Rasco: Resource Allocation and Scheduling Co-design for DAG Applications on Multicore

ABIGAIL EISENKLAM, University of Pennsylvania, United States
ROBERT GIFFORD, University of Pennsylvania, United States
GEORGIY A BONDAR, University of California Santa Cruz, United States
YIFAN CAI, University of Pennsylvania, United States
TUSHAR SIAL, Iowa State University, United States
LINH THI XUAN PHAN, University of Pennsylvania, United States
ABHISHEK HALDER, Iowa State University, United States and University of California Santa Cruz, United States

As multicore hardware becomes increasingly prevalent in real-time embedded systems, traditional scheduling techniques that assume a single worst-case execution time for each task are no longer adequate, as they fail to account for the impact of shared resources—such as cache and memory bandwidth—on execution time. When tasks execute concurrently on different cores, their execution times can vary substantially with their allocated resources. Moreover, the instruction rate of a task during a job execution varies with time, and this variation pattern differs across tasks. Therefore, to improve performance it is crucial to incorporate the relationship between the resource budget allocated to each task and its time-varying instruction rate in task modeling, resource allocation, and scheduling algorithm design. Yet, no prior work has considered the fine-grained dynamic resource allocation and scheduling problems jointly while also providing hard real-time guarantees.

In this article, we introduce a resource-dependent multi-phase timing model that captures the time-varying instruction rates of a task under different resource allocations and that enables worst-case analysis under dynamic allocation. We present a method for constructing estimates of such a model based on task execution profiles, which can be obtained through measurements. We then present Rasco, a co-design technique for multicore resource allocation and scheduling of real-time DAG applications with end-to-end deadlines. Rasco leverages the resource-dependent multi-phase model of each task to simultaneously allocate resources at a fine granularity *and* assign task deadlines. This approach maximizes execution progress under resource constraints while providing hard real-time schedulability guarantees. Our evaluation shows that Rasco substantially enhances schedulability and reduces end-to-end latency compared to the state of the art.

This work was supported in part by NSF grants CNS-1750158, CNS-1955670, CNS-2111688, and CCF-2326606.
Authors' Contact Information: Abigail Eisenklam, University of Pennsylvania, Philadelphia, United States; e-mail: aei@seas.upenn.edu; Robert Gifford, University of Pennsylvania, Philadelphia, United States; e-mail: rgif@seas.upenn.edu; Georgiy A Bondar, University of California Santa Cruz, Santa Cruz, United States; e-mail: gbondar@ucsc.edu; Yifan Cai, University of Pennsylvania, Philadelphia, United States; e-mail: caiyifan@seas.upenn.edu; Tushar Sial, Iowa State University, Ames, United States; e-mail: tsial@iastate.edu; Linh Thi Xuan Phan, University of Pennsylvania, Philadelphia, United States; e-mail: linhphan@cis.upenn.edu; Abhishek Halder, Iowa State University, Ames, United States and University of California Santa Cruz, Santa Cruz, United States; e-mail: halder.abhishek@gmail.com.

ACM 1539-9087/2025/09-ART153
https://doi.org/10.1145/3761814

CCS Concepts: • **Computer systems organization → Embedded software**; **Real-time system architecture**; **Multicore architectures**; • **Theory of computation → Parallel computing models;**

Additional Key Words and Phrases: Real-time scheduling, resource allocation, DAG scheduling

ACM Reference Format:
Abigail Eisenklam, Robert Gifford, Georgiy A Bondar, Yifan Cai, Tushar Sial, Linh Thi Xuan Phan, and Abhishek Halder. 2025. Rasco: Resource Allocation and Scheduling Co-design for DAG Applications on Multicore. *ACM Trans. Embedd. Comput. Syst.* 24, 5s, Article 153 (September 2025), 27 pages. https://doi.org/10.1145/3761814

1 Introduction

Modern real-time embedded applications are inherently complex: they consist of interconnected tasks that depend on one another through input/output data constraints. For instance, in video processing applications, raw video input is processed through a series of dependent tasks, such as decoding, filtering, motion detection, compression, and encoding. These applications can be naturally modeled as *taskgraphs*—i.e., **directed acyclic graph** (**DAG**) tasks—in which nodes represent sequential task execution and edges represent precedence constraints [24].

As resource demands grow, these applications are increasingly deployed on multicore hardware to exploit the parallelism inherent in the DAG structure. Various multicore DAG scheduling techniques have been developed; however, they primarily focus on CPU alone, often neglecting other shared resources, such as the **last-level shared cache (LLC)** and memory bandwidth. Such negligence can lead to unsafe schedulability results, as tasks running on different cores can interfere with each other by concurrently accessing shared resources, causing their actual execution times to exceed the **worst-case execution time** (**WCET**) estimated in the absence of such interference.

Recent research has begun to address this issue by developing overhead-aware scheduling and analysis, e.g. by reducing inter-core communication [20], accounting for memory contention [6], exploiting cache recency [32], or co-locating tasks to reduce execution time [22]. While these approaches consider shared resources, they do not explicitly *control* how resources are allocated to each task, and as a result, they still suffer from interference caused by concurrent resource accesses.

A simple solution to the interference problem is to evenly divide each shared resource among cores and use the WCETs under this assigned resource budget in existing DAG scheduling and analysis techniques. However, this approach overlooks the actual resource requirements of each task, which can lead to inefficient resource utilization—something crucial for resource-constrained embedded systems. Previous work for independent tasks [10] has shown that the resource needs of a task vary significantly throughout its execution, particularly for data-intensive tasks, such as those in video processing or autonomous driving applications. It has also shown that by reallocating resources to tasks at a fine-grain (i.e., throughout a job's execution) to match their changing demands, we can effectively reduce latency and deadline misses [10]. However, no prior work has been able to accurately uphold hard real-time guarantees while performing fine-grained (intra-job) resource reallocation. Similarly, no existing DAG scheduling method allocates multicore shared resources (e.g., cache and memory bandwidth) in conjunction with scheduling.

To bridge this gap, we propose a *dynamic resource allocation and scheduling co-design* approach that exploits fine-grained variability in tasks' resource demands while preserving schedulability guarantees. However, several challenges must be addressed to achieve this goal. First, to effectively adapt to fine-grained changes in tasks' resource needs, we require a concise timing model that captures a task's time-varying execution speed under different resource budgets and how its speed scales with additional resources. The model must also support WCET analysis under dynamic budget changes to enable schedulability analysis. Such a model does not currently exist. Second,

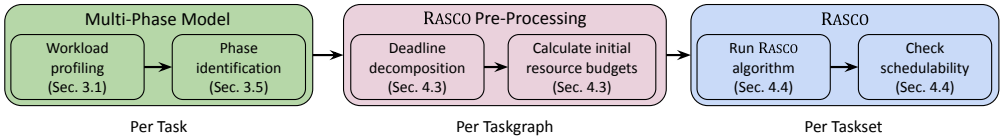


Fig. 1. Workflow of our resource allocation and scheduling co-design algorithm for DAG-based applications.

due to precedence constraints and end-to-end deadlines, resource allocation and scheduling are interdependent. Under deadline-driven DAG scheduling, tasks' deadlines often depend on their WCETs [19], which in turn depend on their allocated resources. A larger task deadline allows for a smaller resource budget, freeing up resources for other concurrent tasks; however, it also tightens deadlines for successor tasks, increasing their resource needs. Finally, the optimal resource budget for a task depends not only on its execution state but also on other concurrently running tasks.

To address these challenges, we present a resource-dependent multi-phase model that captures the time-varying execution behavior of a task under different resource budgets. This model defines a series of execution phases, each corresponding to a distinct worst-case instruction rate, for any given budget of shared resources. Each phase represents a segment of the task's program with a similar instruction rate, typically due to consistent resource usage patterns and needs. To facilitate resource allocation between competing (i.e., concurrently executing) tasks, our model exposes the potential improvement in the worst-case instruction rate when a task is allocated extra resources in its current phase. This information is valuable for determining the tasks that would benefit the most from additional resources. To demonstrate its utility in scheduling and resource allocation, we present a measurement-based method—commonly used for WCET estimation in real-time multicore systems [1, 10, 27]—to construct the multi-phase model.[1] However, this multi-phase model can also be constructed using alternative phase identification and WCET analysis techniques (see Section 3.5).

To leverage our proposed model, we introduce RASCO, the first fine-grained dynamic resource allocation and scheduling co-design method for taskgraphs with schedulability guarantees. Given a set of periodic taskgraphs with end-to-end deadlines, RASCO outputs a schedule of tasks onto cores along with their allocated resources at each point in time. Starting with an initial resource allocation and deadline assignment for tasks, RASCO uses the multi-phase models to iteratively distribute available resources to the tasks that benefit the most. As these tasks' WCETs change, RASCO recomputes task deadlines and redistributes resources. By jointly computing the deadlines and dynamic resource budgets for tasks based on their multi-phase models, RASCO effectively improves WCETs, reduces end-to-end latency, and maximizes schedulability.

Figure 1 shows the workflow for our approach. In summary, we make the following contributions:

- we propose a resource-dependent multi-phase timing model for real-time tasks that enables WCET estimation under dynamic resource budget;
- we present a method for constructing the multi-phase model from task execution profiles;
- we introduce RASCO,[2] a co-design algorithm for fine-grained resource allocation and scheduling of DAG applications that leverages the multi-phase models to improve worst-case instruction rates, resource-use efficiency, and schedulability;
- we present a numerical evaluation of our technique based on real benchmarks;

[1]Prior work on DAG scheduling typically assumes that WCETs are given and uses synthetic WCETs for evaluations. Our work goes beyond by considering real programs and incorporating a way to derive their timing models.
[2]Our implementation of RASCO is available at https://github.com/abbyeisenklam/Rasco

— we present a prototype implementation of RASCO in a **real-time operating system** (**RTOS**) to demonstrate its applicability in practice; and
— we evaluate the runtime overheads of RASCO using this prototype, as well as discuss a way to incorporate such overheads into the RASCO algorithm to ensure safe schedulability.

2 Related Work

CPU scheduling of DAG-based taskgraphs. There exists a large body of work on scheduling and analysis of taskgraphs on multiprocessors (see, e.g., the survey in [24] and references therein). Prior work in this area often focuses on two directions: (1) schedulability/response time analysis methods for a given scheduling algorithm to improve resource augmentation bounds [3, 15] or tighten worst-case response time [12]; and (2) parallel scheduling algorithms that aim to effectively utilize cores, reduce runtime overheads, and improve schedulability (e.g., [7, 13, 30, 31]). Many techniques use deadline decomposition [19] and static priority assignment computed offline to maximize parallelization. For instance, Zhao et al. [30] model DAG tasks as workload distributions and compute an offline priority and core assignment that accumulates the workload distributions to avoid inter-DAG interference. Sun et al. [21] rely on deep learning to statically generate edges between tasks to compress the width of the DAG to fit the number of cores. The majority of existing work, however, focuses on only the CPUs while ignoring other shared resources.

Resource-aware analysis and scheduling. Recent solutions have started to consider resource interferences in DAG scheduling and analysis. For example, [20] proposes a scheduling technique that combines tasks of a DAG into execution groups to reduce inter-core communication. Casini et al. [6] incorporate the potential overhead due to memory contention in the schedulability analysis. The work in [22] co-locates tasks on the same core to reduce cache overhead and improve run-time performance. In [32], jobs of DAG tasks are assigned to cores based on a model which predicts cache recency. Unlike our work, these techniques do not *compute* the resource allocation.

Multicore resource allocation. Several multicore resource allocation techniques have been developed in recent years. For example [26, 27] propose holistic resource allocation techniques that find the assignments of tasks, cache and memory bandwidth to cores. Closely related to our work, DNA/DADNA [10] also exploits the workload characteristics to dynamically adapt resources allocated to a task during an execution. These techniques, however, assume independent tasks whose deadlines are given a priori, and DNA/DADNA also only supports soft real-time systems. Another similar work [28] which does dynamic resource allocation and scheduling co-design (using reinforcement learning) claims to provide hard real-time guarantees, but requires the assumption that the execution rate of a task is constant throughout its lifetime (which we show can result in unsafe WCET estimates in Section 3.3). We are not aware of any prior work that does fine-grained (i.e., intra-job) dynamic resource allocation together with scheduling for taskgraphs (or independent tasks), *while ensuring hard real-time schedulability guarantees.*

3 Resource-Dependent Multi-Phase Modeling of Real-Time Tasks

We first present an empirical study of real workloads under different budgets of shared resources. Based on insights from this study, we then present a multi-phase model that succinctly captures a task's resource-dependent execution behavior. Once obtained (through the proposed measurement-based approach or other techniques), the model can be used for scheduling and resource allocation.

3.1 Profiling of Real-Time Workloads on Multicore

To understand the resource-dependent timing behaviors of real-time workloads, we performed measurements of real benchmarks on multicore hardware under various resource allocations.

Setup. Our benchmarks consisted of programs and inputs taken from the PARSEC [4] and SPLASH2x [25] benchmark suites. These benchmarks have often been used as workloads by prior work on resource allocation (e.g., [10, 16, 29]). We used Ubuntu 20.04.6 on an Intel Xeon CPU E5-2618L v3 machine, which supports Intel's **Cache Allocation Technology** (**CAT**) [14] for cache allocation and MemGuard [29] for memory bandwidth allocation (see Section 6 for details on CAT and MemGuard). The machine has 8 cores and a 20MB 20-way set-associative shared L3 cache, which is partitioned by CAT into $N_{\text{ca}} = 20$ cache partitions. We divided the maximum guaranteed bandwidth of 1.4 GB/s (measured on our machine) into $N_{\text{bw}} = 20$ partitions of 72MB/s each. We disabled hyperthreading, SpeedStep, and hardware prefetching to avoid nondeterministic timing.

