

# CAPA: A Framework for Contention-Aware and Progress-Aware Multi-Core Real-Time Systems

Weifan Chen\*, Ivan Izhbirdeev\*, and Renato Mancuso

**Abstract**—Real-time systems face significant challenges in managing shared resources on multi-core platforms while maintaining temporal predictability. This paper introduces the CAPA (Contention-Aware Progress-Aware) framework for real-time systems to export and leverage runtime progress information to enable informed resource management decisions. CAPA represents a practical model for tracking task progress, extending existing Timely Progress Assessment (TPA) techniques to support complex control flows and concurrent execution on commercial multi-core hardware. Building on this model, it is possible to design progress-aware multi-core schedulers capable of dynamically regulating task execution to meet timing constraints. We implement CAPA on a commercial off-the-shelf (COTS) platform and evaluate its performance. Results demonstrate CAPA’s ability to provide controlled performance degradation, meet timeliness constraints, and improve schedulability. Thus, CAPA represents a significant step forward toward practical, contention-aware scheduling in multi-core real-time systems.

**Index Terms**—Adaptive Control, Real-Time, Scheduling, Resource Contention, Multicore

## I. INTRODUCTION

**T**HE quality of service in many systems degrades nonlinearly as demand increases, particularly as the system’s performance bottleneck approaches saturation. This concept has profound implications in computing systems. When shared hardware resources are accessed by multiple contenders (e.g., processes running on different cores), the execution time of each contender can become highly variable, potentially compromising the temporal correctness of the system, a critical concern in real-time applications.

**Contention and Unpredictability.** Unpredictable performance degradation due to contention is a well-recognized issue in multicore systems [1], [2]. Without proper shared resource management, deploying hard real-time workloads on complex high-performance embedded systems becomes unsafe. On the other hand, over-provisioning can result in system underutilization. This creates a tension between ensuring predictability and maximizing resource utilization.

Several shared resource management paradigms have been studied. In broad categories, (1) strict partitioning/regulation approaches attempt to mitigate contention by enforcing static per-core/per-task reservations of performance-critical hardware to reduce sharing [3]–[6]; (2) profile-driven workload consolidation approaches aim to use offline-acquired data about the expected behavior of applications to tune resource partitioning [7]–[10]; (3) profile-driven dynamic management combines offline analysis and online tracking of performance

metrics to adjust partitioning parameters dynamically [11]–[14]. (4) Design time analysis or principles that quantify or manage interference effects [15], [16].

**Progress-aware Management.** Recently, a new line of work has emerged: progress-driven workload management. The idea is to enable a system to live-track the fraction of work that a task accomplishes in relation to the total needed to complete. Then, contention-induced slowdowns can be promptly assessed, and management decisions can be taken accordingly and well ahead of task deadlines. Early proposals of progress-driven workload management relied on software instrumentation of the tasks [17]–[19], on firmware instrumentation [20], or on custom hardware support [21]. More recently, however, the work in [22] demonstrated that live progress tracking, a.k.a. *Timely Progress Assessment* (TPA), as coined by Chen et al. [22], can be achieved on commercially available hardware, with low overhead, and without any changes to application binaries and OS/hypervisor-level software. While foundational, this work has three important shortcomings. (1) Only tasks whose control flow can be determined prior to execution can be monitored accurately. (2) The original progress monitoring approach could not live-track tasks on multiple cores. (3) The presented management strategy only reacts to tasks suffering unsatisfactory progress, however, does not exploit the situation where some tasks progress faster than expectation.

**Contribution.** In this paper, we acknowledge that there is immense potential in TPA-based workload management and build upon the state-of-the-art in three main directions. (1) We broaden the progress tracking capability to incorporate tasks with complex control flows at runtime and in multicore settings. (2) Importantly, this work pioneers a fractional, linear definition of progress with respect to the execution path leading to the worst-case execution time (WCET). This practically estimates the progress even when the execution does not follow the worst-case path. (3) We reason about integrating live progress information into the design of multicore schedulers. In particular, we propose a class of *timely progress-aware multicore schedulers* capable of regulating the pace at which concurrent applications progress to meet temporal constraints. In summary, our contributions are the following:

- 1) Propose a Contention-Aware-Progress-Aware (CAPA) task model, namely CAPATASK, that can be used to track the progress in percentage of complex control flow.
- 2) Design, implement, and test on commercial hardware a system-level CAPA support, namely CAPASYS. CAPASYS showcases for the first time that TPA of concurrent tasks on a COTS multicore platform is possible by leveraging existing hardware support;
- 3) Propose and evaluate a first class of schedulers that can

\*These authors contributed equally to this work.

Fig. 1: Two jobs illustrated in progress aware diagram.

Fig. 2: The left shows  $\tau_1$  misses its deadline due to contention. The right shows the same jobs but  $\tau_0$  pauses in between to achieve the timeliness of both.

leverage CAPASYS to enact progress-driven multicore scheduling with the purpose-built CAPASIM simulator. The rest of the paper is structured as follows. Section II provides a motivating example, while Section III reviews the related work. Section IV presents the CAPATASK. Next, Section V describes key design elements of CAPASYS, with implementation details and demonstrating CAPASYS in action on a COTS platform. Section VI describes CAPASIM and presents evaluations of CAPASYS employed atop traditional multicore schedulers. The paper concludes in Section VII.

## II. MOTIVATING EXAMPLE

Can runtime progress information unlock better system resource management? To better illustrate the intuition behind, we introduce a progress-aware diagram. Two examples are shown in Fig. 1. In the diagram, x-axis represents the time ticks. In each vertical slice at a time tick, the solid color represents completed progress. The top of the diagram illustrates a job completes around half progress at  $t = 2.5$ , while the bottom shows a job completion at  $t = 4$ . Progress Management. Consider two jobs on separate cores. They may contend for shared resources, causing execution slowdowns—visible as steeper progress slopes due to reduced progress rates. In Fig. 2-left, both jobs finish at  $t = 4.5$ , but  $\tau_1$  misses its deadline without intervention. In contrast, Fig. 2-right shows that with runtime progress awareness, the system can temporarily pause  $\tau_1$  to accelerate  $\tau_0$  and meet its deadline, then resumes and completes on time. These scenarios are demonstrated in both our implementation (Section V-D) and simulation (Section VI-D).

## III. RELATED WORKS

Two aspects play a considerable role in the example above: (i) inter-core shared resource contention can affect progress rate; (ii) the progress can be extracted at runtime. Both are foundational to this work and have been extensively studied by the real-time community. Shared Resource Contention Aware Task Models. In most of these models, a job is typically treated as an “indivisible atom,” i.e., its impact on the performance of the co-runners is uniform regardless of which phase the job resides. For example, Aceituno et al. [23] factor out the worst-case interference time if the application were to encounter resource contention. Bini [24] assumes that the current mode of the system, not those of tasks, determines each task’s progress rate. Kim et al. [25] directly prevent tasks with considerable interference running concurrently. Unfortunately, considering tasks as “indivisible” units of work fails to reflect that jobs at different phases impact shared resources differently.

Workload Profiling. Numerous works have illustrated that shared resource usage can vary throughout the execution of a task, and that profiling is crucial for informed management. For example, approaches to inspect the cache activity as concurrent tasks execute were proposed [7], [8]. Roozkhosh et al. [9] inspect the activity in main memory at the cache-line granularity. Sohal et al. [10] demonstrate using memory consumption envelopes for partition tuning. The ability to inspect at runtime the performance of a target application while minimally disrupting its temporal behavior has been thoroughly investigated in [26].

