the first job of τ_1, which potentially executes for 5 time units. We would need to keep track of the extended LO-mode budgets for all jobs of τ_1, to accurately calculate online response times for τ_2 and τ_3.

To avoid the cost of recording all extended LO-mode budgets for a task, AMC-PAStime simply stores the maximum extended budget for a task. When calculating online response times to check whether to approve an extension to the LO-mode budget, the system uses the maximum extended budget of every high-criticality task. This value is stored in the `max_extended_budget` variable for each HC task. Therefore, when τ_1 asks for 4 time units as its extended LO-mode budget in its second job, the system calculates $R_1^{LO\text{-ext}}$, $R_1^{*\text{-ext}}$, $R_2^{LO\text{-ext}}$, $R_3^{LO\text{-ext}}$, $R_3^{*\text{-ext}}$ with $C_1'(LO) = 5$.

AMC-PAStime uses maximum extended budgets to calculate safe upper bounds for online response times. The `max_extended_budget` task property is reset when a task has not requested a LO-mode budget extension for any of its dispatched jobs within the maximum period of all tasks.

3.4 Online Schedulability Test

AMC-PAStime performs an online schedulability test whenever a high-criticality task asks for an extension to its LO-mode budget. The test calculates the response times ($R_i^{LO\text{-ext}}$ and $R_i^{*\text{-ext}}$) of the delayed high-criticality task and all lower priority tasks. Then, it checks whether the response times are less than or equal to the task periods. If any task's response time is greater than its period, the online schedulability test returns false, and the extension in LO-mode for the high-criticality task is denied. If the schedulability test is successful the high-criticality task is permitted to run for its extended budget in LO-mode.

Algorithm 1 shows the pseudocode for the online schedulability test. The offline response time values (R_i^{LO} and R_i^*) are stored in the properties for each task τ_i. When a high-criticality task τ_k is delayed, it asks for an extension of its LO-mode budget by e time units.

Line 6 in Algorithm 1 determines whether the newly requested extension, e, is more than a previously saved maximum extended budget for τ_k. In lines 8–11, the $C_i'(LO)$ is set to the maximum extended budget for all the lower priority tasks and τ_k. As we have explained above, it is practically infeasible to store the execution times of every job of all the tasks to calculate online response times. Therefore, we store the maximum extended budget of

Algorithm 1 Online Schedulability Test.

```
1:  Input: tasks - set of all tasks in priority order
2:         τ_k - delayed task
3:         e - extra budget for τ_k
4:  Output: true or false
5:  function IsBudgetChangeApproved(tasks, τ_k, e)
6:      e' = max(τ_k.max_extended_budget, C_k(LO) + e) − C_k(LO)
7:      for each task τ_i in {τ_k ∪ lower priority tasks than τ_k in tasks} do
8:          C'_i(LO) = τ_i.max_extended_budget
9:          if τ_i is τ_k then
10:             C'_i(LO) =max(C'_i(LO), C_i(LO) + e)
11:         end if
12:         Initialize R_i^{LO-ext} for Equation 4 with R_i^{LO} + e'
13:         Solve Equation 4
14:         if R_i^{LO-ext} ≤ T_i then
15:             if τ_i is high-criticality then
16:                 Initialize R_i^{*-ext} for Equation 4 with R_i^*
17:                 Solve Equation 5 with the new R_i^{LO-ext}
18:                 if R_i^{*-ext} > T_i then  Return false
19:                 end if
20:             end if
21:         else Return false
22:         end if
23:     end for
24:     τ_k.max_extended_budget = max(C_k(LO) + e, τ_k.max_extended_budget)
25:     Return true
26: end function
```

every task and use this to calculate response times online. For the currently delayed task, τ_k, $C'_k(LO)$ is set to the maximum value between a previously saved `max_extended_budget` and the currently requested $C_k(LO) + e$. Considering the example in Table 1 from the paper, line 10 would translate to $C'_i(LO)$=max(5, 4) for the second LO-mode extension request.

Then, Equation 4 is solved in line 13 by initializing $R_i^{LO\text{-ext}}$ to the R_i^{LO} plus extra budget e' from line 6. If a lower priority task is a high-criticality task, then $R_i^{*\text{-ext}}$ is calculated in line 17, with the newly derived value of $R_i^{LO\text{-ext}}$.

Online response time calculations may take significant time, depending on the number of iterations of Equations 4 and 5. However, the response time values from Equations 1, 2, 3 are already calculated offline to determine the schedulability of a taskset using Audsley's priority assignment algorithm [2] with AMC scheduling. Since AMC-PAStime uses the same priority ordering as AMC, the offline response times for schedulability remain the same.

AMC-PAStime initializes the online $R_i^{LO\text{-ext}}$ in Equation 4 with $R_i^{LO} + e$, where R_i^{LO} is calculated offline by Equation 1 and $e(> 0)$ is the extra budget of the delayed task.

Since the budget of a delayed task is extended by e, $R_i^{LO\text{-ext}}$ must be greater than or equal to $R_i^{LO} + e$. Hence, this is a good initial value to start calculating $R_i^{LO\text{-ext}}$ online. In Section 5, we establish an upper bound on the total number of iterations needed to check the schedulability of random tasksets, if the highest priority task's budget is extended by different amounts.

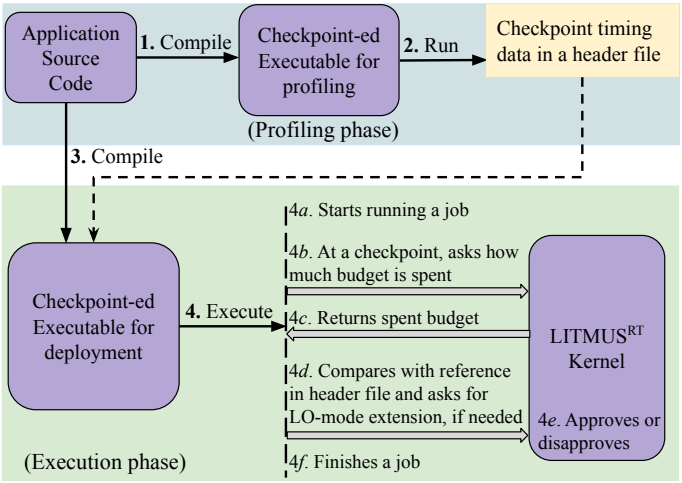


Figure 2 Implementation of AMC-PAStime in LITMUS^{RT}.

4 Design and Implementation of PAStime

In this section, we first describe the overall design of AMC-PAStime in LITMUS^{RT} [10, 14]. This is followed by a description of how checkpoints are instrumented in an application's source code. We then show the algorithm to determine and insert checkpoints, which is integrated into the LLVM compiler infrastructure. A high-criticality task requires profiling to determine the placement of checkpoints, before it is ready for execution with other tasks. We describe how the profiling and execution of a high-criticality task is performed, along with the scheduling mechanism in PAStime. The source code for PAStime in LITMUS^{RT} is publicly available [41].