Measurement. To obtain the execution profile for a given task (benchmark program) under a given budget $\beta = (\beta_{\text{ca}}, \beta_{\text{bw}})$ of β_{ca} cache partitions and β_{bw} bandwidth partitions, we pinned the task on a dedicated core that was isolated from the OS, and assigned β_{ca} cache partitions and β_{bw} bandwidth partitions to that core. Using the CPU's performance counters, we periodically measured the number of retired instructions every 10 milliseconds throughout each run of the task. From these measurements, we can compute the *instruction rate*, defined as the number of instructions retired per millisecond, and the instruction count at which this rate is observed.

We recorded these metrics for 100 runs of each task under each possible resource budget. By repeating the measurement process for all possible allocated budgets β (i.e., a total of $(N_{\text{ca}} - 1) \times N_{\text{bw}} = 19 \times 20 = 380$ different configurations of β), we obtained the complete resource-dependent execution profile of the task. Note that we measured $N_{\text{ca}} - 1$ cache partitions because our CPU's CAT implementation limits the minimum LLC allocation to two partitions.

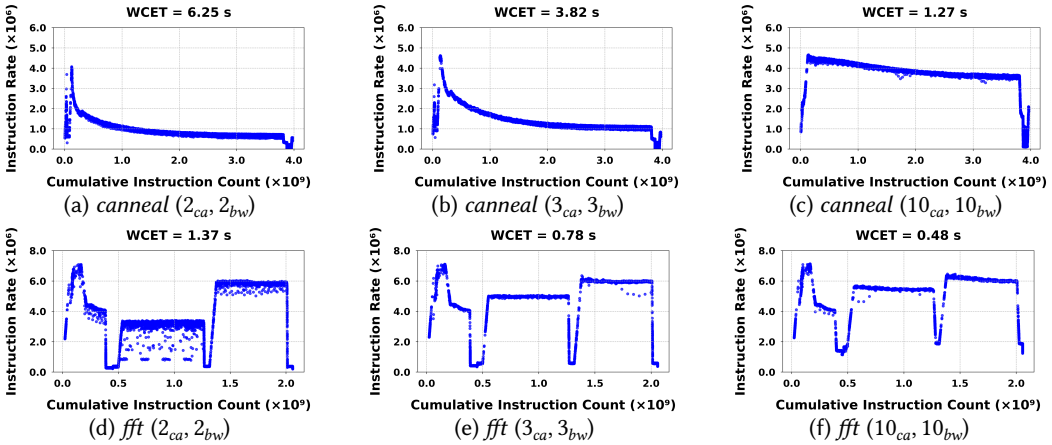
3.2 Results and Discussions: The Case for Phase-Based Resource Allocation

Figure 2 shows the profiling results for two example tasks, *canneal* and *fft*. From these figures, we can identify several key observations. First, a task's WCET depends on its resource budget. As shown in the top horizontal axes of Figure 2(a) and (c), *canneal*'s WCET improves from 6.25 to 1.27 seconds when going from 2 to 10 partitions of each type. Likewise, *fft* improves from 1.37 to 0.48 seconds for the same increase in resources (Figure 2(d) and (f)). This decrease in execution time is directly linked to the increase in instruction rate, as shown on the vertical axis, which can be attributed to a reduction in LLC misses and memory bandwidth bottlenecks when allocated more resources.

Insight 1. *The total execution time of a task is highly dependent on its allocated resource budget.*

Next, we observe that a task's instruction rate follows a common pattern, regardless of the allocated resource budget. For example, *fft* clearly shows three distinct periods with high instruction rates, separated by valleys with a lower rate. Intuitively, we call each period of distinct behavior a *phase* of execution. As the allocated resource budget increases, the phase boundaries continue to occur roughly at the same points in the program, but the rate of each phase varies significantly. For instance, when *fft* is allocated a resource budget of $(10_{ca}, 10_{bw})$ instead of $(2_{ca}, 2_{bw})$, the instruction rate during the "middle" phase (between instruction counts 0.5×10^9 and 1.25×10^9) increases from 3×10^6 to around 5×10^6, representing a 66% increase in the instruction rate. However, the first phase (between instruction counts 0 and 0.4×10^9) does not experience a rate increase with additional resources. This leads to our second key insight.

Insight 2. *Different phases of the same task exhibit different levels of resource sensitivity, resulting in different improvement in instruction rate when given the same additional resource budget.*



(a) *canneal* (2_{ca}, 2_{bw}) (b) *canneal* (3_{ca}, 3_{bw}) (c) *canneal* (10_{ca}, 10_{bw})

(d) *fft* (2_{ca}, 2_{bw}) (e) *fft* (3_{ca}, 3_{bw}) (f) *fft* (10_{ca}, 10_{bw})

Fig. 2. Instruction rate (number of instructions retired per ms) for two benchmarks, *canneal* (top) and *fft* (bottom), given three different resource budgets (2_{ca}, 2_{bw}), (3_{ca}, 3_{bw}), and (10_{ca}, 10_{bw}) shown from left to right. Each data point shows the instruction rate (obtained from profiling) at a specific point in the program (the cumulative instruction count). Each graph plots data points across all 100 runs.

Next, not all tasks have clearly defined phase boundaries. Unlike *fft* where the valleys indicate clear start and end points between phases, *canneal* shows a much more gradual change in the instruction rate during its execution.

Insight 3. *The instruction rate pattern differs across tasks.*

Finally, the instruction rate of a task often does not increase linearly with the number of resource partitions allocated. Instead, many tasks experience a significant increase in instruction rate only after a certain critical threshold of resource budget is reached (e.g., an amount sufficient to hold their working set). For example, with *canneal*, there is only a small increase in the instruction rate going from (2_{ca}, 2_{bw}) to (3_{ca}, 2_{bw}) (the WCET decreases by 0.38s in this case), but a much larger increase going from (3_{ca}, 2_{bw}) to (3_{ca}, 3_{bw}) (a WCET decrease of 2.05s). In contrast, with *fft*, going from (2_{ca}, 2_{bw}) to (2_{ca}, 3_{bw}) results in the same reduction in WCET as going from (2_{ca}, 2_{bw}) to (3_{ca}, 3_{bw}), indicating a different critical threshold than *canneal*.

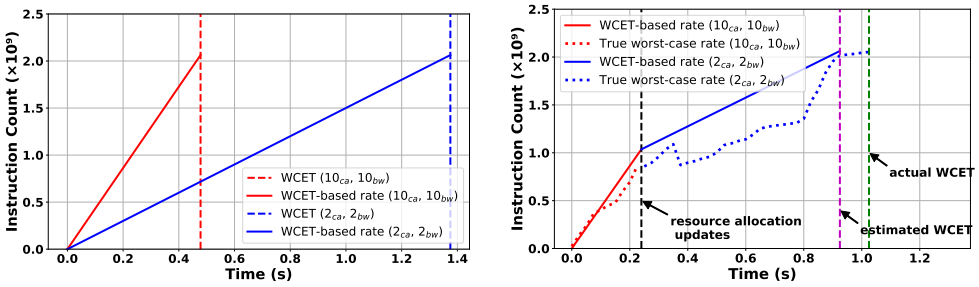
Insight 4. *The instruction rate of a task does not scale linearly with the resource budget allocated, and each task has a different critical threshold of resources at which the rate improves significantly.*

These insights underscore the need for a phase-based model that accurately captures a task's time-varying, resource-dependent execution behavior.

3.3 Challenges of Achieving Timing Guarantees Under Dynamic Resource Allocation

Fine-grained dynamic allocation of shared multicore resources—e.g., shared caches and memory bandwidth—has been explored in recent work, but only for *independent* tasks in *soft real-time* systems [10]. Our goal is to enable such allocation with *worst-case guarantees*.

At first glance, it might seem that we could assume a constant worst-case instruction rate for each resource budget and compose the overall WCET from these rates. However, this approach is not safe: the WCET estimate based on constant rates can be smaller than the measured WCET with dynamic resource allocation! To illustrate this issue, Figure 3 shows the measured worst-case instruction rates of *fft*. In Figure 3(a), the red and blue solid lines represent the cumulative instruction count over time *when assuming a constant instruction rate* for static budgets (10_{ca}, 10_{bw})



(a) Constant rates of *fft* based on WCETs (b) Rate change with dynamic resource allocation

Fig. 3. Unsafe WCET estimation when assuming a constant instruction rate under each resource budget.

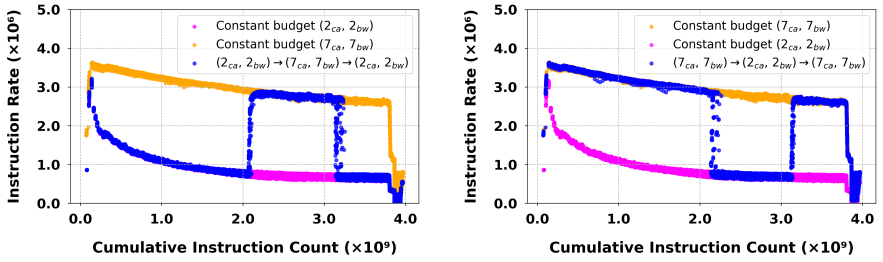
and $(2_{ca}, 2_{bw})$, respectively. The dashed vertical lines indicate when the task completes its execution under these static resource budgets (i.e., corresponding to their measured WCETs). In Figure 3(b), the *actual fine-grained* instruction count of the task under dynamic resource budget is shown as the dotted curve, where the red portion corresponds to the initial execution of *fft* under $(10_{ca}, 10_{bw})$ and the blue corresponds to the subsequent execution after changing the resource budget to $(2_{ca}, 2_{bw})$. As shown in this figure, the overall WCET estimated by composing the assumed constant rates for the two resource budgets (the time at the vertical purple dashed line) is much smaller than a measured completion time (the time at the vertical green dashed line). Thus, we cannot safely estimate the WCET under dynamic resource allocation by assuming a constant rate for each resource budget.

Another approach is to use the WCET under the minimum budget the task ever receives. However, this will result in an overly-conservative WCET. In our example, the measured WCET under the minimum budget is almost 1.4 seconds (Figure 3(a)), which is much larger than the measured WCET under dynamic resource allocation (just above 1 second, in Figure 3(b)). To avoid overly-conservative WCETs, we must consider the fine-grained instruction rates under each resource budget.

Basic idea. We model a task's execution under a given resource budget as a series of phases, each with its own worst-case instruction rate. Then, when the resource budget allocated to a task changes, we can simply lookup the worst-case rate for the current phase under the new resource budget to understand the worst-case behavior of the task under dynamic resource allocation.

To illustrate this, Figure 4(a) shows (in blue) the instruction rate of *canneal*, measured over 100 profiling runs, under a dynamic resource allocation. In these runs, *canneal* was allocated an initial budget of $(2_{ca}, 2_{bw})$, which was increased to $(7_{ca}, 7_{bw})$ at instruction count 2,035,574,822, and subsequently reverted to $(2_{ca}, 2_{bw})$ at instruction count 3,091,669,630. Importantly, we observe that the instruction rate of *canneal* after its resource budget changes to $(7_{ca}, 7_{bw})$ closely matches the rate measured when running *canneal* under the *constant* resource budget of $(7_{ca}, 7_{bw})$ (shown in yellow). Similarly, when the budget reverts to $(2_{ca}, 2_{bw})$, its instruction rate closely matches the rate measured under the constant resource budget of $(2_{ca}, 2_{bw})$ (shown in pink). The same trend holds for the inverse budget assignment, $(7_{ca}, 7_{bw}) \rightarrow (2_{ca}, 2_{bw}) \rightarrow (7_{ca}, 7_{bw})$, as illustrated in Figure 4(b).

Therefore, after accounting for the transient effect of resource reconfiguration on the execution rate (e.g., the overhead associated with filling additional cache partitions; see Section 6 for further discussion), we can obtain the worst-case rates under a dynamic resource budget by composing the phase-based worst-case rates under the individual constant resource budgets.



(a) $(2_{ca}, 2_{bw}) \rightarrow (7_{ca}, 7_{bw}) \rightarrow (2_{ca}, 2_{bw})$ (b) $(7_{ca}, 7_{bw}) \rightarrow (2_{ca}, 2_{bw}) \rightarrow (7_{ca}, 7_{bw})$

Fig. 4. Profiled instruction rates of *canneal* under dynamic (blue) vs. constant (yellow and pink) budgets.

3.4 Resource-Dependent Multi-Phase Model for Real-Time Tasks on Multicore

Let $\beta = (\beta_{\text{ca}}, \beta_{\text{bw}})$ denote an allocated resource budget, which is a vector of the number of shared cache partitions and the number of bandwidth partitions. For example, $\beta = (2_{\text{ca}}, 5_{\text{bw}})$ represents a resource budget with 2 cache and 5 bandwidth partitions. Then, each task τ's timing behavior under a resource budget β can be modeled as

$$\Theta_{\tau|\beta} = (\theta_1, \theta_2, \ldots, \theta_k), \tag{1}$$

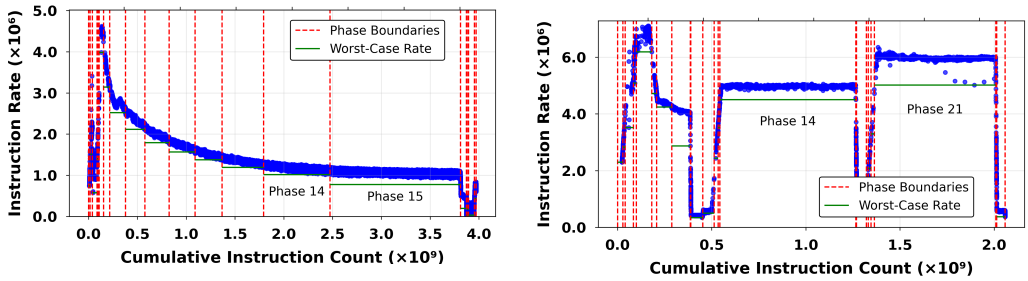
where k is the number of consecutive execution phases that capture non-negligible changes in the instruction rate of the task. Notice that $\Theta_{\tau|\beta}$ is conditional on an allocated budget β, since we know the phase characteristics are highly dependent on β.

