DYNAMIC RESOURCE ALLOCATION AND ITS APPLICATIONS TO MULTICORE REAL-TIME SYSTEMS

Robert Gifford

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2025

ACKNOWLEDGEMENT

Firstly, I am deeply grateful to my committee for their time, feedback, and dedication to helping me improve this dissertation. I owe special thanks to my advisors, Professor Linh Thi Xuan Phan and Professor Andreas Haeberlen, and to my undergraduate advisor, Professor Gabriel Parmer. I could not have asked for better mentors. Linh has challenged me over the past six and a half years to be critically minded and independent—to look more closely, think more deeply, and grow as a researcher in real-time systems. Her generous investment of time and her patient guidance are gifts for which I will be forever grateful. Andreas has been a fantastic mentor whose insight and enthusiasm helped me learn to design and build better systems, lessons that heavily influenced the infrastructure that enabled our research. I am grateful for his steady guidance on some of the most challenging engineering problems I have faced. Gabe set me on this path in the first place, and without his encouragement and mentorship throughout my undergraduate years I would not have discovered my passion for research and systems.

I would like to acknowledge the many friends I have made while living in Philadelphia. From late nights studying and celebrating to our adventures in and around the city, you have made this place feel like home. I am especially thankful to Dr. Michael D'Agati, whose regular check-ins created countless chances to reconnect, reset, and enjoy life outside the lab. I am grateful to the friends I lived with over the years—Dr. Nick Rioux and Rachel and Ryan Kellner—for making our home welcoming and fun. I also thank my mentor and lab mate, Dr. Neeraj Gandhi, for helping me learn to be a successful student here at Penn.

I am forever thankful to my parents, Anne and Whitney Gifford, for their lifelong support and for encouraging me to pursue an advanced degree. They have always done everything in their power to support me, and I will be endlessly grateful.

Lastly, I want to thank my partner, Zoe, whose love and support carried me through the hardest parts of this journey. Your patience, understanding, attention, and encouragement helped make this work possible, and I could not be more grateful.

ABSTRACT

DYNAMIC RESOURCE ALLOCATION AND ITS APPLICATIONS TO MULTICORE REAL-TIME SYSTEMS

Robert Gifford

Linh Thi Xuan Phan

Andreas Haeberlen

Multicore architectures have become prevalent in real-time embedded and cyber-physical systems due to their ability to address increasing computational demands. However, these architectures introduce challenges related to shared resource interference, particularly in last-level caches and memory bandwidth, which adversely affect system predictability and performance. Traditional static resource allocation methods, which isolate resources to prevent interference, lack the adaptability necessary to respond to dynamic task requirements, resulting in suboptimal performance and reliability.

This dissertation investigates dynamic resource allocation techniques across a variety of settings to overcome these limitations, emphasizing their effectiveness in improving schedulability, latency, and robustness in multicore real-time systems. It presents techniques that adaptively reallocate shared resources at runtime by leveraging knowledge of task execution behavior, such as distinct phases with varying resource needs. These methods demonstrate that dynamic resource allocation can effectively reduce contention and improve performance even under fluctuating workloads.

The dissertation explores dynamic allocation in a variety of system contexts. It begins with systems consisting of independent tasks under soft timing constraints, where fine-grained resource reallocation can substantially improve throughput and responsiveness while ensuring safety constraints. It then extends to multi-mode systems—systems that adjust their tasksets and execution behavior based on changes in system state—where dynamic allocation is used to optimize resource usage across and during mode transitions. Further, it incorporates systems with real-time control tasks, where the ability to co-design controller implementations and resource allocations leads to enhanced safety and schedulability. Finally, it investigates systems with complex task interdependencies, modeled as directed acyclic graphs (DAGs), where resource allocation must be co-designed with the scheduling algorithm to ensure end-to-end timing guarantees.

Together, these contributions demonstrate that dynamic resource allocation is a powerful and general technique for addressing the challenges of multicore real-time systems. By adapting to changing workloads and system conditions, dynamic allocation improves resource efficiency, predictability, and overall system performance across a wide spectrum of application scenarios.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER 1

INTRODUCTION

1.1 The Importance of Real-Time Systems

Computers now mediate an ever-growing portion of everyday life, from transportation and healthcare to finance, manufacturing, and the devices in our homes. These systems do not merely compute in the abstract; they sense the physical world, make decisions, and actuate effects that flow back into the world through brakes, motors, pumps, displays, and networked services [154, 109].

In many of these interactions, being merely functionally correct is not enough; results must also arrive on time. A braking command can be calculated perfectly, decelerating the car at the correct rate, but if it is issued too late, the result is moot; the car has already traveled farther than is safe. Likewise, a pacemaker pulse delivered at the incorrect time can destabilize a patient's heart rhythm even if the pulse amplitude and waveform are correct [111, 100]. This coupling between not just *what* the result is, but also *when* a result is produced gives rise to the notion of *real-time* computing.

At a high level, a real-time system is one in which *timeliness* is an explicit requirement alongside functional correctness. Timeliness means that each computation completes before a specified time bound, often called a deadline, so that the surrounding physical or cyber context can safely use the result. When the consequence of missing a bound is degraded quality (e.g., a frame drop in video or a longer response time in a web service), we speak of *soft* or *firm* real-time. When the consequence may be loss, damage, or hazard (e.g., control instability and failure, or collision risk), we speak of *hard* real-time [32]. Across this spectrum, the shared idea is that correctness includes constraints on *time*.

The importance of real-time thinking has grown as computing has moved closer to people and physical processes. Modern vehicles now contain dozens of networked processors coordinating powertrain, braking, and advanced driver-assistance; industrial robots share workcells with humans; medical devices monitor and intervene in real time; and even cloud services increasingly host user-facing interactions (gaming, telepresence, financial trading) where latency *and its variability* are business- and safety-relevant [225, 202, 56]. In each case, deadlines arise from physics, human factors, or service-level objectives. Because time is a requirement, not merely a resource, platforms must ultimately provide mechanisms and policies that align computation with temporal constraints.

Crucially, real-time does *not* mean “as fast as possible.” Instead, it means “fast enough, predictably, every time.” Where general-purpose systems often optimize average throughput, real-time systems should emphasize predictability and tight control of worst-case and tail behavior [110]. This shift in emphasis changes how systems are designed, analyzed, and validated.

In summary, real-time systems are important because they align computation with the time scales of the physical world and human interaction, making the *when* as critical as the *what*. When results arrive within guaranteed windows, control loops remain stable, machines coordinate safely with people, and risks can be bounded; when they do not, functionally correct outputs can still lead to failure.

1.2 Trends in Hardware Evolution

Against this requirement for timeliness, we examine the hardware that executes real-time workloads. Early real-time systems achieved predictability largely because they executed on single-core processors with relatively simple microarchitectures and strictly local resource usage [119, 205]. With only one hardware thread of execution, there were few factors that could introduce timing variance from one task run to another. It was mainly limited to hardware interrupts, scheduling preemption and I/O that could be bounded or disabled during critical sections [106]. This isolation, together with simpler pipelines and cache hierarchies, made the analysis of worst-case execution times (WCETs) tractable and enabled early scheduling theory—such as rate-monotonic and earliest-deadline-first to connect cleanly to implementation-time guarantees [119]. These early scheduling policies provided simple, closed-form schedulability tests that linked design-time task models to provable deadline guarantees on uniprocessors.

In practice, many deployed real-time products consisted of a few dedicated single-core controllers, each assigned to a narrow function and interacting over simple, time-triggered buses or field networks [106].

This architectural separation reduced interference by design, simplified certification arguments, and allowed engineers to compose end-to-end timing from well-understood building blocks.

Two long-run trends forced a shift away from that model. First, power and thermal limits curtailed frequency scaling as described by the end of Dennard scaling, so additional performance could no longer come “for free” from faster clocks [27]. Second, application demand rose dramatically, as perception, planning, connectivity, and analytics required more compute than a single core could supply at acceptable power consumption and heat output [13]. Multicore and many-core processors answered these pressures by offering higher throughput per watt, the ability to exploit parallel workloads, and the integration of heterogeneous accelerators on a single platform.

However, these platforms and their modern microarchitectural features are optimized for average-case performance rather than predictable timing. Multicore systems do not have fully isolated hardware resources. Shared last-level caches, memory controllers, and DRAM channels couple the timing of software running on different cores to one another. Unmanaged, this coupling increases latency, inflates variance, and erodes schedulability, because one task’s cache or memory activity can perturb another’s execution time—even when they run on different cores [204, 101, 199]. On the CPU itself, out-of-order execution, speculative pipelines, hardware prefetchers, simultaneous multithreading, and dynamic voltage and frequency scaling (DVFS) further complicate WCET reasoning by changing execution behavior based on recent history and system state [205]. As a result, the once-neat separation between per-task analysis and system-level interference no longer holds. Task execution times can vary widely when executed in the presence of other tasks on the same platform [138, 204, 101].

1.3 Trends in Software

As hardware complexity evolved, the needs of real-time software evolved in parallel. Older real-time applications frequently implemented compact, well-specified control algorithms—for example, boiler and turbine governors in power plants, engine control units, and flight-control loops—whose inputs, control periods, and failure modes were narrowly defined and stable over long lifetimes [32]. Their software architectures reflected this stability: a small number of periodic tasks, statically configured priorities, minimal dynamic behavior, and carefully bounded interrupt activity running on dedicated controllers with fixed interfaces [121]. By contrast, contemporary systems increasingly integrate perception, estimation, planning, communication, and sometimes learning-enabled components, interact continuously with networks and users, and adapt behavior at runtime, which multiplies the number of tasks, data paths, and timing dependencies that must be coordinated. A prominent trend within this growth is the rise of *multi-modal* systems whose workload, timing characteristics, and even control objectives shift over time. An autonomous vehicle switches among highway cruising, urban stop-and-go, and emergency maneuvers [225]. A robotic system alternates between exploration, manipulation, and safe-stopped states [202]. Datacenter services dynamically enable or disable low-latency inference, video processing, and interactive analytics based on user demand and system health [69].

These *dynamic* multi-modal systems create even more challenges to maintain safety assurances. First, the needs of the active task set and their inter-task communication patterns can change across modes; saftey critical timing analyses that assume a fixed set of jobs and rates must be revalidated whenever these elements change [156]. Second, *mode transitions* themselves must be safe and schedulable: enabling tasks for a new mode of operation and retiring tasks from the old one must preserve strict timing requirements throughout the transition [149]. Third, runtime variability—arising from factors such as input dependent code paths or external asynchronous I/O and networking complicate worst-case reasoning and inflates latency variance. Fourth, modern tasks are often connected by data dependencies, where the output of one stage becomes the input of the next. A delay or deadline miss in an upstream task propagates downstream as late arrivals, buffer buildup, or stale data, leading to cascading failures of end-to-end timing even when later tasks are locally schedulable [62].

Taken together, these trends in both hardware and software have shattered the clean abstractions that real-time theory historically relied on in the uniprocessor era. The simple assumption that each task runs in isolation with fixed overheads no longer holds; execution time is no longer an independent property of a task but instead depends on the system as a whole.

1.4 The Rise of Static Allocation

Given these growing hardware and software pressures, designers sought to suppress interference and regain deterministic runtime behavior. Advanced hardware features such as dynamic frequency scaling and simultaneous multithreading could be disabled in commodity processors. And crucially, vendors and system designers introduced mechanisms to partition shared hardware resources such as last level cache and memory bandwidth, dividing them into isolated slices to regulate their use. Examples on the hardware side include Intel Cache Allocation Technology (CAT) for cache capacity and ARM MPAM for bandwidth partitioning [94, 91]. On the software side, cache page coloring, cgroup-based throttling, and memory-bandwidth regulators such as MemGuard expose control levers at the operating-system level. [218, 223].

These mechanisms enable *static resource allocation*: engineers select fixed cache partitions, bandwidth budgets, and processor affinities so that tasks can no longer interfere directly with one another. These static configurations greatly simplify system complexity, and offer a straightforward path to deployment on modern hardware.

Mechanisms alone, however, do not answer *how* to divide resources to best meet system goals. A substantial body of work therefore computes *static* policies offline—from profiling data, requirement constraints, or analytical models—and installs the resulting fixed partitions at runtime as a conservative, predictable baseline [214, 142, 42].

1.5 The Need for Dynamic Resource Management

While static partitioning can simplify analysis, it also hard-codes assumptions about task behavior and system conditions. As software systems grow more dynamic—featuring bursty demand, shifting modes, and unpredictable data dependencies—static resource divisions can quickly become outdated or suboptimal. Overprovisioning isolates tasks at the cost of utilization, stranding valuable resources when demand is low. Underprovisioning leads to contention and latency spikes when demand unexpectedly rises.

To maintain predictability while improving efficiency, systems must adapt resource allocations at runtime. They must recognize when demand shifts, when tasks enter new phases or modes, and when interference patterns change. This dissertation advances the thesis that *dynamic* resource allocation—guided by runtime measurements and coordinated with the scheduler—is both necessary and practical on modern multicore platforms.

By moving cache capacity and memory bandwidth to the tasks that benefit most *now*, dynamic policies can reduce tail latency, increase schedulability, and preserve analyzability, even in the face of evolving workloads and mode transitions. The remainder of this dissertation presents dynamic allocation frameworks that meet these goals, evaluate their overhead and performance, and demonstrate their applicability to real-time systems.

1.6 Thesis Statement and Goals

Thesis. Dynamic resource allocation, co-designed with scheduling and informed by phase and mode structure, improves predictability and efficiency in multicore real-time systems relative to static approaches.

Goals. We pursue this thesis across four distinct and increasingly complex settings. (i) Soft independent real-time workloads where phase-aware, fine-grained reallocation reduces average and tail latency. (ii) Multi-modal systems where allocations must adapt both *within* a mode and *during* mode transitions. (iii) Real-time control systems where various controller implementations (e.g., sampling periods) can be co-chosen with resource allocations to preserve safety and maximize robustness while improving schedulability. (iv) DAG-structured applications where end-to-end timing hinges on coupling resource decisions with the scheduler's placement and ordering of dependent tasks.

1.7 Contributions

This dissertation contributes both new algorithms and models for dynamic resource management as well as experimental platforms that realize these ideas on PREEMPT-RT enabled Linux and LITMUS^{RT}, helping to bridge the gap between real-time theory and modern operating systems.

DNA: Phase-aware dynamic allocation for soft real-time. DNA profiles tasks to learn phase structure and resource sensitivity, then reallocates cache capacity and memory bandwidth at runtime to the tasks that benefit most. A deadline-aware variant (DADNA) incorporates slack and virtual-deadline reasoning so that allocations reflect both immediate speedups and proximity to deadlines. Together, they show that fine-grained reallocation can cut latencies and reduce deadline misses with small overheads [76].