AMC-PAStime has two phases: a Profiling phase for high-criticality tasks, and an Execution phase. In the Profiling phase, one or more checkpoints are placed at key stages in a program's source code. The *average* time to reach each checkpoint is then measured. After profiling all high-criticality tasks, the system switches into the Execution phase. The time taken to reach each checkpoint in every high-criticality task is observed by the system at runtime. Each observed time is compared against the profiled time to reach the same checkpoint. Any high-criticality task lagging behind its profiled time to a checkpoint is tentatively given increased LO-mode budget, according to the approach described in Section 3.

Figure 2 shows an overview of the design of AMC-PAStime in LITMUS^{RT}. Step 1 is the compilation of a high-criticality application's source code in the Profiling phase, which uses our compilation procedure [41]. The compiled executable has checkpoints embedded into its code for profiling. Step 2 executes the program to generate timing metadata for each checkpoint in a *timeinfo.h* file. Step 2 is performed multiple times with different program inputs to generate an average time to reach each checkpoint.

Step 3 compiles the source code along with the checkpoint timing metadata header file (*timeinfo.h*) for the Execution phase, producing a binary image that is used for deployment under working conditions. Finally, Step 4 runs the code in the Execution phase along with all other tasks. At some point after the system is started, Step 4a starts running a job for a high-criticality task. When a checkpoint is reached, Step 4b asks the LITMUS^{RT} kernel how much budget it has consumed. Step 4c returns the spent budget from the LITMUS^{RT} kernel to the application.

After receiving the spent budget, t_{spent}, the application compares it to the reference timing, t_{ref}, for the checkpoint from the *timeinfo.h* header file. It calculates the extra budget, e, needed in LO-mode, using an execution time prediction model as described later in Section 5.8. If $e > 0$, Step 4d asks LITMUS$^{\text{RT}}$ for extra budget. The AMC-PAStime scheduling policy in the LITMUS$^{\text{RT}}$ kernel runs an online schedulability test. If the test returns true, the LO-mode budget of the current task is extended by e. If the test algorithm returns false, the task budget is not altered. Finally, Step 4f finishes the current job of the running task.

4.1 Checkpoint Instrumentation and Detection

A checkpoint is a key stage in an application's code, used to evaluate the progress of a currently running task. Well placed checkpoints balance the number of instructions that are executed prior to the checkpoint, with those that remain to the next checkpoint or the end of the program. Ideally, there should be a meaningful number of instructions leading up to a checkpoint to determine progress. Likewise, there should be sufficient instructions after a checkpoint to increase the likelihood that a task is adequately compensated for execution delays using an extended budget.

A developer of a high-criticality application finds a potential checkpoint location in the program's source code for its Execution phase, after trying out multiple locations in the Profiling phase. PAStime includes a development library to instrument checkpoints for the two different phases. Additional modifications to the LLVM compiler [29] are used to automatically detect and instrument checkpoints in the Profiling phase.

4.2 Checkpoint Library

We have developed a C library to instrument checkpoints in the two PAStime phases. The main purpose of the library is to generate the necessary checkpoint timing information during the Profiling phase and then request a task's extended LO-mode budget from the LITMUS$^{\text{RT}}$ kernel during the Execution phase.

In the Profiling phase, a `writeTime` function call from our library is inserted into the application code at a desired checkpoint. `writeTime` takes a unique ID for each checkpoint. The function logs the time to reach that checkpoint in the source code since the start of a job. The average of multiple such timing entries is saved in *timeinfo.h* after the Profiling phase.

During the Execution phase, a preprocessor macro called `ANNOUNCE_TIME` is inserted at a checkpoint. This macro obtains the spent budget from the LITMUS$^{\text{RT}}$ kernel via a `get_current_budget` system call. Then, it compares the spent budget with the reference budget from the *timeinfo.h* header file, and calculates the extra budget using an execution time prediction model. If extra budget is needed, the macro makes a `set_rt_task_param` LITMUS$^{\text{RT}}$ system call.

4.3 Manual Checkpoint Instrumentation

A developer may attempt various strategies to identify a key stage [13,22,50] of an application to instrument a checkpoint. The developer uses either the `writeTime` function for the Profiling phase, or the `ANNOUNCE_TIME` macro for the Execution phase to instrument a checkpoint. Checkpoints should generally be avoided inside tight loops. Visiting a checkpoint every loop iteration incurs a small overhead that is accumulated across multiple iterations.

Algorithm 2 Determine and Insert Checkpoints.

```
1: isLoopBefore: identifies if a loop is in the paths from the starting BB to another BB
2: visited: set of already visited BBs in DFS
3: function INSERTCHECKPOINT(function)
4:     startingBB = function.getEntryBlock()
5:     DODFS(startingBB)
6: end function
7: function DODFS(currentBB)
8:     if currentBB is in visited then  return
9:     end if
10:    LoopID = getLoopFor(currentBB)
11:    if LoopID != null then
12:        if isLoopBefore[currentBB] ∧ isLoopHeader(currentBB) then
13:            insert checkpoint before currentBB
14:        end if
15:    end if
16:    insert currentBB in visited
17:    for each s in successors of currentBB do DODFS(s)
18:    end for
19: end function
```

4.4 Automatic Checkpoint Instrumentation

We have written a compiler pass in LLVM to automatically instrument checkpoints for the Profiling phase of PAStime. The instrumented code is run in the Profiling phase, and multiple checkpoint timing information is generated. Finally, the developer chooses one such checkpoint for the Execution phase.

The compiler pass automatically inserts checkpoints in the basic block preceding each loop in a function, except the first loop. The first loop is excluded so that enough instructions are executed before a checkpoint to determine meaningful delays.

For nested loops, we consider only the outer loop. Automatic instrumentation works with only simple program structures and ignores intersecting loops. LLVM's `LoopInfo` analysis identifies only natural loops [37]. We utilize the `LoopInfo` class in our checkpoint instrumentation implementation.

Algorithm 2 identifies checkpoint locations for the Profiling phase. It starts a Depth First Search (DFS) from the starting Basic Block (BB) of a function, by calling `DoDFS` in line 6. The algorithm then checks whether a BB is part of a loop using `LoopInfo`'s `getLoopFor` member function. This function returns a unique `LoopID` for every new loop. A nested inner loop has the same ID as its outer loop. If a BB is not part of a loop, then it returns `null`.

If a BB is part of a loop, line 12 first checks whether there is any loop before the current loop using a dictionary `isLoopBefore`. `isLoopBefore` is pre-populated for every BB in the CFG to indicate whether there is at least one loop seen in the paths from the starting BB to the current BB. To pre-populate `isLoopBefore`, the algorithm checks whether there is a path to a BB from the loop BBs.

Algorithm 2 then verifies that the current BB is a header of a loop using `LoopInfo`'s `isLoopHeader` member function. A header is the entry-point of a natural loop. We insert a checkpoint in the predecessor BB of a header.

Finally, if there is at least one loop before the current header BB, a checkpoint is added before the current BB in line 13. As natural loops have only one header, a checkpoint is avoided for any inner loop within a nested loop. We continue the DFS by marking the current BB as visited.