Progress-aware Task Models. Various expressions for progress exist, e.g. the number of instructions retired [13], [27], deadline partitioning [28], [29]. However, the most relevant approach to this work involves TPA [17]–[22], whose essence is the selection of specific addresses within the task through of the profiling. These addresses are typically the entries/exits of single-entry-single-exit code blocks, and addresses inside loops. Then, a runtime monitoring mechanism tracks the time when the addresses are reached, thereby adaptively assessing progress timeliness. Most of the works [17], [18], [20], [22] aim at protecting the timeliness of a single task in a multicore environment via TPA by throttling the activities of the co-runners on different cores. Only Kritikakou et al. [19], [21] investigate protecting the timeliness of multiple tasks in a multicore environment. Nevertheless, these works assume that the execution order of visits to the selected addresses can be determined prior to runtime, and none define a notion of universal progress. As a result, the progress of different tasks cannot be effectively compared. This work addresses these limitations.

## IV. THE CAPA TASK MODEL

Contention-Aware-Progress-Aware task model (CAPATASK for short) is presented in this section. The goal of CAPATASK is to express the progress of a task as a percentage. Despite its simple form, the definition of progress must remain robust even for tasks with complex control flow, e.g., in the presence of branches that cannot be statically predicted before runtime, or in the presence of loops where the number of iterations is only known at runtime. The key intuition to achieve this is:

Intuition I: Normalizing progress over all possible execution paths with respect to the worst-case path identified during of the task profiling or static analysis.

Task profiling, in this context, refers to measuring execution times at specific control flow “observation points” during its

execution as the task is repeatedly executed in a contention-free environment during the analysis phase. This procedure is used in all existing TPA-related methods [17]–[22]. We use the terms contention-free, without contention, solo, and isolation interchangeably to describe the condition where a task is running alone in the system. While WCET traditionally refers to the end-to-end execution time of a task, we also use WCET to denote the execution time of a portion of a task. When doing so, the specific context will make this distinction clear. While we adopt a measurement-based approach, equivalent definitions apply for static analysis-based methods.

### A. Task Model

The proposed CAPATASK integrates support for trackable progress within traditional real-time taskset semantics. Each CAPATASK task  $\tau_i$  is a sequence of jobs with sporadic or periodic release where the generic job can be described as the tuple  $\tau_{i,j} = fT_{i,j}; D_i; G_i$ . Here,  $T$  represents a minimum inter-arrival time in the case of sporadic tasks, or an exact release period for periodic tasks. Next,  $D_i$  is the relative deadline, i.e., a constraint on the job's response time once it is released. Finally,  $G_i$  is a Timed Milestone Graph (TMG) proposed in [22]. A recap of TMG is presented in the next section. In CAPATASK, TMGs replace typical WCET information. CAPATASK is not intended to replace WCET analysis; instead, it leverages WCET estimates—whether from static or measurement-based methods—for progress estimation. Moreover, it mitigates WCET pessimism by detecting surplus progress (or lack thereof) for better resource management.

### B. Recap: Timed Milestone Graph

CAPATASK is inspired by the Timed Milestone Graph (TMG) and is a transformation over an existing TMG. In this work, we follow the same method as in [20] to construct the initial TMG and then transform it into CAPATASK. This section presents the existing definition and construction method of TMG. Conceptually, a TMG retains two types of timing information acquired through profiling/analysis: (1) a tail time  $T_{\text{tail}}$  indicates the WCET for execution to reach a specific instruction; (2) a move time  $T_{\text{move}}$  indicates the maximum time it takes for the execution to transfer from one specific instruction to another.

#### Definition I: Timed Milestone Graph

A Timed Milestone Graph (TMG) is a weakly connected graph  $G = (V; A)$ , where  $V$  and  $A$  are sets of vertices and arcs respectively. The graph has the following properties:

- There exists only one vertex  $v_{\text{entry}} \in V$  that has no predecessors—in-degree is zero;
- There exists only one vertex  $v_{\text{exit}} \in V$  that has no successors—out-degree is zero.

Additionally, two maps are defined:

- $T_{\text{tail}} : V \rightarrow \mathbb{R}_{\geq 0}$  assigns a tail time to each vertex;
- $T_{\text{move}} : A \rightarrow \mathbb{R}_{\geq 0}$  assigns a move time to each arc.

Typically, a vertex  $u \in V$ , a.k.a., a milestone in this context, represents an instruction location, such as the address of the

assembly instruction in  $\tau_i$ . At runtime, we assert that a milestone  $u$  is reached as the program counter (PC) points to  $u$ . We also name it a vertex/milestone-hit. Experimental Construction. Like all other TPA methods,  $T_{\text{tail}}$  and  $T_{\text{move}}$  are derived by collecting the behavior of the task across multiple runs/samples in solo execution. A milestone might be reached multiple times in the same run if it is part of a loop. Let's define the random variable  $T(u; i; j)$  indicating the time elapsed since the job release till  $u$  is reached for the  $i^{\text{th}}$  time in the  $j^{\text{th}}$  run. Then  $T_{\text{tail}}(u)$  is assigned to be the maximum in all  $i; j$  for  $T(u; i; j)$ :

$$T_{\text{tail}}(u) = \max_{i; j} T(u; i; j)g \quad (1)$$

Thus, the tail time  $T_{\text{tail}}(u)$  indicates the worst (experimentally observed) execution time to reach  $u$  for the very last time. By construction,  $T_{\text{tail}}(v_{\text{entry}}) = 0$ .

The move time  $T_{\text{move}}(u; v)$  is a conservative estimation of the maximum time for the execution to transfer from  $u$  to  $v$ . The largest time elapse seen during the profiling is taken as its move time. Formally, let  $T(u; v; i; j)$  be the time elapsed for the execution to transfer from  $u$  to  $v$  for the  $i^{\text{th}}$  time in the  $j^{\text{th}}$  run. Then

$$T_{\text{move}}(u; v) = \max_{i; j} T(u; v; i; j)g \quad (2)$$

**Milestone Selection.** The milestone selection method we use in this paper is akin to what is described in [20]. We hereby present the key steps. (1) Given the task's binary, construct its control flow graph. Several tools are available, such as angr [30] and GNU cflow. (2) In the main function, tentatively place initial milestones on all the return addresses of top-level function calls and all the successors of any strongly connected components (SCC). (3) Measure  $T_{\text{tail}}$  and  $T_{\text{move}}$  for these initial milestones. If the temporal distance between any two subsequent milestones is too small, then randomly remove one of them. If too large, place more milestones in between. Two cases exist in the latter scenario: (i) If the successor milestone is the return address of a function call, then unfold the function and repeat from step (2), thus placing milestones within the unfolded function instead of main; (ii) If the milestone corresponds to the successor of an SCC, this typically implies that one of the loops in the SCC takes a large number of iterations. In this case, randomly place one milestone inside the SCC, and repeat from step (3). The procedure above is applied iteratively until a satisfactory TMG is obtained. This approach is empirically practical because a TMG with fine granularity can usually be obtained by placing one or two milestones within key SCCs that the outlined methodology identifies by progressively annealing the distance between milestones. The iterative nature of the procedure also allows a user to control the granularity of the TMG on demand. In Table II, we quantitatively summarize the TMGs of the benchmarks used in this work.<sup>2</sup> In the next section, we present our method to transform a TMG into a CAPATASK,

<sup>1</sup>Precisely speaking, it is the execution to transfer from  $u$  to  $v$  without reaching any other vertices during the transfer.

<sup>2</sup>Appendix I offers a detailed discussion on the of fine cost of TMG construction.

Fig. 3: A TMG and its transformed graph. Each SCC (in dashed boxes) is replaced with arcs (in red).

which not only provides a normalized progress value at each vertex, but also conservatively evaluates the progress when control flow diverges at runtime.

### C. Normalized Progress over TMG

To enhance readability, we provide intuitive explanation at key stages of the transformation and include a glossary Table I to facilitate reading.