Omni: Dynamic allocation for multi-modal systems. Omni lifts dynamics across *modes*. It computes allocations that are feasible for each mode and provides a protocol that preserves schedulability during transitions, mitigating contention spikes when enabling or disabling tasks. Omni demonstrates that dynamic, mode-aware allocation outperforms single-configuration baselines and naive per-mode partitions that ignore transition costs [77].

DECNTR: Control and resource co-design for safety and schedulability. DECNTR couples resource allocation with controller selection. Rather than fix control implementations and then squeeze them into a scheduler, DECNTR jointly chooses controller variants (e.g., sampling periods) and resource budgets so that the closed-loop system remains within its safe set while meeting deadlines across modes and transitions [75].

RASCO: Co-design of resource allocation and scheduling for DAGs. RASCO considers applications with task dependencies. It provides a model and algorithm that co-designs per-phase resource budgets with the scheduler's placement of DAG nodes, obtaining end-to-end guarantees while accounting for runtime overheads and the non-linear effects of cache and bandwidth on execution rates [65].

1.8 Organization of the Dissertation

Chapter 2 reviews foundational background on hardware, real-time task models, shared-resource mechanisms, and real-time operating systems used later. Chapter 3 surveys the state of the art in interference-aware real-time systems. Chapters 4–7 present DNA/DADNA, Omni, DECNTR, and RASCO, including models, algorithms, prototypes, and empirical evaluation. Chapter 8 discusses system implementations leveraged across this dissertation in detail. Lastly, Chapter 9 summarizes contributions and lessons learned as well as highlights remaining research questions for the future.

CHAPTER 2

BACKGROUND

This chapter provides the basic conceptual and technical groundwork for the remainder of the dissertation. The goal is to give the reader a compact but thorough overview of related hardware architectures, including processors, shared hardware resources, interrupts and timers, and foundational real-time task and scheduling models. Where possible, we emphasize the aspects that matter most for predictability on modern multicore systems, and that motivate the mechanisms and analyses developed in later chapters.

The chapter proceeds as follows. Section 2.1 surveys processor and memory hardware architectures, beginning with simple uniprocessors and tracing the evolution to multicore CPUs, cache hierarchies, and DRAM organization. Section 2.2 introduces real-time task models, deadlines, and guarantees, which form the basis of schedulability analysis. Section 2.3 then surveys foundational scheduling algorithms for both uniprocessors and multiprocessors, focusing on fixed-priority and earliest-deadline-first scheduling and their variants. Section 2.4 discusses worst-case execution time (WCET), reviewing classic static and measurement-based estimation methods, explaining why multicore platforms make exact WCET analysis intractable, and motivating the profiling-based approach used in this dissertation. Section 2.5 briefly introduces the concept and basic notation for multimodal systems. Section 2.6 discusses static and dynamic approaches to resource allocation in real-time systems, motivating the need for co-design with schedulers. Finally, Section 2.7 outlines real-time operating systems (RTOS), contrasting purpose-built RTOSes with general-purpose kernels extended with real-time capabilities, and describes how interrupts, timers, and kernel overheads shape predictability in practice.

2.1 Hardware Architectures

This section introduces the essential hardware concepts required to understand the timing behavior of tasks on modern processors. We begin with early single-core CPUs, whose simple designs allowed for nearly deterministic execution. We then trace the evolution to multicore architectures and highlight the rise of shared caches and memory hierarchies. Finally, we explain the organization of cache and main memory.

2.1.1 History of the Single-Core CPU

Early processors were designed around simplicity. A single hardware "thread" was supported at a time, corresponding to one sequential instruction stream. This thread was realized by a register file, a program counter, and a status register that together defined the execution state. The core fetched, decoded, executed, and retired instructions one at a time in a straightforward pipeline. Pipelines were shallow—often just a few stages—and instructions were executed in-order, which meant latencies could be treated as fixed. Main memory was accessed directly or through very small caches, so access times could often be modeled deterministically. Interrupt sources were few and could be masked, ensuring that asynchronous effects were minimal or absent. Because of this simplicity, executing the same instruction sequence on the same hardware yielded nearly identical timing outcomes [205]. This determinism was essential for early embedded and control systems, as it allowed developers to reason about timing with confidence.

2.1.2 The Shift to Multicore CPUs

By the early 2000s, the growth in single-core performance stalled. Dennard scaling broke down, meaning that smaller transistors no longer brought proportional reductions in power density. Raising clock speeds beyond a few gigahertz led to excessive heat and power consumption, while deeper pipelines and speculative execution offered diminishing returns [90, 193]. Processor vendors therefore shifted to multicore architectures, replicating cores on the same die[1]. Multicore designs delivered better throughput per watt and allowed parallel workloads to scale performance. Mainstream desktop chips soon doubled to two or four cores, and server-class processors scaled to tens of cores. Simultaneous multithreading (SMT) was also introduced, exposing multiple hardware contexts per core to increase utilization when one thread stalled.

From a performance perspective, these changes were effective. But for timing analysis, they introduced new *cross core interferences*. Each core retained its own private pipeline and registers, but many subsystems—caches, memory controllers, and the DRAM interface—became shared. As a result, the execution time of a job now depended not only on its own control flow and data but also on the activity of co-runners on neighboring cores.

[1]A *die* is a single piece of silicon cut from a semiconductor wafer during fabrication. It contains the processor's transistors and interconnects, and defines the level of integration at which cores share thermal and power constraints.

2.1.3 Cache Hierarchies

Caches bridge the gap between fast processors and comparatively slow main memory by storing recently used data close to the core. Modern multicore CPUs implement hierarchical caches: small L1 caches private to each core, larger L2 caches that may be private or semi-private, and a large last-level cache (LLC) shared by all cores on a die.

The LLC is organized into *cache sets*, each consisting of multiple *ways*. A memory block maps to exactly one set, typically determined by a subset of its physical address bits, and can be stored in any of the ways of that set. Because the LLC is shared, blocks from any core may occupy the ways of a given set, and thus one core can overwrite or *evict* another core's cache contents. When a block is evicted, subsequent accesses by the evicted core become cache misses that must be served from higher-latency levels of the hierarchy or from DRAM. This mechanism is the root of cache interference: the activity of one task can increase the miss rate and elongate access times for another.

Which block is chosen for eviction depends on the cache's *replacement policy*. There exist many replacement policies, including least-recently-used (LRU), pseudo-LRU approximations, random replacement, and more sophisticated adaptive schemes [139]. Although policies vary in detail, the principle is the same: when a set is full and a new block maps to it, one existing block is removed according to the policy's rules. This means that even if a task's working set fits comfortably into the cache in isolation, co-running tasks may displace its lines under contention, causing costly cache misses and subsequent main memory accesses. Such evictions can increase execution times in multicore systems compared to uniprocessors.

Interference at the LLC is therefore both common and difficult to predict. It depends on how task memory footprints map to sets, on the degree of overlap between tasks' active working sets (the complete set of data accessed at any given time), and on the replacement policy's dynamics. Later sections will discuss mechanisms to partition or control cache usage, but fundamentally the shared nature of caches creates a channel by which tasks contend for resources and disturb one another's execution timing.

2.1.4 Main Memory and DRAM Organization

Beyond caches lies main memory, which most commodity platforms implement with dynamic random-access memory (DRAM). A modern processor does not connect directly to raw DRAM chips. Instead, requests from the cores flow through a *memory controller*, a logic unit that translates physical addresses into DRAM commands, arbitrates among competing requests, and enforces electrical and timing constraints. Early controllers were separate chips on the motherboard, but in modern processors they are integrated directly on the die, close to the cores. This integration reduces latency and gives the processor tighter control over memory bandwidth allocation.

DRAM itself is organized hierarchically. A *channel* corresponds to a wide data bus to a group of DRAM devices, and multiple channels can operate in parallel to increase aggregate bandwidth. Each channel contains one or more *ranks*, which are groups of DRAM chips that respond in lockstep to provide a large amount of data. Within each rank are multiple *banks*, each of which is a two-dimensional array of rows and columns. Every bank contains a *row buffer* that caches the most recently accessed row of data.

When a core issues a load or store that misses in the cache hierarchy, the request travels over the processor's internal interconnect to the memory controller. The controller maps the physical address to a specific channel, rank, and bank. If the request targets the row currently held in that bank's row buffer, it can be served quickly as a *row hit*. If it targets a different row, the controller must close the current row and open the new one, which takes longer and consumes bandwidth. If multiple cores issue requests that map to the same bank, they must be serialized, and requests may experience queuing delays. Meanwhile, requests that map to different banks or channels can often proceed in parallel.

This organization introduces two broad classes of contention. *Spatial conflicts* arise when different streams map to the same bank but different rows, forcing frequent row buffer evictions. *Temporal conflicts* arise when the aggregate request rate from all cores exceeds what the controller and channels can sustain, filling queues and delaying service even without direct row conflicts. In both cases, the result is increased memory access latency for some tasks, which in turn increases their execution time compared to running in isolation [101, 220, 166].

2.2 The Classic Real-Time Task Model

We often model applications as sets of real-time tasks. Let $\mathscr{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ denote the task set. Each task τ_i is characterized by three timing attributes: a period (or minimum inter-arrival time), a relative deadline, and a worst-case execution time (WCET). The period T_i specifies the minimum separation between successive jobs of τ_i. The relative deadline D_i specifies the maximum allowed response time of a job of τ_i from its release to its completion. The WCET C_i denotes an upper bound on the execution requirement of any job of τ_i when run in isolation on a dedicated processor.

Each task τ_i generates an infinite sequence of jobs $\{J_{i,1}, J_{i,2}, \ldots\}$. The k-th job $J_{i,k}$ arrives at time $a_{i,k}$ and must complete no later than its absolute deadline $d_{i,k} = a_{i,k} + D_i$. Each job may execute for any duration up to C_i, but no more. The average processor demand of task τ_i is measured by its utilization $U_i = C_i/T_i$, and the total system utilization is $U = \sum_{i=1}^{n} U_i$ where n is the number of tasks. These parameters form the basis for schedulability analysis in subsequent sections [119].

Systems are often categorized by the relation between D_i and T_i. In *implicit-deadline* systems, deadlines equal periods ($D_i = T_i$). In *constrained-deadline* systems, deadlines are no larger than periods ($D_i \leq T_i$).

2.2.1 Periodic vs. Sporadic Arrivals

Task arrival models differ in how strictly they constrain job releases. In the *periodic* model, a task releases jobs exactly every T_i time units, yielding a perfectly regular stream of arrivals. In the more general *sporadic* model, jobs may arrive irregularly, but successive arrivals are separated by at least T_i. The sporadic model subsumes the periodic model while also capturing event-driven workloads where arrivals depend on external stimuli. In this dissertation, we adopt the periodic model for simplicity, which is sufficient to illustrate the resource allocation and scheduling mechanisms developed in later chapters.

2.3 Scheduling Fundamentals

At its core, scheduling is the process of deciding which job runs at any given instant. The scheduler maintains a set of ready jobs, usually sorted based on different ideas of priority, and chooses one to execute on each processor according to a well-defined policy. In general-purpose systems, the scheduler's goal is often to balance responsiveness, throughput, and fairness across users and applications. In real-time systems, the objective is stricter: the scheduler must ensure that every job meets its specific deadline.

Real-time scheduling theory provides a framework for answering two fundamental questions: (i) given a set of tasks, is there a schedule in which all jobs meet their deadlines (the *schedulability problem*), and (ii) what scheduling algorithm can construct such a schedule at runtime. Seminal work by Liu and Layland [120] established much of the classical foundation by analyzing periodic task systems under preemptive scheduling. Their results showed that deadlines could be guaranteed using either fixed-priority or dynamic-priority scheduling, depending on how priorities were assigned.

At runtime, the notion of the "highest priority" job is central. Priority can be static or dynamic. In static (fixed-priority) scheduling, each task is assigned a fixed priority in advance, and every job of that task inherits the same priority. In dynamic-priority scheduling, a job's priority can change over time, usually as a function of its deadline or remaining slack. These two approaches give rise to the most studied uniprocessor schedulers: rate-monotonic (RM) scheduling as the canonical fixed-priority policy [120], and earliest-deadline-first (EDF) scheduling as the canonical dynamic-priority policy [58]. Both have been shown to be optimal within their respective classes: RM among fixed-priority assignments, and EDF among all uniprocessor scheduling algorithms for implicit-deadline task sets.

The remainder of this section first briefly reviews the basics of uniprocessor fixed-priority scheduling, then earliest-deadline-first scheduling, before moving on to multiprocessor scheduling models and briefly ending with demand bound scheduling.

2.3.1 Fixed-Priority Scheduling on a Uniprocessor

In fixed-priority (FP) scheduling, each task is assigned a static priority that does not change over time. RM assigns higher priority to tasks with shorter periods.

As mentioned before, Liu and Layland proved that RM is optimal among all fixed-priority assignments for implicit-deadline task sets and also derived a utilization-based schedulability bound: any task set with total utilization

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

is guaranteed to be schedulable under RM. There are many other variations of fixed-priority scheduling policies, including Deadline Monotonic and Slack Monotonic which set static priorities based on relative deadline and slack time respectively [14]. The details of these policies fall outside the scope of this dissertation.

2.3.2 Earliest-Deadline First on a Uniprocessor

Earliest-deadline-first (EDF) scheduling is one of the most popular dynamic-priority scheduling policy for real-time systems and we use it throughout this dissertation. At every scheduling point, the scheduler selects the job with the closest absolute deadline to execute. Unlike fixed-priority schemes where tasks are statically ordered, EDF dynamically reorders jobs as deadlines evolve.

The optimality of EDF for uniprocessors was first established in the work of Liu and Layland [120], who proved that EDF is optimal for scheduling periodic task sets with implicit deadlines. A task set is said to be *feasible* if there exists *any* schedule—regardless of algorithm—that allows all jobs to meet their deadlines and EDF is optimal in the sense that if a task set is feasible, then EDF will produce a schedule in which all deadlines are met. They also showed that the schedulability condition under EDF for uniprocessors is both necessary and sufficient: a task set is schedulable under EDF if and only if the total utilization does not exceed one:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

This result was extended and generalized by Dertouzos [58], who showed that EDF is optimal not just for periodic or sporadic tasks, but for arbitrary collections of jobs: among all preemptive uniprocessor scheduling algorithms, EDF produces a schedule in which all jobs meet their deadlines if any such schedule exists.

In practical terms, EDF offers two key advantages. First, it eliminates the need for manual or static priority assignment, avoiding the need for heuristics like rate-monotonic or deadline-monotonic ordering. Second, its utilization-based schedulability test is exact for implicit-deadline tasks, providing a simple and tight check for feasibility.