We have implemented the above algorithm in a compiler pass within LLVM [29], to automatically detect and instrument appropriate function calls as checkpoints in a C language program. This uses our previously described Checkpoint library for the Profiling phase. A developer uses our modified LLVM compiler with their high-criticality application written in C. Our compiler pass operates at the LLVM Intermediate Representation (IR) level. It takes a piece of IR logic as input, figures out the points of interest according to the above algorithm for checkpoints in a particular function, and generates the instrumented IR. These IRs are compiled into executable machine code. As the compiler pass operates at the IR level, it is easily extensible to other high-level languages and back-ends supported by LLVM.

4.5 Profiling and Execution Phases

The Profiling phase of PAStime determines viable checkpoints for use in the Execution phase, and also the LO- and HI-mode budgets for a high-criticality application. A checkpointed program is run multiple times in the Profiling phase against a set of test cases. The program is allowed to have multiple test checkpoints, which are either generated automatically using our modified LLVM compiler, or manually by a developer. Each checkpoint has a unique ID (function ID, BasicBlock ID) given to the checkpoint function call `writeTime`. A Python package then runs the Profiling phase to collect the average times (LO-mode) to reach different checkpoints.

Step 4 in Figure 2 is the start of the Execution phase. One checkpoint from those generated in the Profiling phase for a high-criticality application is instrumented with an `ANNOUNCE_TIME` macro. Although in general it is possible to use multiple checkpoints within the same application in the Execution phase, our experience shows that one is sufficient to improve LO-mode service. Multiple checkpoints add overhead to the task execution. Moreover, later checkpoints account for execution delays that make earlier checkpoints redundant, as long as they are reached before the LO-mode budget expires.

The key issue in deciding on a single checkpoint for the Execution phase is to ensure it is not placed too late in the instruction stream. If it is placed too far into the program code a mode switch may occur before the task's LO-mode budget is extended due to delays. We show in Section 5.6 the effects of using checkpoints at different locations in a program's code.

4.6 LITMUS^{RT} Implementation

As a first step to applying PAStime for use in adaptive mixed-criticality scheduling, we extended LITMUS^{RT} with AMC support. This required modifications to the existing partitioned fixed-priority scheduling policy, to include the following new variables in the task properties: `c_lo`, `c_hi`, `r_lo`, `r_star`, `c_extended`, `max_extended_budget`.

Tasks are divided into low- and high-criticality classes. By default, the HI-mode budget for low-criticality tasks, `c_hi`, is set to zero. If desired, we also support a reduced, non-zero HI-mode execution budget for low-criticality tasks, as discussed in the work on Imprecise Mixed-criticality scheduling [35]. Task priorities are determined offline, along with all response times for a system operating in LO-mode (`r_lo`) and during a mode switch (`r_star`). These parameters are then initialized in the kernel when the system starts executing a set of tasks.

The system is started in LO-mode, with all tasks assigned their `c_lo` budgets. Whenever a high-criticality task is out of its LO-mode budget, an enforcement timer handler (based on Linux's high-resolution timer [33]) is fired, and the system is switched to HI-mode.

For AMC-PAStime, `c_extended` is the extended LO-mode budget for a delayed job of a high-criticality task. An enforcement timer handler is therefore triggered only when a high-criticality task is still unfinished after the depletion of it `c_extended` time.

As Baruah et al. suggest [7], an AMC system switches back to LO-mode when none of the high-criticality tasks have been running for more than their LO-mode budgets. In the case of AMC-PAStime, the system will switch to a lower criticality level when none of the high-criticality tasks have been running for more than their extended LO-mode budgets. A list is used to keep track of which high-criticality tasks complete within their (extended) LO-mode budgets, to determine when to revert to a lower system criticality level.

A task's LO-mode budget is extended by AMC-PAStime by making a `set_rt_task_param` system call inside the `ANNOUNCE_TIME` macro. We have implemented the `task_change_params` callback of a LITMUS^{RT} `sched_plugin` interface to support runtime adjustment of task parameters. In the `task_change_params` callback, the system runs an online schedulability test. If the test returns *true*, the budget extension is approved, and the enforcement timer for the current task is adjusted accordingly. The task's `c_extended` variable is updated, along with the maximum value of its extended budget in `max_extended_budget`.

It is worth noting that a budget extension could be delayed until a mode switch is about to happen. However, this strategy needs the scheduler to save the delay at a checkpoint and later decide about the budget extension at a mode switch point. This approach potentially increases runtime overhead while repeatedly accumulating task delays. Conversely, redundant budget extensions in PAStime are avoided by following a lazy budget extension approach. Such strategies are open to further study.

5 Evaluation

We tested our implementation of AMC and AMC-PAStime in LITMUS^{RT} with real-world applications on an Intel NUC Kit [24]. The machine has an Intel Core i7-5557U 3.1GHz processor with 8GB RAM, running Linux kernel 4.9. We use three applications in our evaluations: a high-criticality object classification application from the Darknet neural network framework [44], a high-criticality object tracking application from dlib C++ library [16], and a low-criticality MPEG video decoder [20]. These applications are chosen for their relevance to the sorts of applications that might be used in infotainment and autonomous driving systems. The dlib object tracking application is only used for the last set of experiments to test two different execution time prediction models.

For object classification, we use the COCO dataset [32] images for both profiling and execution. The dataset for the Profiling phase was chosen from random images from the dataset and used to determine the LO- and HI-mode budgets of the Darknet high-criticality application. The dataset for the Execution phase was also randomly chosen but similarly distributed in terms of image dimensions as the data of the Profiling phase was. For the video decoder application, we use the *Big Buck Bunny* video [8] as the input. We have turned off memory locking with `mlock` by the `liblitmus` library, as multiple object classification tasks collectively require more RAM than the physical 8GB limit.

5.1 Task Parameters

Table 2 shows the LO- and HI-mode budgets for the three applications. Each object classification and tracking task consist of a series of jobs that classify objects in a single image. Each video decoder task decodes 30 frames in a single job.

Table 2 Applications and their Budgets.

Application	C(LO)	C(HI)
Darknet object classification	345 ms	627 ms
dlib object tracking	10.7 ms	18 ms
video decoder	250 ms	-

The LO-mode budget, $C(LO)$, is the average time that a task takes to complete its job. In Section 5.5, we also present experiments where we increase our LO-mode budget estimate. The HI-mode budget, $C(HI)$, accounts for the worst-case running time of the high-criticality task for any of its jobs. The LO-mode utilization of each individual task is generated by the UUnifast algorithm [9]. We then calculate a task's period by dividing its LO-mode budget by its utilization.

In all experiments, we observed that none of the tasks exceed their HI-mode budget. Consequently, none of the high-criticality tasks miss their deadlines in any of our tests. Hence, we assume that our derivation of LO- and HI-mode budgets are safe and correct for our experiments. The main focus of our evaluation now is to compare the QoS of the LC tasks.