Symbol	Description
$G$	A timed milestone graph (TMG)
$G^{DAG}$	An acyclic TMG, a.k.a., birdseye TMG
$w$	A generic walk or subwalk on $G$
$\bar{w}$	Walk leading to solo WCET; a.k.a., the nominal walk
$C(w)$	The WCET in a solo run of executing walk $w$
$C_i$	The WCET in a solo run from start to $i$ th vertex-hit
$C_i = C_{i-1} \cdot C_{i-1}$	The solo WCET between $(i-1)$ th and $i$ th vertex-hit
$P_{i-1} \in [0; 1]$	Progress between $(i-1)$ th and $i$ th vertex hits
$P(v_i) \in [0; 1]$	Progress at $i$ th vertex hit
$P(t) \in [0; 1]$	Progress at time $t$
$\bar{P}(t) \in [0; 1]$	Nominal progress over time $t$
$k$	Nominal progress rate (reciprocal of solo WCET)

TABLE I: Summary of notation.

A TMG captures all the potential control flow transfers. However, it cannot predict whether a branch is taken. This accompanies the presence of loop components, making pressing progress as a single percentage quantity hard.

Intuition II: Loops complicate progress tracking, but if we focus only on entry and exit of a loop—ignoring iterations—the problem becomes simpler. Progress within loops can then be addressed separately.

A strongly connected component (SCC) is a subgraph in which every vertex is reachable from every other vertex. A loop is just a simple form of SCC. If no SCCs are present, building the desired progress would be simpler, because the TMG will be acyclic. Notice that a vertex is either in an SCC or not. We can “encapsulate” each SCC and replace it with a set of arcs that reflect the net result of the control transfer. The transformation is illustrated in Fig. 3. With that, the problem becomes: (1) how to define the progress over an acyclic TMG, and (2) in a SCC. We now focus on solving (1). Said SCC encapsulation is intuitive. We now define it formally. Let  $\mathcal{S}$  denote the set of all vertices in an SCC. A vertex  $u$  is a predecessor of the SCC if and only if  $u \notin \mathcal{S}$  and there exists an arc  $(u; v)$  such

that  $v \in \mathcal{S}$ . A vertex  $v$  is a successor of the SCC if and only if  $v \notin \mathcal{S}$  and there exists an arc  $(u; v)$  such that  $u \in \mathcal{S}$ . Let  $P(\mathcal{S}_i)$  be the set of predecessors of  $\mathcal{S}_i$ . Similarly,  $S(\mathcal{S}_i)$  for the set of successors. The resulting transformed graph, called the birdseye TMG  $G^{DAG}$ , is formally defined as follows. The acyclic nature of  $G^{DAG}$  facilitate progress definition.

#### Definition II: Birdseye TMG

Given a TMG

$$G = fV_{scc} [ V_{scc}^c : A_{scc} [ A_{scc}^c ] g \quad (3)$$

where  $V_{scc}$  is the set of vertices of some SCCs  $\mathcal{S}_{scc}$ ,  $V_{scc}^c$  is the complementary set;  $A_{scc}$  is the set of arcs, each of which has at least one endpoint within some SCC  $\mathcal{S}_{scc}$ ;  $A_{scc}^c$  is the complementary set. A birdseye TMG

$$G^{DAG} = fV_{scc}^c \cdot A_{scc}^c [ A^0 ] g \quad (4)$$

is an acyclic weakly connected graph, where  $A^0$  consists of additional arcs added by the following procedures. For an SCC  $\mathcal{S}$  in  $G$ , the arcs added are defined as:

$$A_i = f(u; v) \mid u \in P(\mathcal{S}_i) \wedge v \in S(\mathcal{S}_i) g \quad (5)$$

With  $T_{move}(u; v) = T_{tail}(v) - T_{tail}(u)$ . All  $T_{tail}$  remain the same. Then  $A^0$  is the union of all  $A_i$

$$A^0 = \bigcup A_i \quad (6)$$

Walk over TMG. A TMG or Birdseye TMG do not contain the information regarding actual execution for a concrete execution. For any concrete execution, it can be represented as a sequence of vertices on a TMG or Birdseye TMG. We name the said sequence as walk. A walk is a path corresponding to the control transfers in a concrete run, defined as

#### Definition III: Walk over TMG

Given a TMG  $G = (V; A)$ , a walk over  $G$  is a sequence of vertices  $w = (v_1; \dots; v_N)$  such that for all  $i \in \{1; \dots; N-1\}$ , the arc  $(v_i; v_{i+1}) \in A$ .

The subscript denotes the order in which PC visits a vertex. For example,  $v_i$  is the  $i$ th vertex reached by PC. A vertex can be reached more than once due to the presence of loops. The walk is well-defined on both  $G$  and  $G^{DAG}$ . Now define a function  $f_w : V \times V \rightarrow W$  that maps two vertices to the set of all possible walks from the former to the latter. On a TMG,  $f_w(u; v)$  can be an infinite set due to the existence of SCC(s), but on  $G^{DAG}$  the set is always finite. The finiteness allows us to find the walk that results in contention-free WCET.

Intuition III: The acyclic property of a birdseye TMG allows us to enumerate all possible walks, inspect the timing information, and find the walk that results the contention-free WCET.

current progress  $P(v)$  is the summation of each fractional progress contribution:

$$P(v_j) = \sum_{i=0}^j \dots \quad (11)$$

This definition of progress has the following properties. First (1) it is normalized to  $[0; 1]$  with

$$P(v_{\text{entry}}) = 0 \quad (12)$$

$$P(v_{\text{exit}}) = 1 \quad (13)$$

Fig. 4: Left: red arcs indicate  $w$ ; evaluate the progress along with the alternative walk indicated by the green arrow; Middle: from the sub-graph (solid line), find the new nominal walk  $w$  represented by the solid red lines; Right: take the relevant sub-walk  $w^0$ , represented by the solid red lines.

Given a  $G^{\text{DAG}}$ , and denote  $W^{\text{DAG}} = f_w(v_{\text{entry}}, v_{\text{exit}})$  as the set of walks from entry to exit. We want to find the walk  $w \in W^{\text{DAG}}$  that corresponds to the path which leads to the contention-free WCET. We name  $w$  as the nominal walk. In the scope of this work, “nominal” means the worst-case behavior in a contention-free environment. Now we present the nominal walk normalization procedure which (1) identifies  $w$ ; (2) calculates the ratio of total progress made between two vertices in  $w$ ; (3) defines the progress at each of the vertices in  $w$ . To identify  $w$ , we need to evaluate the nominal execution time for a generic walk as follows. For a walk  $w \in W^{\text{DAG}}$ , denote the nominal execution time when PC reaches  $v_i$ ,  $C_i$ , whose value is defined inductively

$$C_0 = 0 \quad (7)$$

$$C_{i+1} = \min\{T_{\text{tail}}(v_{i+1}); C_i + T_{\text{move}}(v_i; v_{i+1})\} \quad (8)$$

Eq. 8 takes the minimum of the two, because  $T_{\text{move}}$  is the longest time observed to transfer between  $v_i$  and  $v_{i+1}$  across all profiling data, and, unlike  $T_{\text{tail}}$ , it does not take into account potential temporal dependencies. Thus, merely summing up all  $T_{\text{move}}$  might result in a pessimistic estimation. The use of  $T_{\text{tail}}$  here not only mitigates the pessimism, but also ensures safety, because  $T_{\text{tail}}$  captures the contention-free WCET until execution reaches  $v_i$ .

For all possible walks from entry to exit, the one with the largest nominal time is chosen as the nominal walk, formally

$$w = \underset{w \in W^{\text{DAG}}}{\text{argmax}} C(w) \quad (9)$$

Progress over Nominal Walk. Now define the progress when the execution follows  $w$ . Define  $i \in [0; 1]$  as the fractional progress contributed by the execution from  $v$  to  $v_i$ :

$$i = \frac{C_i}{C(w)} \quad (10)$$

where  $C_i = C_i - C_{i-1}$ . Thus, if at runtime the task's execution follows exactly  $w$  and has reached  $v_i$  then its

Second, (2) if the execution does follow  $w$ , the progress remains in a linear relationship w.r.t. the task's worst-case execution time in isolation. This concludes the nominal walk normalization procedure.