Each individual phase is characterized by $\theta_i = [\theta_i^s, \theta_i^e, \theta_i^r, \theta_i^\Delta]$, where θ_i^s and θ_i^e specify the start and end instruction of the phase, and θ_i^r specifies the *worst-case* instruction rate (i.e., the minimum number of instructions retired per millisecond) of the task in this phase. By definition, θ_k^e is τ's total number of instructions, $\theta_1^s = 0$, and $\theta_i^e = \theta_{i+1}^s$ for all $1 \leq i < k$. Lastly, θ_i^Δ is a lookup table expressing the potential to increase the worst-case instruction rate of τ in its current phase given the remaining resource budget available for allocation. (More details on θ_i^Δ are in Section 3.5.) Next, we describe a method for constructing the model from execution profiles, such as those in Figure 2.

3.5 Constructing Multi-Phase Models

Phase identification and WCET analysis. The multi-phase model requires identifying program phases and determining the worst-case instruction rate for each phase under each resource budget. To construct this model, any existing phase identification technique and WCET analysis—provided that it is deemed safe by the system designer—may be used. For example, phases can be identified via program inspection, clustering on profile data, or changepoint detection (as done in this work). Similarly, a wide range of WCET analysis techniques exist, including static analysis, measurement-based timing analysis, and hybrid approaches (see [1] for a survey of each category and a discussion of their tradeoffs). Regardless of the techniques employed, the worst-case instruction rate of each phase is simply the smallest instruction rate achievable within that phase.

In this work, we employ measurement-based timing analysis—commonly used in multicore real-time systems [1]—to estimate the worst-case instruction rate of each phase. Although measurement alone cannot enumerate all paths through an arbitrary program, it can be used to estimate the WCET of a deterministic path when shared resources are partitioned. Therefore, we fixed the **worst-case execution path (WCEP)** in each benchmark using deterministic inputs and estimated the WCET of this path via measurement.



(a) *canneal* (3_{ca}, 3_{bw}, $k = 20$) (b) *fft* (3_{ca}, 3_{bw}, $k = 25$)

Fig. 5. Results of changepoint detection for *canneal* and *fft* with k = 20 and k = 25, respectively. The worst case rate of each phase is shown in green and the boundary between each phase is shown in red.

For programs with nondeterministic, input-dependent execution paths, static analysis must first be used to identify the WCEP (i.e., the longest path in the control flow graph—see [1]). To use the multi-phase model for such programs, one must then verify that this path holds over all resource budgets. If the WCEP depends on the resource budget, then under dynamic resource budget, we cannot safely compose the rates from multiple WCEPs (c.f. Section 3.3). This requirement can be verified using prior work that incorporates allocations of shared resources into multicore timing analysis [11, 17]. Once the WCEP is identified, our proposed measurement-based approach may be used to construct the multi-phase model. As we will show in Section 6, this approach—combined with runtime overhead accounting—enables worst-case schedulability guarantees with Rasco.

Changepoint detection for identifying phases. For each task τ and resource budget β, the execution profile records the instruction rate and the cumulative instruction count at which this rate is observed (as shown in Figure 2). Our goal is to construct a series of execution phases $\Theta_{\tau|\beta} = (\theta_1, \theta_2, \ldots, \theta_k)$ for each τ under each resource budget β by identifying consecutive segments of its profile that display similar instruction rates. To achieve this, we use changepoint detection [23], an approach commonly used to detect the time points where certain properties change in time-series data. In our case, we use it to identify the cumulative instruction count values at which the rate changes significantly. Specifically, we use the kernel changepoint detection method [2] from the `ruptures` [23] library with an `l2` kernel. Given a number of changepoints as input, the `l2` kernel aims to minimize the least-squared deviation of the measured rates between any two changepoints. Therefore, the output of this algorithm is a series of k phases $(\theta_1, \theta_2, \ldots, \theta_k)$, where the ith phase θ_i is defined by a start and an end instruction (θ_i^s and θ_i^e, respectively), and the instruction rate changes minimally within each phase. Since the rate changes minimally in each phase, the worst-case rate θ_i^r is a tight lower bound on the task's instruction rate in phase θ_i.

Figure 5 shows the results of applying the algorithm to *canneal* and *fft*. We can observe that it successfully identifies the instruction counts at which the instruction rate changes (indicated by vertical red dotted lines), and that the worst-case instruction rate provides a relatively tight lower bound on the instruction rates across all profiling runs in each phase (horizontal green lines).

Choosing the number of phases. The number of phases k introduces a tradeoff between the multi-phase model's precision and the runtime of our resource allocation algorithm. Intuitively, a smaller value of k results in greater variation in the instruction rate in each phase, leading to a looser (i.e., more conservative) lower bound on the instruction rate. This can in turn affect the precision of our resource allocation and scheduling co-design algorithm. Conversely, a larger k increases the

number of phases, and thus, the runtime of our resource allocation algorithm. Therefore, we aim to choose the smallest possible k that still maintains tight worst-case rates for the phases.

To empirically assess the marginal benefit of increasing k, we first define the metric *WCET Amplification Ratio* for a fixed k as follows:

$$\textit{WCET Amplification Ratio} = \frac{\textit{phase-based WCET}}{\textit{profiled WCET}}. \tag{2}$$

Here, the phase-based WCET of a task τ with multi-phase model $\Theta_{\tau|\beta} = (\theta_1, \theta_2, \ldots, \theta_k)$ is given by

$$\textit{phase-based WCET} = \sum_{i=1}^{k} \left(\frac{\theta_i^e - \theta_i^s}{\theta_i^r} \right). \tag{3}$$

For each phase θ_i of τ under resource budget β, we find the WCET in that phase using $(\theta_i^e - \theta_i^s)/\theta_i^r$. We then sum this value for all phases $(\theta_1, \ldots, \theta_k)$ to obtain the phase-based WCET under resource budget β. This sum is then divided by the observed WCET across all runs of the task under resource budget β, yielding a ratio that indicates the accuracy of our phase-based WCET estimate for a given value of k. Ideally, this ratio should be as close to 1 as possible, meaning that the phase-based WCET closely matches the profiled WCET.

For each resource budget β (380 in total), we calculate the WCET Amplificiation Ratios and plot the median. The plot for the benchmark task *fft* in Figure 6 shows that the median WCET Amplificiation Ratio converges to just below 1.1, and that the reduction in the ratio indicates diminishing returns as k increases. We observed a similar trend across all workloads (omitted here). Therefore, to determine the optimal number of phases for each task, we select the value of k corresponding to the "elbow point"—the point at which further increases in k yield diminishing returns in the median WCET Amplificiation Ratio.

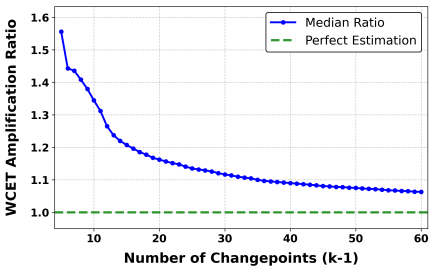


Fig. 6. The median WCET amplification ratio over all possible resource budgets for *fft*.

Computing the rate increase lookup table. The final parameter of the multi-phase model is θ_i^Δ, which is used by RASCO as a heuristic to decide which task τ^* among the ready tasks would benefit the most from an additional resource given their current phases.

One approach for selecting this task τ^* would be to give the resource to the task whose worst-case rate improves the most from the additional resource. However, this approach does not account for the fact that a *critical threshold* of resource budget is often required to achieve significant improvements in the instruction rate. For example, the *canneal* task does not experience significant improvement between the resource budgets $(2_{ca}, 2_{bw})$ and $(3_{ca}, 2_{bw})$. If resources are allocated iteratively (one partition at a time, as in RASCO), this naïve heuristic would not select *canneal* to receive additional resources, even though its worst-case instruction rate improves significantly with budget $(3_{ca}, 3_{bw})$ (and even more with $(10_{ca}, 10_{bw})$). Therefore, θ_i^Δ should incorporate not only the rate of the phase each task would enter by getting *one* additional resource partition but also the rates of the phases that could be entered with any amount of additional resources (up to the maximum resource budget available currently).

This observation raises another crucial point: since the platform resources are shared, there may not be sufficient resources for some tasks to ever reach their critical thresholds. To illustrate this point, suppose there are $R = (0_{ca}, 1_{bw})$ remaining resources, but that *canneal* requires $(1_{ca}, 0_{bw})$ additional resources to reach its critical resource threshold. In this case, some other task might

benefit more from the additional budget, despite *canneal*'s high resource sensitivity. Therefore, our heuristic θ_i^{Δ} must take into account the remaining resource budget, R, currently available.

Thus, to fully express how a task τ's worst-case rate can change given (1) its current phase θ_i, (2) each phase θ_j it could enter by receiving additional budget β', and (3) the constraint on the amount of remaining available resources $R = (R_{\text{ca}}, R_{\text{bw}})$, we let θ_i^{Δ} be a table and we set each entry $\theta_i^{\Delta}[R]$ equal to the average change in the worst-case rate $(\theta_j^r - \theta_i^r)$ over each phase θ_j that τ could enter by receiving some additional budget β' where $\beta' \in [(0,0), (R_{\text{ca}}, R_{\text{bw}})]$.

4 Resource-Allocation and Scheduling Co-Design

Using the multi-phase model, our co-design algorithm focuses on multicore scheduling and resource allocation for systems comprising one or more periodic taskgraphs, an application class of recent interest in real-time embedded systems [7, 18, 21, 32]. Rasco imposes no constraints on the number of taskgraphs or the number of tasks per taskgraph.

4.1 Problem Statement

Taskset. We consider a system of n periodic taskgraphs, $T = \{G_1, G_2, \ldots, G_n\}$, scheduled on a multicore platform ($n \in \mathbb{N}^+$). Each taskgraph G_i is a DAG whose nodes represent tasks $\{\tau_1, \tau_2, \ldots, \tau_j\}$ and whose edges represent precedence constraints between tasks. Each G_i has a period P_i and a relative end-to-end deadline D_i. For each taskgraph, a source task—a task with no predecessors—is released whenever a new instance of the taskgraph is released (i.e., once every P_i time units), whereas a non-source task is released when all its predecessor tasks have completed. We assume that all taskgraphs release their first instance synchronously at time $t = 0$, although the algorithm can easily be extended to allow fixed taskgraph release offsets, as long as a hyper-period is preserved.

Let H be the hyper-period of all the taskgraphs in T, i.e., the least common multiple of their periods, and let K_i be the number of releases of taskgraph G_i in one hyper-period (thus, $K_i = H/P_i$ where P_i is G_i's period). Then, we have a series of fixed release points $A_i = \{(k-1)\cdot P_i \mid 1 \leq k \leq K_i\}$ of the instances of G_i in one hyper-period. We call A_i the set of *anchor points* of G_i. Therefore, each source task of G_i releases a job at each of the anchor points. We denote by $\mathcal{J}$ the set of all jobs of all tasks of the taskgraphs in T that are released in a hyper-period. Let $J_{i,j,k}$ denote the kth job of task τ_j belonging to taskgraph G_i. Then, the system is schedulable if all jobs $J_{i,j,k} \in \mathcal{J}$ complete their executions by time $A_{i,k} + D_i$ where $A_{i,k} = (k-1)\cdot P_i$.

Note that each task executes a sequential workload (e.g., a function or program), running on at most one core at a time, since taskgraph-level parallelism is inherently expressed in the taskgraph's DAG structure. We assume that a WCEP that holds for all resource budgets can be determined for each task's workload. Thus, a resource-dependent multi-phase timing model $\Theta_{\tau|\beta}$ can be constructed for each task τ, using techniques such as those discussed in Section 3.5.

Platform. The platform contains m identical cores that share a set of b different resource types. For concreteness, we focus on two types of shared resources: the LLC and the memory bandwidth, as described in Section 3. However, our algorithm generalizes to other types of shared resources that can be partitioned. As stated before, we assume the shared cache is partitioned into N_{ca} equal partitions, and the memory bandwidth is partitioned into N_{bw} equal partitions.

Goal. Given the above setting, our objective is to develop a co-design algorithm that leverages tasks' multi-phase models to holistically compute a schedule and fine-grained resource budget allocations for the taskgraphs in T to maximize the system's schedulability and resource utilization.

4.2 Overview of Rasco Co-Design Algorithm

At a high level, Rasco schedules tasks by decomposing the end-to-end deadline of each taskgraph into independent deadlines and release offsets for its tasks. Unlike traditional deadline decomposition techniques, Rasco allocates deadlines *and* resources to tasks jointly, adjusting deadlines based on resource allocations and vice versa.

Towards this, Rasco first assigns initial values for the release time, deadline, and base ("minimum") resource budget for each task in each taskgraph. It then constructs the set of all job releases $\mathcal{J}$ in a hyper-period. Starting with the initial release time, deadline, and base resource budget β_J^{init} for each job J in $\mathcal{J}$, Rasco uses an iterative algorithm to compute new deadlines and fine-grained resource budgets for each job. The release times and completion times of the jobs form a set of *decision points*, at which we make scheduling and resource allocation decisions. We use the term *segment* to denote the time between consecutive decision points.