5.2 QoS Improvements for Low-criticality Tasks

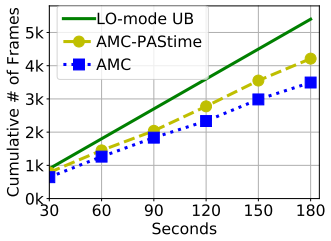
We compare the QoS for low-criticality tasks using AMC and AMC-PAStime in different cases. In every case, each taskset has an equal number of high-criticality object classification tasks and low-criticality video decoder tasks. We experiment with ten schedulable tasksets in all cases except the base case described below. We run each of the tests ten times, and we report the average of the measurements for low-criticality tasks. As stated earlier, all high-criticality tasks meet their timing requirements in each case. Execution time prediction at a checkpoint for PAStime is done with a linear extrapolation model for all the experiments ($f(C_i(LO), X) = C_i(LO) + \frac{C_i(LO) \times X}{100}$ from Section 3) except in Section 5.8 where we discuss other approaches.

Our base case is to run one high-criticality object classification task and one low-criticality video decoder task for 180 seconds. Here, we set the periods of both tasks to 1000 ms rather than using the UUnifast algorithm, yielding a total LO-mode utilization of ~60%.

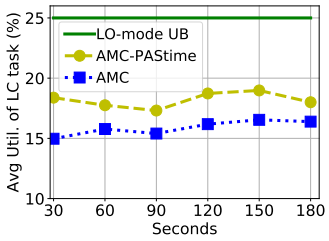
Figure 3a shows the cumulative number of frames decoded by the video decoder task. The LO-mode Upper Bound (UB) line shows the cumulative number of decoded frames if the system is kept in LO-mode all the time. This line represents a theoretical UB for a decoded frame playback rate of 30 frames per second over the entire experimental run.

We see in Figure 3a that AMC-PAStime has a 9–21% increase in the cumulative number of decoded frames compared to AMC scheduling. The performance of the low-criticality task is related to the number of HI-mode switches in the two scheduling policies. AMC-PAStime decreases the number of HI-mode switches by 35%, compared to AMC scheduling.

Although the number of decoded frames is an illustrative metric for a video decoder's QoS, the average utilization of an application is a more generic metric. Average utilization represents the CPU share a task receives over a period of time. Figure 3b shows that AMC-



(a) Cumulative # of Frames.



(b) Average Utilization.

Figure 3 Video Decoder Performance.

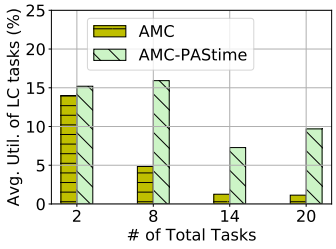


Figure 4 Varying # of Tasks.

Table 3 Number of Mode Switches.

(a) Varying Number of Tasks.

# of tasks	AMC	AMC-PAStime
2	4	2
8	9	4
14	7	5
20	5	3

(b) Varying Initial LO-mode Utilization.

Utilization (%)	AMC	AMC-PAStime
40	11	4
50	11	4
60	9	4
70	10	5
80	11	9

PAStime achieves 10% more utilization on average for the video decoder task, compared to AMC scheduling. The LO-mode UB line is the maximum utilization of the video decoder task, which is 25% (i.e., $C(LO)/Period$ = 250 ms/1000 ms).

5.3 Scalability

To test system scalability, we increase the number of tasks in a taskset up to 20 tasks. As explained above, we generate the periods of the tasks by distributing the total LO-mode utilization of ~60% to all the tasks using the UUnifast algorithm. This setup is inspired by the theoretical parameters in previous mixed-criticality research work [7]. The LO-mode utilization bound for low-criticality tasks remains between 25–35%.

Figure 4 shows the average utilization of the low-criticality tasks, when the total tasks vary from 2 to 20. Each task in this case consists of 20 jobs. We see that the average utilization drops for AMC scheduling as the number of tasks increases.

AMC-PAStime achieves significantly greater average utilization for the low-criticality tasks, by deferring system switches to HI-mode until much later than with AMC scheduling. This is because the LO-mode budgets for the high-criticality tasks are extended due to runtime delays.

AMC-PAStime decreases the number of mode switches by 28–55%. AMC-PAStime's resistance to switching into HI-mode allows low-criticality tasks to make progress. This in turn improves their QoS. In these experiments, AMC-PAStime improves the utilization of the low-criticality tasks by a factor of 3, 5 and 9, respectively, for 8, 14 and 20 tasks. Table 3a shows that AMC-PAStime reduces the number of mode-switches compared to AMC.

5.4 Varying the Initial Total LO-mode Utilization

In this test, we vary the initial total LO-mode utilization for 8 tasks from 40% to 80% by adjusting the periods of all tasks. The initial utilization does not account for increases caused by LO-mode budget extensions to high-criticality tasks.

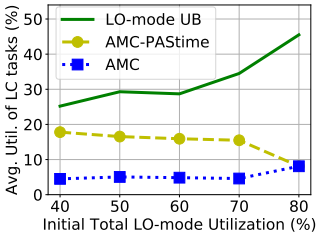


Figure 5 Varying LO-mode Utilization.

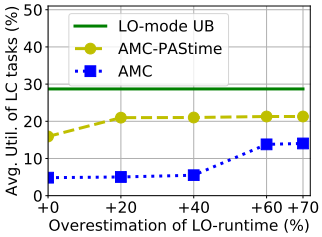


Figure 6 Overestimated $C(LO)$.

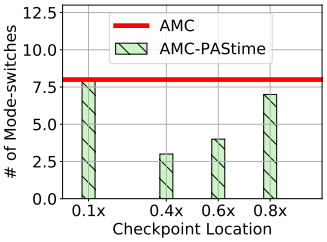


Figure 7 Checkpoint Location.

Figure 5 demonstrates that AMC-PAStime improves average utilization of the low-criticality tasks by more than 3 times, up to 70% total LO-mode utilization. After that, AMC-PAStime and AMC scheduling converge to the same average utilization for low-criticality tasks. This is because there is insufficient surplus CPU time in LO-mode for AMC-PAStime to accommodate the extended budget of a high-criticality task. Therefore, the LO-mode extension requests are disapproved by AMC-PAStime. The reduced number of mode switches for AMC-PAStime, shown in Table 3b, also corroborates the rationale behind AMC-PAStime's better performance than AMC.

We note that the schedulability of random tasksets decreases with higher LO-mode utilization in AMC scheduling. Therefore, many real-world tasksets may not be schedulable because of their HI-mode utilization. Thus, AMC-PAStime's improved performance is significant for practical use-cases.

5.5 Estimation of LO-mode Budget

In our evaluations until now, we estimate a LO-mode budget based on the average execution time of the high-criticality object classification application. In the next set of experiments, we estimate the LO-mode budget of a high-criticality task to be a certain percentage above the average profiled execution time. An increased LO-mode budget for high-criticality tasks benefits AMC scheduling. This is because high-criticality tasks are now given more time to complete in LO-mode, and therefore low-criticality tasks will still be able to execute as well. As a result, the utilization of low-criticality tasks is able to increase.