Progress over Alternative Walks. As part of the progress normalization procedure, each arc along  $w$  is assigned a fractional progress  $i$ , and at each vertex  $v_i$ ,  $P(v_i)$  is also well defined. However, progress for arcs and vertices along alternative walks is not yet defined. To define this quantity, we make the following observation.

Intuition IV: An alternative walk always has at least one vertex down the path shared by  $w$ . When reaching a shared vertex, the current progress must be the same as if the execution had been following  $w$  thus far.

We therefore present an iterative algorithm to set values for unassigned arcs and vertices. The idea is to progressively focus on arcs  $(u; v)$  where the predecessor  $u$  is already associated a notion of progress, while the successor's progress  $P(v)$  is still undefined. For the very first call of the algorithm, all arcs  $(u; v)$  such that  $u \in w \wedge v \notin w$  are eligible—see left side of Fig. 4. Consider all paths starting from  $u$  where  $v$  is visited next. All of them will eventually merge into some  $v_{\text{merger}}$  that already has progress assigned. For example,  $v$  in Fig. 4 is  $u^0$ . During the first iteration of the algorithm,  $v_{\text{merger}}$  is guaranteed to be a vertex along  $w$ , but the subsequent iteration of the algorithm can merge into a vertex that has been assigned a progress value even if not necessarily along  $w$ . Among all such paths, we pick the path whose  $v_{\text{merger}}$  has the least progress. If multiple paths with the same  $v_{\text{merger}}$  exist, we reuse the aforementioned nominal walk normalization procedure to pick the one with the largest nominal time—i.e., the worst-case sub-path starting with arc  $(u; v)$ . We can form a new birdseye TMG  $G^{\text{DAG}}_{(u,v)}$  which includes  $u$  as its entry vertex, the arc  $(u; v)$ , and all vertices and arcs reachable from  $v$  through some path to  $v_{\text{merger}}$ . Next we apply nominal walk normalization procedure on  $G^{\text{DAG}}_{(u,v)}$  to identify nominal walk  $w^0$  of  $G^{\text{DAG}}_{(u,v)}$ . Before applying the procedure, we need to subtract  $T_{\text{tail}}(u)$  from the  $T_{\text{tail}}$  of each vertex in  $G^{\text{DAG}}_{(u,v)}$ , because  $T_{\text{tail}}$  is the time elapsed since the execution of the TMG.

Let  $u^0$  be the first vertex where convergence, i.e., the  $v_i$  occurs (center of Fig. 4). We want the sub-walk  $w^0$  from  $u^0$  to  $u^0$  (right side of Fig. 4) to contribute the same amount of progress as if the original nominal walk  $w$  from  $u$  to  $v_{\text{merger}}$ , i.e., the progress contribution equals to  $P(v_{\text{merger}}) - P(u)$ , where  $P(u) = 0$ .

<sup>3</sup>This enumeration does not suffer from exponential complexity in practice because the number of vertices and their out-degree is generally small for a TMG. A TMG with less than 20 vertices can already offer satisfactory monitoring granularity [20], [22].

is the progress at  $v_i$  following  $w$ . Now, we treat  $w$  as a TMG, where and apply the nominal walk normalization procedure again to calculate the fractional progress  $\frac{P(v_i)}{T_{scc}}$  for each edge in  $w$  (Eq. 10). The superscript  $00$  indicates this fractional progress is based on the TMG  $w$ . The nominal fractional progress from  $v_{i-1}$  to  $v_i$  along the alternative path is

$$p_i = \frac{w_{i-1,i}}{T_{scc}} (P(v_i) - P(v_{i-1})) \quad (14)$$

With  $p_i$  assigned, the progress at each vertex can also be assigned according to Eq. 11. This algorithm is invoked iteratively until the progress at all the vertices has been assigned. At runtime, when the PC hits a vertex  $v$  on the application's DAG, the progress is  $P(v)$ . The progress has the following properties: For any paths leading to  $v$ , the sum of all along that path equals to  $P(v)$ , i.e., the progress at each vertex is unique regardless of path taken, and the progress always monotonically increases due to  $t_q > 0$ .<sup>4</sup> Progress over an SCC. Now that the progress over DAG is well-defined. We focus our attention on the progress while executing inside an SCC. Given the arbitrary control flow, the execution can enter and exit the SCC at any predecessor and successors, and iterates inside SCC for a nondeterministic number of times. To define the progress under this circumstance, we realize

Intuition V: While the execution is inside an SCC, treat it as if it will eventually leave the SCC at the successor which has the least progress.

Despite the many possible control flows within an SCC, this complexity does not affect the reasoning at the level of birdseye TMG, because any SCCs are squashed into arcs. Thus, as per intuition above, when the execution enters the SCC from a predecessor, we can conservatively assume that it will leave the SCC via the successor with the least progress calculated in Eq. 11. At runtime, if it exits via other successors, it is always possible to in-state the progress to match the corresponding successor. This guarantees that the worst-case solo progress is never overestimated.

Assume the execution enters it via predecessor  $u$ , and the successor corresponding to the least amount of progress is  $v$ . Denote the least possible progress contributed by the SCC as  $T_{scc} = P(v) - P(u)$ . Then the question becomes how to distribute  $T_{scc}$  while the execution is inside the SCC. Similar to Eq. 10 and Eq. 14, where the denominator is the solo worst-case time elapsed for  $w$  or some sub-walk, we also need a similar quantity for the SCC,  $T_{scc}$ . We can conservatively approximate  $T_{scc}$  as

$$T_{scc} = T_{tail}(v) - T_{tail}(u) \quad (15)$$

Inside an SCC, the progress contributed by the execution from the  $(i-1)$ th to  $i$ th vertex-hit is inductively calculated as

$$p_i = \frac{A_i}{T_{scc}} \quad (16)$$

$$A_0 = 0 \quad (17)$$

$$A_{i+1} = \min_{scc} \{ A_i + T_{move}(v_i; v_{i+1}) \} \quad (18)$$

The intuition is that Eq. 18 ensures  $A_i$  is at most some  $T_{move}$ , i.e., the longest time PC takes to move from one vertex to the other. The ratio between  $A_i$  and  $T_{scc}$  is then used to assign a fraction of  $T_{scc}$  to the progress. The upper bound of  $A_i$  is  $T_{move}$ , which makes the progress estimation conservative.

If at runtime a loop exceeds the maximum number of iterations ever seen during the profiling phase, Eq. 18 will no longer increase, indicating that no further progress is being made. This is the desired behavior because, for each milestone hit, the progress and the job's CPU time are compared to determine whether the execution satisfies the temporal requirements (see Section V-D). In the event of unexpected higher loop iterations, the ever-increasing CPU time compared with stagnant progress will signal a temporal abnormality.

Effective and Nominal Progress The three expressions for Eq. 10, 14, and 16 define how to evaluate the progress when the execution is on the worst-case path, on some alternative paths, and inside some SCCs. The evaluation is conducted when PC reaches any of the vertices

$$P(v_0) = 0 \quad (19)$$

$$P(v_{i+1}) = P(v_i) + p_{i+1} \quad (20)$$

We name the above progress evaluation as effective progress, which is a function of the  $i$ th reached vertex.<sup>5</sup> We can also define the effective progress as a function of wallclock time,  $P(t)$ . Assume the  $j$ th vertex-hit is the latest one prior to  $t$ . We conservatively define

$$P(t) = P(v_j) \quad (21)$$

In contrast, we name the progress made as if the execution takes the worst-case path without contention as nominal progress  $\bar{P}$ . The property of CAPATASK ensures that this is linear to execution time. Assume the contention-free WCET is  $C$ . Then when the effective nominal progress ratio is  $1=C$ . Then the expression for nominal progress is

$$\bar{P}(t_{CPU}) = t_{CPU} / C = kt_{CPU} \quad (22)$$

which is a function of processor time  $t_{CPU}$ , i.e., the time spent by the task under analysis while actively running on the CPU. In Section V-D and Section VI, we shall see how to leverage  $P(v)$  and  $\bar{P}(t)$  to compute slack at runtime to achieve progress-aware scheduling.