However, EDF also introduces several implementation considerations. Because job priorities change over time, the system must support dynamic priority queues or other mechanisms to efficiently select the next job to run. EDF is also sensitive to release jitter and execution-time overruns: if jobs do not arrive or finish when expected, the tight packing of EDF schedules can leave less room to recover.

2.3.3 Multiprocessor Scheduling: Partitioned, Global, and Clustered

On multiprocessors, three broad categories of schedulers exist.

Partitioned schedulers assign each task statically to a core and use uniprocessor scheduling within each core. This avoids migrations and preserves cache locality but suffers from bin-packing inefficiencies. Deciding an optimal partition is NP-hard (analagous to bin-packing, even for fixed-priority real-time tasks– a fact established by Leung and Whitehead [72].

Global schedulers maintain a single run queue of all jobs and allow jobs to migrate freely across cores. Global EDF (G-EDF) and global FP are good examples. In G-EDF, the *m* jobs with the earliest deadlines are scheduled across *m* cores. G-EDF allows job migration and can avoid bin-packing limitations, but incurs additional preemption and migration costs and is not optimal [59].

Clustered schedulers form groups of cores (often aligned with last level cache slices) and schedule globally within each cluster. This hybrid approach balances scalability with migration costs.

Each model has benefits and drawbacks, and no consensus exists on a universally best approach. Later chapters of this dissertation assume either partitioned or global EDF as a baseline, consistent with much of the theoretical literature.

2.3.4 Demand-Bound Functions and Demand-Based Schedulability

Thus far, every scheduling policy we have discussed has come with a *utilization*-based schedulability check: for fixed priorities we relied on per-task utilizations and classical bounds, and for EDF we noted the simple $\sum_\tau U_\tau \leq 1$ test (exact only under implicit deadlines). These tests are attractive because they are fast, easy to apply, and provide immediate intuition about load (often $O(1)$ or linear-time checks). However, this convenience comes at a cost: utilization bounds are generally *sufficient but not necessary*. On constrained-deadline task sets—especially in multiprocessor contexts—they can be overly pessimistic, rejecting feasible task sets [52]. To reason more precisely while still analyzing offline, we can switch viewpoints from *capacity used on average* to *worst-case demand due in any window*. Demand-based schedulability tests formalize this idea via demand-bound functions (DBFs): if, for every interval length t, the maximum cumulative work that could be both released and due within that interval does not exceed the processor supply t, then all deadlines are met [19]. The remainder of this subsection develops DBFs and the associated exact uniprocessor EDF test; later chapters leverage this in more complex settings.

Setup and notation. Consider a uniprocessor core k with task set $\mathscr{T}_k$ scheduled by EDF. Each task $\tau \in \mathscr{T}_k$ is characterized by its worst-case execution time C_τ, period T_τ, and relative deadline $D_\tau \leq T_\tau$ (constrained-deadline model). Define the utilization of τ as $U_\tau \triangleq C_\tau / T_\tau$ and the total core utilization as $U_k \triangleq \sum_{\tau \in \mathscr{T}_k} U_\tau$. Let $D_k \triangleq \max_{\tau \in \mathscr{T}_k} D_\tau$ denote the largest relative deadline on core k.

Demand-bound function (DBF). For any interval length $t > 0$, the demand-bound function of task τ is

$$dbf_\tau(t) \;=\; \max\left\{0,\; \left\lfloor \frac{t - D_\tau}{T_\tau} \right\rfloor + 1\right\} C_\tau,$$

which upper-bounds the maximum cumulative execution demand of jobs of τ that can both arrive and be *due* within some length-t window. The total demand on core k is the sum over the tasks assigned to it:

$$dbf_k(t) \;=\; \sum_{\tau \in \mathscr{T}_k} dbf_\tau(t).$$

EDF demand-based schedulability test. A necessary and sufficient condition for schedulability of $\mathscr{T}_k$ under EDF on a single core is

$$U_k \le 1 \quad \text{and} \quad \forall t \in (0, L_k] : \; dbf_k(t) \le t,$$

where a finite *check horizon* L_k can be chosen as

$$L_k \;=\; \max\left\{ D_k,\; \frac{1}{1-U_k} \sum_{\tau \in \mathscr{T}_k} (T_\tau - D_\tau)\, U_\tau \right\}.$$

Intuitively, $dbf_k(t) \le t$ certifies that in every length-t window, the worst-case cumulative demand does not exceed the processor supply on a unit-speed core.

Special cases and practical checks. For implicit-deadline systems ($D_\tau = T_\tau$), the per-task DBF simplifies to $dbf_\tau(t) = \lfloor t/T_\tau \rfloor C_\tau$, and the horizon reduces to $L_k = D_k = \max_\tau T_\tau$. In practice, it suffices to check $dbf_k(t) \le t$ at a finite set of *critical* window lengths no greater than L_k, typically values of the form $t = q\,T_\tau + D_\tau$ for integers $q \ge 0$ with $t \le L_k$.

Why DBF matters (and what comes next). DBF turns feasibility into a finite set of demand-vs.-supply checks, capturing effects of phasing and non-implicit deadlines that utilization bounds miss. Later chapters leverage this demand-centric view in richer settings (e.g., bounding carry-in across multi-mode events and linking C_τ to resource assignments).

2.3.5 Overheads and Capacity Loss

Classical theoretical schedulability bounds assume zero-cost preemptions, migrations, and scheduling decisions. In reality, all of these incur overhead as we will see in later chapters. Preemptions and migrations flush pipelines and caches, invalidating useful state and leading to *implicit* overheads due to the time needed to refill a task's working set. The preemption and context switch mechanism itself, consume CPU cycles and are example of *explicit* overheads. Accurate analysis must therefore inflate execution budgets by overhead bounds and treat them as capacity loss. [33] Later evaluations in this dissertation explicitly measure and account for such overheads on Linux.

2.4 Worst-Case Execution Time Estimation

The worst-case execution time (WCET) of a task is the maximum time that any of its jobs may require to complete on a given platform. WCET bounds are the foundational input to schedulability analysis and admission control: without them, one cannot prove that all deadlines will be met under a given scheduling policy.

2.4.1 Definition and Role of WCET

A task's WCET bound summarizes the most demanding circumstance under which the task may execute, including all relevant control-flow paths and microarchitectural effects that can influence runtime. Underestimation of WCET can lead to deadline misses and safety hazards, whereas excessive overestimation wastes capacity, reduces utilization, and inflates cost margins. In practice, WCET is always computed relative to a specified platform configuration, including processor core type, frequency settings, cache and memory allocation, and the operating-system kernel features that may introduce overheads or jitter. These details are crucial because changes at any of these layers can alter timing behavior substantially.

2.4.2 Static WCET Analysis

Static WCET analysis estimates a task's worst-case execution time without running it, by combining program-level structure with a model of the underlying hardware [205]. At the program level, the analysis constructs a control-flow graph (CFG), identifies loops and derives bounds on their iteration counts, and determines feasible execution paths. A control-flow graph consists of *basic blocks*, which are maximal sequences of instructions with a single entry and exit point and no internal branches. At the hardware level, analysis incorporates timing models of the pipeline, instruction latencies, and cache behavior to estimate execution cost for each basic block.

Cache analysis classifies memory references according to their predicted cache behavior—e.g., always-hit, always-miss, first-hit, first-miss, or not-classified—under a given replacement policy and cache configuration [68]. These classifications are used to bound the time cost of memory accesses. Once all basic blocks have been annotated with timing bounds, the total WCET is computed by composing them with the CFG structure. A common method is the Implicit Path Enumeration Technique (IPET), which formulates the WCET problem as an integer linear program that maximizes execution time over all feasible paths [115].

Several commercial and academic toolchains implement these static WCET analysis techniques, including *aiT*, *Bound-T*, and *OTAWA*, and are widely used in certification contexts for safety-critical embedded systems [205].

However, as hardware has evolved, increasing levels of complexity have made sound static analysis more difficult. Modern features violate the compositional assumptions required for static bounding, and full hardware models are often unavailable, especially on commercial multicore processors.

2.4.3 Measurement-Based and Probabilistic Approaches

Due to these difficulties measurement-based WCET estimation can be used in lieu of offline static WCET analysis. Measurement-based WCET estimation executes the program under test on the target hardware using carefully controlled system settings. Engineers then take the maximum observed execution time as an estimate of WCET, optionally adding margins to account for residual uncertainty. The key strengths of this approach are its ability to leverage real hardware behavior and to capture effects that are difficult to model analytically, such as proprietary harware behavior (Intel Speed-step), speculative execution side effects, or undocumented microarchitectural behaviors. However, pure measurement lacks formal completeness guarantees unless one can argue that the profiling covers all timing-relevant paths and hardware states, which is rarely feasible on complex processors with non-trivial programs.

Lastly, an alternative to deterministic WCET estimation is *probabilistic worst-case execution time* (pWCET) analysis. Rather than seeking a single absolute upper bound, pWCET methods aim to characterize the distribution of execution times and derive probabilistic bounds with associated exceedance probabilities. That is, instead of asserting that a job will never exceed time C_i, these methods might guarantee that a job will exceed C_i with probability less than 10^{-9}. This probabilistic framing is particularly attractive for complex hardware where precise deterministic modeling is infeasible due to deep speculation, unpredictable cache replacement, or lack of full microarchitectural disclosure. By embracing statistical variation and hardware-induced randomness, pWCET techniques can often produce tighter bounds while retaining high confidence. Analysis approaches include both static probabilistic methods, which use abstract models and probabilistic semantics, and measurement-based probabilistic techniques that apply extreme value theory (EVT) to the tail of the observed execution-time distribution. A comprehensive survey of these methods, including their theoretical foundations, tool implementations, and challenges for certification, is provided by Davis and Cucu-Grosjean [53].

2.4.4 Relevance to This Dissertation

This dissertation adopts a measurement and profiling methodology for execution-time estimation on our multicore platforms. We profile how execution time responds to changes in shared-cache capacity, memory bandwidth, and core assignment, and we use these measurements to guide dynamic allocation decisions. When relevant, our analyses and mechanisms are explicitly *overhead-aware*: they treat interrupt costs, scheduling and migration overheads, and enforcement costs as first-class terms in the budget and schedulability accounting. The objective is predictable, robust performance via resource control and co-design with schedulers, not certification-grade static WCET bounds on opaque microarchitectures. We found this to be suitable for our goal of exploring diverse dynamic resource allocation strategies.

2.5 Multimodal Systems

Real-time systems often operate under changing environmental conditions, operational goals, or application phases. To accommodate these variations, designers partition system behavior into a finite set of *modes*, each representing a distinct operational configuration. Such systems are referred to as *multi-mode systems*. In each mode m, the system executes a mode-specific set of real-time tasks with corresponding timing and resource requirements.

Formally, a multi-mode system is modeled as a tuple $(\mathscr{M}, \mathscr{T}, \mathscr{G})$, where:

- $\mathscr{M} = \{m_0, m_1, \ldots, m_{|\mathscr{M}|-1}\}$ is the set of system modes.
- $\mathscr{T}$ is the complete set of tasks that may be active in some mode.
- $\mathscr{G} \subseteq \mathscr{M} \times \mathscr{M}$ is a directed mode-transition graph.

Each mode $m \in \mathscr{M}$ is associated with a taskset $\tau^m \subseteq \mathscr{T}$, where each task $\tau_i^m \in \tau^m$ is parameterized by its worst-case execution time C_i^m, period T_i^m, and relative deadline D_i^m.

Mode transitions. During execution, the system may change from a source mode m' to a target mode m, triggering a *mode transition*. Transitions may be initiated by external events (e.g., faults, mission phases) or by internal timing logic, and may occur at predictable or irregular intervals. Transitions may introduce *carry-over jobs*—tasks that were active in m' but remain incomplete at the moment of the switch—and *new jobs*, released under m. To preserve temporal correctness, the system must remain schedulable both:

1. within each individual mode (i.e., m and m'),
2. during the transition interval (m', m), where carry-over and new jobs may coexist.

That is, all jobs must meet their deadlines, even if released across mode boundaries.

Mode-change protocols. The scheduling semantics during mode transitions are governed by the system's *mode-change protocol*. Several classes of protocols exist:

- **Synchronous protocols** delay the release of new-mode tasks until all old-mode tasks have completed. This avoids temporal overlap, but may lead to underutilization.
- **Asynchronous protocols** allow new-mode tasks to be released immediately, coexisting with carry-over jobs. These protocols offer higher utilization but require careful analysis to ensure schedulability during transitions.

Challenges. Multi-mode systems introduce two key challenges:

1. The task set, resource requirements, and timing characteristics may differ across modes, complicating static analysis.
2. Mode transitions introduce interference between new and old jobs, increasing the potential for deadline misses.

Later Chapters 5 and 6 cover multi-mode systems in greater detail.

2.6 Resource Allocation in Real-Time Systems

As highlighted in Chapter 1, on multicore platforms schedulability depends not only on the order of execution but also on how shared resources are divided. Unmanaged interference inflates WCETs and undermines assumptions in schedulability analysis. Resource allocation is therefore not a luxury but a necessity in many real-time deployments.

2.6.1 Allocation Mechanisms

Modern processors and operating systems expose a range of mechanisms for allocating shared resources such as cache capacity, memory bandwidth, and processor time. These mechanisms exist both in hardware and software, and they form the foundation on which real-time resource management strategies can be implemented.

Cache partitioning. Hardware support for cache partitioning is available on several commercial platforms. For example, Intel's Cache Allocation Technology (CAT) allows software to assign cores to disjoint subsets of cache ways via Model Specific Registers (MSRs), limiting cross-core eviction interference [94]. ARM's MPAM specification provides similar cache and memory partitioning interfaces [91].

In the absence of hardware support, *page coloring* can be used to achieve software-enforced cache isolation [227]. This technique leverages the fact that physical memory addresses determine which cache sets they map to. By controlling the allocation of physical pages, a page-coloring-aware memory allocator can ensure that different tasks use disjoint subsets of the cache. This provides a form of cache partitioning without requiring hardware support. However, updating the cache allocation for a running task typically requires migrating its memory to pages with a new color, which incurs expensive memory copying overhead. Because of this, page coloring is most effective for static isolation and is not well-suited for dynamic reallocation. For that reason, this dissertation focuses on hardware-assisted cache partitioning mechanisms to support dynamic resource allocation at runtime.

Memory bandwidth throttling. Inter-core interference at the memory controller arises when multiple cores simultaneously generate high memory traffic, leading to contention for DRAM channels and queues. As Chapter 1 introduced, this interference can be broadly classified into two categories: *spatial* and *temporal*.

Spatial interference arises when multiple cores access memory regions that fall into the same DRAM bank, leading to eviction and increased access latencies, similar to shared last level cache eviction. This can be mitigated through page allocation strategies that ensure threads are mapped to distinct banks [132, 102]. Similar to page-coloring, these mechanisms are not the focus of this dissertation due to high memory copying costs for dynamic reallocation.