Suppose that $C(LO)$ is an average execution time estimate for the LO-mode budget of a high-criticality task. Let $(C(LO) + o)$ be an overestimate of the LO-mode budget. As before, AMC-PAStime detects an $X\%$ lag at a checkpoint and predicts the total execution time to be $C(LO) + e$. If the actual LO-mode budget is $(C(LO) + o)$ then AMC-PAStime requests for an extra budget of $(e - o)$, assuming $(e - o) > 0$.

Even for overestimated $C(LO)$, Figure 6 shows that AMC-PAStime still improves utilization for the low-criticality tasks by more than a factor of 3 up to an overestimation of 40%. Overestimation helps in reducing the number of mode switches for AMC scheduling after 40%, as high-criticality tasks have larger budgets in LO-mode.

There is no improvement by AMC scheduling after 60% LO-mode budget overestimation. AMC-PAStime also shows no benefits with increased overestimation, because the LO-mode budget extensions are disapproved by the online schedulability test. Therefore, the system is switched to HI-mode by an overrun of a high-criticality task.

5.6 Checkpoint Location

We use our modified LLVM compiler in the Profiling phase to determine a viable checkpoint for the high-criticality object classification task. We instrument checkpoints in the

`forward_network` function of the Darknet neural network module. We consider four checkpoint locations in the Profiling phase, which are both automatically and manually instrumented. In the Execution phase, we measure performance for each of these checkpoint locations. Figure 7 shows the variation in the number of mode switches against the location of a checkpoint. The x-axis is the approximated division point of a checkpoint location with respect to $C(LO)$. For example, $0.1\times$ means that the checkpoint is at $(0.1 \times C(LO))$.

We see that the number of mode switches decreases if the location of a checkpoint is more towards the middle of the code. However, a checkpoint near the start and the end of the source code have nearly the same number of mode switches, as with AMC scheduling. A checkpoint near the beginning of a program is not able to capture sufficient delay to increase the LO-mode budget enough to prevent a mode switch. Likewise, a checkpoint near the end of a program is often too late. A HI-mode switch may occur before the high-criticality task even reaches its checkpoint. Hence, a checkpoint at $0.8\times$ in the source code of a program reduces the number of mode switches by just 1.

5.7 Overheads

The main overheads of AMC-PAStime compared to AMC are the online schedulability test and budget extension. We first derive an upper bound on the overhead by offline analysis and compare with the experimental measurements.

Our offline upper bound is the total number of iterations in solving the response time recurrence relations during the schedulability test in AMC-PAStime. We generate 500 random tasksets of 20 tasks for different initial LO-mode utilizations. Initial LO-mode utilizations range from 40% to 90%. The utilization of each individual task is generated using the UUnifast algorithm, and each period is taken from 10 to 1000 simulated time units, as done in previous work [7, 23]. As our experimental taskset has a criticality factor ($\text{CF} = \frac{C(HI)}{C(LO)}$) of ~1.8, we also test with a CF of 1.8.

Among the schedulable tasks with AMC scheduling, we increase the demand in LO-mode budget of the highest priority task. Then, we calculate the total number of iterations needed to determine whether an extension of the budget can be approved by an offline version of the online schedulability test. Here, one iteration is a single update to the response time in any one of the recurrence equations (in Equations 4 and 5) used to test for schedulability.

We increase the demand for extra budget from 10 to 80% in this analysis because the CF is 1.8. We check the maximum number of iterations to decide the schedulability of a taskset across the 500 tasksets. We carry out this offline analysis with 40-90% initial LO-mode utilization.

In Figure 8, we show the maximum number of iterations to decide the schedulability of a taskset, against a demand of 10 to 80% extra budget in LO-mode by the highest priority task. Each point in the figure represents the maximum iterations across the 500 tasksets to either approve or disapprove of schedulability.

We have observed the number of iterations to be as high as 120. Therefore, we set 120 as the highest number of allowed iterations for the online schedulability test. When the number of iterations exceed 120 at runtime, we disapprove a LO-mode budget extension. This strategy maintains a safe and known upper bound on the online overhead of AMC-PAStime.

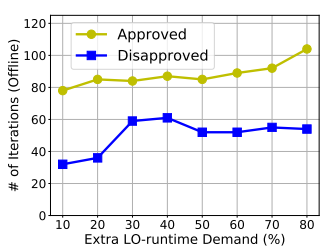


Figure 8 Offline Iterations.

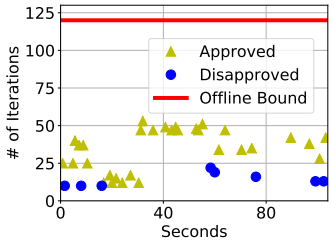


Figure 9 Online Iterations.

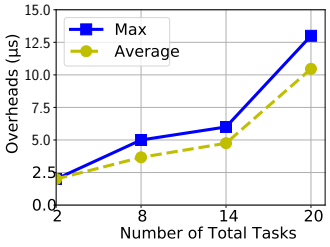


Figure 10 Budget Extension.

5.7.1 Microbenchmarks

Each iteration of a response time calculation has a worst-case time of 1μs, for our implementation of the online schedulability test in LITMUS$^{\text{RT}}$. Therefore, we bound the worst-case delay for the online schedulability test in LITMUS$^{\text{RT}}$ at 120μs for our test cases. Additionally, the worst-case execution time for the `ANNOUNCE_TIME` macro is 10μs. Hence, the maximum total overhead of an `ANNOUNCE_TIME` call, accounting for the schedulability test, is 130μs. The 130μs overhead is factored into the LO-mode extension inside the LITMUS$^{\text{RT}}$ kernel.

Figure 9 shows the number of iterations for the online schedulability test for a taskset of 20 tasks with 60% initial total LO-mode utilization, in cases where an extension was approved and disapproved. We see that the offline bound of 120 iterations is much higher than the actually observed number of iterations. Hence, we never need to abandon the schedulability test because of excessive overheads. In addition, disapproval takes less time than approval online, which corroborates our offline observations. In addition, disapproval takes less time than approval online, which corroborates our offline observations in Figure 8.

Figure 10 shows the maximum and average times for a LO-mode budget extension decision including the online schedulability test. It demonstrates that the extension approval decision takes more time with increasing number of tasks. However, the maximum times are still significantly lower than the offline upper bound of 130μs for 20 tasks. In general, budget extension overheads can be bounded according to the number of tasks in the system.

5.8 Execution Time Prediction Model

As we have explained in Section 3, the execution time after a checkpoint is predicted by a function $f(C_i(LO), X)$, where $C_i(LO)$ is the LO-mode budget, and $X\%$ is the observed delay percentage relative to the LO-mode time to reach the checkpoint. The parameter X in f is a *timing progress metric*, which is used to make runtime scheduling decisions in PAStime.