## V. CAPASYS DESIGN: SYSTEM-LEVEL CAPA SUPPORT

This section presents our design of CAPASYS, a runtime infrastructure to instantiate the CAPATASK on one commercial multi-core hardware platform. We first review key platform assumptions (Section V-A), next we describe the main responsibilities of CAPASYS and the corresponding implementation details (Section V-B and V-C). Beyond implementing progress

<sup>4</sup>A proof is provided in Appendix II.

<sup>5</sup>The computation of effective progress at runtime is also summarized as pseudo-code in Appendix III.

Fig. 5: A TMG used by tracer at runtime. From left to right, three snapshots are provided in temporal order: (1) PC is somewhere on the arc ( $\mu_2$ ), and the TU is waiting for the event indicating  $\mu_1$  was hit. (2) When PC reaches  $\mu_2$ , the TU notices the system and the tracer starts to re-configure the TU. (3) While execution is on the arc ( $\mu_3$ ), the TU is waiting for  $\mu_3$  or  $\mu_4$  to be the next hit.

tracking, CAPASys also uses the progress information exposed by CAPATASK to add support for progress-driven CPU management primitives, namely self-halting and assurance halt. Finally, we evaluate the overhead and capabilities of CAPASys in Section V-D.

#### A. Assumptions for CAPASys Deployment

For a practical instantiation of CAPASys, following assumptions are made. (i) Black-box binaries of the task are available for offline analysis. (ii) The target is a multi-core platform with high-performance cores. The high-performance cores hosting the managed tasks are called tracees. (iii) Spare (co-)processors exist to instantiate the CAPASys runtime. The latter, referred to as tracers, can have a low-power/low-performance profile compared to the tracees. (iv) The tracees can conduct self-hosted tracing, i.e., it can generate an execution trace and save it to on-chip memory using a hardware Trace Unit (TU). A trace reports the processor control flow at the level of basic blocks, as described more in-depth in [22]. (v) We assume that the TMGs have been constructed and focus on the instantiation and use of the CAPATASK.

#### B. Multi-core CAPASys Responsibilities

The proposed CAPASys implements multi-core support for TPA. It has three key responsibilities.

(R1) Milestone Detection and TU (Re)configuration. TUs can inform the system when the PC of a given CPU hits a pre-defined virtual address. Nevertheless, the number of addresses that can be concurrently monitored is limited. To overcome this limitation, Fig. 5 depicts the approach used by CAPASys. As shown in the figure, the module uses the TMG to (re)configure the TUs to only monitor the vertices that can be potentially hit next, i.e., the immediate successors of the current milestone. Thus, CAPASys must enable efficient milestone detection and TU reconfiguration.

(R2) Multi-core Trace Data Processing. A TU is a per-core to asset. When properly configured, each TU emits trace packets upon a vertex hit. Typically, trace packets from different TUs are multiplexed before being routed downstream. CAPASys must demultiplex the trace stream to recover individual packets

Fig. 6: Job releases at  $t_1$ . The slopes of nominal progress and setpoint are  $k$  and  $k'$  respectively. The vertical gap between the green line and setpoint is  $\delta$ . The effective progress is the blue line. At  $t_1$ , the slack (vertical gap between the blue line and setpoint) is larger than  $\delta$ . Thus the task self-halts until  $t_2$ . Job completes at  $t_3$ .

and identify the originating core, task, and vertex associated with the hit. Next, the event is timestamped using the cycle counter, and TU reconfiguration is triggered.

(R3) Progress-driven CPU Management. CAPASys also exports key progress-driven CPU management capabilities, namely (1) self-halting and (2) external blocking. Self-halting refers to the ability of CAPASys to slow down the progress of a task on a given CPU if its progress is ahead of schedule. This allows for relieving contention effects on shared resources when it is safe to do so. External blocking refers to the ability of CAPASys to slow down progress on cores hosting less important tasks or ahead-of-schedule tasks if the progress of the task on the current core falls below the desired progress setpoint. We will quantify this in Section V-C (R3).

#### C. Implementation of Key Responsibilities

We implemented CAPASys on the ZCU102 development board featuring an Xilinx UltraScale+ MPSoC. It consists of two CPU clusters: (1) the APU cluster contains four ARM Cortex-A53 CPUs at 1.3GHz serving as the tracees; (2) the KPU cluster contains two ARM Cortex-R5 CPUs at 500MHz, used as the tracers. The target includes ARM CoreSight infrastructure modules that allow self-hosted tracing, in which the Embedded Trace Macrocells (ETMs) serve as TUs. Fig. 7 shows an overview. When ARM CoreSight is not available, several alternatives exist to perform multi-core progress tracking. (i) First, equivalent hardware tracing capabilities are

offered by Intel Processor Trace, RISC-V E-Trace and N-Trace, MIPS PDtrace, etc. (ii) Second, hardware breakpoints can be used to achieve monitoring [20]. (iii) Finally, software-based approaches have been explored [17], [19].

(R1) Milestone Detection and TU (Re)configuration. In Chen et al. [22], vertex hits are detected by dynamically reprogramming ETM filters, which frequently toggles ETM state and emits extraneous metadata packets. We improved this by utilizing ETM events instead: the PC reaching a user-defined virtual address can be programmed as an ETM event. When it occurs, only one 1-byte packet is emitted.

The ETM filter can be set to a null range (e.g. [0x0,0x0]) to suppress any unnecessary trace data. Once the CPU that originated the event is recovered (see R2 below), the TMG of the corresponding foreground task is queried for the set of possible next milestones. Next, the local ETM is temporarily disabled, reconfigured, and re-enabled.

Fig. 7: Green arrows are trace data stream paths. Gray shapes are trace data interconnects—CoreSight funnels and replicators. The first Cortex-R5 polls the first TMC for trace data. Red arrows are hardware signals for just-in-time flush. Blue arrows represent re-configuration and halt/resume operations.

(R2) Multi-core Trace Data Processing. The ETM only traces the activity on the local core. It does so by producing a stream of trace packets for downstream consumers. When multiple ETMs are active in a multicore platform, trace packets from different ETMs are multiplexed into 16-byte frames according to the Arm CoreSight formatter protocol. The consumer must demux the stream to recover the packets of individual cores. We use the first Cortex-R5 to demux. The recovered packets are written to local memory that is accessible by both Cortex-R5s. The second Cortex-R5 parses the packets which include the ETM event just occurred, indicating the virtual address that PC just reaches (see Fig. 6). With this information, tracer can identify the vertex hit in a TMG, thus tracking the execution path.

Polling Trace Data. In Chen et al. [22], the trace data go through three trace memory controllers (TMCs) to reach the consumer. We discovered that the trace data can be acquired by using software to poll any of the TMCs in software FIFO mode. Thus, only one TMC is involved in this implementation. Fig. 7 shows the said two paths.

Just-in-Time Flush. Multiple CoreSight components handle trace routing and must be flushed promptly for real-time delivery. However, frequent flushes can overload the channels with dummy data [22]. To avoid this, CAPASYS leverages the ETM event outputs to trigger flushes only on vertex hits, using external pins routed to the TMC (red arrow in Fig. 7).

(R3) Progress-driven CPU Management. The aforementioned self-halting and external blocking rely on CAPASYS's low-level capability to halt/resume cores, which is enabled by the Cross-Trigger Interface (CTI). The per-core CTI can be used to instruct each core to enter/leave debug state, a technique used in [4], [5], [31] that prevents cache pollution on the traces and that can be kept transparent to the OS.

Below are the detailed behaviors of the regulation. CAPASYS allows one to specify a per-task degradation factor  $\alpha \in (0; 1)$ . The progress setpoint is then computed as  $P(t)$ . Upon a vertex hit, CAPASYS recomputes the current slack as  $s(i) = P(v_i) - P(t)$ . A positive slack indicates a task's progress ahead of the setpoint, while a negative slack indicates that it is falling behind. With this information and the ability to halt/resume CPU activity, CAPASYS implements the following two regulation mechanisms.