At each decision point (from time $t = 0$ onwards), Rasco considers the set of all ready jobs and decides which m jobs to execute in the current segment, as well as what resource budgets they should get. The goal is to maximally reduce execution times by redistributing resources to the jobs that would most benefit from them, while respecting job deadlines (by ensuring they are allocated sufficient resources to complete before their deadlines, if at all possible). For this, Rasco utilizes the multi-phase model $\Theta_{\tau \equiv J}$ to keep track of each job J's current phase θ_i and the parameter θ_i^{Δ} to determine which jobs among the ready jobs would benefit the most from the extra resources. As more resources are allocated to these jobs, their WCETs (computed using the per-phase worst-case rates under each budget it has received) shrink and Rasco adjusts their deadlines accordingly. It then determines which jobs should be scheduled on the cores based on a global **earliest-deadline-first** (**EDF**) policy. As the deadlines of some jobs are shortened, the set of jobs with the earliest deadlines may change, causing the set of jobs that are chosen by EDF to change. Our algorithm iteratively distributes resources to jobs until there are no more resources to give out. When this happens, the set of jobs with the earliest deadlines, their resource budgets, and the next decision point can be determined (from the next earliest job release or completion time). Rasco then moves to the next decision point and repeats this process.

The output of our algorithm is a static schedule for one hyper-period, which is made of a series of consecutive segments. Each segment contains ($\leq m$) jobs that will be executed on the m cores and the allocated resource budget for each job. Note that a job execution may span multiple consecutive or non-consecutive segments (if it was preempted), and its allocated resource budget may also change across these segments. At run time, the scheduler can schedule jobs and allocate resources simply by repeating Rasco's output schedule at each hyper-period.

Remark: While optimization might seem like a natural approach for computing a static schedule, encoding the time-varying, resource-dependent behavior of each task into a constraint formulation not only is highly challenging but also significantly increases complexity. In fact, prior work [27] has shown that this approach is too expensive even for much simpler scenarios, with coarse-grained static resource allocation. Therefore, we adopt an iterative, heuristic-based approach instead.

Figure 7 shows the high-level overview of Rasco. Next, we describe the algorithm in detail, starting with the computation of the base resource budget and initial deadline for each task.

4.3 Computing Base Resource Budgets and Initial Deadlines

Our goal is to assign the minimum resource budget that each task would need for the overall taskgraph to complete by its end-to-end deadline, assuming independent execution on sufficient cores. Towards this, we first apply a deadline decomposition method to assign release times and deadlines to tasks. We use the method of [15] which decomposes each taskgraph by placing the tasks into time segments, such that the load in each segment is minimized. We begin by setting

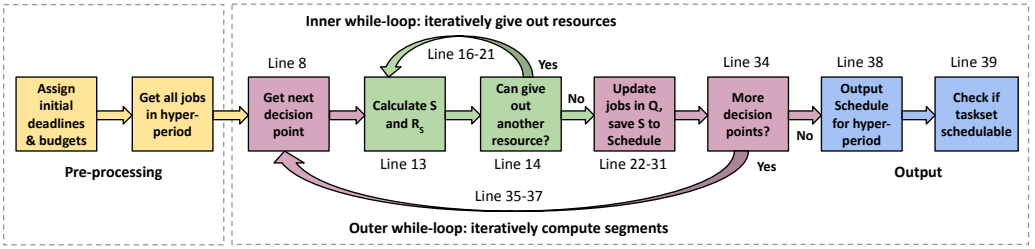


Fig. 7. Overview of Rasco.

the WCET of each task equal to its WCET with the maximum amount of resources (obtained via profiling). After applying the deadline decomposition method, we iteratively take away resource partitions until we find the *minimum* resource budget $\beta_J^{init} \geq (2_{\text{ca}}, 2_{\text{bw}})$ that each job J of task τ needs to ensure its resulting execution time plus its release time does not exceed its deadline. The computed job release times, deadlines, and base budget allocation β_J^{init} serve as inputs to Rasco.

4.4 Rasco Algorithm Details

Algorithm 1 shows the pseudocode for Rasco. Using the initial release time, deadline and base allocation, Rasco computes a static schedule, Schedule. Each element Schedule$[t_i]$ contains a set of (at most m) jobs to execute at decision point t_i and their assigned resource budgets (where $0 = t_1 < \cdots < t_i < H$, and H is the hyper-period). The algorithm takes as input the following parameters: the set of all job releases $\mathcal{J}$ in the hyper-period; the set $A = \{A_i \mid G_i \in T\}$ that contains the anchor points (fixed release times) of the source tasks in the hyper-period; and the release time r_J, the deadline d_J, and the base resource budget β_J^{init}, for all $J \in \mathcal{J}$. For its co-allocation, Rasco also takes as input the total number of instructions $\text{maxIns}_{\tau \equiv J}$ and the multi-phase model $\Theta_{\tau \equiv J}$ for each J, where J is a job of task τ. The last two inputs are the numbers of cores m and resource types b.

Notation. $R_{\max}$ denotes the vector of maximum number of partitions per resource type – e.g., $R_{\max} = (N_{\text{ca}}, N_{\text{bw}})$ on a platform with N_{ca} cache partitions and N_{bw} bandwidth partitions. Throughout the algorithm, t denotes the current decision point at which we compute the schedule, t_{next} denotes the next decision point, $\mathcal{A}$ denotes the future anchor points, ins_J denotes the number of instructions J has already retired at t, Q denotes the set of ready jobs at t, and S and R_s denote the set of $\leq m$ jobs with the earliest deadlines (to execute on the cores from t to t_{next}) and the total resource budget assigned to these jobs, respectively. Finally, β_J denotes the current resource allocation (initialized to β_J^{init}), and r_J, c_J and d_J are updated throughout to denote the release time, completion time and deadline of J as it is scheduled and allocated budgets in segments.

Initialization: The algorithm begins by initializing the variables (Lines 4–9). It first sets Schedule to be empty. Line 5 then constructs a set of b unit vectors that are used to indicate/select between resource types. Therefore, with $b = 2$ resource types (cache and bandwidth), $I = \{(1, 0), (0, 1)\}$. For each job $J \in \mathcal{J}$, Rasco then initializes the retired instruction count ins_J to 0 and the flag done_J to false to indicate that J has not completed its execution. It also sets the current budget β_J to β_J^{init} and calculates the completion time c_J under this budget. Rasco then sets $t = 0$ as the current decision point and sets the next decision point t_{next} to the earliest release time or completion time of any of the jobs in $\mathcal{J}$ that occurs after t. Next, Rasco initializes the set of future anchor points $\mathcal{A}$ and constructs the set of ready jobs Q to consider for scheduling and resource allocation at time t.

ALGORITHM 1: RASCO

1: **Input:** $\mathcal{J}$, A, $\{r_J,\ d_J,\ \beta_J^{init}, \text{maxIns}_J \mid J \in \mathcal{J}\}$, $\{\Theta_\tau \mid \tau \in T\}$ ▷ Job info. and multi-phase execution models
2: m, b ▷ Number of cores and number of resource types
3: **Output:** Schedule ▷ Schedule with $\leq m$ job-budget pairs at each decision point
4: Schedule $\leftarrow \{\}$
5: $I \leftarrow \{e_j \mid j, i \in [\![b]\!],\ e_j[i] \leftarrow 1$ **if** $i = j,\ e_j[i] \leftarrow 0$ **otherwise**$\}$ ▷ Set of unit vectors to indicate resource types
6: **for all** $J \in \mathcal{J}$ **do**
7: $\text{ins}_J \leftarrow 0$; $\text{done}_J \leftarrow$ **false**; $\beta_J \leftarrow \beta_J^{init}$; $c_J \leftarrow$ GETFINISH$(J, \beta_J, \text{ins}_J, r_J, \infty)$ ▷ Initialize per-job variables
8: $t \leftarrow 0$; $t_{\text{next}} \leftarrow \min\{r_J > 0, c_J > 0 \mid J \in \mathcal{J}\}$ ▷ Current and next decision points
9: $\mathcal{A} \leftarrow A \setminus \{0\}$, $Q \leftarrow \{J \in \mathcal{J} \mid r_J = 0\}$ ▷ $\mathcal{A}$: set of future anchor points, Q: set of ready jobs at $t = 0$
10: **while true do**
11: **for all** $J \in Q$ **do** $d_J^{init} \leftarrow d_J$; $\beta_J \leftarrow \beta_J^{init}$ ▷ Save initial deadline, assign base budgets
12: **while true do**
13: $S, R_s, t_{\text{next}} \leftarrow$ GETSCHEDSET$(Q, \{\beta_J, \text{ins}_J | J \in Q\}, I, R_{\max}, t_{\text{next}}, m)$ ▷ Get m jobs to schedule, budget used by S
14: $(J, \delta_J) \leftarrow$ RESOURCEALLOC$(Q, \{\beta_J, \text{ins}_J | J \in Q\}, S, R_s, t, t_{\text{next}}, I, b)$ ▷ Select job J to get resource δ_J
15: **if** $J =$ **null then break** ▷ Cannot give more resources, go to Line 22
16: $\beta_J \leftarrow \beta_J + \delta_J$ ▷ Else give extra resource to J
17: $d_J \leftarrow d_J - (c_J - \text{GETFINISH}(J, \beta_J, \text{ins}_J, t, t_{\text{next}}))$; $c_J \leftarrow \text{GETFINISH}(J, \beta_J, \text{ins}_J, t, t_{\text{next}})$ ▷ Update d_J and c_J
18: **if** $J \notin S$ **then**
19: $J_{\max} \leftarrow \text{argmax}_{J' \in S}\ d_{J'}$ ▷ Get job with max. deadline in S
20: **if** $(d_J > d_{J_{\max}} \vee R_s - \beta_{J_{\max}} + \beta_J > R_{\max})$ **then** ▷ Check if J can and should enter S
21: $d_J \leftarrow d_J^{init}$ ▷ J cannot enter S, reset d_J
22: **for all** $J \in Q \setminus S$ **do**
23: $\beta_J \leftarrow \beta_J^{init}$; $d_J \leftarrow d_J^{init}$; $c_J \leftarrow$ GETFINISH$(J, \beta_J, \text{ins}_J, t_{\text{next}}, \infty)$ ▷ Reset any unscheduled jobs
24: **for all** $J \in S$ **do** ▷ Update scheduled jobs and successors
25: $\text{ins}_J \leftarrow$ COMPUTEINS$(J, \beta_J, \text{ins}_J, t, t_{\text{next}})$
26: **if** $\text{ins}_J = \text{maxIns}_{\tau \equiv J}$ **then** ▷ J finished
27: $c_J \leftarrow t_{\text{next}}$; $\text{done}_J \leftarrow$ **true** ▷ Check for any new job releases
28: ReadySucc $\leftarrow \{J_{\text{succ}} \mid J_{\text{succ}} \in \text{successors}(J) \wedge \text{done}_{J_{\text{pred}}} =$ **true** $\forall J_{\text{pred}} \in \text{predecessors}(J_{\text{succ}})\}$
29: **for all** $J_{\text{succ}} \in$ ReadySucc **do** $r_{J_{\text{succ}}} \leftarrow c_J$; $c_{J_{\text{succ}}} \leftarrow \text{GETFINISH}(J_{\text{succ}}, \beta_{J_{\text{succ}}}^{init}, 0, r_{J_{\text{succ}}}, \infty)$
30: $Q \leftarrow (Q \setminus \{J\}) \cup$ ReadySucc ▷ Add any newly released jobs to Q, remove J
31: Schedule$[t] \leftarrow \{(J, \beta_J) \mid J \in S\}$ ▷ Save jobs and budgets for decision point t
32: **if** $Q = \emptyset \wedge \mathcal{A} = \emptyset$ **then break** ▷ No more jobs, complete algorithm
33: **if** $Q = \emptyset$ **then** $t \leftarrow \min(\mathcal{A})$ **else** $t \leftarrow t_{\text{next}}$ ▷ Prepare for next segment, update current decision point
34: **for all** $(J \in \mathcal{J} \mid r_J = t \wedge \text{parents}(J) = \emptyset)$ **do**
35: $Q \leftarrow Q \cup \{J\}$ ▷ Release ready source jobs, add to Q
36: $\mathcal{A} \leftarrow \mathcal{A} \setminus \{t\}$ ▷ Remove t from set of future anchor points
37: $t_{\text{next}} \leftarrow \min\{\ \min(\mathcal{A}),\ \min_{J \in Q} c_J\}$ ▷ Get next decision point, segment complete, return to Line 10
38: **Output:** Schedule ▷ Schedule with $\leq m$ job-budget pairs at each decision point
39: **return** $c_{J_{i,j,k}} \leq A_{i,k} + D_i\ \forall J_{i,j,k} \in \mathcal{J}$ ▷ Schedulability test: $J_{i,j,k}$ is the k^{th} job of task τ_j in taskgraph G_i

Main outer while-loop: After initialization, it proceeds to the outer while-loop (Lines 10–37), which iterates through each segment determining the set of jobs to schedule and their resource budgets. This resource-deadline co-allocation is done in three main steps:

Step 1 (Calculate S and R_s): After initializing the current budget β_J to the base budget and saving the initial deadline for each $J \in Q$ (Line 11), RASCO calls the function GETSCHEDSET (shown in Algorithm 2). This function returns the m jobs (if any) with the smallest deadlines in Q to be the scheduled set S, and computes the total budget R_s used by these jobs (Line 3 of GETSCHEDSET).