5.8.1 Linear Extrapolated Delay

We have already shown in the previous experiments how a straightforward linear extrapolated delay improves the utilization of LC tasks compared to AMC for an object classification application. In such a case, $f(C_i(LO), X) = C_i(LO) + \frac{C_i(LO) \times X}{100}$. A linear extrapolation of delay at a checkpoint applied to the entire LO-mode time makes sense in the absence of additional knowledge that could influence the remaining execution time of the task.

We now investigate further whether the linear extrapolation is effective in detecting the amount of delay in our Darknet high-criticality object classification task. We compare the predicted and actual times taken by the high-criticality tasks when LO-mode is extended by

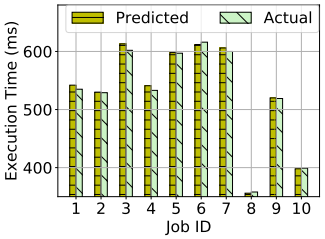


Figure 11
Prediction Accuracy.

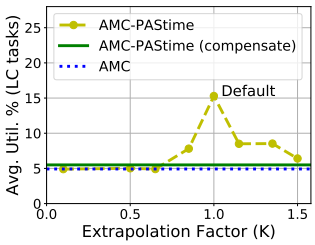


Figure 12
Linear Extrapolation.

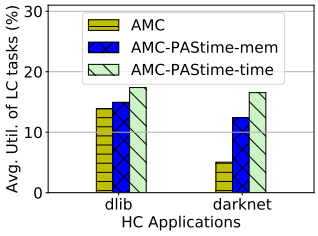


Figure 13
Prediction Models.

AMC-PAStime. This experiment is performed with 8 tasks for 40 jobs each. The initial total LO-mode utilization is 60%, before applying budget extensions.

We show ten of the extended jobs in Figure 11. We see that the predicted execution times are close to the actual execution times. For cases where the predicted times exceed the actual budget expenditure, the predictions overestimate execution by just 0.88% on average for this experiment. In Figure 11, Job ID 6, 8 and 10 show lower predicted times than the actual spent budgets. For these jobs, the system is switched to HI-mode because the extended LO-mode is not enough for a task to complete its job. In these cases, the predicted times are smaller than the spent budgets by 0.49%.

This experiment shows that the checkpoint is effectively being used to predict the execution time of a high-criticality task in most cases. The budget extensions are a reasonable estimate of the actual task requirements.

5.8.2 Alternative Execution Time Compensation Models

In our experiments, we compare the linear extrapolation model with an alternative compensatory execution time prediction model. The alternative approach compensates for the observed delay at a checkpoint, by adding the delay amount to the predicted execution time. If the observed delay is Y, then the estimated total execution time is $C_i'(LO) = C_i(LO) + Y$.

We have also investigated variations to the linear extrapolation model, even though it has proven effective in our previous experiments. A further experiment multiplies the extrapolated delay by a factor K, such that the predicted execution time is $f(C_i(LO), X) = C_i(LO) + K \times \frac{C_i(LO) \times X}{100}$.

Figure 12 compares AMC-PAStime with the alternative compensatory model `compensate`, and linear extrapolation model where K ranges from 0.1 to 1.5. The linear extrapolation model ($K = 1$) outperforms the other approaches, while the compensatory model improves the utilization slightly compared to AMC.

5.8.3 Prediction based on Memory Access Time

Memory accesses by different processes compete for shared cache lines and consequently cause unexpected microarchitectural delays. There are previous works that model the memory [38, 42, 56] and cache [53, 55] accesses in a multicore machine to deal with the issue of predictable execution. Here, we demonstrate PAStime with an execution time prediction model based on the number of memory accesses, to increase the LO-mode budget of a high-criticality task.

In this model, we note the number of memory accesses by a high-criticality task as we measure the LO- and HI-mode execution time of the task during the Profiling phase. We

measure the average number of memory accesses in LO-mode for each profiled run of a task with different inputs, and denote it by $M(LO)$.

During the Profiling phase, we also measure the number of memory accesses in LO-mode before and after a checkpoint: $M_{\text{pre\_}CP}(LO), M_{\text{post\_}CP}(LO)$. These values are also averaged across all runs of a given task. Therefore, $M_{\text{pre\_}CP}(LO) + M_{\text{post\_}CP}(LO) = M(LO)$

We define a new *memory instructions progress metric* to be the ratio of task execution time to the number of memory accesses. The LO-mode value of this metric is $Pr_{mem,LO} = \frac{C(LO)}{M(LO)}$.

In the Execution phase, we measure the number of memory accesses and the used budget up to a checkpoint, respectively denoted by M_{CP} and C_{used} for a given task. During Execution, we calculate the memory instructions progress metric at a checkpoint, $Pr_{mem,CP} = \frac{C_{used}}{M_{CP}}$. If $Pr_{mem,CP} \leq Pr_{mem,LO}$, we predict the task is progressing as expected in LO-mode in terms of memory access-related delays. There are other factors such as I/O-related delays that affect the execution time of a task. However, I/O should be budgeted separately from the main task, as shown in prior work for AMC [39]. Notwithstanding, PAStime is capable of supporting even richer prediction models based on a multitude of factors that cause delays due to microarchitectural and task interference overheads.

When $Pr_{mem,CP} > Pr_{mem,LO}$, we predict a task's memory access delays. We calculate the expected number of memory instructions after a checkpoint: $M_{\text{expected\_post\_CP}} = \frac{M_{CP} \times M_{\text{post\_CP}}(LO)}{M_{\text{pre\_}CP}(LO)}$, and increase the LO-mode budget by $(M_{\text{expected\_post\_}CP} \times (Pr_{mem,CP} - Pr_{mem,LO}))$. This increment helps cover future memory delays after a checkpoint.

We tested both high-criticality dlib object tracking and Darknet object classification applications with this model. Figure 13 shows experimental results with 4 HC tasks (either dlib or Darknet) and 4 LC video decoder tasks, with initial 50% LO-mode utilization. We see that `AMC-PAStime-mem` with this memory access progress metric provides better utilization to the LC tasks than AMC. However, it is not better than `AMC-PAStime-time`, which uses the default linear extrapolated execution time prediction. Nevertheless, the result shows that some form of progress metric dynamically improves the utilization of LC tasks.

6 Related Work

The problem of determining tight worst-case execution time (WCET) bounds for tasks [54] is compounded by timing variations caused by caches, buses and other hardware features. Recently, mixed-criticality systems (MCSs) [5–7,12,15,52] have gained popularity as they allow tasks to have multiple estimates of execution time at different criticality, or assurance, levels. Baruah et al. proposed Adaptive Mixed Criticality (AMC) as a fixed-priority scheduling policy for mixed-criticality systems [7]. AMC dominates other fixed-priority scheduling schemes, such as Static Mixed-criticality with Audsley's priority assignment and Period Transformation for random tasksets [7, 23].