Self-halting Regulation. When a task accumulates enough positive slack, the core hosting it will be temporarily halted so that tasks on other cores can progress faster due to reduced contention. CAPASYS accepts a parametric safety margin  $\beta \in (0; 1)$  capturing the required slack before self-halting.

External Blocking. CAPASYS accepts a configurable ranking on each CPU in relation to the other ones. This can be used to describe differences in CPU-wide time assurance levels in the presence of non-real-time tasks on some CPUs. This information is then considered by CAPASYS to halt lower-ranking CPUs if tasks on higher-ranking CPUs are not making sufficient progress. Specifically, upon the vertex-hit for a task/CPU with higher ranking  $i$  if  $s(i) < 0$ , CAPASYS will halt other lower-ranking cores. Next, when the  $(i+1)$ th hit occurs, three possible outcomes exist: (a) if  $s(i+1) > 0$ , the cores halted by the considered task are resumed; (b) if  $0 > s(i+1) > s(i)$  (negative slack shrinking) no action is taken, awaiting future vertex hits; (c) if  $s(i+1) < s(i)$  (negative slack growing) an additional eligible core to halt is chosen until outcome (a) or (b) occurs.

State Change due to Halt/resume. Although halting and resuming are conducted via the debug interface, not altering the core state during the freeze, some shared resources such as the last-level cache (LLC) can still be affected by contenders. When a task is resumed after a period of halting, it might need to warm up the LLC, incurring additional overhead. This effect is negligible in our setup; however, if it becomes significant, the overhead should be subtracted from the slack.

Watchdog Enforcement. Since CAPASYS does not track progress between vertex hits, long gaps may jeopardize the system. To prevent this, CAPASYS can optionally program a per-task watchdog to trigger if the next milestone is not reached within the expected time.

Safety Implication. The safety implication of the ranking configuration is that only tasks running on the highest-ranked core are guaranteed to meet their deadlines, as they can run in isolation when necessary. In contrast, cores with lower rankings receive resources on a best-effort basis. Therefore, the current regulation is more suitable for scheduling soft/ firm real-time tasks. Section VI-D further explores how progress information can be leveraged to improve scheduling by combining the spare utilization of some existing scheduling algorithm.

External blocking favors CPUs of higher ranking. Thus, a fairness metric is provided in Appendix V.

<sup>7</sup>A pseudo-code is provided in Appendix IV.

Fig. 8: Three real-time disparity tasks with degradation factors (0:87; 0:8; 0:71).

Fig. 9: Three real-time disparity tasks with degradation factors (0:92; 0:83; 0:69).

Hardware Consideration. Compared with other software-suite, texturesynthesis, stitch and localization were excluded based progress tracking methods [17], [18] in which a core because largely CPU-bound, while multicore was excluded hosts both user tasks and regulation logic, CAPASYS uses because it does not run to completion on the selected platform. additional core(s) to host progress tracking and regulation from the TACLeBench suite, we selected the most memory-logic. This is a trade-off between hardware requirements intensive benchmark, namely mpeg2. All the selected bench- and timing overhead. If the target platform does not have marks, except mpeg2, are considerably impacted by interfe- spare (co)processors, the regulation logic of CAPASYS can effects. Additionally, mser is particularly representative be hosted on one of the application cores as well. In this case, illustrate the limitation of CAPASYS when the TMG has the time overhead should be accounted for by discounting the granularity.

Overhead and Latency. The overhead of CAPASYS is computed by comparing the runtime of all the considered applications with and without active progress tracking. We

#### D. CAPASYS Evaluation

We evaluate CAPASYS 's ability to support progress-driven multi-core management without relying on a scheduler. Our observe that the overhead of CAPASYS on task executions is upper bounded by 0:4%. To measure the latency of milestone-task is activated per core in sync, using a xed ranking bit detection, we use a synthetic task that self-reports the (CPU1 > CPU2 > CPU3 > CPU4). We extract the TMG and absolute timestamps of milestones. We then compute how instantiate the CAPA model using realistic benchmarks from many clock cycles later CAPASYS can detect the same event. SD-VBS [32] and TACLeBench [33] as real-time workloads. In our measurements, the latency is upper-bounded by 16.4 s on CPUs 1–3. A memory-intensive best-effort task from with 4.5 s on average.

Multi-core Management. Table II presents the TMG prop- degradation factors ( $\delta_0$ ;  $\delta_1$ ;  $\delta_2$ ) and a xed safety margin erties and runtime information of the considered benchmarks. = 0:01 for all real-time tasks. Though a smaller value of We further present the plots for the most representative exper- will enact more frequent regulation, we nd, experimentally, ments<sup>8</sup>. In our rst experiment, depicted in Fig. 8, three dis- the system performance depends considerably more on the parity(s) benchmarks from SD-VBS are deployed as the real- granularity of TMG than the value of .

Benchmark Selection and Characteristics. disparity, track- <sup>8</sup>Due to space constraints, we omit the plots of benchmarks such as tracking ing, svm, and mser were selected from SD-VBS. From this and svm since they are very similar to what is depicted in Fig. 9 and Fig. 10.

Fig. 10: Three real-time mpeg2 tasks; all of them have sufficient positive slack, thus enacting self-halting.

Fig. 11: Three real-time msr(s) tasks; due to the low-granularity TMG, management is not effective. Additional horizontal dotted lines are added for each milestone hit to visualize the low-granularity issue. It can be seen that many milestone hits are concentrated above  $\rho = 0.5$

benchmarks	vertices	arcs	SCCs	vertex-hits	min/max/avg $T_{move}$ (s)	nominal WCET (s)
disparity	11	12	1	137	1270 / 20828 / 11611	1579172
mpeg2	3	3	1	18	6 / 5546 / 5127	87173
msr	10	11	1	10	185 / 207880 / 46447	418025
svm	6	9	1	45	195 / 114485 / 12791	562839
tracking	11	12	1	17	110 / 104201 / 48219	771506

TABLE II: Properties of TMGs for each benchmark. The vertex-hits column is measured at runtime. All other columns are static data obtained during the profiling phase.

time tasks on CPU 1–3. Each task is assigned a color profile in a marker on the effective progress indicates that agreement with the CPU ranking: green (CPU1,  $\rho_1 = 0.87$ ), CAPASys enacts self-blocking. A color-coded 6 marker yellow (CPU2,  $\rho_2 = 0.8$ ), red (CPU3,  $\rho_3 = 0.71$ ), and indicates that external blocking is performed. For example, gray (memory-bomb, CPU4). The degradation factor values gray 6 on the green effective progress line indicates that here are chosen to demonstrate scenarios in which CAPASys halts the gray task because the green task has a exhibits interesting behaviors. Because the evaluation in the negative slack. A I marker indicates that halted cores are section focuses on inter-core contention effects and considered. Thus 6 and I typically come in pairs. By setting only one active job per core, schedulability considerations distinct degradation factors, the first two disparity tasks can not be meaningfully discussed yet. In Section VI-D we setpoints above the interference line, the third below. The will explore the advantages of leveraging progress information zoom-in shows various CAPASys actions to keep the progress tion when multiple tasks per core are active, thus enabling of each task to the vicinity of their setpoints. progress-aware, contention-aware multi-core scheduling. Behavior under Unsatisfiable Constraints. In this experi-

Fig. 8-left describes the scheduling diagram with progress annotated: The gray vertical bars interposed on the green degradation factors:  $\rho_1 = 0.92$ ,  $\rho_2 = 0.83$ ,  $\rho_3 = 0.69$ . This yellow, and red indicate the duration of the self-blocking. The places the system slightly outside the feasibility envelope. In gap indicates the task is halted due to another task on a higher ranking CPU. The task on the higher-ranking CPU triggering the halt is shown as color strips on top of the gap. miss) and it is halted a number of times to allow higher