Step 2 (Give out a resource): In Step 2, RASCO calls the function RESOURCEALLOC (shown in Algorithm 3), which uses the multi-phase execution model $\Theta_{\tau \equiv J}$ to determine the job J in Q that

ALGORITHM 2: GetSchedSet

Input: $Q, \{\beta_J, \text{ins}_J | J \in Q\}, I, R_{\max}, t_{\text{next}}, m$
Output: S, R_s, t_{next} ▷ S: m jobs with earliest deadlines, R_s: budget used by S
$S \leftarrow \text{argmin}_{Q' \subset Q, |Q'|=m} \sum_{J \in Q'} d_J$; $R_s \leftarrow \sum_{J \in S} \beta_J$
for $J \in S$ **do**
  if $c_J < t_{\text{next}}$ **then** ▷ Changing β_J caused the segment boundary to change
    $t_{\text{next}} \leftarrow c_J$ ▷ Update t_{next}
    for all $J' \in Q \setminus \{J\}$ **do** ▷ Reset all jobs in Q aside from J
      $\beta_{J'} \leftarrow \beta_{J'}^{init}$; $d_{J'} \leftarrow d_{J'}^{init}$; $c_{J'} \leftarrow$ GetFinish$(J', \beta_{J'}, \text{ins}_{J'}, t, \infty)$
    $S \leftarrow \text{argmin}_{Q' \subset Q, |Q'|=m} \sum_{J' \in Q'} d_{J'}$; $R_s \leftarrow \sum_{J' \in S} \beta_{J'}$ ▷ Recompute S and R_s
while $(R_s > R_{\max})$ **do** ▷ Check if initial budgets exceed $R_{\max}$, if so take away resources
  $J_{\text{save}} \leftarrow \text{argmin}_{J \in S | c_J = t_{\text{next}}} d_J$ ▷ If any jobs define t_{next}, save one of them from resource removal
  $J \leftarrow \text{argmax}_{J' \in S \mid J' \neq J_{\text{save}}} d_{J'} - c_{J'}$ ▷ Job with largest slack (that does not specify t_{next})
  $\theta_i \leftarrow \Theta_{\tau \equiv J | \beta}[\text{ins}_J]$ ▷ Current phase of J
  $\delta_J \leftarrow \text{argmin}_{e_j \in I \mid e_j R_s > e_j R_{\max}} \theta_i^{\Delta}[R_{\max} \cdot e_j - e_j]$ ▷ Unit vector for least impactful resource type
  $\beta_J \leftarrow \beta_J - \delta_j$; $c_J \leftarrow$ GetFinish$(J, \beta_J, \text{ins}_J, t, t_{\text{next}})$ ▷ Take away this resource from J
  $R_s \leftarrow R_s - \delta_J$ ▷ Update R_s
Output: S, R_s, t_{next}

ALGORITHM 3: ResourceAlloc

Input: $Q, \{\beta_J, \text{ins}_J | J \in Q\}, S, R_s, t, t_{\text{next}}, I, b$
Output: $(J_{\text{best}}, \delta_{\text{best}})$ ▷ Select job J_{best} to get additional budget δ_{best}
$R_{\text{avail}} \leftarrow R_{\max} - R_s$ ▷ Check how much of each resource type is available
for all $J \in Q$ **do** ▷ For each job J
  for all $j \in [\![b]\!]$ **do** $\text{ScoreVec}_J[j] \leftarrow 0$ ▷ Initialize score vector: $\text{ScoreVec}_J[j]$ stores the score for resource type j
  if $J \notin S \wedge \beta_J = R_{\max}$ **then continue** ▷ J is maxed out, could not be swapped into S
  if $R_{\text{avail}} = 0 \wedge J \in S$ **then continue** ▷ Max. budget already used by S, only check jobs not in S to swap
  $\text{ins}_J^{t_{\text{next}}} \leftarrow$ ComputeIns(J, t, t_{next}) ▷ Get instruction count at t_{next}
  $\theta_i \leftarrow \Theta_{\tau \equiv J | \beta_J}[\text{ins}_J]$ ▷ Find current phase θ_i of J using $\Theta_{\tau | \beta_J}$ and ins_J, where J is a job of τ
  while $\theta_i^s < \text{ins}_J^{t_{\text{next}}}$ **do** ▷ Iterate through all phases in current segment
    $\text{startIns} \leftarrow \max(\theta_i^s, \text{ins}_J)$; $\text{endIns} \leftarrow \min(\theta_i^e, \text{ins}_J^{t_{\text{next}}})$
    for all $e_j \in I$ **do** ▷ For each resource type
      if $\beta_J \cdot e_j = R_{\max} \cdot e_j$ **then continue** ▷ If J already has the max. for this resource type, skip
      $\text{ScoreVec}_J[j] \leftarrow \text{ScoreVec}_J[j] + \theta_i^{\Delta}[R_{\text{avail}} \cdot e_j]/(\text{endIns} - \text{startIns})$ ▷ Increase score by weighted θ_i^{Δ}
    $i \leftarrow i + 1$ ▷ Move to next phase in segment
$J_{\text{best}} \leftarrow \text{argmax}_{J \in Q} \|\text{ScoreVec}_J\|_{\infty}$ ▷ Give resource to the job J with the single largest score in its ScoreVec
$\delta_{\text{best}} \leftarrow \text{argmax}_{e_j \in I} \|\text{ScoreVec}_{J_{\text{best}}} \cdot e_j\|_1$ ▷ Resource type that corresponds to the largest score in $\text{ScoreVec}_{J_{\text{best}}}$
if $\|\text{ScoreVec}_{J_{\text{best}}}\|_{\infty} = 0$ **then** $J_{\text{best}} \leftarrow$ **null** ▷ If all ScoreVec's are zeros, no more resources could be given
Output: $(J_{\text{best}}, \delta_{\text{best}})$

would benefit the most from extra resources during the timing segment from t to the immediate next decision point t_{next} and which resource type (indicated by δ_J) to give to this job (Line 14).

If no more resources can be given, ResourceAlloc returns null for J and Rasco skips the rest of this step (Line 15), moving to Step 3. Otherwise, it will add the extra budget δ_J to J's current budget β_J (Line 16), recalculate the completion time, and shorten the deadline of J by an amount equal to the reduction in execution time under the newly increased budget (Line 17).

The completion time, which is recomputed after each resource allocation, and used to define future decision points, is computed by the GetFinish function, shown in Algorithm 4.

After computing the new completion time and deadline of J (the task chosen to receive an additional resource), Rasco checks whether J should be swapped into S. This will occur if the new

ALGORITHM 4: GetFinish

Input: $J, \beta_J, \text{ins}_J, t, t_{\text{next}}$ ▷ Given that J is at instruction count ins_J, and that β_J resets to β_J^{init} at t_{next}
Output: c_J ▷ Compute maximum finish time (in absolute time since $t = 0$)
$\text{ins}_J^{t_{\text{next}}} \leftarrow \text{ComputeIns}(J, \beta_J, \text{ins}_J, t, t_{\text{next}})$ ▷ Get instruction count at t_{next}
$\theta_i \leftarrow \Theta_{\tau \equiv J \mid \beta_J}[\text{ins}_J]$ ▷ Find current phase θ_i using ins_J, where J is a job of task τ
$t_{\text{left}} \leftarrow 0$; $k \leftarrow |\Theta_{\tau \equiv J \mid \beta_J}|$ ▷ Initialize t_{left}; get total number of phases in $\Theta_{\tau \mid \beta_J}$
while $i \leq k$ **do**
  $\text{startIns} \leftarrow \max(\theta_i^s, \text{ins}_J)$; $\text{endIns} \leftarrow \min(\theta_i^e, \text{ins}_J^{t_{\text{next}}})$ ▷ Get instructions retired in this phase
  $t_{\text{left}} \leftarrow t_{\text{left}} + (\text{endIns} - \text{startIns})/\theta_i^r$ ▷ Get worst-case execution time using worst-case rate θ_i^r
  if $\theta_i^e > \text{ins}_J^{t_{\text{next}}}$ **then** ▷ Check if we have reached $\text{ins}_J^{t_{\text{next}}}$
    $\theta_i \leftarrow \Theta_{\tau \equiv J \mid \beta_J^{init}}[\text{ins}_\tau^{t_{\text{next}}}]$ ▷ If so, switch to multi-phase model for β_J^{init}, get new current phase
    $k \leftarrow |\Theta_{\tau \equiv J \mid \beta_J^{init}}|$ ▷ Update the number of phases
    $\text{ins}_J^{t_{\text{next}}} \leftarrow \text{maxIns}_{\tau \equiv J}$ ▷ Continue iterating through phases until we reach $\text{maxIns}_{\tau \equiv J}$
  else
    $i \leftarrow i + 1$ ▷ Otherwise, go to next phase in multi-phase model for β_J
$c_J \leftarrow t + t_{\text{left}}$
Output: c_J

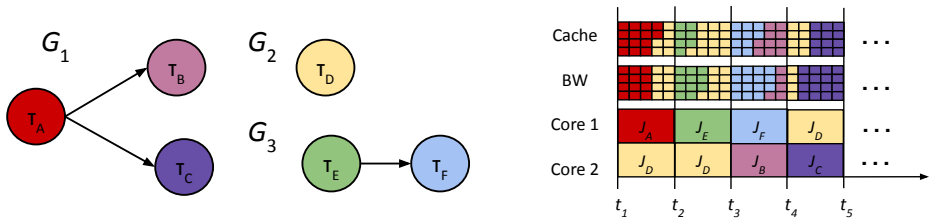
ALGORITHM 5: ComputeIns

Input: $J, \beta_J, \text{ins}_J, t, t_{\text{next}}$ ▷ Given that J is at instruction count ins_J at time t with resource budget β_J
Output: ins_J ▷ Compute instruction count at t_{next}
$\theta_i \leftarrow \Theta_{\tau \equiv J \mid \beta_J}[\text{ins}_J]$ ▷ Find current phase θ_i using ins_J, where J is a job of task τ
$t_{\text{left}} \leftarrow (t_{\text{next}} - t)$; $\text{insRetired} \leftarrow 0$ ▷ Get time until t_{next} and initialize instructions retired
while $t_{\text{left}} > 0$ **do**
  $\text{startIns} \leftarrow \max(\theta_i^s, \text{ins}_J)$; $\text{endIns} \leftarrow \theta_i^e$ ▷ Get number of instruction in current phase
  $t_{\text{phase}} \leftarrow \min(t_{\text{left}}, (\text{endIns} - \text{startIns})/\theta_i^r)$ ▷ Compute time that J will spend in this phase
  $\text{insRetired} \leftarrow \text{insRetired} + (\theta_i^r \cdot t_{\text{phase}})$ ▷ Update insRetired
  $t_{\text{left}} \leftarrow t_{\text{left}} - t_{\text{phase}}$ ▷ Subtract time spent in this phase from t_{left}
  $i \leftarrow i + 1$ ▷ Go to next phase
$\text{ins}_J \leftarrow \text{ins}_J + \text{insRetired}$
Output: ins_J

deadline of J is smaller than the latest deadline in S and if the swap does not lead to the scheduled jobs having more total budget than $R_{\max}$ (Lines 18–20). If either condition fails, the deadline of J is reset (Line 21); otherwise, the job will be added to S in the next call to GetSchedSet.

While there are more resources to give out, Rasco will repeat Step 1 (to update S and R_s) and Step 2 (to pick another job to receive a resource). When no more resources can be given, Rasco then moves to Step 3. Note that for ResourceAlloc to return null and for the inner while-loop to terminate, $R_{\max} - R_s$ must equal the zero vector (all resources allocated), and for every $J \in Q, J \notin S$, β_J must be equal to $R_{\max}$. Intuitively, this indicates that the number of resources required to sufficiently shrink the deadlines of these jobs was too large for the job to ever be swapped into S.

Step 3 (Update Q, save S): Once the algorithm cannot give out any more resources to jobs, the scheduled set S is fixed for the current decision time point t. Rasco will reset the current resource budget for all ready jobs that are not in the scheduled set to be their base budget and reset to their initial deadlines in case they were changed (by entering S at any point). It then computes their new completion times, given that they were not scheduled in the current segment (Line 23). Rasco then traverses through all jobs in the scheduled set S (Line 24). For each scheduled job J, it uses the function ComputeIns (shown in Algorithm 5) to compute the number of instructions ins_J that J would have completed by t_{next} (i.e., after executing in the segment $[t, t_{\text{next}})$) under its current allocated budget (Line 25). If ins_J is equal to τ's total number of instructions (where J is a job of τ),



(a) Example system with 3 taskgraphs (b) Schedule with $N_{ca} = 20$, $N_{bw} = 20$, and $m = 2$

Fig. 8. RASCO input and output visualized. Note J_A represents the first job of task τ_A, and so on.

then J would have finished its execution by t_{next}; hence, we set J's completion time to be t_{next} and done$_J$ to true to indicate that J has completed at the next decision point (Line 27). RASCO will then check whether any successor jobs of J should be released (Line 28), shift their release times and completion times based on the completion of J (Line 29), add them to Q, and remove J (Line 30).

Finally, RASCO saves the current set of job-budget pairs in S to Schedule for decision point t (Line 31). If the ready set Q is empty and there is no future anchor point, then the static schedule is completed (Line 32) and the algorithm returns whether the taskset was schedulable (Line 39). Otherwise, RASCO prepares for the allocation in the next segment, starting from the new decision point t (Line 33). It marks all source jobs whose release times are equal to t as ready and adds them to the ready set Q, then updates the set of future anchor points $\mathcal{A}$ (Lines 34–36). Finally, it updates the immediate next decision point after t to be the minimum of any completion time of the jobs in the ready queue or the earliest future anchor point (Line 37). The algorithm then continues on to the next iteration to compute the next timing segment $[t, t_{next})$ (restarting from Step 1 onwards).