AMC scheduling affects the QoS of low-criticality tasks by dropping them in HI-mode. Further research work explored adjustments to the task model, including stretching the periods [21, 46–49], and using reduced budget in HI-mode [4, 5, 35, 43], to provide improved service to the low-criticality tasks. AdaptMC employs control-theoretic feedback to manage the budgets of dual-criticality workloads [40]. In contrast, PAStime adjusts the budget of a currently running task based on the observed delay at a checkpoint, as we want a simple budget adjustment mechanism to minimize the runtime overhead. However, future work may explore integrating control-theoretic feedback, such as the one used by AdaptMC, into PAStime's runtime strategy.

Santy et al. proposed the idea of a statically-calculated task allowance [45], for the theoretical modeling and scheduling of mixed-criticality tasks. In contrast to Santy's work, PAStime dynamically decides whether a task is given extra budget in LO-mode based on the

observed runtime delay at a checkpoint. PAStime is then able to decide which task's budget to extend in LO-mode, given the slack in computational resources [15].

The work by Kritikakou et al. uses run-time monitoring and control in mixed-criticality systems, to increase task parallelism [26–28]. The authors run high- and low-criticality tasks together and monitor high-criticality tasks at multiple *observations points* embedded into their control flow graphs. If interference from the low-criticality tasks is too prohibitive for the high-criticality tasks, low-criticality tasks are stopped to ensure that the high-criticality tasks meet their deadlines. The authors use static execution time analysis to decide whether to run the low-criticality tasks after an observation point in a high-criticality task. In our work, we dynamically adjust LO-mode budgets of the high-criticality tasks when we detect delays at intermediate checkpoints. We decide about the LO-mode budgets based on the observed progress, instead of using static offline remaining time as used by other work. In addition, we have implemented an LLVM compiler pass to *automatically* instrument checkpoints in C and C++ programs, which have been tested with a PAStime implementation in LITMUS^{RT} for real-world applications developed for Linux. Kritikakou et al.'s research was implemented for a specific DSP platform, which is yet to be tested for real-world applications. Notwithstanding, the authors provide an important WCET analysis using CFGs for high-criticality tasks.

Previous ideas of progress-based scheduling were proposed to improve GPU performance [1, 25], fairness among multiple threads [18, 19] and to account for instruction cycles [17]. Most of these works run a task in an isolated environment and compare its progress to an online parallel execution with other tasks. Somewhat similar to the motivation behind Jeong et al's work [25], we also meet the deadlines of high-criticality tasks. However, PAStime uses CFGs to monitor progress rather than application-specific features such as frame processing rates in multimedia applications as done in the other works.

7 Conclusions and Future Work

This paper presents PAStime, a scheduling strategy based on the execution progress of a task. Progress is measured by observing the time taken for a program to reach a designated checkpoint in its control flow graph (CFG). We integrate PAStime in mixed-criticality systems by extending AMC scheduling. PAStime extends the LO-mode budget of a high-criticality task based on its observed progress, given that the extension does not violate the schedulability of any tasks. Our extension to AMC scheduling, called AMC-PAStime, is shown to improve the QoS of low-criticality tasks. Moreover, we implement an algorithm in the LLVM compiler to automatically detect and instrument viable program checkpoints for use in task profiling.

This paper presents the first implementation of AMC scheduling in LITMUS^{RT}. We have also implemented AMC-PAStime in LITMUS^{RT} and compared it against AMC. While both meet deadlines for all high-criticality tasks, AMC-PAStime improves the average utilization of low-criticality tasks by 1.5–9 times for 2–20 total tasks. AMC-PAStime is shown to improve performance for low-criticality tasks while reducing the number of mode switches. Finally, we have shown that different progress metrics also improve the LC tasks' utilization.

In future work, we will explore other uses of progress-aware scheduling in timing-critical systems. We plan to extend the Linux kernel `SCHED_DEADLINE` policy [30, 31, 34] to support progress-aware scheduling. We believe that PAStime is applicable to timing-sensitive cloud computing applications, where it is possible to adjust power (e.g., via Dynamic Voltage Frequency Scaling) based on progress. Application of PAStime to domains outside real-time computing will also be considered in future work.

References

1 Jayvant Anantpur and R Govindarajan. PRO: Progress-aware GPU Warp Scheduling Algorithm. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 979–988, 2015.

2 Neil C Audsley. On Priority Assignment in Fixed Priority Scheduling. *Information Processing Letters*, 79(1):39–44, 2001.

3 Sanjoy Baruah and Alan Burns. Fixed-priority Scheduling of Dual-criticality Systems. In *Proceedings of the 21st International conference on Real-Time Networks and Systems*, pages 173–181. ACM, 2013.

4 Sanjoy Baruah, Alan Burns, and Zhishan Guo. Scheduling Mixed-criticality Systems to Guarantee Some Service under All Non-erroneous Behaviors. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 131–138. IEEE, 2016.

5 Sanjoy Baruah, Haohan Li, and Leen Stougie. Towards the Design of Certifiable Mixed-criticality Systems. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 13–22, 2010.

6 Sanjoy Baruah and Steve Vestal. Schedulability Analysis of Sporadic Tasks with Multiple Criticality Specifications. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, pages 147–155, 2008.

7 Sanjoy K Baruah, Alan Burns, and Robert I Davis. Response-time Analysis for Mixed Criticality Systems. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS)*, pages 34–43, 2011.

8 Big Buck Bunny. `https://www.bigbuckbunny.org`, 2018.

9 Enrico Bini and Giorgio C Buttazzo. Measuring the Performance of Schedulability Tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

10 Björn Brandenburg and James H Anderson. *Scheduling and Locking in Multiprocessor Real-time Operating Systems*. PhD thesis, PhD thesis, The University of North Carolina at Chapel Hill, 2011.

11 Alan Burns and Sanjoy Baruah. Towards a more practical model for mixed criticality systems. In *Workshop on Mixed-Criticality Systems (colocated with RTSS)*, 2013.

12 Alan Burns and Robert I. Davis. A Survey of Research into Mixed Criticality Systems. *ACM Comput. Surv.*, 50(6):82:1–82:37, November 2017. `doi:10.1145/3131347`.

13 Kevin Burr and William Young. Combinatorial Test Techniques: Table-based Automation, Test Generation and Code Coverage. In *Proceedings of the International Conference on Software Testing Analysis & Review*, 1998.

14 John M Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C Devi, and James H Anderson. LITMUS^{RT}: A Testbed for Empirically Comparing Real-time Multiprocessor Schedulers. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 111–126. IEEE, 2006.

15 Dionisio De Niz, Karthik Lakshmanan, and Ragunathan Rajkumar. On the Scheduling of Mixed-criticality Real-time Task Sets. In *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS)*, pages 291–300, 2009.

16 dlib C++ Library. dlib: Video Tracking. `http://dlib.net/video_tracking_ex.cpp.html`, 2019.

17 Stijn Eyerman and Lieven Eeckhout. Per-thread Cycle Accounting in SMT Processors. *ACM Sigplan Notices*, 44(3):133–144, 2009.

18 Josue Feliu, Julio Sahuquillo, Salvador Petit, and José Duato. Addressing Fairness in SMT Multicores with a Progress-aware Scheduler. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 187–196, 2015.