Fig. 8-right focuses on the per-task progress. The plot zoomed in to show the activities when execution time is between 125M to 185M cycles. Important progress lines, as in Fig. 6, are shown: nominal, setpoint, and effective progress. Self-blocking for Positive Slack. We evaluate CAPASys on interference progress line is also added for reference, which is obtained by running the same setup but where CAPASys takes no management actions. The first experiment has  $\rho_1 = 0.83$ ,  $\rho_3 = 0.71$ ). Here, the Interference line closely tracks interference line ends 210M cycles, which is significantly larger than its contention-free WCET (150M cycles). slack. CAPASys enacts self-blocking to align progress with

setpoints. While we currently use core halting, other strategies—e.g., executing lower-priority tasks—are also viable. The frequency of monitoring on mpeg2 is much higher (300 kilo-cycles per invocation), compared to those of disparity and mser (1333 and 3000 kilo-cycles per invocation). In the  $n^{\text{th}}$  segment,  $a_n$  denotes the number of progress units (PUs) required to complete the segment. A task with  $N$  segments requires a total of  $\sum_{n=1}^N a_n$  PUs to complete. While high monitoring frequency. However, if the excessive accesses to debug infrastructure is a concern in practice, then some invocations can be skipped to proactively constraint the access. We leave this as future research.

Limitations from Low TMG Granularity. The natural experiment highlights how low-granularity TMGs can limit CAPASYS's ability to make fine-grained decisions. Using the mser benchmark from SD-VBS for all tasks ( $\alpha_1 = 0:91$ ,  $\alpha_2 = 0:83$ ,  $\alpha_3 = 0:71$ ), where most milestones occur before progress 0.6, Fig. 11 shows that coarse milestone spacing hinders maintaining progress above the setpoint. Improving TMG granularity is part of our future work.

## VI. CAPASIM DESIGN: CAPA-BASED MULTI-CORE SCHEDULING SIMULATOR

Up to and including Section V, all evaluations were conducted on real hardware without assuming a specific scheduler. This section explores the potential of integrating CAPASYS's management capabilities into a multi-core scheduler—a direction that opens a broad design space for future research. To enable rapid prototyping and hypothesis testing, we also introduce and implement CAPA-based Multi-core Scheduling Simulator (CAPASIM). With the help of CAPASIM, the performance of CAPASYS's regulation is further evaluated in this section, and a comparative study of shared resource partitioning techniques is also presented.

### A. Motivation for CAPASIM

While many scheduling simulators exist [35], [36], these treat jobs as indivisible atoms and do not precisely model shared resource contention effects. Conversely, the proposed CAPASIM models shared resource contention effect. Admittedly, shared resource contention modeling is challenging. The type and topology of the shared resources produce different contention patterns [37]. Thus, our CAPASIM provides the contention model as an interface so that different models can be explored. Nevertheless we propose a baseline statistical contention model. The CAPASIM is open-source to encourage community reuse. The design of CAPASIM is purposefully de-coupled from CAPATASK and CAPASYS, and is a self-contained simulator, so that CAPASIM can be used in a broader scope of research. We will first describe how CAPASIM functions, and then apply CAPATASK and the regulation techniques of CAPASYS in the simulation setup.

### B. Simulation Model

Due to space constraints, the complete simulation model can be found in Appendix VI. Here, we only present the key

parameters necessary to follow the rest of the paper. A task  $\tau_i$  is represented as a sequence of tuples:  $(a_n; b_n) \mid n \in \mathbb{N}^+; a_n \in \mathbb{N}^+; b_n \in [0; 1]; n \in \mathbb{N}^+$  (23)

In the  $n^{\text{th}}$  segment,  $a_n$  denotes the number of progress units (PUs) required to complete the segment. A task with  $N$  segments requires a total of  $\sum_{n=1}^N a_n$  PUs to complete. While high monitoring frequency. However, if the excessive accesses to debug infrastructure is a concern in practice, then some invocations can be skipped to proactively constraint the access. We leave this as future research.

Limitations from Low TMG Granularity. The natural experiment highlights how low-granularity TMGs can limit CAPASYS's ability to make fine-grained decisions. Using the mser benchmark from SD-VBS for all tasks ( $\alpha_1 = 0:91$ ,  $\alpha_2 = 0:83$ ,  $\alpha_3 = 0:71$ ), where most milestones occur before progress 0.6, Fig. 11 shows that coarse milestone spacing hinders maintaining progress above the setpoint. Improving TMG granularity is part of our future work.

Augmenting P-RM and P-EDF with CAPASYS

As a first look at the benefits of integrating the progress-aware management primitives implemented by CAPASYS into multi-core schedulers, we simulate an integration with Partitioned Rate-Monotonic (P-RM) [39] and Partitioned Earliest-Deadline First (P-EDF) [40] scheduling. We refer to the P-RM (resp., P-EDF) scheduler augmented with CAPASYS as CP-RM (resp., CP-EDF). To do so, suitable degradation factors ( $\beta_i$ ) have to be set for each task. For the remaining section, any mentioning of utilization corresponds to the contention-free WCET<sup>9</sup>. Spare utilization  $U^0$  is used as a heuristic to set  $\beta_i$ . Recall that a sufficient condition for a fully preemptive periodic taskset to be schedulable under RM on single-core is  $U \leq m(2^{\frac{1}{m}} - 1)$ , in which  $U$  is the total utilization, and  $m$  is the number of task(s). We define per-core spare utilization  $U^0 = m(2^{\frac{1}{m}} - 1) U$ , which can be uniformly distributed to each task. For EDF on a single core, the taskset is schedulable if  $U \leq 1$ . Thus for each core under P-EDF, the spare utilization is  $U^0 = 1 - U$ . The safe execution time of each task can, therefore, be safely increased to

$$C_i^0 = T_i (U_i + U^0 = m) \quad (24)$$

With  $C_i^0$  being the setpoint, it can be calculated  $C_i = C_i^0$ . Experimentally, the following two optimizations achieve better performance: (1) Use only self-halting, not external-blocking. Using both improves timeliness on high time-assurance cores, but increases deadline misses on low-assurance cores. While this yields better timeliness on high time-assurance cores than plain P-RM/EDF, the overall schedulability under CP-RM/EDF worsens. In contrast, applying only self-halting leads to better overall performance than

<sup>9</sup>Using contention-free utilization is the choice of the proposed algorithm. Methods that absorb the contention/interference/noisy effect into the utilization exist, such as mixed criticality systems [41].

Fig. 12: Total utilization is increased in increments of 1%. For each utilization value, 100 tasksets are generated and tested against each scheduler. This process is repeated for 100 rounds. The envelope represents the maximum and minimum percentages of schedulable tasksets observed across the 100 rounds. From left to right, the utilization balances are 0:33; 0:43; 0:75, and the consumption factors, are sampled from Gaussian distribution with mean 0:3; 0:2; 0:5, and standard deviation 0:025. The  $U_{max}$  is is in ated by 6 ; 3 ; 3 . <sup>10</sup>

Fig. 13: For each Gaussian mean and utilization balance pair, all possible total utilization (from 0% to 400% with 1% incremental) are run with 100 independent tasksets for each. Let  $y$  in the heat maps on left be the total number of schedulable tasksets for P-RM/EDF, and on right for P-RM/EDF with resource partitioning or CP-RM/EDF. The score in the entry is  $(y^0 - y)/N$  in percentage, where  $N = 40000$  is the total number of tasksets tested to produce one entry (100 tasksets for each total utilization 100 400). At  $y = 0:25$  on left, all entries are zero, because the optimization from Eq. 25. Six colored boxed entries correspond to the scenarios tested as in Fig. 12.

P-RM/EDF. Moreover, since P-RM/EDF does not distinguish balance factor  $y_b$ , indicates the percentage of total utilization assurance levels, comparing it with self-halting alone—which is assigned to the first core. Given a total utilization  $U$ , the also lacks such differentiation—is more appropriate. (2) Let  $U_i$  be the average per-core utilization. Only cores with  $U_i < U/z$  enact self-halting, where  $z = 1:1$  is a cutoff value. The intuition is that although self-halting causes processor time to be lost, cores with below-average utilization can yield shared resources via self-halting.