Schedulability test: The schedulability of the output schedule is determined by checking that the completion time $c_{J_{i,j,k}}$ of every job $J_{i,j,k} \in \mathcal{J}$ is no later than $A_{i,k} + D_i$, where $J_{i,j,k}$ is a job of task τ_j in taskgraph G_i, $A_{i,k}$ is the kth release of G_i, and D_i is the relative deadline of G_i. If this condition holds, the taskset is deemed schedulable for the output schedule computed by RASCO.

Figure 8 shows a simple taskset with three taskgraphs and the Schedule computed by RASCO.

Details of key functions. We next discuss the key functions used in RASCO in greater detail.

GETSCHEDSET **(Algorithm 2):** The function GETSCHEDSET computes the m jobs (if any) with the smallest deadlines in Q to be the scheduled set S and their total allocated budget R_s (Line 3). It then checks if the new completion time of any $J \in S$ is earlier than the next decision point t_{next}, as a result of getting an additional resource (Line 5). If this is the case, we have a newly created future decision point (at the new c_J) that is earlier than the immediate next decision point t_{next}. Therefore GETSCHEDSET updates t_{next} to be c_J (Line 6) and resets the resource-deadline allocations at the current decision point t for all jobs in Q *except* for the job J (Line 7).

The intuition behind this reset is that, since $c_J < t_{next}$, the current allocation for the jobs, which is to be applied for the timing segment from t to t_{next}, may not be the most resource efficient if it is used instead for the shorter timing segment $[t, c_J)$. This is because the improvement in the execution rates, which we aim to maximize, varies depending on the timing segment for which we compute the schedule. It is influenced by where in the program each job is (i.e., which phase), the phases it will enter until the next decision point, and how much improvement in execution rate can be achieved in these phases. Therefore, we redo the allocation for the jobs to avoid an inefficient allocation. Note that we only reset the allocation for other jobs in Q but not J, since we successfully shrunk J's completion time by giving it δ_J extra budget. We then compute a new scheduled set S and its total budget R_s (Line 9).

Note that if the base budgets of the jobs in S are large, R_s may initially be larger than $R_{\max}$ (the maximum platform resources) at the beginning of a segment, or after the budgets have been reset as described above. In this case GetSchedSet iteratively takes away resources from the job with the most slack in S (unless this job's completion time uniquely defines t_{next}) until the total budget R_s used by S is within $R_{\max}$ (Lines 10–16 of GetSchedSet).

ResourceAlloc **(Algorithm 3):** Intuitively, ResourceAlloc uses the multi-phase execution model $\Theta_{\tau \equiv J}$ to determine which job $J \in Q$ would benefit the most from an additional resource given the remaining resource budget $R_{\text{avail}} = R_{\max} - R_s$. This is done by considering the value of the lookup table θ_i^{Δ} for each phase θ_i that each job J will enter between its current instruction until reaching the next decision point (given its current budget β_J). Using the numbers of instructions that J will execute within each phase as the weights, we then compute a weighted sum of the $\theta_i^{\Delta}[R_{\text{avail}} \cdot e_j]$ values for each resource type indicated by $e_j \in I$ (Lines 10–15). The job with the highest weighted sum (called a score) is then selected to receive the resource type that produced the highest score (Lines 16–17). If no more resources can be given, ResourceAlloc returns null.

GetFinish **(Algorithm 4):** The GetFinish function computes the completion time of a job J under the current budget. It works by considering all the phases θ_i that J would execute under the current budget, starting from the current instruction count. It computes the total time that J will spend in these phases, assuming that J would be given the current budget β_J up to the next decision point and its base budget β_J^{init} afterwards. The time spent at each phase θ_i is calculated based on the worst-case instruction rate θ_i^r and the number of instructions in θ_i that J will execute.

4.5 Termination and Complexity Analysis

For Rasco to terminate, the algorithm must break out of both while-loops. The outer while-loop, which computes the segments, executes at most $2 \cdot |\mathcal{J}|$ times since each iteration corresponds to a decision point, and there are at most two unique decision points for every job in $\mathcal{J}$ (the job's release and completion times). For termination, each of these iterations then requires the inner-while loop to be broken, which occurs when the function ResourceAlloc returns null for J. This occurs trivially if $Q = \emptyset$, or when both of the following conditions are met: (1) for every $J \in Q$ such that $J \notin S$, $\beta_J = R_{\max}$, and (2) $R_s = R_{\max}$. Assuming the worst-case scenario where $\beta_J^{init} = 0 \ \forall J \in Q$, meeting these conditions for a single segment requires $(|Q| - m + 1) \cdot ||R_{\max}||_1$ iterations. Recall, however, that budgets can be reset when giving a resource to a job J causes the segment boundary t_{next} to change (i.e., $c_J < t_{\text{next}}$). Nonetheless, this can only happen a finite number of times because GetSchedSet guarantees that t_{next} will never increase once a smaller completion time is found. Therefore, the number of resets in a segment is bounded and the above two conditions will be met eventually. At this point, the inner while-loop is broken, which enables the outer while-loop to advance to the next iteration. Since these iterations are bounded, Rasco is guaranteed to terminate.

In particular, the number of resets in a single segment is upper bounded by $|Q| \cdot ||R_{\max}||_1$, since there are at most $|Q| \cdot ||R_{\max}||_1$ unique values of t_{next}. Therefore, for a fixed number of resource partitions and cores, the runtime complexity of Rasco is $O(|Q|^2 \cdot |\mathcal{J}|)$. In other words, Rasco has a linear dependency on the number of jobs in the hyper-period, and a quadratic dependency on the number of tasks in the taskset (since $|Q|$ is the number of concurrently released job, which is upper bounded by the number of unique tasks in implicit/constrained deadline systems).

5 Numerical Evaluation

To evaluate the effectiveness of Rasco, we conducted a series of experiments using synthetic real-time DAG taskgraphs with resource-intensive benchmarks as workloads. Our goal was to evaluate the schedulability and end-to-end latency for tasksets, as well as Rasco's running time.

Taskset generation. We randomly generated tasksets with varying target taskset utilization, as in [31]. To generate a taskset T with a target utilization U_T, we constructed n taskgraphs $G_1, \ldots, G_n$ whose individual utilizations $U_{G_i} = \sum_{\tau \in G_i} \frac{\text{WCET}_\tau}{P_i}$ sum to the target utilization (i.e., $\sum_{i=1}^{n} U_{G_i} = U_T$). We leveraged the DAG creation tool [8] developed by [31] to generate 100 unique tasksets for each target utilization. The taskset target utilizations are set in the ranges $[0.2, 5.0]$, $[0.2, 8.0]$, and $[0.2, 10.0]$ at steps of 0.2, for $m = 4$, $m = 6$, and $m = 8$ cores, respectively. This produced a total of 2500, 4000, and 5000 tasksets for the three settings of m.

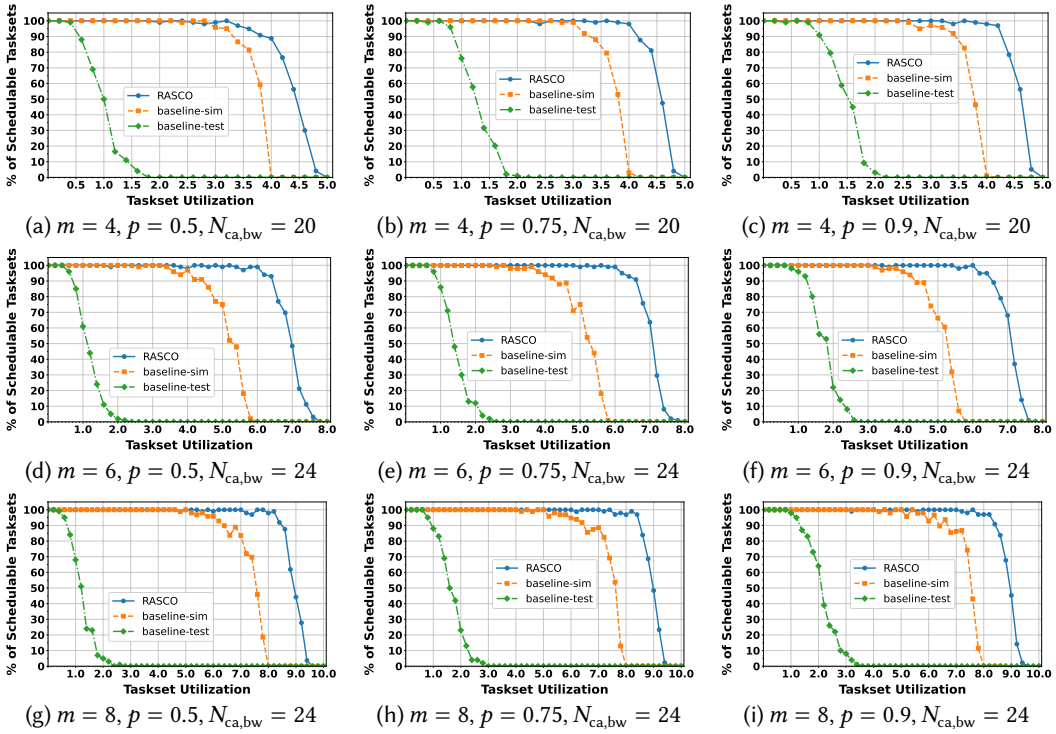
Each taskset consists of $n = 5$ taskgraphs, with each taskgraph's utilization uniformly distributed in $[0, m]$ (calculated by the classic UUniFast-Discard algorithm [9]). To create each taskgraph, the DAG generation tool randomly selects a number of layers in $[minlayer, maxlayer]$, populates each layer with a random number of nodes in $[1, 4]$, and then assigns edges between these nodes with probability p. It then creates a single source node and a sink node. (These can be dummy nodes, e.g., if a taskgraph application has multiple source and/or sink nodes). For our experiments, we set $minlayer = 3$, $maxlayer = 8$, and considered three different values of p, at 0.50, 0.75, and 0.90. Since p is the probability of adding an edge between nodes, we used it as a proxy metric for how sequential the resulting taskgraphs are. For $m = 8$ cores, we increased the number of taskgraphs per taskset to 10 to achieve higher taskset utilizations and to increase taskset-level parallelism.

We next assigned a workload to each node (task) in the taskgraph. We picked the workload uniformly at random from our set of benchmarks (*canneal*, *dedup*, *fft* and *streamcluster*). We set the reference WCET of each task τ to be the WCET of its assigned workload under uniform resource allocation $\beta_{\text{even}} = (N_{\text{ca}}/m, N_{\text{bw}}/m)$, obtained through measurements (as discussed in Section 3.1). Since we used real benchmarks, we could not select each taskgraph's period from a candidate set of periods and simply adjust the execution times to match the target utilization (e.g., as done in [31]). Therefore, we calculated a period for each DAG G_i such that the utilization U_{G_i} assigned by the DAG creation tool is maintained: $P_i = \left(\sum_{\tau \in G_i} \text{WCET}_\tau\right) / U_{G_i}$. We then rounded the period to the nearest harmonic period defined by the closest power of 2. We kept only the tasksets T whose resulting utilization after this procedure is within 0.05 of the target utilization U_T.

Finally, we applied our proposed algorithm in Section 3.5 to construct a multi-phase model for each benchmark from its execution profile, obtained via measurements (Section 3.1). The model of each benchmark was then used for every task that was assigned the benchmark as its workload. The number of phases k for *canneal*, *dedup*, *fft*, and *streamcluster* were 70, 40, 40, and 15, respectively.

Implemented algorithms. The output of Rasco is a static schedule for a hyper-period which follows global EDF, where the job deadlines are computed based on the fine-grained resource allocation and deadline decomposition co-design. As we are not aware of any prior work on the co-design of resource allocation and scheduling, or on dynamic fine-grained resource allocation, that provides hard timing guarantees, we compared Rasco with global EDF when each core is statically assigned an even partition of the resources $\beta_{\text{even}} = (N_{\text{ca}}/m, N_{\text{bw}}/m)$ to understand the impact of our co-design and fine-grained dynamic resource allocation on schedulability. For a fair comparison, we applied the same deadline decomposition method [15] used in our pre-processing to assign job deadlines. We then implemented two different baselines based on this method.

The first is Baseline-Test, which is the utilization-based schedulability test derived in [15]. We used this as a lower bound on our schedulability results and to analytically evaluate the effect of varying our configuration parameters on schedulability. However, since Rasco computes a static schedule with synchronous job release at $t = 0$, for a better comparison, we implemented another algorithm called Baseline-Sim that uses the same deadline decomposition technique under even resource partitions and simulates a full hyper-period for each taskset under global EDF.



(a) $m = 4, p = 0.5, N_{\text{ca,bw}} = 20$ (b) $m = 4, p = 0.75, N_{\text{ca,bw}} = 20$ (c) $m = 4, p = 0.9, N_{\text{ca,bw}} = 20$

(d) $m = 6, p = 0.5, N_{\text{ca,bw}} = 24$ (e) $m = 6, p = 0.75, N_{\text{ca,bw}} = 24$ (f) $m = 6, p = 0.9, N_{\text{ca,bw}} = 24$

(g) $m = 8, p = 0.5, N_{\text{ca,bw}} = 24$ (h) $m = 8, p = 0.75, N_{\text{ca,bw}} = 24$ (i) $m = 8, p = 0.9, N_{\text{ca,bw}} = 24$

Fig. 9. Percent of schedulable tasksets for increasing p values (L to R) and increasing cores (top to bottom).