Instead, this dissertation addresses *temporal* interference, which occurs when the aggregate request rate exceeds the capacity of the memory controller or DRAM bus. Temporal interference is particularly relevant for memory-bound applications, as even well-separated address streams can cause queuing delays and bandwidth contention.

A widely used software mechanism for controlling temporal interference is *MemGuard* [223], which enforces per-core memory bandwidth reservations on COTS hardware. MemGuard uses performance monitoring counters to track last-level cache (LLC) misses, a proxy for memory accesses, and triggers throttling interrupts when a core exceeds its allocated budget. Enforcement occurs at fixed intervals and prevents memory-hungry tasks from overwhelming the bus and degrading other tasks' performance.

Because MemGuard relies only on performance counters and interrupt injection, it is portable and works on most commodity systems. However, it partitions bandwidth statically unless reconfigured manually or by a higher-level runtime system. This dissertation builds on MemGuard as the enforcement mechanism for dynamic memory bandwidth allocation policies developed in later chapters.

CPU affinity and scheduling domains. Processor time itself can be partitioned via CPU affinity masks, cpusets, and domain-based scheduling mechanisms in Linux and other RTOSes. These controls restrict which cores a given task can execute on, helping to localize interference and reduce migration overheads. When combined with cache and memory bandwidth partitioning, affinity settings enable end-to-end isolation between real-time and best-effort workloads.

Isolation tradeoffs. While these mechanisms enable strong isolation, they also reduce flexibility. Partitioning may leave resources underutilized when demand fluctuates, and enforcing hard limits can increase deadline misses under transient overloads. This motivates dynamic allocation approaches—explored in the rest of this dissertation—that adapt resource assignments at runtime based on current demand, mode, and scheduling constraints.

2.7 Real-Time Operating Systems

Real-time operating systems (RTOS) provide the software layer that implements scheduling, isolation, and timing services atop modern processors. They must balance predictable execution with the practical realities of devices, interrupts, and multiprogramming. This section contrasts purpose-built RTOSes with general-purpose operating systems (GPOS) extended for real-time use, and explains how interrupts, timers, and kernel overheads shape predictability in practice.

2.7.1 Purpose-Built RTOS and GPOS with RT Extensions

Purpose-designed RTOSes such as VxWorks, FreeRTOS, and QNX are built for predictability and analyzability. They offer minimal operating abstractions, bounded system calls, and tightly controlled interrupt paths to support certification and real-time guarantees. To reduce timing variability, such systems may forego complex features like virtual memory, dynamic scheduling classes, or full file systems. For example, FreeRTOS—one of the most popular open-source embedded RTOS kernels—does not support virtual memory, treating execution units simply as tasks rather than processes. Similarly, it contains no built-in filesystem, security model or seperation between user and kernel space, embodying a highly stripped-down design intended for resource-constrained environments [70].

General-purpose operating systems such as Linux, Windows, and BSD kernels were not designed for strict predictability, but have been extended with real-time capabilities. Linux, in particular, supports real-time scheduling policies such as SCHED_FIFO and SCHED_DEADLINE, for fixed priority and earliest deadline first respectively, and with the PREEMPT-RT patch set can provide a nearly fully preemptible kernels. Research extensions such as LITMUS$^{\text{RT}}$ add additional scheduling plugins and low latency logging facilities, making Linux a popular experimental platform. GPOS retain broad device driver support and large scale widespread adoption, but this comes at the cost of more complex kernels and higher baseline jitter compared to purpose-built RTOSes.

2.7.2 Interrupts and Timers in RTOS Kernels

Interrupts and timers are central to how real-time kernels operate, but they are also major sources of overhead and variability. An interrupt is an asynchronous event that diverts control to privileged code to service devices or internal timers. On modern systems, interrupts are delivered via a controller hierarchy (e.g., APICs), dispatched by vector, and handled by an interrupt service routine (ISR), a small configurable block of code. Many kernels split ISR work into two parts: a short *top half* handler that executes in *hard interrupt context*, and a deferred *bottom half* handler (softirq, tasklet, or kernel thread) that executes later in *processor context*. This design bounds non-preemptible time while deferring longer work to lower-priority contexts.

The *hard interrupt context* is a non-preemptible execution mode in which normal task scheduling is suspended and only other high-priority interrupts can be handled. Code executing in this context cannot block, sleep, or perform operations that depend on the scheduler, and it runs with most kernel preemption mechanisms disabled. In contrast, *processor context* refers to normal kernel thread execution, where code runs with preemption enabled and may be scheduled like a regular task. Deferred interrupt handling (e.g., softirqs or threaded IRQs) runs in processor context, which allows longer or blocking operations to execute without stalling the system.

Linux also supports *threaded interrupts*, in which most ISR work executes in a dedicated kernel thread under a real-time scheduling policy. This allows interrupt activity to be preemptable and even pinned to specific CPUs, reducing interference with real-time workloads on other cores, one of the major benefits of running Linux with its PREEMPT-RT patch.

Timers provide the mechanism for releasing periodic and sporadic jobs. High-resolution timers (hrtimers) can generate interrupts at fine granularity, allowing precise job releases. However, timers themselves incur overhead: programming the timer, handling the interrupt, executing the callback, and waking the target thread all consume cycles and may be delayed by higher-priority interrupts. In addition, Linux and other kernels may coalesce timers to reduce wakeups, or operate in tickless modes that trade a fixed periodic tick for more complex idle and wakeup handling. These factors contribute to *release jitter*, where jobs are released later than their programmed deadlines. Many of these issues can be avoided with careful compile-time kernel configuration.

2.7.3 Implications for This Dissertation

The later chapters of this dissertation rely on Linux with real-time extensions as the experimental platform. Interrupts and timers are treated as realistic sources of overhead and jitter that must be explicitly included in analysis and evaluation. We configure our real-time tasks with cpu affinity to minimize involuntary interference on cores reserved for real-time workloads, and we incorporate interrupt and timer costs directly into job budgets and schedulability conditions. By doing so, we ensure that the improvements attributed to our resource allocation mechanisms are measured under realistic kernel activity, not in an idealized environment free of asynchronous events.

CHAPTER 3

STATE OF THE ART

This chapter surveys the state of the art most relevant to this dissertation's goal: *resource-aware execution for multicore real-time systems* with resource allocation, multi-mode operation, dependency-aware workloads, and control-informed co-design. We omit general scheduling theory, which is covered in Chapter 2, as well as resource allocation for non–real-time systems (data centers, high-performance computing), since these fall outside this dissertation's scope. Chapter-specific related work appears in the corresponding chapters. Our focus is on general, modern mechanisms and analyses for cache and memory bandwidth isolation, and on algorithms that allocate these resources—statically or dynamically—under the feasibility checks used in Chapters 4–7.

3.1 Static Resource Allocation

Static cache partitioning reduces contention interference by assigning sets or ways to cores or tasks. Mechanistically, way-based methods exploit Intel RDT/CAT and Arm MPAM primitives, while set-based methods rely on page coloring and memory placement [94, 95, 211, 11, 118]. Beyond capacity partitioning, LLC *throughput* can also be regulated to mitigate interference on shared banks. Per-bank LLC bandwidth control limits traffic to individual banks (rather than throttling the whole LLC), reducing bank-focused denial-of-service and unnecessary throttling on uncongested banks [189]. This control complements way/set partitioning (CAT/page coloring), which governs cache *capacity* but not per-bank service rates. Industrial interfaces (`resctrl`, hypervisors like Jailhouse/Xen) have matured to expose these controls in deployable form on x86 and arm64 [118, 11].

Static memory bandwidth isolation enforces per-core caps to curb DRAM bus contention and queuing effects, with MemGuard as a representative mechanism in commodity systems, using performance monitoring counters (PMCs) for enforcement [224]. Beyond PMC-based enforcement, finer-grained regulators exist but generally require specialized hardware rather than just commodity CPUs. *MemPol* achieves microsecond-scale policing from outside the application cores, requiring an auxiliary policing context or dedicated monitoring hardware [235]. The coherence-aided regulator by Izhbirdeev et al. leverages coherence-interface visibility on tightly integrated SoC/FPGA platforms to infer and limit memory activity with lower overhead than PMC polling [97]. These mechanisms illustrate the upper bound of what can be enforced when the hardware exposes precise hooks, but their portability is limited compared to PMU-only approaches.

From a policy perspective, combined cache+bandwidth schemes recognize that C_τ depends on both *capacity* (LLC occupancy) and *service rate* (memory bandwidth). Static co-allocation approaches tackle *both* resource partitioning and task-to-core assignment under partitioned scheduling, producing per-core budgets and mappings that satisfy feasibility constraints. Some formulations cast the problem as an optimization and solve it with integer/linear solvers to maximize schedulability or minimize latency given cache and bandwidth budgets [212, 192]. Others develop heuristic algorithms that iteratively refine task partitions and resource splits—e.g., greedily reallocating ways and bandwidth partitions or rebalancing cores—to approach the same objectives with lower runtime [192]. Static approaches are attractive for certification and for workloads with stable phases, but they suffer from fragmentation and leave performance on the table when demand varies across applications or over time.

Lastly, there is substantial recent work that manages one, but not both, shared resources—either last-level cache or memory bandwidth—by computing static per-core budgets from policy objectives and feasibility constraints [42, 105, 104, 194, 5, 128, 209, 217, 190]. Cache-only policies include static coloring and isolation trade-offs for mixed-criticality workloads [42, 194], as well as task/core partitioning that is explicitly cache-interference-aware [209]. On the memory side, static bandwidth budgeting and admission/use-cap tests have been proposed for COTS platforms [5], and earlier execution-model and co-scheduling work accounts for memory activity alongside CPU scheduling constraints [144, 128]. NUMA-aware memory placement via memory coloring has also been explored to reduce cross-socket interference [141]. While these single-resource policies improve predictability and are attractive for certification, they can leave performance untapped when interference arises from the combination of cache capacity and memory-service limits, motivating the joint cache+bandwidth co-allocation approaches introduced later in this dissertation.

3.2 Dynamic Resource Allocation

Beyond static budgets, a small but growing number of recent solutions propose *dynamic* policies that adjust resource allocations, for one resource type, over time to track contention while preserving timing. E-WarP [182] offers a system-wide memory bandwidth management policy that profiles demand and then *adapts per-core caps at run time* to keep the DRAM subsystem below saturation while protecting real-time tasks. [163] introduces a feedback-control policy that *continuously tunes per-core memory bandwidth caps* to maintain utilization below a configured threshold, improving responsiveness compared to fixed reservations. FARRE [129], dynamically controls shared cache to both improve system performance but also a user-defined fairness objective. For heterogeneous platforms, [4] demonstrates a policy that *dynamically reallocates memory bandwidth between CPU and GPU workloads* based on observed progress of a real-time GPU kernel, tightening guarantees without over-throttling best-effort work. Collectively, these techniques exemplify dynamic *policies*—not just enforcement mechanisms—that use online measurements or offline profiles to revise budgets during execution or according to a schedule, yet comprehensive multi-resource controllers with explicit EDF/FP feasibility guarantees do not yet exist.

3.3 Multi-Mode Resource Allocation

Multi-mode systems can change both the active task set $\mathscr{T}$ and its resource needs across modes. This creates two requirements: each mode must be schedulable in isolation, and mode transitions must also be schedulable as the system moves from one mode to another. In other words, it is not sufficient to find a feasible allocation for each steady state; the system must also avoid overload during the transient period when jobs from two modes overlap.

Recent multi-mode policy work that *decides per-mode resources* is comparatively sparse. Most prior work on multi-mode systems focuses on mode-change *analysis* (e.g., ensuring that a given set of per-mode task parameters yields a safe transition) rather than on computing concrete per-mode resource allocations.

One line of work focuses on cache-only per-mode policies. Kwon *et al.* [107] formulates per-mode cache partitioning as an optimization problem and computes mode-specific LLC allocations that remain feasible both within steady states and across mode changes. At runtime, cache partitions are reconfigured when the system switches modes, so each mode sees an appropriate LLC allocation; however, other shared resources such as memory bandwidth are not explicitly allocated.

A second line of work exposes mode-specific CPU configurations. M2-Xen, for example, allows a VM's resource requirements to change substantially over time by switching between different vCPU and isolation profiles at mode changes [114]. Here, the emphasis is on selecting a different CPU and isolation configuration per mode; shared caches and DRAM bandwidth remain uncontrolled, implicit platform characteristics.

A third line embeds per-mode allocations inside a broader real-time framework. Chisholm *et al.* [43] integrate per-mode LLC and DRAM budgets into the MC2 framework to provide hardware isolation across modes. They solve a mixed-integer linear program (MILP) to select allocations that respect mixed-criticality constraints together with mode-change constraints. This yields stronger per-mode isolation, but is tied to a specific framework and does not consider memory bandwidth.

Very recent joint allocators (e.g., static co-allocation of cache and memory bandwidth under partitioned scheduling) provide strong *per-mode* solvers that can, in principle, be invoked in a multi-mode setting [212, 192]. These works compute high-quality allocations of multiple shared resources (cache and memory bandwidth) for a given task set and scheduling policy. However, they typically do not model mode transitions explicitly. They assume that if each mode is feasible in isolation, then the system is acceptable, without reasoning about the transient load when jobs from the old and new modes coexist during a transition.

As we will see in Chapter 5, this lack of explicit mode-awareness can result in poor schedulability and performance. It is possible for every mode to be schedulable on its own, yet for the system to miss deadlines during transitions because the allocator never budgets for the combined transition load. This motivates a comprehensive multi-resource, multi-mode allocation solution that plans CPU, cache, and memory bandwidth across all modes and mode-transitions.

3.4 Resource Allocation Co-Design Techniques

Co-design, in this context, means *jointly* choosing resource allocations together with other system-design decisions rather than treating the rest of the stack as a fixed black box. Instead of first fixing task periods, core mappings, or controller implementations and only later "fitting" cache or memory-bandwidth budgets, co-design exposes and couples policy control on *both* sides: platform resources (cache partitions, memory-bandwidth caps, GPU shares, network/QoS budgets, DVFS states) and application or scheduling choices (task periods/priorities, core mappings, control variants, mode-switch rules). The aim is to meet *real-time objectives* like deadline satisfaction and bounded response time while improving secondary goals such as latency, throughput, control stability or energy consumption. Concretely, co-design policies decide *what resources to give and when* together with scheduling, controller design, accelerator orchestration, or networking, supported by models that link those resources to timing and quality. This stands in contrast to designs mentioned previously, where each component is tuned in isolation and the resulting plan may be unnecessarily conservative. Here, we focus on co-design techniques with explicit real-time or control objectives.