19 Feliu, Josue and Sahuquillo, Julio and Petit, Salvador and Duato, Jose. Perf&Fair: A Progress-aware Scheduler to Enhance Performance and Fairness in SMT Multicores. *IEEE Transactions on Computers*, 66(5):905–911, 2017.

20 FFmpeg Multimedia Framework. `https://www.ffmpeg.org/`, 2019.

21 Chris Gill, James Orr, and Steven Harris. Supporting Graceful Degradation through Elasticity in Mixed-Criticality Federated Scheduling. In *Proc. 6th Workshop on Mixed Criticality Systems (WMC), RTSS*, pages 19–24, 2018.

22 Google. trucov. `https://code.google.com/archive/p/trucov/`, 2018.

23 Huang-Ming Huang, Christopher Gill, and Chenyang Lu. Implementation and Evaluation of Mixed-criticality Scheduling Approaches for Sporadic Tasks. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):126, 2014.

24 Intel. Intel® NUC Kit NUC6i7KYK (Skull Canyon): Features and Configurations. `https://www.intel.com/content/www/us/en/products/docs/boards-kits/nuc/nuc-kit-nuc6i7kyk-features-configurations.html`, 2019.

25 Min Kyu Jeong, Mattan Erez, Chander Sudanthi, and Nigel Paver. A QoS-aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC. In *Proceedings of the 49th annual Design Automation Conference*, pages 850–855. ACM, 2012.

26 Angeliki Kritikakou, Olivier Baldellon, Claire Pagetti, Christine Rochange, and Matthieu Roy. Run-time Control to Increase Task Parallelism in Mixed-critical Systems. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

27 Angeliki Kritikakou, Thibaut Marty, and Matthieu Roy. DYNASCORE: DYNAmic Software COntroller to increase REsource Utilization in Mixed-critical Systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 23(2), 2017.

28 Angeliki Kritikakou, Christine Rochange, Madeleine Faugère, Claire Pagetti, Matthieu Roy, Sylvain Girbal, and Daniel Gracia Pérez. Distributed Run-time WCET Controller for Concurrent Critical Tasks in Mixed-critical Systems. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*. ACM, 2014.

29 Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, page 75. IEEE Computer Society, 2004.

30 Juri Lelli, Giuseppe Lipari, Dario Faggioli, and Tommaso Cucinotta. An Efficient and Scalable Implementation of Global EDF in Linux. In *Proceedings of the 7th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, pages 6–15, 2011.

31 Juri Lelli, Claudio Scordino, Luca Abeni, and Dario Faggioli. Deadline Scheduling in the Linux kernel. *Software: Practice and Experience*, 46(6):821–839, 2016.

32 Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft COCO: Common Objects in Context. In *Proceedings of the European Conference on Computer Vision*, pages 740–755. Springer, 2014.

33 Linux. hrtimers - Subsystem for High-resolution Kernel Timers. `https://www.kernel.org/doc/Documentation/timers/hrtimers.txt`, Last Accessed: May 2019.

34 Linux. SCHED_DEADLINE Scheduling Policy. `https://www.kernel.org/doc/Documentation/scheduler/sched-deadline.txt`, Last Accessed: May 2019.

35 Di Liu, Nan Guan, Jelena Spasic, Gang Chen, Songran Liu, Todor Stefanov, and Wang Yi. Scheduling analysis of imprecise mixed-criticality real-time tasks. *IEEE Transactions on Computers*, 67(7):975–991, 2018.

36 Di Liu, Jelena Spasic, Nan Guan, Gang Chen, Songran Liu, Todor Stefanov, and Wang Yi. Edf-vd scheduling of mixed-criticality systems with degraded quality guarantees. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 35–46. IEEE, 2016.

37 LLVM LoopInfo Class. `https://llvm.org/doxygen/LoopInfo_8h_source.html`, Last Accessed: May 2019.

38 Renato Mancuso, Roman Dudko, and Marco Caccamo. Light-PREM: Automated Software Refactoring for Predictable Execution on COTS Embedded Systems. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10. IEEE, 2014.

39 Eric Missimer, Katherine Missimer, and Richard West. Mixed-criticality scheduling with i/o. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 120–130. IEEE, 2016.

40 Alessandro Papadopoulos, Enrico Bini, Sanjoy Baruah, and Alan Burns. AdaptMC: A control-theoretic approach for achieving resilience in mixed-criticality systems. In *Proceeding ECRTS Conference*, page 14. LIPICS, 2018.

41 PAStime. Source Code, 2020. URL: `http://cs-people.bu.edu/soham1/pastime/`.

42 Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A Predictable Execution Model for COTS-based Embedded Systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279. IEEE, 2011.

43 Saravanan Ramanathan, Arvind Easwaran, and Hyeonjoong Cho. Multi-rate Fluid Scheduling of Mixed-criticality Systems on Multiprocessors. *Real-Time Systems*, 54(2):247–277, 2018.

44 Joseph Redmon. Darknet: Open Source Neural Networks in C. `http://pjreddie.com/darknet/`, 2013–2016.

45 Francois Santy, Laurent George, Philippe Thierry, and Joël Goossens. Relaxing mixed-criticality scheduling strictness for task sets scheduled with fp. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 155–165. IEEE, 2012.

46 Hang Su, Peng Deng, Dakai Zhu, and Qi Zhu. Fixed-priority Dual-rate Mixed-criticality Systems: Schedulability Analysis and Performance Optimization. In *2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 59–68. IEEE, 2016.

47 Hang Su, Nan Guan, and Dakai Zhu. Service guarantee exploration for mixed-criticality systems. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10. IEEE, 2014.

48 Hang Su and Dakai Zhu. An Elastic Mixed-criticality Task Model and its Scheduling Algorithm. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 147–152. IEEE, 2013.

49 Hang Su, Dakai Zhu, and Scott Brandt. An Elastic Mixed-Criticality Task Model and Early-Release EDF Scheduling Algorithms. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 22(2):28, 2017.

50 Mustafa M Tikir and Jeffrey K Hollingsworth. Efficient Instrumentation for Code Coverage Testing. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 86–96. ACM, 2002.

51 Manohar Vanga, Andrea Bastoni, Henrik Theiling, and Björn B Brandenburg. Supporting Low-Latency, Low-Criticality Tasks in A Certified Mixed-Criticality OS. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, pages 227–236. ACM, 2017.

52 Steve Vestal. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. In *Proceedings of the 28th IEEE Real-Time Systems Symposium (RTSS)*, pages 239–243, 2007.

53 Richard West, Puneet Zaroo, Carl A Waldspurger, and Xiao Zhang. Online Cache Modeling for Commodity Multicore Processors. *ACM SIGOPS Operating Systems Review*, 44(4):19–29, 2010.

54 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The Worst-case Execution-time Problem - Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36, 2008.

55 Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. COLORIS: A Dynamic Cache Partitioning System using Page Coloring. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 381–392. IEEE, 2014 .

56 Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. PALLOC: DRAM Bank-aware Memory Allocator for Performance Isolation on Multicore Platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166. IEEE, 2014.