We implemented all three algorithms, RASCO, BASELINE-TEST, and BASELINE-SIM, within 1,273 lines of Python code and 675 lines of C code (approximately 2,000 LoC in total).

Schedulability results. Figure 9 shows the schedulability results (the percent of the 100 tasksets that were found schedulable) for RASCO, BASELINE-SIM, and BASELINE-TEST across taskset utilizations.

The first row of the figure shows the results for $m = 4$ cores, with $N_{\text{ca}} = 20$ cache partitions and $N_{\text{bw}} = 20$ bandwidth partitions. Therefore, both baselines have static resource partitions of size $\beta_{\text{even}} = (5_{ca}, 5_{bw})$ for each task on each core. Figure 9(a), (b), and (c) shows the schedulability results for $p = 0.50$, $p = 0.75$, and $p = 0.90$, respectively. Notice that the BASELINE-TEST schedulability increases as p increases. As more edges are added, the critical path of each taskgraph increases. Since all of the generated workloads must have a period larger than their critical path (otherwise, the taskgraph is trivially unschedulable), the taskgraphs generated with more edges (larger p) tend to have fewer nodes and therefore are easier to schedule.

Across all m values and p values, RASCO significantly outperforms BASELINE-SIM at high utilizations. In Figure 9(c), for example, RASCO can schedule approximately 55% more tasksets than BASELINE-SIM at utilization 3.8. At utilization 4.0, BASELINE-SIM is unable to schedule any of the tasksets, whereas RASCO can schedule almost all of them. More surprisingly, RASCO continues to maintain high schedulability as the taskset utilization increases beyond the platform capacity. For instance, at taskset utilization of 4.5, RASCO can schedule over 65% of the tasksets. The results clearly illustrate the combined benefit of co-design and fine-grained resource allocation, especially at heavy loads.

Overall, however, the performance of BASELINE-SIM is high. Since the number of shared cores is small, the size of the even resource partitions ($\beta_{\text{even}} = (5_{ca}, 5_{bw})$) is relatively large when we

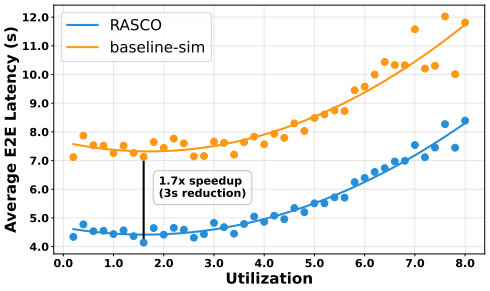
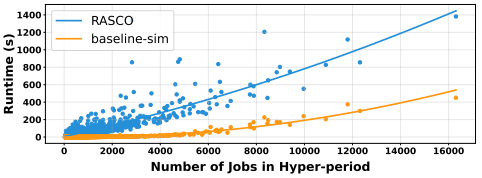


Fig. 10. Latency of Rasco vs. Baseline-Sim ($m = 6$).



Algorithm	Min. (s)	Median (s)	Max. (s)
Rasco	1.506	19.284	1382.011
Baseline-Sim	0.004	0.112	450.137

Fig. 11. Runtime data ($m = 6$)

keep N_{ca} and N_{bw} fixed. For some tasks, setting $\beta = (5_{ca}, 5_{bw})$ is enough for them to reach their critical resource thresholds. This suggests that an even resource partitioning strategy can perform reasonably well on a platform with sufficient shared resources. If the resources are constrained, however, Rasco will become highly beneficial.

For example, recall that the platform that we collected profiles on (which has $N_{\text{ca}} = 20$ and $N_{\text{bw}} = 20$ partitions) has $m = 8$ cores. This means that an even split of these resources to cores would give $\beta_{\text{even}} = (2.5_{ca}, 2.5_{bw})$, which we observed was suboptimal for resource-sensitive tasks on this platform. Therefore, we expect performance to degrade significantly for Baseline-Sim on the reference platform. Since we cannot give out fractional partitions with CAT, however, we increase the number of resource partitions in the remaining experiments to $N_{\text{ca}} = N_{\text{bw}} = 24$ to have the same total amount of resources for all three algorithms when $m = 6$ and $m = 8$.

The second row of Figure 9 shows the schedulability results for $m = 6$ cores across varying p. Already, we see a large performance gap between Rasco and Baseline-Sim, since each job of Baseline-Sim is limited to $\beta_{\text{even}} = (4_{ca}, 4_{bw})$ while Rasco is able to dynamically reallocate all $N_{\text{ca}} = N_{\text{bw}} = 24$ resource partitions between jobs across their various execution phases. Notably, Baseline-Sim cannot schedule any taskset at utilization 6.0, while Rasco can schedule close to 100%, further confirming the consistent high performance benefits of Rasco as we scale the system.

We also notice that, under the same CPU load, Baseline-Sim's ability to schedule tasksets diminishes as the resources per core under even partitioning become more constrained (even when the total amount of resources in the system increases). This is demonstrated by the overall downward trend in schedulability when moving from the first row to the second row of the figure. For instance, consider the middle column: while Baseline-Sim can schedule all tasksets at 3.0 utilization on 4 cores (i.e., 75% CPU load), it can only schedule around 90% of the tasksets at the same CPU load (4.5 utilization on 6 cores) when it has one fewer resource partition. In contrast, Rasco is able to achieve the same or even better schedulability performance.

Latency. Importantly, we observed significant decreases in the average end-to-end latency of all taskgraph releases in the hyper-period for Rasco compared to Baseline-Sim, *across all utilizations*. Figure 10 shows these results for $m = 6$. Notice that at utilization 1.6, the average end-to-end latency is reduced from over 7 seconds to just over 4 seconds, and the magnitude of reduction is similar across all utilizations. This is due to Rasco's ability to efficiently allocate resources to the jobs that most benefit from them, thus reducing overall execution times and improving latency performance.

Runtime comparison. Figure 11 compares the runtime of Rasco to Baseline-Sim for all generated tasksets for $m = 6$ cores. Since Rasco and Baseline-Sim both compute static schedules, their runtimes scale linearly with the number of jobs in a hyper-period (as discussed in Section 4.5). However, Rasco's runtime also depends on the number of tasks per taskset, which varied

significantly across tasksets in our experiments, explaining the wider runtime variation compared to Baseline-Sim's. The table in Figure 11 compares the runtime summary statistics of the two algorithms. Overall, Rasco can efficiently compute schedules for tasksets with a very large hyper-period (with 16,000 jobs).

Runtime and storage scalability. To evaluate the scalability of Rasco, we performed an additional study with exponentially increasing taskset sizes. To show the quadratic dependency on the number of tasks, we fixed the number of tasks per taskgraph and set the number of taskgraphs per taskset to 2^i+1 for each step $i \in [1, 6]$. We then randomly generated 20 tasksets at each step size, ran Rasco on each taskset (with $m = 8$), and plotted the average runtime. Figure 12 shows the results. The largest taskset size contained 65 taskgraphs with 715 tasks and had a mean runtime of ~16 hours, which, although large, is feasible for an offline algorithm with known runtime complexity. Finally, the memory required to store a Rasco scheduling table as a C struct is $290 \cdot |\text{Schedule}|$ bytes, which is relatively small. For instance, the largest scheduling table in this study—with 10,834 segments and 5,765 jobs—required only ~3.14 MB of memory.

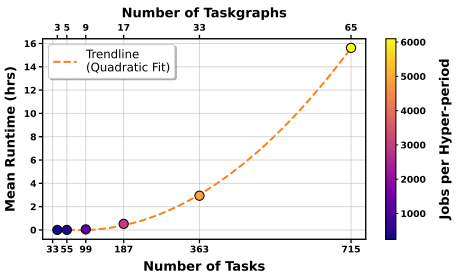


Fig. 12. Rasco runtime vs. taskset size ($m = 8$)

6 Prototype, Runtime Overheads, and Experimental Evaluation

To evaluate the runtime overheads, safety, and practical utility of Rasco, we implemented a prototype of a Rasco runtime scheduler within an RTOS and measured its runtime overheads. We first describe the prototype, then present a method for incorporating these overheads into Rasco, and experimentally evaluate the safety and utilization overhead of its overhead-aware schedules.

6.1 Rasco Runtime Scheduler Prototype

Our prototype was implemented in LITMUS^{RT} [5], a real-time scheduling framework built on top of Linux. LITMUS^{RT} enables developers to implement real-time schedulers as plugins, which are dynamically loaded and executed within the Linux kernel. Our prototype contains ≈ 880 LoC.

Core scheduling logics. The Rasco runtime scheduler operates by referencing a scheduling table computed offline by the Rasco algorithm. At the start of each experiment, the scheduler initializes a cross-core global segment index to track the current scheduling segment, along with individual core-specific counters. It then uses a newly added system call to load the scheduling table into the kernel, launches all DAG tasks, and finally releases all source tasks simultaneously via LITMUS^{RT}. Whenever a core's scheduler is invoked, it checks whether its local segment index is behind the global index (i.e., a new segment has reached). If so, the core updates its local counter and applies the new resource allocation for the new segment. It then references the scheduling table to determine the task it should execute and schedules it accordingly. At the first scheduler invocation (right after the initial task release), the scheduler also initializes a timer to fire at each subsequent decision point in the static schedule. The interrupt handler for this timer updates the global segment index and marks all cores to be preempted, prompting them to invoke their schedulers.

Work-stealing via under-run handler.[3] Since the static schedule assumes worst-case rates, a job may complete earlier than the time indicated by the schedule. To maximize resource utilization, we implemented an under-run handler to work-steal ready jobs (whose predecessors have all

[3]For robustness to system faults, our prototype also includes a WCET overrun handler (omitted here due to space constraints).

completed) that are scheduled in future segments to run on the idle core during the remaining of the current segment. If multiple such jobs exist, we pick the one with the earliest deadline, which can be efficiently determined since jobs are already sorted by deadlines in the scheduling table. Note that we simply utilize the idle core along with whichever resources it is currently assigned for early execution of the selected job(s), without changing the resource allocation.[4] Once the current segment ends, the core's scheduler schedules jobs based on the scheduling table as before.

Resource allocation. Using Intel's CAT [14], cache partitions can be assigned to CPUs via MSR registers: we first assign to each CPU a distinct CLOS (class of service) register value, and then associate with each CLOS a set of cache partitions (which must be contiguous), in the form of a bitmask. As Rasco outputs only the number of partitions per CPU, we convert its output to a bitmask value as follows: Suppose CPU i is allocated n_i partitions. Then, we assign the bottom n_0 bits to CPU 0, the next n_1 bits to CPU 1, and so on. Since Rasco keeps jobs pinned to the same core between segments, this strategy helps improve cache locality, preserving as many warm cache partitions as possible under dynamic resource allocation.

MemGuard [29] allocates memory bandwidth budget by using hardware performance counters to monitor L3 cache misses—as a proxy for bandwidth usage–on each CPU during each period. If a CPU exceeds its cache miss budget, MemGuard throttles it by running a *memguard* thread that spins until the next replenishment period. Since this thread must run at the highest priority, which is infeasible in LITMUS^{RT}, we instead set a special throttled bit on the CPU and trigger a scheduler invocation. We extended the scheduler to detect this bit and run our custom throttle thread that spins until the bit is cleared, after which normal scheduling logic resumes.

6.2 Experimental Setup

Platform. We ran our prototype on a CAT-enabled Intel Xeon E5-2683 v4 processor with 16 cores, 40MB 20-way set-associative shared L3 cache, and 3 single-channel 16GB PC-2400 DDR4 DRAMs. The shared cache and memory bandwidth are divided into $N_{\text{ca}} = N_{\text{bw}} = 20$ partitions each. We used $m = 4$ cores for our experiments. We disabled cache prefetching and CPU hyperthreading.

Tasksets. We used the same benchmarks, profiling technique, multi-phase model construction, and taskset generation as presented in Sections 3.1, 3.5, and 5, respectively. We generated 100 tasksets between utilization 0.2 and 5.0, at steps of 0.2. For each taskset, we applied Rasco algorithm to compute the static schedule, which was then provided as input to the Rasco runtime scheduler.

6.3 Runtime Overhead Evaluation and Overhead Accounting in Rasco Algorithm

Direct runtime overheads. We measured the direct overheads of our prototype through a series of microbenchmark experiments. Each taskset was executed on the experimental cores using the Rasco runtime scheduler for one minute, covering at least one full hyper-period. During each run, at each scheduler invocation—triggered by Rasco's timer interrupt handler at a decision point or by preemption from the underlying Linux scheduler—we recorded the time taken by the scheduler to perform all scheduling and resource allocation operations. The results are shown in Table 1.

In Table 1, "Rasco" refers to the overhead incurred by the core scheduling logics (i.e., the time required to look up the scheduling table and to schedule tasks). "Resource Allocation" denotes the time needed to configure the new cache and memory bandwidth partitions for all CPUs. "Under-run Handler" represents the overhead for performing work-stealing when a job finishes early.

[4]Although the selected job might execute under a different resource budget from its intended one, any work done during this work-stealing interval merely reduces its total amount of work and thus has no adverse effect. Each job is guaranteed to receive (at least) the resources determined by the static schedule for sufficient time and in the correct order.