CPU/memory and accelerator co-design. Profile-driven co-design of task placement and memory-bandwidth budgeting appears in E-WarP [182], which profiles memory demand and then plans saturation-aware co-locations and per-workload bandwidth limits that are applied at runtime. On heterogeneous SoC platforms, Aghilinasab *et al.* [4] co-design shared GPU and CPU memory allocation: the goal is to protect real-time GPU kernels by updating best-effort budgets for CPU tasks online based on the observed progress of the GPU kernel. DART [207] performs co-design of CPU/GPU scheduling via stage partitioning and mapping, yielding bounded response times for real-time deep neural networks while maximizing throughput for other best-effort tasks. These approaches demonstrate that jointly choosing task placement, throttling, and accelerator use can improve utilization compared to tuning each component independently. However, they typically target a specific smaller combination of resources (e.g., CPU+memory bandwidth or CPU+GPU) on a fixed platform or do not consider real-time CPU tasks.

Control-aware co-design. In control-aware co-design, Sudvarg *et al.* [187] integrate controller selection (periods/variants) with schedulability, solving a constrained optimization to maintain safety while meeting timing guarantees. However, this work does not consider cache or memory-bandwidth allocation and instead assumes a given underlying platform. Zhu *et al.* [233] describe a broader view of cyber-physical co-design, emphasizing unified modeling, synthesis, and verification workflows that are coupled with scheduling decisions. The emphasis here is on jointly reasoning about control quality and scheduler behavior, but again resource allocation is largely implicit.

Networked and wireless control co-design. Within networked control, Phan *et al.* [183] provide overrun-aware co-design to select platform arbitration and control strategies. They derive bounds on skip/abort rates and provide a numerical algorithm that tunes platform and controllers to maintain performance. Wireless control–communication co-design [167] allocates radio resources adaptively based on application state and requirements, demonstrating that application-aware, runtime resource management can reduce provisioning while preserving control performance. For time-sensitive networks (TSN), Xiaotian *et al.* [49] integrate controller synthesis with TSN timing analysis, co-selecting controller parameters and packet priorities/schedules to guarantee closed-loop stability and network schedulability. Taken together, these methods exemplify policies that decide what subset of network or radio resources to give and when *in concert with* scheduling or control design.

Gaps and relevance to this dissertation. Existing co-design techniques show that coupling resource decisions with scheduling, control, or accelerator orchestration can yield better performance and robustness than tuning each layer in isolation. However, most of the above work is *domain-specific* (e.g., GPU scheduling, wireless control, TSN) and focuses on a limited set of resources at a time. Many approaches rely on profiling and numerical optimization to choose parameters, but do not provide general multi-core schedulability guarantees for arbitrary task sets, and they often treat shared resources such as caches or memory bandwidth as fixed platform characteristics rather than as first-class, allocatable quantities. In contrast, the frameworks presented later in this dissertation apply co-design ideas to multi-core real-time resource allocation on commodity platforms: they jointly plan CPU, cache, and memory-bandwidth budgets with scheduling and application-level structure (phases, modes, control models, and DAGs), and they provide analytic guarantees on deadline satisfaction and stability for the resulting configurations.

3.5 Task-Graph / DAG–Aware Resource Allocation

Many application domains exhibit *task dependencies* in which the output of one computation feeds the input of another. These precedence constraints can be modeled as a *directed acyclic graph (DAG)*, where vertices represent tasks or stages and directed edges capture data or control dependencies. There is a substantial real-time literature on scheduling and analysis of task graphs, [200] offers a strong survey. Here we focus on recent work most relevant to *resource-allocation* policies for DAGs.

Under federated scheduling[2], holistic *static* co-allocation assigns multi-threaded tasks core reservations together with cache and memory bandwidth budgets, yielding a concrete resource plan for heavy parallel graphs [137]. Within a single mode, *holistic DAG budgeting* policies redistribute CPU-time execution budgets across a processing graph to minimize graph-level stalls/aborts, explicitly deciding how much budget each node receives to improve end-to-end behavior [197]. Run-time cache-aware co-location policies place DAG threads to increase cache reuse and reduce interference, effectively allocating cache capacity and memory-service opportunity by controlling which nodes share hardware at a time [196]. Orthogonally, policies that tune the *degree of parallelism* for parallel real-time tasks guide how much compute to allocate across ready nodes to balance critical-path progress against resource contention [87]. Lastly, energy-aware DAG policies co-decide execution and platform power states, allocating CPU frequency or power budgets to nodes to minimize energy while preserving real-time guarantees [22, 164]. Together, these solutions move beyond fixed-cost assumptions by shaping placement, reservations, and budgets for DAG nodes; however, comprehensive frameworks that *jointly* allocate shared resources (e.g., cache and memory bandwidth) across the DAG with end-to-end real-time guarantees remain limited.

3.6 Summary

Across the four themes above, several patterns emerge. First, enforcement mechanisms are now practical on commodity platforms (CAT/MPAM, `resctrl`, MemGuard), which means resource budgets are highly configurable. Second, policy-level allocation has matured from single-resource, static splits to joint cache and memory bandwidth *static* co-allocation and early *dynamic* controllers, but most results still optimize one

[2]Federated scheduling dedicates an exclusive set of cores to each "heavy" parallel task or DAG, while "light" tasks share the remaining cores under a traditional policy (e.g., FP/EDF); see [200] for a concise overview.

resource at a time or lack ties to formal EDF/FP feasibility. Third, multi-mode policies typically optimize *within* modes and leave transitions implicit or do not closely control resources at all, even though real systems spend time in flux where carry-over work and reconfiguration latency matter. Fourth, DAG work increasingly acknowledges resource sensitivity, but comprehensive models that allocate cache *and* memory bandwidth *across nodes* with end-to-end guarantees are still very rare.

These gaps motivate the technical arc of this dissertation. Chapter 4 makes dynamic allocation practical by modeling resource-sensitive execution and adjusting cache/memory bandwidth online with the goal of preserving per-core schedulability. Chapter 5 extends allocation to multi-mode systems with explicit, certified transition handling so guarantees hold both *within* and *across* modes. Chapter 6 couples allocation with control objective *co-design*, showing how resource choices interact with safety/stability and how to co-tune them. Chapter 7 generalizes allocation to DAGs, assigning resources at node/stage granularity and delivering end-to-end guarantees for dependency-rich pipelines. Together, these contributions provide an integrated framework that treats execution cost as *resource dependent*, co-allocates CPU/cache/memory bandwidth under formal tests, scales across modes, and supports dependency-aware workloads.

CHAPTER 4

DNA: DYNAMIC RESOURCE ALLOCATION FOR SOFT REAL-TIME MULTICORE SYSTEMS

As emphasized in Chapter 1, shared resource allocation is required to ensure that the increasingly complex hardware and software behavior of multicore platforms and real-time tasks remain predictable and safe. Partitioning resources statically simplifies reasoning but leaves efficiency on the table, since fixed budgets cannot adapt when tasks' demands fluctuate.

This naturally raises several questions.

- Q1. How wasteful is static allocation in practice?
- Q2. Can dynamic allocation actually reclaim wasted capacity without undermining predictability?
- Q3. To what degree is it possible to leverage changing cache and bandwidth needs in practice?
- Q4. And what kind of mechanisms are required to make such reallocation feasible at runtime?

This chapter explores these questions in detail. It demonstrates that tasks often exhibit recurring phases with distinct resource sensitivities and that adapting cache and memory bandwidth allocations across these phases yields substantial gains in timeliness. The algorithms presented in this chapter show how profiling and lightweight online adjustment can expose these opportunities while remaining compatible with soft real-time scheduling constraints. In doing so, the chapter establishes a first proof-of-concept that dynamic allocation is not only viable but beneficial, setting the stage for the richer forms of adaptation developed in later chapters.

In Section 4.1, we examine in greater detail the issues to real-time predictability introduced by multicore platforms. Section 4.2 introduces an overview of our model, a motivating case study and deeper relevant background knowledge. In Sections 4.3 - 4.4, we introduce the concept of phases and our DNA framework and show how it leverages these phases for dynamic allocation. Section 4.5, extends this approach with deadline-awareness in DADNA, and in Section 4.7, we evaluate both approaches to quantify their benefits and limitations.

This chapter is based on work that first appeared as: Robert Gifford, Neeraj Gandhi, Linh Thi Xuan Phan, and Andreas Haeberlen. DNA: Dynamic resource allocation for soft real-time multicore systems. In *RTAS*, 2021.

4.1 Introduction

Today, multiple cores are a common feature of both desktop and embedded CPUs, and latency-sensitive and real-time systems are taking advantage of them to support their increasingly complex workloads. This is mostly a good thing, since having multiple cores means better performance. However, this trend also creates some new challenges because the cores are not independent – they share certain hardware resources, including the memory bus and certain cache levels. Thus, tasks on different cores can influence each other's runtime.

Consider, for instance, a system whose workload includes the following two kinds of tasks: (1) a control task, which multiplies a vector with a large matrix and then uses the result to make a complicated control decision, and (2) a stream-processing task, which computes a filter and then applies the filter to a stream of sensor inputs. (Both matrix multiplication and filter are basic functions in many real-time control systems.) Matrix multiplication is memory-intensive; on a single-core CPU, it can be fast if the matrix is already in the cache, or fairly slow if the matrix needs to be fetched from main memory. But on a multicore CPU, it can be *even slower* if other cores happen to generate heavy traffic on the memory bus at the time the matrix needs to be fetched – for instance, if the stream-processing task happens to read the data stream at the exact same time. If this kind of interference happens unexpectedly, it can cause increased latencies and deadline misses.

One way to do better is to statically partition the resources (memory bandwidth and cache capacity) between the cores. For instance, MemGuard [223] regulates the memory bandwidth each core can use by counting memory accesses using hardware performance counters, and by interrupting workloads when a specific limit is reached; similarly, Intel's Cache Allocation Technology (CAT) [95] can restrict cores to a certain subset of the available cache partitions. These techniques can be used to prevent interference and to isolate the cores from each other. In our example, if the matrix-multiplication task and the stream-processing task run on different cores, we can allocate most of the last-level cache to the control task, since matrix multiplication benefits from caching but stream processing does not (and would in fact thrash the working set of the control task), and we can split the memory bandwidth between the two tasks, to make their execution times more predictable. It is well known that different kinds of tasks can extract different benefits from resources (see, e.g., the animalistic taxonomy from [210]), and this approach has been used for static partitioning [212].

However, static partitioning is *still* very conservative and often fails to fully utilize the available resources. The reason is its assumption that neither the set of tasks nor the characteristics of these tasks can change over time. In practice, the set of tasks often does change: for instance, the controller on a car might launch a new task when the driver enables cruise control, and stop it again when the driver switches back to manual mode.

More importantly, the characteristics of the tasks themselves can change as well! To see why, consider again our earlier example. During the matrix multiplication, the control task heavily depends on memory bandwidth, and we can speed it up considerably by allocating more bandwidth to it. However, computing the actual control decision afterwards is much less memory-intensive; during this time, it would be better to allocate most of the bandwidth to the stream-processing task. In other words, tasks can have different *phases* during their execution [178], and these phases can differ in their resource demands. Because of this, a static allocation – perhaps based on the demands of the worst phase, or on the average demand across phases – almost inevitably leads to suboptimal utilization and/or higher response times.

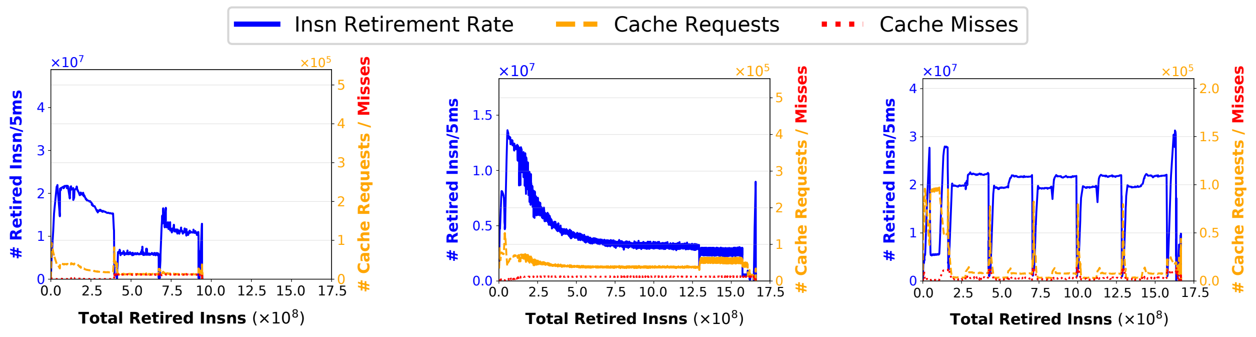
In this chapter, we present a pair of resource allocation techniques, Dynamic Allocation (DNA) and deadline-aware DNA (DADNA), that allocate memory bandwidth and cache capacity while explicitly taking the phases into account. Both techniques consist of two parts. The first is an offline profiler that runs each task with different combinations of resources to build an *execution profile* – essentially a function that maps different points in the task to the *rate of progress* (i.e., instructions retired per unit time) at that point. For instance, the profiler might find that the control task's matrix multiplication runs at 10^9 instructions/s if it can have the entire cache and the entire memory bus, but only at 10^8 instructions/s if it is restricted to 10% of the memory bus, and only at 10^7 instructions/s if it is additionally restricted to 50% of the cache (because of thrashing).

As a next step, the profiler then uses a simple machine-learning technique (clustering) to identify phases with similar behavior. This saves space – since we only need to store the phases and not the entire, detailed execution profile – and, more importantly, it identifies potential decision points at which it may make sense to adjust the resource allocation at runtime. The sequence of phases depends not only on the program but also on the resources the task has been allocated: for instance, memory-intensive phases can disappear when the tasks are given more cache space, or they can move around when cache partitions of different sizes cause different conflict patterns.