D. CP-RM and CP-EDF Simulation Experiments

The simulated system consists of 4 cores, each running EDF scheduling against their CAPA variants, i.e., CP-RM and 4 tasks, with the following setup: the utilization of the first core is no less than that of the remaining three cores; the three remaining cores have identical utilization; The utilization under maximum contention is unknown. We will use feasible to describe the absence of deadline misses within a finite simulation duration of a taskset. Thus Fig. 12 represents

<sup>10</sup>The advancement model of CAPASIM causes the slowdown to be at most twofold. However, experiments show the contention-induced slowdown can be much more severe [20]. Thus we in ated by  $U_{max}$  to offset the underestimation.

feasibility curves instead of schedulability curves.

In most cases, CP-RM/EDF outperform their vanilla counterparts. The rightmost diagram shows CP-RM under-performs when the utilization is highly unbalanced. We conduct more extensive experiments by swiping a range of parameters. The results are visualized using heat maps on the left in Fig. 13, in which each cell is an aggregation of a feasibility curve like Fig. 12. In heat maps, the y-axis represents the utilization balance factor  $y_{ub}$ . The mean of the  $b_n$  factors is varied in the set  $\{0.05, 0.1, 0.2, 0.3, 0.4, 0.5\}$ ;  $g_{\max}$  is multiplied by  $\{1, 3, 6\}$ . Green/red-shifted entries indicate more/less tasksets are feasible under CAPA variant.

CP-EDF consistently outperforms P-EDF across most test cases. However, results from (C)P-RM—and some extreme (C)P-EDF scenarios—exhibit counterintuitive behavior: under high contention in unbalanced systems, P-RM can outperform CP-RM. This occurs because brief periods of intense contention can yield more total progress than sustained moderate contention over the same window. While even simple integrations of CAPA into basic multi-core schedulers yield significant gains, a systematic investigation is needed to fully harness progress information in multi-core scheduling.

We compare our progress-based scheduling with shared resource partitioning, including way/set-based cache partitioning [43] and memory bandwidth regulation (e.g., MemGuard, MemPol, MemCoRe [3]–[5]). These methods assign cores specific cache slices or bandwidth quotas, suspending execution upon quota violations. We choose these comparisons for two reasons: (1) our self-halting mechanism resembles how these methods conduct regulation, i.e. via core throttling; and (2) other progress-estimation methods either do not include schedulability/feasibility [18]–[20], [22] or are limited to single-core systems with few milestones [17].

We must stress that our implementation of shared-resource partitioning in the simulation does not faithfully replicate real hardware behavior, because the simulation does not explicitly model hardware resources. Instead, we focus on modeling the effect that access to a slice of shared memory resources has on applications' runtime. This is intended as an illustrative demonstration showing that indiscriminately slowing down the memory phases of applications performs worse than purposefully managing their behavior by leveraging progress information. The heat maps on the right of Fig. 13 show the result. CP-RM/EDF continue to outperform P-RM/EDF with resource partitioning. We model shared resource partitioning as follows. Recall that at each simulation cycle, a running task has a shared resource-consumption factor  $b$  that parametrizes the density of memory operations in the current phase. As such, if for  $g_{\text{window}}$  consecutive simulation cycles, the values of  $b$  of a task at each cycle are all larger than  $b_{\text{thresh}}$ , a memory stall is introduced in the form of a 1-simulation-cycle CPU stall. This has the effect of slowing down only the memory-intensive phases of applications, which is consistent with the inherent slowdown introduced by shared resource partitioning techniques. Otherwise, the underlying scheduler remains P-RM/P-EDF. We set  $b_{\text{thresh}}$  equal to the mean of the Gaussian where  $b_n$  is sampled, and  $g_{\text{window}}$  equal to 65. Although not shown in the heatmap, we also experiment with  $g_{\text{window}}$  equal

to 16; 33; 120; 240, corresponding to the slowdown of memory phase by 6.25%; 3%; 0.83%; 0.41%. Their performance is very similar to what we observed with  $g_{\text{window}} = 65$ .<sup>11</sup>

## VII. CONCLUSION

This paper presents the CAPA framework—comprising the CAPATASK, CAPASYS, and CAPASIM—to advance progress-aware real-time systems along four dimensions. First, the CAPATASK enables runtime tracking of task progress with complex control flows. Second, CAPASYS extends state-of-the-art TPA implementations to support concurrent multi-core execution with minimal overhead. Third, CAPASIM models task progress and shared resource contention. Fourth, we explore using live progress information to drive system-wide scheduling decisions under CP-RM and CP-EDF, showing promising improvements in feasibility under contention.

While CAPA opens new directions in progress-driven scheduling, further research is needed to fully realize its potential. Future work will explore global scheduling policies and leverage CAPA for enhanced task observability and profile-driven resource management under contention.

## REFERENCES

- [1] C. Maiza, H. Rihani, J. Rivas, J. Goossens, S. Altmeyer, and R. Davis, "A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems," *ACM Comput. Surv.*, vol. 52, no. 3, Jun. 2019.
- [2] T. Lugo, S. Lozano, J. Fernández, and J. Carretero, "A survey of techniques for reducing interference in real-time applications on multicore platforms," *IEEE Access*, vol. 10, pp. 21 853–21 882, 2022.
- [3] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.
- [4] A. Zuepke, A. Bastoni, W. Chen, M. Caccamo, and R. Mancuso, "Mem-pol: Policing core memory bandwidth from outside of the cores," in *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2023, pp. 235–248.
- [5] I. Izhbirdeev, D. Hoornaert, W. Chen, A. Zuepke, Y. Hammad, M. Caccamo, and R. Mancuso, "Coherence-Aided Memory Bandwidth Regulation," in *2024 IEEE Real-Time Systems Symposium (RTSS)*. Los Alamitos, CA, USA: IEEE Computer Society, Dec. 2024.
- [6] D. Ottaviano, F. Ciraolo, R. Mancuso, and M. Cinque, "The Omnivisor: A Real-Time Static Partitioning Hypervisor Extension for Heterogeneous Core Virtualization over MPSoCs," in *36th Euromicro Conference on Real-Time Systems (ECRTS 2024)*, vol. 298, 2024, pp. 7:1–7:27.
- [7] J. Marshall, R. Gifford, G. Bloom, G. Parmer, and R. Simha, "Precise cache profiling for studying radiation effects," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 3, Mar. 2021.
- [8] D. Tarapore, S. Roozkhosh, S. Brzozowski, and R. Mancuso, "Observing the invisible: Live cache inspection for high-performance embedded systems," *IEEE Transactions on Computers*, mar 2022.
- [9] S. Roozkhosh, D. Hoornaert, and R. Mancuso, "Caesar: Coherence-aided elective and seamless alternative routing via on-chip fpga," in *2022 IEEE Real-Time Systems Symposium (RTSS)*, 2022, pp. 356–369.
- [10] P. Sohal, R. Tabish, U. Drepper, and R. Mancuso, "E-warp: A system-wide framework for memory bandwidth profiling and management," in *2020 IEEE Real-Time Systems Symposium (RTSS)*, 2020.
- [11] S. Baruah, A. Burns, and R. Davis, "Response-time analysis for mixed criticality systems," in *2011 IEEE 32nd Real-Time Systems Symposium*, 2011.
- [12] S. Baruah, V. Bonifaci, G. D'angelo, H. Li, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie, "Preemptive uniprocessor scheduling of mixed-criticality sporadic task systems," *J. ACM*, May 2015.
- [13] R. Gifford, N. Gandhi, L. T. X. Phan, and A. Haeberlen, "DNA: Dynamic resource allocation for soft real-time multicore systems," in *27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '21)*, May 2021.

<sup>11</sup>The plots can be found in Appendix VII.