Table 1. Measured Direct Runtime Overheads of RASCO Prototype Over All Utilizations

Overhead Type	# Observations	Min (µs)	Mean (µs)	95th (µs)	99th (µs)	Max (µs)
RASCO	14,536,633	0.02	0.03	0.04	0.05	18.10
Resource Allocation	84,656	1.66	2.53	4.29	5.80	23.30
Under-run Handler	13,182,104	0.02	0.12	0.38	0.69	22.16
Complete Prototype	18,782,307	0.56	2.33	4.92	6.55	61.45

Table 2. Time to Fill a 2MB L3 Cache Partition by Memory Bandwidth Budget ($\beta_{\text{bw}} \cdot 72$ MB/s)

β_{bw}	2	4	6	8	10	12	14	16	18	20
Mean (ms)	0.145	0.085	0.099	0.087	0.141	0.146	0.127	0.122	0.169	0.119
Max (ms)	1.134	0.395	0.383	0.474	0.818	0.775	0.819	0.818	0.729	0.591

Finally, "Complete Prototype" reports the total end-to-end overhead per scheduler invocation of our prototype, summing all of the above overhead components. Overall, we observe that the total scheduling and resource allocation overheads are very small, at only 2.33 microseconds on average, including the cost of work-stealing (which is not required by RASCO).

Indirect cache-related overheads caused by preemption and dynamic resource allocation. In addition to direct overheads, we also quantified the indirect cache-related overheads caused by preemption and the transient effects of dynamic resource reconfiguration on execution rate (c.f. Figure 4), both of which are workload- and resource-dependent. These overheads occur at decision points, either when a job resumes execution after being preempted or when its physical cache partitions are reallocated. They represent the maximum time required by a job to refill its useful cache content that was evicted by other jobs while it was preempted or that was on a cache partition that has been reassigned to other jobs. Since RASCO controls the cache and memory bandwidth partitions assigned to a job at each decision point, we can upper bound this delay with relative precision for each workload under each possible resource budget. For each benchmark program, we determined its **working set size** (**WSS**)—in terms of the number of cache partitions—and the refill cost for one cache partition under each memory bandwidth budget. The maximum cache-refill overhead incurred by a job under a given budget of $(\beta_{\text{ca}}, \beta_{\text{bw}})$ can then be safely estimated by multiplying the refill cost for one cache partition under the bandwidth budget of β_{bw} partitions by the maximum number of cache partitions to refill, i.e., the minimum of its WSS and β_{ca}.

To bound a program's WSS, we allocated to it the maximum memory bandwidth, then identified the number of cache partitions at which the program's execution time "levels off", i.e., no longer benefits from additional cache partitions. To estimate the maximum refill cost per cache partition under a given memory bandwidth budget β_{bw}, we implemented a synthetic benchmark that accesses a 2MB (the size of one cache partition on our platform) heap-allocated array. This benchmark performs a variety of read and write operations, including sequential, deterministic random, and strided memory accesses. We executed the benchmark twice back-to-back: the first execution with a cold cache, and the second with a warm cache. We measured the time taken by each execution and computed their difference, which indicates the time required to refill one cache partition. For each bandwidth budget, we repeated the above measurement over 100 runs, and recorded the maximum time required to fill one cache partition under that bandwidth budget. Table 2 shows sample results.

Incorporating runtime overheads into RASCO algorithm. Direct runtime overheads can be incorporated into the RASCO algorithm by inserting the maximum total end-to-end overhead per scheduler invocation (i.e., "Complete Prototype" overhead, measured at $\leq 61.45\mu s$) as a delay on each core after each decision point. Indirect overheads can be accounted for in the static schedule

at decision points where a job resumes after preemption or experiences a change in its physical cache partitions—both cases are identifiable from our static schedule and deterministic policy for mapping physical cache partitions to cores (c.f. Section 6.1). To ensure safe overhead accounting, we make no assumption about the memory layout of the program within its cache partitions. Instead, we conservatively assume that a job loses its cache content whenever its physical partitions are reallocated—even if some of its original partitions are retained. By adding the maximum overhead for cache refill as a delay, the job's worst-case instruction rate under the new resource allocation is guaranteed to hold after this delay. Finally, we enforce a minimum segment length (equal to the worst-case sum of the direct and indirect overheads) to ensure that the resource budget is only ever reconfigured to execute useful work. By explicitly accounting for both direct and indirect overheads, Rasco produces an overhead-aware schedule that ensures timing guarantees in practice.

Overall effect of overhead accounting. To evaluate the end-to-end impact of these overheads, we computed a new overhead-aware schedule for each generated taskset in Section 6.2 and calculated the percentage increase in CPU utilization compared to the original schedule. Figure 13 shows the results. Notably, under the overhead-aware Rasco, all but 9 of the 2,500 tasksets evaluated incur less than a 2% increase in utilization. Interestingly, in some cases, the utilization actually *decreases*. This is because the overhead-aware implementation of Rasco accounts for the job-level overheads *as it makes resource allocation and scheduling decisions*. Since the measured overheads are relatively small, the compounding effects of different resource allocation/scheduling decisions made under the overhead-aware algorithm can, in some cases, outweigh the costs of the overheads themselves.

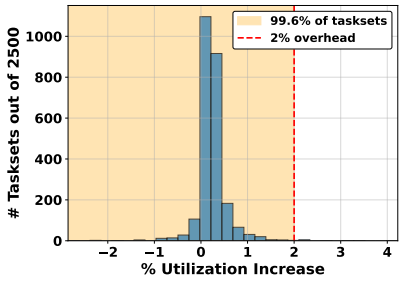


Fig. 13. Percent utilization increase: overhead-aware vs. standard Rasco.

6.4 Experimental Evaluation: Schedulability in Theory vs. in Practice

To evaluate the safety of the overhead-aware Rasco algorithm and its analysis, we conducted experiments by running the generated tasksets on our experimental platform under the Rasco runtime scheduler. The scheduler schedules each taskset based on its overhead-aware schedule (computed earlier). Each taskset was executed for one hyper-period, lasting at most 1 minute. During each run, we recorded the job completion times and determined their schedulability.

Figure 14 shows the percentage of schedulable tasksets observed experimentally, compared to those predicted through numerical analysis, across all *actual* taskset utilizations. The actual utilization of a taskset is defined as the total execution time of all its jobs in the static schedule, divided by its hyper-period. Notably, the percentage of schedulable tasksets observed experimentally is always equal to or greater than the theoretical results. Furthermore, any taskset deemed schedulable by Rasco theoretical analysis was also schedulable experimentally, and we observed no overruns. These results confirm that our overhead-aware algorithm achieves safe schedulability in practice.

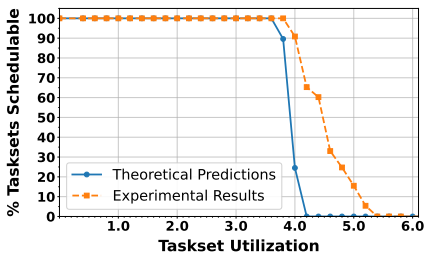


Fig. 14. Schedulability for overhead-aware Rasco: experimental results vs. theoretical prediction.

7 Conclusion

We presented RASCO, a co-design algorithm for taskgraphs that leverages resource-dependent multi-phase task models to jointly optimize resource allocation and scheduling. Evaluations on real benchmarks show that RASCO reduces latency, improves schedulability under high utilization, and supports much heavier loads than prior work. We demonstrated its practical utility with a prototype RASCO scheduler in an RTOS, evaluated runtime overheads, and integrated those overheads into the RASCO algorithm. Experimental evaluation confirms that the overhead-aware schedule preserves safe schedulability with minimal utilization cost. While we focused on taskgraphs, both the multi-phase model and RASCO are broadly applicable, including to non-DAG applications.

References

[1] Jaume Abella, Carles Hernandez, Eduardo Quiñones, Francisco J. Cazorla, Philippa Ryan Conmy, Mikel Azkarate-askasua, Jon Perez, Enrico Mezzetti, and Tullio Vardanega. 2015. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *SIES*.

[2] Sylvain Arlot, Alain Celisse, and Zaid Harchaoui. 2019. A kernel multiple change-point algorithm via model selection. *Journal of Machine Learning Research* 20, 162 (2019), 1–56.

[3] Sanjoy Baruah. 2014. Improved multiprocessor global schedulability analysis of sporadic DAG task systems. In *ECRTS*. DOI: https://doi.org/10.1109/ECRTS.2014.22

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*.

[5] John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. 2006. LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In *RTSS*.

[6] Daniel Casini, Alessandro Biondi, Geoffrey Nelissen, and Giorgio Buttazzo. 2020. A holistic memory contention analysis for parallel real-time tasks under partitioned scheduling. In *RTAS*.

[7] Nan Chen, Shuai Zhao, Ian Gray, Alan Burns, Siyuan Ji, and Wanli Chang. 2023. Precise response time analysis for multiple DAG tasks with intra-task priority assignment. In *RTAS*. DOI: https://doi.org/10.1109/RTAS58335.2023.00021

[8] Xiaotian Dai. 2022. dag-gen-rnd: A randomized multi-DAG task generator for scheduling and allocation research. (March 2022). DOI: https://doi.org/10.5281/zenodo.6334205. Accessed: November 7, 2024.

[9] P. Emberson, R. Stafford, and R.I. Davis. 2010. Techniques for the synthesis of multiprocessor tasksets. In *WATERS*.

[10] Robert Gifford, Neeraj Gandhi, Linh Thi Xuan Phan, and Andreas Haeberlen. 2021. DNA: Dynamic resource allocation for soft real-time multicore systems. In *RTAS*.

[11] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. 2009. Cache-aware scheduling and analysis for multicores. In *EMSOFT*.

[12] Qingqiang He, Nan Guan, Mingsong Lv, Xu Jiang, and Wanli Chang. 2022. Bounding the response time of DAG tasks using long paths. In *RTSS*. DOI: https://doi.org/10.1109/RTSS55097.2022.00047

[13] Qingqiang He, Xu Jiang, Nan Guan, and Zhishan Guo. 2019. Intra-task priority assignment in real-time scheduling of DAG tasks on multi-cores. *IEEE Transactions on Parallel and Distributed Systems* 30, 10 (2019), 2283–2295. DOI: https://doi.org/10.1109/TPDS.2019.2910525

[14] Intel. 2015. Improving Real-Time Performance by Utilizing Cache Allocation Technology. (April 2015). White Paper. Intel Corporation.

[15] Xu Jiang, Nan Guan, Xiang Long, and Han Wan. 2020. Decomposition-based real-time scheduling of parallel tasks on multicores platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2020), 2319–2332. DOI: https://doi.org/10.1109/TCAD.2019.2937820

[16] Hyoseung Kim and Ragunathan (Raj) Rajkumar. 2016. Real-time cache management for multi-core virtualization. In *EMSOFT*.

[17] Jakob Rosen, Alexandru Andrei, Petru Eles, and Zebo Peng. 2007. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *RTSS*.

[18] Abusayeed Saifullah, Sezana Fahmida, Venkata P. Modekurthy, Nathan Fisher, and Zhishan Guo. 2020. CPU energy-aware parallel real-time scheduling. In *ECRTS*.

[19] Abusayeed Saifullah, David Ferry, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D. Gill. 2014. Parallel real-time scheduling of DAGs. *IEEE Transactions on Parallel and Distributed Systems* 25, 12 (2014), 3242–3252.

[20] Junjie Shi, Mario Günzel, Niklas Ueter, Georg von der Brüggen, and Jian-Jia Chen. 2024. DAG scheduling with execution groups. In *RTAS*.

[21] Binqi Sun, Mirco Theile, Ziyuan Qin, Daniele Bernardini, Debayan Roy, Andrea Bastoni, and Marco Caccamo. 2024. Edge generation scheduling for DAG tasks using deep reinforcement learning. *IEEE Transactions on Computers* 73, 4 (2024), 1034–1047. DOI: https://doi.org/10.1109/TC.2024.3350243

[22] Corey Tessler, Prashant Modekurthy, Nathan Fisher, Abusayeed Saifullah, and Alleyn Murphy. 2023. Co-located parallel scheduling of threads to optimize cache sharing. In *RTSS*.
[23] Charles Truong, Laurent Oudre, and Nicolas Vayatis. 2020. Selective review of offline change point detection methods. *Signal Processing* 167, C (2020), 107299.
[24] Micaela Verucchi, Ignacio Sañudo Olmedo, and Marko Bertogna. 2023. A survey on real-time DAG scheduling, revisiting the global-partitioned infinity war. *Real Time Systems* 59, 3 (2023), 479–530.
[25] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*.
[26] Meng Xu, Robert Gifford, and Linh Phan. 2019. Holistic multi-resource allocation for multicore real-time virtualization. In *DAC*.
[27] Meng Xu, Linh Thi Xuan Phan, H. Choi, Y. Lin, H. Li, C. Lu, and Insup Lee. 2019. Holistic resource allocation for multicore real-time systems. In *RTAS*.
[28] Zijin Xu, Yuanhai Zhang, Shuai Zhao, Gang Chen, Haoyu Luo, and Kai Huang. 2023. DRL-based task scheduling and shared resource allocation for multi-core real-time systems. In *ICITES 2023*.
[29] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. 2016. Memory bandwidth management for efficient performance isolation in multi-core platforms. *IEEE Transactions on Computers* 65, 2 (Feb 2016), 562–576.
[30] Shuai Zhao, Xiaotian Dai, and Iain Bate. 2022. DAG scheduling and analysis on multi-core systems by modelling parallelism and dependency. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 4019–4038. DOI: https://doi.org/10.1109/TPDS.2022.3177046
[31] Shuai Zhao, Xiaotian Dai, Iain Bate, Alan Burns, and Wanli Chang. 2020. DAG scheduling and analysis on multiprocessor systems: Exploitation of parallelism and dependency. In *RTSS*. DOI: https://doi.org/10.1109/RTSS49844.2020.00022
[32] Shuai Zhao, Xiaotian Dai, Benjamin Lesage, and Iain Bate. 2023. Cache-aware allocation of parallel jobs on multi-cores based on learned recency. In *RTNS*.

Received 10 August 2025; revised 10 August 2025; accepted 11 August 2025