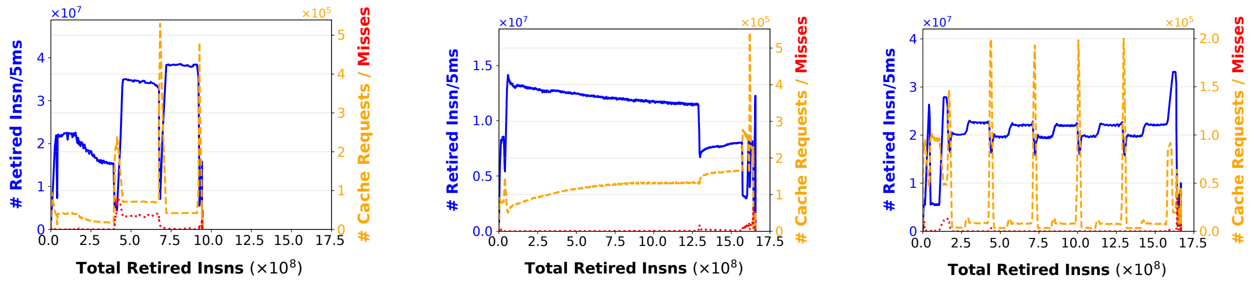
The second part of DNA/DADNA is a resource allocator that uses the collected execution profiles to dynamically reassign resources at runtime. Somewhat analogous to Antfarm [145], which dynamically allocates network bandwidth to BitTorrent swarms, our allocator gives memory bandwidth and cache capacity to the tasks that can benefit the most. Thus, in the above example, the control task would get most of the cache, since it benefits heavily from caching but the stream-processing task does not, and it would initially get most of the memory bandwidth, but only until the matrix has been loaded into the cache; after that, much of the bandwidth would be reallocated to the stream-processing task. The deadline-aware variant (DADNA) is additionally able to allocate resources not only based on the immediate needs of a task but also based on the slack time of current and future deadlines.

We have built a prototype implementation of DNA and DADNA in the Xen hypervisor [17] by modifying Xen's existing Real-Time Deferrable Server (RTDS) scheduler [161], and we report results from an experimental evaluation on real hardware. Our results show that DNA and DADNA incur only small run-time overhead, and that they can substantially improve schedulability, reduce deadline miss ratios, and cut latencies by more than a factor of two compared to a state-of-the-art solution. In summary, this chapter makes the following contributions:

- the concept of phase-aware allocation (Section 4.2);
- a phase-based task model (Section 4.3);
- the DNA resource allocation technique (Section 4.4);
- DADNA, a deadline-aware variant of DNA (Section 4.5);
- a prototype implementation in Xen (Section 4.6); and
- an experimental evaluation (Section 4.7).



(a) `fft` (2 cache, 2 bandwidth). (b) `canneal` (2 cache, 2 bw.). (c) `fluidanimate` (2 c., 2 bw.).



(d) `fft` (18 cache, 18 bandwidth). (e) `canneal` (18 cache, 18 bw.). (f) `fluidanimate` (18 c., 18 bw.).

Figure 4.1: Execution patterns for three benchmark tasks under two different resource allocations: two cache and two bandwidth partitions (top row), and 18 cache and 18 bandwidth partitions (bottom row). Each point on the horizontal axes represents a particular point in a program, identified by the number of instructions since that program was started, and the lines show the behavior of the program at that point: its execution speed (blue) and the rate of cache requests (orange) and cache misses (red).

4.2 Overview

We assume that the system consists of a set of tasks that execute on a shared multicore platform, with a shared cache and a shared memory bus that are accessible by all cores. As in existing work, the set of tasks that can *potentially* run on the system is known before the system is launched. This is necessary because our approach involves some offline profiling to identify the phases and their resource requirements (Section 4.3). However, the set of tasks that are *actually* running on the system can change over time (e.g., in multi-mode systems [39]). We assume that the tasks are short and typically perform similar operations on similar inputs, which is true in most real-time systems where each task periodically executes a control function on streams of sensor inputs. This assumption allows us to use a relatively simple method to identify the phases (Section 4.3) but is not fundamental; if the tasks are more complex, there are other ways to find phases (e.g., [175]) that can be used instead.

We focus on latency-sensitive and soft real-time systems (though it should be possible to extend our solution to hard real-time systems). For soft real-time systems, we assume that deadline misses are acceptable, and jobs that miss their deadlines are allowed to continue their executions until completion. Our goals are 1) to reduce average, tail, and worst-case latencies, and 2) to minimize job deadline miss ratios when tasks have (soft) deadlines.

4.2.1 Background: CAT and MemGuard

Intel's Cache Allocation Technology [95] is a hardware feature that is present in newer Intel CPUs; it allows the hypervisor to control how the shared last-level cache (LLC) is allocated between the physical cores. CAT divides the shared cache into N equal-size partitions (e.g., $N = 20$ on our evaluation machine), which can be allocated to one of several classes of service (COS). For each COS, there is a model-specific register with a bitmask – the capacity bitmask, or CBM – that controls which partition(s) should be used; for each logical core, there is another register (PQR) that specifies the COS for this core. CAT enforces the property that all new cache allocations from a logical core are made only in cache partitions specified in the CBM of that core's COS. For instance, if we set bits 0..3 in the bitmask for COS 1, and then set the PQR register for core #5 to 1, new cache allocations from core #5 will be made in one of the first four cache partitions.

MemGuard [223] is a software-based mechanism for enforcing per-core limits on memory bandwidth consumption. It uses the CPU's performance counters to count the LLC misses on each core; since memory bandwidth is consumed in response to LLC misses, this number is a good proxy for memory bandwidth consumption. Each core and/or each task can be assigned a memory bandwidth budget, and MemGuard periodically configures each core's counters so that they will generate an interrupt if the core's LLC misses exceed the budget for that period; if this interrupt is received, it throttles the core, or switches to a different task. In this chapter, we use MemGuard to assign memory bandwidth in small, discrete partitions, just like CAT. (We could enforce limits at the granularity of a single LLC miss, but small differences in the bandwidth usually do not cause big differences in performance, and the small number of partitions helps us keep the number of configurations small.)

4.2.2 Case Study: Setup

Since our approach is based on the fact that tasks often go through different phases with different resource requirements, we ran an experiment to illustrate this, and to show which phases exist in a typical workload. Specifically, the workloads we examined were the PARSEC [23] and SPLASH2x [206, 185] benchmark suites, which have often been used as workloads by prior work in this area [103, 101, 214, 199, 212, 213].

To get a platform where we could vary both the cache and memory bandwidth allocations, we used a Xen modification from our earlier work [212] that already supports MemGuard. We extended it to additionally support Intel's CAT. We ran the modified Xen on a CAT-capable Intel Xeon E5-2683 v4 processor with 16 cores and a 40MB 20-way set-associative L3 cache that is shared among the cores. (Each core has its own L1 and L2 caches.) This processor has 16 COS registers and supports 20 L3 cache partitions. The machine also had three single-channel 16GB PC-2400 DDR4 DRAMs. Using the method in [224], we measured a maximum guaranteed bandwidth of 1.4 GB/s, which we divided into 20 partitions of 70MB/s each. This is lower than the peak bandwidth that the platform supports, but it results in better isolation between the cores. To avoid nondeterministic timing, we disabled hyperthreading, SpeedStep, and hardware prefetching.

To collect data for a given benchmark task and a given cache/bandwidth allocation, we booted the Xen hypervisor with its built-in RTDS real-time scheduler [161] and then launched one guest VM running LITMUS^{RT} [34] (a real-time OS) with two VCPUs that were each pinned to a dedicated core; one to run essential guest OS tasks and the other to run our benchmark in isolation. We then ran the benchmark task on this VCPU, synchronized its release with the VCPU's release (as in [212]), and we measured three performance metrics, using the CPU's performance counters: (i) the total number of cache requests, including both hits and misses; (ii) the number of cache misses, as an indication of the traffic on the memory bus; and (iii) the number of retired instructions. For each configuration of cache and memory bandwidth resources, we took a set of measurements every Δ milliseconds and collected the sets of measurements for 100 runs. (We used $\Delta = 5$ms, as it was small enough to capture fine-grained changes in resource use patterns for our workloads, without creating too much noise.)

4.2.3 Case Study: Results

Figure 4.1 shows our results for three representative benchmark tasks – the `fft` program from SPLASH2x, which performs signal processing, and the `canneal` and `fluidanimate` programs from PARSEC, which perform simulated annealing and fluid dynamics for animation purposes, respectively – as well as for two different resource allocations: one in which resources are scarce and the task is given only two (10%) of the 20 available cache and memory bandwidth partitions (top row), and another in which resources are plentiful and the task is given 18 (90%) of the 20 partitions (bottom row). Each graph shows three curves: the blue one shows the number of instructions retired in the preceding 5ms window, the orange one is the number of cache requests, and the red one is the number of cache misses, which corresponds to memory traffic. The horizontal axes identify particular points in each program: for instance, the `fft` program executes about $9 \cdot 10^8$ instructions, so the lines in both Figures 4.1(a) and (d) end at that point; the speed (blue line) is roughly comparable during the first $4 \cdot 10^8$ instructions, but the rest of the program runs much faster when more cache and bandwidth partitions are available.

A look at the top row of Figure 4.1 shows evidence that different execution phases do exist: for instance, `fft` has three clearly separated phases, of which the first shows quick progress and almost no cache misses (due to high locality), while the other two show slower progress (but at different rates!) and very high miss rates. `fluidanimate` shows a cyclic behavior with three alternating phases that vary substantially in the number of cache accesses, and `canneal` has one long phase, followed by a much shorter one. This leads to our first finding:

Finding 1. *A task's resource usage patterns vary throughout its execution and can be broadly divided into phases, where each phase exhibits a distinct cache and memory bandwidth resource demands.*

Our next observation is that, while each of the three tasks does exhibit different phases, both the number and the characteristics of the phases are quite different: in `fft`, there are three large phases, with rates of execution changing little within a phase but substantially between phases; `fluidanimate` has a sequence of shorter phases, including some brief spikes; `canneal` has some variation within its long first phase; etc. This leads to our second finding:

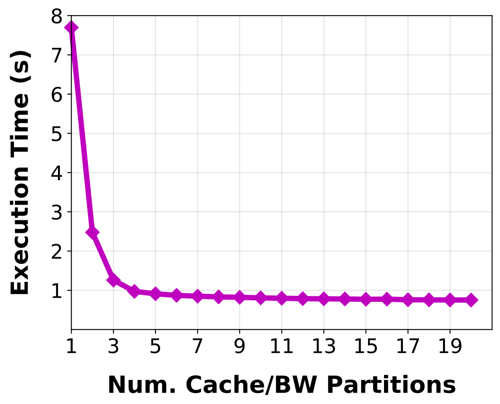


Figure 4.2: Runtime for `canneal` for different cache and bandwidth allocations.

Finding 2. *The number of phases and resource demand patterns in each phase are different across different programs.*

A third observation is that spikes in the orange and red curves (cache and memory bandwidth) generally correspond to dips in the blue curve (progress). This should not be surprising: the more often a task needs to access the cache and/or main memory in a given phase, the slower it will be. Conversely, we can *generally* expect to speed up a task by giving it more space in the cache or more memory bandwidth. (There are exceptions – e.g., when a task's working set already fits into the allocated cache space, as well as some unusual cases we discuss below.) We summarize this in our third finding:

Finding 3. *The execution rate in each phase is closely related to the resource demands in that phase: more cache misses or cache requests tend to lower the execution rate, and fewer requests tend to increase it.*

So far, we have focused mostly on the top row of Figure 4.1. We now compare the top row (scarce resources) to the bottom row (plentiful resources); notice that the horizontal axis is the number of instructions retired, so the same horizontal point in the two rows corresponds to the same point in the execution. Notice how *some* phases change their characteristics substantially (e.g., the third phase of `fft`, which now has no more cache misses and runs much faster), while others change little, if at all (e.g., the phases of `fluidanimate`, which have fewer cache misses but run at pretty much the same speed). This is expected: once the resource demands of a phase are satisfied, increasing the allocation further should not have much of an impact on the rate of execution.

To reinforce this point, Figure 4.2 shows how the overall execution time of `canneal` changes as we vary the number of cache and memory-bandwidth partitions it can use. (For simplicity, we use the same number of partitions for both.) Once the number of partitions reaches a certain threshold (around 5), further allocations do not change the execution time very much. This leads to our final observation:

Finding 4. *Allocating extra cache and memory bandwidth resources to a task can help improve its execution time, but only up to a certain number of partitions.*

In this section, we have focused on the typical behavior we have seen in our experiments. However, we have also noticed a number of atypical events. For instance, new phases can appear, or existing ones disappear, as the resource allocation is changed – e.g., because a larger cache allocation can, in combination with certain cache replacement strategies, cause a form of thrashing. Also, while Figure 4.2 shows "smooth" changes in runtime as the allocations are changed, this behavior is not universal; sometimes there is a threshold effect where thrashing persists until a certain cache allocation is reached, but then disappears abruptly. This will become important in Section 4.4.

4.3 DNA: Phase Generation

Next, we describe how, based on the findings from the previous section, we build a model that captures the phases of a given task, along with their resource requirements.

4.3.1 What is a Phase?

Since the resource allocation algorithm will need to make decisions based on a task's current phase, we need a way to quickly tell, from the "outside", which phase a given task is currently in. The instruction pointer is not a good option for this – partly because of loops, but also because the same function can be invoked from different contexts. For instance, a `matrixMultiply` function could be compute-bound when invoked with a small matrix, and memory-bound when invoked with a large one.

Because of this, we use the *number of retired instructions* to estimate what part of the program is executing, and we define a phase to be a range of instructions – for instance, a phase could last from the 10,000th instruction to the 15,000th one. The number of retired instructions can be easily measured using the performance counter on Intel CPUs, and many other CPUs have a similar counter.

This definition is not precise: for instance, the count could be widely off if a task does busy waiting, if it is invoked with inputs of different sizes, or if the control flow varies widely depending on the inputs. However, for real-time systems, this approach is plausible because they often involve periodic tasks that perform the same operations again and again, on similar inputs. (For instance, the operations might be reading data from a particular sensor, filtering data, or making a control decision.) Small variations in the control flow are not problematic because we do not need to change the resource allocation precisely at a particular instruction, but rather when the task is entering a certain (long) phase, so all we really need is an approximate point.

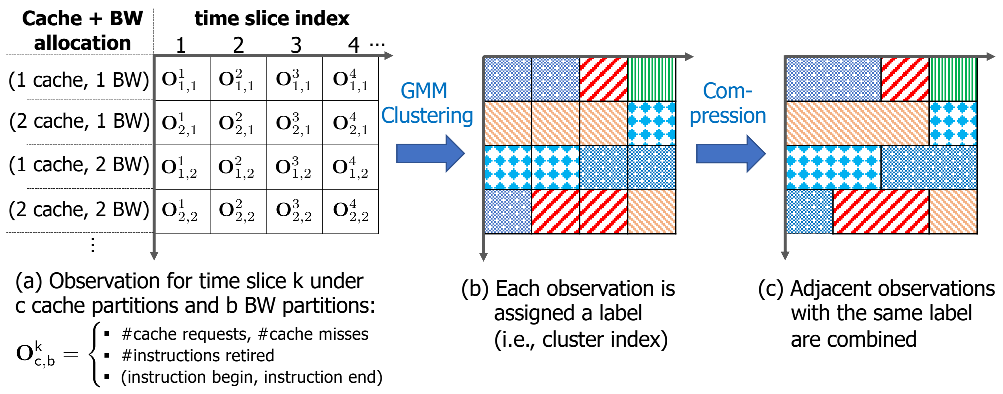
For more complex workloads, this simple approach would not work, but there are other, more sophisticated techniques in the literature (e.g., basic-block vectors [175]) that could be used instead. (Precisely how the phases are delimited is somewhat orthogonal to our work; our main focus is resource allocation.)

4.3.2 Step #1: Profiling

At a high level, the DNA algorithm aims to allocate resources to the task(s) that will "benefit the most" from them – that is, the tasks whose rate of execution will increase the most. To do this, it needs to know, for a given task and a given instruction count within that task, what the rate of execution would be if the task were allocated a certain set of resources. We obtain this information through profiling.

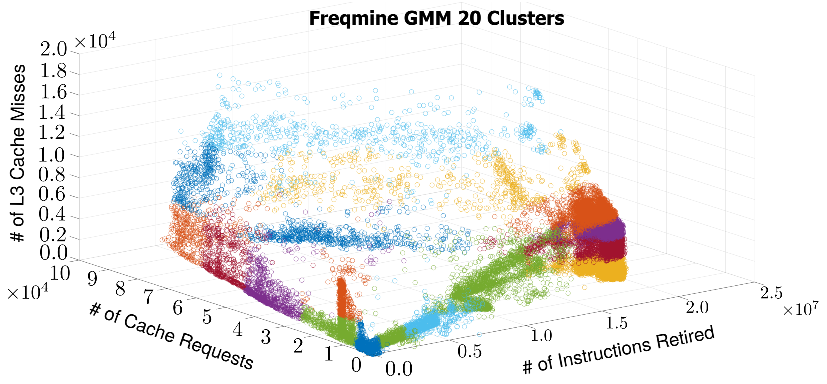
The profiling process is the same as the one from our case study (Section 4.2.2). The first step is to set up a carefully controlled environment in which 1) tasks can be run without interfering with other tasks, and in which 2) we can control the resources that each task has access to. In our experiments, we profile the tasks one by one, on a dedicated CPU core that no other task may access, and we disable all other workloads and all nonessential OS features, to prevent resource consumption by other, unrelated tasks; we also use CAT and MemGuard to control the number of cache partitions and the memory bandwidth the task has available to it.

Next, we run each task in this environment for $N = 100$ runs per resource allocation configuration (there are 400 configurations in total). In each run, for every $\Delta = 5$ milliseconds, we collect (1) the number of



Cache + BW allocation
time slice index
1
2
3
4 ⋯
(1 cache, 1 BW)
(2 cache, 1 BW)
(1 cache, 2 BW)
(2 cache, 2 BW)
$O^1_{1,1}$ $O^2_{1,1}$ $O^3_{1,1}$ $O^4_{1,1}$
$O^1_{2,1}$ $O^2_{2,1}$ $O^3_{2,1}$ $O^4_{2,1}$
$O^1_{1,2}$ $O^2_{1,2}$ $O^3_{1,2}$ $O^4_{1,2}$
$O^1_{2,2}$ $O^2_{2,2}$ $O^3_{2,2}$ $O^4_{2,2}$
GMM Clustering
Com-pression
(a) Observation for time slice k under c cache partitions and b BW partitions:
$O^k_{c,b}$ =
#cache requests, #cache misses
#instructions retired
(instruction begin, instruction end)
(b) Each observation is assigned a label (i.e., cluster index)
(c) Adjacent observations with the same label are combined

(a) Phase generation for a single task.



(b) Clusters for `freqmine`.

Figure 4.3: Phase generation process (top) and clustering result (bottom).

instructions completed; (2) the number of L3 cache requests (hits + misses); and (3) the number of L3 cache misses. This information can be easily gathered from the hardware performance counters on Intel CPUs, and similar counters are present on many other processors.

Finally, we replace each measurement with the delta over the measurement before it – that is, the number of instructions completed, cache requests made, and cache misses *since the last measurement*. We refer to each set of deltas as an *observation*; intuitively, an observation describes the activity of a task during a given Δ window. Together, the observations from the different runs form a matrix that is illustrated in Figure 4.3a.

4.3.3 Step #2: Clustering

The next step is to automatically identify groups of observations that show similar behavior. This step could be done using a variety of clustering techniques; for our prototype, we used expectation-maximization (EM) [130], in combination with a Gaussian Mixture Model (GMM) [157], on a three-dimensional feature space (since each observation contains three metrics). A GMM model simply contains k Gaussian distributions, and the EM technique discovers a mean and a covariance for each that are a good fit for the specific data set. The process is somewhat similar to the classical k-means clustering: EM produces, for each observation, a posterior probability that the observation belongs to each of the k Gaussians, then it updates the mean and covariance for each Gaussian to the maximum-likelihood values for the associated observations, and it repeats these steps until convergence.

As is usually the case with clustering, the correct value of k is not known a priori, but we can generate clusterings with a range of different values for k ($2 \leq k \leq 30$ in our experiments) and then evaluate their output quality using the Davies-Bouldin index [51] and the Calinski-Harabasz index [35]. (The two metrics differ in how they weigh cluster density and separation.) We use the clustering with the highest quality; if multiple clusterings have the same quality, we use the one with the largest k (i.e., the most phases) among those to provide more fine-grained knowledge of the resource needs of the task.

As an illustration, the bottom side of Figure 4.3b shows the clustering for the `freqmine` task, with different colors representing different clusters.

4.3.4 Step #3: Identifying Phases

The final step is to identify the phases. Recall that we need the phases to serve two purposes: they are a more compact representation of the (very verbose) profiling data, and they identify possible decision points for DNA and DADNA, where it “makes sense” to potentially change the resource allocation.

Finding phases in the original profiling data would be difficult because the observations usually are all different, and it is not clear which differences are significant. However, once we replace each observation with the label of the cluster it belongs to, we typically find long sequences of identical labels. (This is illustrated in

Figure 4.3(b); recall that each row represents a run with a different set of resources.) We can then simply collapse each contiguous sequence into a single phase, keeping track only of 1) the instruction where the phase began, 2) the instruction where the phase ended, and 3) the average rate of execution (instructions executed per unit time) during the phase. The result is illustrated in Figure 4.3(c). This is the information the DNA and DADNA algorithms need.

4.4 DNA: Resource Allocation

In this section, we describe how DNA performs resource allocation. DNA is agnostic to deadlines and merely optimizes for overall system throughput and latencies. A deadline-aware version, DADNA, is presented in the next section.

4.4.1 Invocation and Output

The purpose of DNA is to find an allocation of (cache and memory bandwidth) resources to cores, so as to maximize the total rate of execution for the entire system. DNA itself does not perform scheduling; it is designed to be used in combination with an existing scheduler. At the point DNA is invoked, the scheduler has already picked a task for each core to run, and DNA allocates resources to these running tasks. In our prototype, the scheduler is partitioned Earliest Deadline First (EDF), but other schedulers can be used as well. In principle, a single, more complex algorithm could make both decisions simultaneously; we do not consider this approach here, but it could be an interesting direction for future work.

DNA is deterministic, that is, given the same mapping of tasks to cores and the same parameters about the current phases, it will output the same resource allocation. Thus, it generally makes no sense to invoke it again unless there is a change in one of the two. In other words, DNA should run if either (a) the scheduler has changed the task on at least one core, or (b) one of the running tasks encounters a phase transition for *any* resource allocation.

The last point is a bit subtle; it is related to the observation from Section 4.2.3 that the same task can go through different phases if given different resource allocations. For instance, suppose a task has just finished a (memory-bound) matrix multiplication and now begins a (compute-bound) cryptographic signature, and

Algorithm 1 The DNA algorithm

```
1: function AllocateCachePartitions($\tau$, $\delta c$)
2:     c[$\tau$] = c[$\tau$] + $\delta c$
3:     $rem\_c = rem\_c - \delta c$
4:
5: function AllocateMemoryBandwidth($\tau$, $\delta b$)
6:     b[$\tau$] = b[$\tau$] + $\delta b$
7:     $rem\_b = rem\_b - \delta b$
8:
9: function GiveResourceTo($\tau$)
10:     $c\_gain = \theta(\tau, i(\tau), c[\tau], b[\tau], rem\_c, 0)$
11:     $b\_gain = \theta(\tau, i(\tau), c[\tau], b[\tau], 0, rem\_b)$
12:     if $b\_gain < c\_gain$ then
13:         AllocateCachePartition($\tau$,1)
14:     else
15:         AllocateMemoryBandwidth($\tau$,1)
16:
17: function DNA($\mathscr{T}$, $i$)                    ▷ $\mathscr{T}$: running tasks, $i$: instr. completed
18:     $rem\_c = C$                                        ▷ Max cache partitions
19:     $rem\_b = BW$                                       ▷ Max mem bandwidth partitions
20:
21:     /* Assign initial resources */
22:     for $\tau \in \mathscr{T}$ do
23:         c[$\tau$] = b[$\tau$] = 0
24:         AllocateCachePartitions($\tau$, $min\_c$)
25:         AllocateMemoryBandwidth($\tau$, $min\_b$)
26:
27:     /* Iteratively refine allocations */
28:     while $rem\_c > 0 \,||\, rem\_b > 0$ do
29:         $\tau = \arg\max_{\tau}\{\theta(\tau, i(\tau), c[\tau], b[\tau], rem\_c, rem\_b)\}$
30:         GiveResourcesTo($\tau$)
31:     return $(c, b)$                                     ▷ c,b map cache/bw partitions to tasks
```

consider the scenarios where the task is either given a large number of cache partitions L or a small number S. If the task had L partitions before, it might not have generated any memory traffic during the multiplication, and its overall rate of execution might not change much at the transition point; however, with only S partitions, its rate of execution would have been low before and would be high now. Because of this, DNA should run again at the transition point *even if* the task currently has L partitions and is still in the middle of its current phase – simply because there is another allocation (S) that would exhibit a phase transition, and because it may now make sense to switch to S partitions and allocate the remaining L-S partitions to another task.

DNA outputs a mapping of cache and memory bandwidth partitions to tasks. This mapping can then be enforced by the OS or hypervisor, e.g., with CAT and MemGuard.

4.4.2 Algorithm

In principle, a good resource allocation could be found using a form of multidimensional bin packing. However, this kind of computation is expensive and would generate a high overhead, especially since, as discussed above, DNA needs to run frequently. Because of this, we instead opt for a heuristic that can be evaluated quickly.

Algorithm 1 shows the algorithm for DNA. As a first approximation, DNA is a greedy heuristic: it starts by giving only the minimal allocation to each running task (lines 21–25) and then iteratively assigns an additional cache or bandwidth partition to the running task that can "benefit the most" – in other words, the task whose rate of execution would increase the most on average, relative to the allocation it has so far (lines 27–30). Intuitively, the function $\theta(\tau, i(\tau), c, b, \delta c, \delta b)$ is the "gradient" in the rate of execution of a task τ, after executing i instructions, when adding δc cache allocations and δb bandwidth allocations. θ can be thought of as the *sensitivity* of τ to a change in its resource allocation. θ can be computed from the data that is gathered during profiling.

However, this simple approach would not work very well by itself. The reason is that some tasks benefit very little from extra resources, unless and until they reach an allocation of a certain size (say, enough cache partitions to fit their entire working set) but at that point the benefit could be very large. If DNA made decisions based on only the *local* gradient – that is, the benefit from getting *one* extra cache or memory bandwidth allocation – it might never be able to reach the large benefit, since it would always seem that allocating one extra resource makes little difference.

To avoid this, we use a slightly different definition of θ that takes larger increases into account as well. Let $\rho(\tau, i, c, b)$ be the rate of execution of τ after i instructions, using c cache partitions and b bandwidth partitions. Then we define θ as:

$$\theta(\tau, i, c, b, \delta c, \delta b) := \frac{1}{\delta c \cdot \delta b} \sum_{j=0}^{\delta c} \sum_{k=0}^{\delta b} \rho(\tau, i, c+j, b+k) - \rho(\tau, i, c, b)$$

where δc and δb represent the amount of remaining available resources on the system to be assigned.

In other words, θ represents the *average* increase in the rate of execution when adding *up to* δc cache partitions and *up to* δb bandwidth partitions. This is the function used in line 29 of Algorithm 1. After a resource has been assigned, δc or δb will decrement by 1 and thus the task's sensitivity to more resources will update to be realistic with the amount of resources it can still receive. Notice that, in practice, we do not explicitly record θ; instead, θ can be computed efficiently from the (compact) phase information we derived in Section 4.3.

There is one final complication, which has to do with the fact that Algorithm 1 allocates the resources one by one, rather than in larger increments. Once DNA has picked, in line 29, a task to give additional resources, it must still decide *which* resource (cache partition or memory bandwidth partition) to allocate. We make this decision by comparing, in line 12, the *marginal* improvement each resource provides.

4.5 Deadline-Aware DNA

In this section, we present an extension of DNA, called DADNA, for soft real-time workloads that aims to minimize deadline misses (in addition to improving latencies).

4.5.1 Basic Operation

Once tasks have deadlines, it is no longer enough to just allocate resources to the tasks that "benefit the most", in terms of rate of execution; we sometimes need to allocate extra resources to certain tasks just to enable them to finish before their deadlines. In other words, the optimization function becomes the total rate of execution, subject to the constraint that all tasks should meet their deadlines.

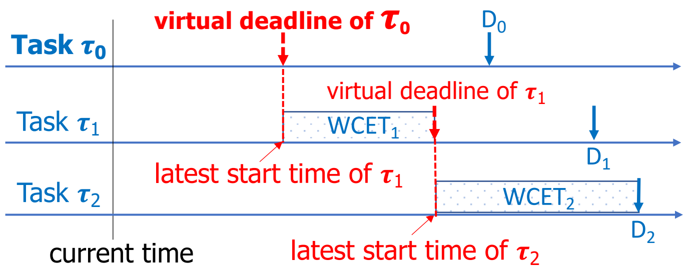
Algorithm 2 shows how DADNA achieves this goal. (Functions that were already defined in Algorithm 1 have been omitted for brevity.) The beginning is similar to DNA: in lines 13–20, we begin again by allocating the minimum resources to each task. However, we now also extrapolate, using a function called TIMELEFT, for how much time each task τ would still need to finish, if given only this minimum allocation, and we compare this time to the slack $S(\tau)$ – that is, the amount of time τ has left before its next deadline. If the task cannot finish in time, it gets added to a set *prio* and will be prioritized during the rest of the algorithm. The only exception, in line 19, is for tasks that cannot finish in time at all, even if given all the available resources.

Algorithm 2 The DADNA algorithm

```
 1: function TIMELEFT($\tau$, $c$, $b$)
 2:     $p = \{x \,|\, P(\tau,c,b).start \le i(\tau) \le P(\tau,c,b).end\}$
 3:     left = $(P(\tau,c,b).end - i(\tau))/P(\tau,c,b).\rho$
 4:     for each $j$ with $p < j \le$ maxPeriod($\tau,c,b$) do
 5:         left += $(P(\tau,c,b).end - P(\tau,c,b).start)/P(\tau,c,b).\rho$
 6:     return left
 7:
 8: function DADNA($\mathscr{T}$, $i$, $S$)                          ▷ $S(\tau)$: Slack of $\tau$
 9:     $rem\_c = C$                                       ▷ Max cache partitions
10:     $rem\_b = BW$                           ▷ Max mem bandwidth partitions
11:     $prio = \emptyset$
12:
13:     /* Assign initial resources */
14:     for $\tau \in \mathscr{T}$ do
15:         c[$\tau$] = b[$\tau$] = 0
16:         ALLOCATECACHEPARTITIONS($\tau$, $min\_c$)
17:         ALLOCATEMEMORYBANDWIDTH($\tau$, $min\_b$)
18:         if TIMELEFT($\tau$,c[$\tau$],b[$\tau$]) > S($\tau$) then
19:             if TIMELEFT($\tau$,$rem\_c$,$rem\_b$) $\le$ S($\tau$) then
20:                 $prio = prio \cup \{\tau\}$
21:
22:     /* Help tasks meet their deadlines */
23:     while ($rem\_c > 0 \,||\, rem\_b > 0$) $\wedge$ ($prio \neq \emptyset$) do
24:         $\tau = \arg\max_\tau\{$TIMELEFT$(\tau, c[\tau], b[\tau]) - S(\tau)\}$
25:         GIVERESOURCESTO($\tau$)
26:         if TIMELEFT($\tau$,c[$\tau$],b[$\tau$]) $\le$ S($\tau$) then
27:             $prio = prio \setminus \{\tau\}$
28:
29:     /* Iteratively refine allocations */
30:     while $rem\_c > 0 \,||\, rem\_b > 0$ do
31:         $\tau = \arg\max_\tau\{\theta(\tau, i(\tau), c[\tau], b[\tau], rem\_c, rem\_b)\}$
32:         GIVERESOURCESTO($\tau$)
33:     return $(c,b)$                        ▷ c,b map cache/bw partitions to tasks
```

These tasks would use up all the resources if they were added to *prio*.

Next, in lines 22–27, the DADNA algorithm allocates resources to the tasks in *prio*, starting at the task with the "greatest need" (that is, the greatest difference between its projected completion and its deadline). Once a task has enough resources to finish before the deadline, it is removed from the *prio* set. If there are resources left over when the set is empty, DADNA allocates them in the same way as DNA, by giving them to the tasks that can benefit the most (lines 29–32).


virtual deadline of τ_0
D_0
Task τ_0
virtual deadline of τ_1
Task τ_1
$WCET_1$
D_1
latest start time of τ_1
Task τ_2
$WCET_2$
D_2
current time
latest start time of τ_2

$$\begin{aligned} \mathsf{start}(\tau_n) &= \mathsf{D}_n - \mathsf{WCET}_n \\ \mathsf{start}(\tau_i) &= \min\{\mathsf{start}(\tau_{i+1}), \mathsf{D}_i\} \\ &\quad - \mathsf{WCET}_i,\ \forall 1 \le i < n \\ \mathsf{virtualDL}(\tau_0) &= \min\{\mathsf{start}(\tau_1), \mathsf{D}_0\} \end{aligned}$$

Virtual deadline computation

Figure 4.4: Virtual deadline illustration and computation.

4.5.2 Virtual Deadlines

So far, we have considered only the tasks that are *currently* running on the available cores. If the core scheduler is EDF, these will be the tasks whose deadlines are currently the closest. However, by allocating resources to these tasks based *only* on how much *they* need to finish in time, we are potentially harming other tasks that are in the ready queue and are not currently running.

The left picture of Figure 4.4 illustrates the problem. Here, task τ_0 has the earliest deadline and will currently be running. But if τ_0 is given just enough resources to finish by its deadline D_0, tasks τ_1 and τ_2, which have deadlines shortly thereafter, will be doomed: if they start to run at D_0, there is simply not enough time left to finish both by D_2, let alone D_1.

To fix this problem, we use a variant of an old trick: *virtual deadlines*. At a high level, this works as follows. We begin with the task in the ready queue that has the largest deadline (τ_2 in our example) and compute the latest point in time at which this task would need to be started in order to finish by its deadline, assuming it is given the maximum possible resource allocation. If the next-highest deadline (τ_1's, in our example) is after that point, we replace it with a *virtual* deadline at that point. We then repeat this process with the earlier deadlines, until we arrive at a (possibly virtual) deadline for the currently running task τ_0. In other words, we (recursively) compute the virtual deadlines as shown in Figure 4.4, where τ_n denotes the task in the ready queue with the highest deadline, and D_i and WCET_i denote the absolute deadline and the worst-case execution time of τ_i under the maximum possible resource allocation. These virtual deadlines can then be used to compute the slack S for DADNA, as before. (By definition, the virtual deadline of the running task should be recomputed whenever a new job with a larger absolute deadline is released on the same core.)

Note that the virtual deadlines are a heuristic that boosts tasks that are urgent, and they are internal to DADNA only (i.e., the CPU scheduler never sees them).

Remarks: Like all existing multicore resource allocation algorithms (that we aware of), our algorithms are not optimal; there are cases where a schedule is theoretically possible, but DNA/DADNA will not find it. However, our experimental results suggest that DNA and DADNA work substantially better than the state-of-the-art technique in terms of schedulability, deadline miss ratios, and average/tail/worst-case latencies. Our experiments use DNA/DADNA with partitioned EDF, but DNA/DADNA should work with any CPU scheduler (though the benefits could vary).

In this work, we focus on reducing latencies and minimizing job deadline miss ratios; however, with a schedulability test, our solution can be adapted to hard real-time systems as well. Since DNA/DADNA is deterministic, one way to obtain a simple schedulability analysis for periodic tasks is to run DNA/DADNA for an entire hyperperiod, and to assume that, in each phase, each task runs for the maximum time we observed for that phase during profiling. A closed-form analysis would not be trivial, because the execution time depends on the allocated resources, but should still be possible.

4.6 Prototype Implementation

To evaluate our solution and to show that it can be integrated into a practical run-time system, we built a prototype of our solution on top of the Xen hypervisor. In this section, we describe some key aspects of this prototype.

Partitioning mechanisms: For partitioning the cache and the memory bandwidth, we built on top of a patch to Xen 4.8 from our earlier work [212]. This patch contains support for the MemGuard [223] technique, and we extended it to additionally support Intel's CAT. As discussed in Section 4.2.2, we artificially partition MemGuard's (continuous) memory bandwidth limits into fixed-size "partitions", whose number is equal to the number of cache partitions.

Soft-real-time support: We further extended Xen's RTDS scheduler to enable multiple instances of a VCPU to co-exist in the run queue. This is necessary to support soft real-time systems, where jobs may execute beyond their deadlines.

Thread support: Our current prototype is restricted to single-threaded workloads. This is not inherent; the reason is simply that our phase characterization (Section 4.3.1), which is based on the number of retired instructions, works best if the programs are deterministic. However, we could use deterministic multithreading – e.g., Dthreads [123] – to add thread support without losing this property (and with comparable performance), or we could use a different way to identify where a phase begins and where it ends.

Phase generation: Our prototype includes the phase generation technique from Section 4.3. To collect observations, we extended Xen's RTDS scheduler with a configurable timer, and we added a timer handler that recorded three CPU performance counters every $\Delta = 5$ ms. (Note that profiling is done one task at a time, so EDF scheduling is not necessary.) As discussed in Section 4.3.2, we set up the performance counters to track, on each core, (i) the number of instructions retired, (ii) the total number of L3 cache requests, and (iii) the number of L3 cache misses. To prevent interference from the hypervisor itself, we configured the performance counters to prevent counting at the hypervisor's privilege level.

DNA/DADNA with partitioned EDF: We implemented DNA/DADNA in our extended Xen's RTDS scheduler. The scheduler uses partitioned EDF scheduling, where tasks are restricted to a specific core, as it has smaller run-time overhead. We use worst-fit bin packing to assign tasks to cores (though other bin-packing algorithms can also be used). This has the effect of simplifying the virtual-deadline calculation for DADNA, since the number of tasks that can run on a given core and must be considered in this calculation is typically small. Overall, our implementation consists of approximately 960 lines of code for DNA and an additional 200 for the extension to consider deadlines in DADNA. For simplicity, our implementation of DADNA made one small simplification to Algorithm 2: instead of using the `TimeLeft` function, which estimates the remaining time based on the current *and* future phases, our code extrapolates based on just the current phase. This sometimes causes DADNA to make suboptimal decisions, so our results in Section 4.7 are slightly conservative.

Thrashing avoidance: To avoid cache thrashing, we set the minimum number of cache partitions a task can receive to $min\_c := 3$. We also take care to minimize the number of cache partitions that need to be reallocated when an allocation changes. This is not trivial because Intel's CAT requires each core to have a contiguous range of partitions [96, §17.19.2]: for instance, a core can get partitions #5–10, but not partitions #4–6 and #8–10. Fortunately, DNA gives us some flexibility because it only assigns each task a certain *number* of cache partitions, without specifying which ones. Thus, we can use the following simple heuristic to allocate contiguous ranges: core #0's range always starts at partition #0, the last core's range always ends at the last partition, and the ranges in between are ordered by core number. For instance, if there are four cores and DNA assigns (7,6,4,3) partitions to the tasks on these cores, the cores will get ranges #0–6, #7–12, #13–16, and #17–19. Thus, if DNA next assigns (6,7,4,3), we can simply reassign partition #6 from core #0 to core #1. Hypothetically, #14–19, #0–6, #7–10, and #11–13 could also be used, but this would involve reassigning every single partition to a different core.

Decision points: For ease of implementation, our DNA/DADNA prototype made one simplification: instead of running precisely at phase boundaries of the scheduled task, we run DNA/DADNA (i) periodically at 1ms intervals and (ii) whenever a new task is scheduled onto a core. This choice adds a small performance penalty, since DNA/DADNA may run more often than strictly necessary, and since a task may need to wait for a few microseconds after a phase change before its allocation changes accordingly, but we do not expect these costs to be significant. When DNA returns an allocation that is different from the current one, we update MemGuard's bandwidth limits directly from within the hypervisor, and we use the `wrmsr` instruction to update the bitmasks in the COS registers with the new mapping of cache partitions to cores.

4.7 Experimental Evaluation

To evaluate our solution, we performed an experimental evaluation using our prototype. Our key questions were: (1) What is the run-time overhead of DNA/DADNA? And (2) Can DNA and DADNA indeed improve latency, job miss ratio and schedulability, compared to a state-of-the-art solution?

4.7.1 Experimental Setup

Baseline: We compared DADNA and DNA to vC^{2}M [212], a state-of-the-art resource allocation technique that supports real-time workloads and can handle both cache partitions and memory bandwidth, but *cannot* take phases into account. vC^{2}M is an extension of [213], which has already been shown to substantially outperform systems without resource management, so we omit a "free-for-all" baseline in which the tasks compete for resources without any constraints. vC^{2}M takes the WCETs for each task in the workload as an input; once tasks have been assigned to cores, it *statically* assigns a number of cache and memory bandwidth partitions to each core, so as to maximize resource utilization while meeting the deadlines, but without considering in detail the behavior of the tasks. vC^{2}M comes with a schedulability analysis and is thus able to support both soft real-time and hard real-time workloads; here, we focus on the former, since DNA and DADNA support only soft real-time workloads.

Workload: Since resource allocation techniques need to work for a wide variety of workloads, it is customary to evaluate them with synthetic workloads, so that a large part of the design space can be covered. Following the approach from [212], we randomly pick our tasks from a widely used multithreaded benchmark suite; however, while [212] considered only PARSEC [23], we also use tasks from SPLASH2x [185], to get a somewhat larger variety. Both benchmark suites support a single-threaded execution mode, which we used. We profiled the tasks as described in Section 4.2.2, using the `simsmall` input type, and we extracted the phase information as discussed in Section 4.3; the profiling step also yields the WCET for each resource allocation, which is required by vC^{2}M, as well as a *reference WCET*, which is the task's WCET when it is allocated the *entire* cache and the *entire* memory bus. We picked the tasks' utilizations uniformly at random from $[0.1, 0.4]$, and we set the period (deadline) of each task to be its reference WCET divided by its utilization.[3] We generated task sets with taskset utilizations ranging between 1.0 and 3.8, at steps of 0.2. Notice that these utilizations are calculated using reference WCETs as well, i.e., on the assumption that each task can have the entire cache and memory bus to itself, when in practice the tasks have to share. Because of this, a utilization of 2.6 on four cores is already heavy and a utilization of 3.0 fully overloads the system (as they would correspond to a utilization of 3.6 and 4.1, respectively, if we assumed the cache and memory bandwidth were divided evenly among the cores). For each taskset utilization, we generated 15 independent tasksets, for a total of 225 tasksets.

Platform: We ran the generated workloads on the machine described in Section 4.2.2, using the exact same platform setup. Each task was pinned to its own dedicated VCPU, which in turn was pinned to one of four cores that was selected by worst-fit bin packing. A fifth core was reserved for running essential OS services.

[3]We also evaluated the algorithms using tasksets with bimodal utilization distributions, and the results were consistent with that of tasksets with uniform distributions. Due to space constraints, we omit the details.

	Average	99th Percentile	Maximum
DNA	16.07μs	46.11μs	103.44μs
DADNA	16.36μs	46.48μs	110.23μs

Table 4.1: Run-time cost of DNA and DADNA.

Experiments: For each taskset, we released jobs during a two-minute interval and ran them until completion under each of the three algorithm settings, and we collected the response times of all jobs (that is, the interval from the instant the job is released until the instant it is completed).

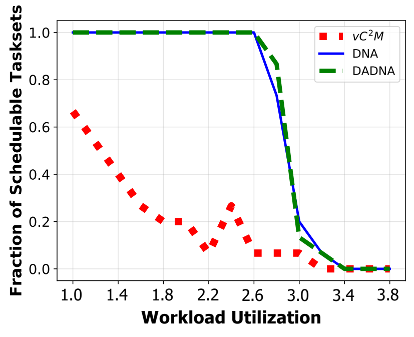
4.7.2 Run-time Overhead

Both DNA and DADNA must run frequently, so they can respond quickly to phase changes and task additions or terminations. Thus, it is important that they can run quickly and do not add a significant overhead. In general, DNA's overhead depends on the number of cores, while DADNA's overhead depends on the number of tasks per core. To examine the cost, we performed the following experiment: we used Xen's time interface to measure the duration of each DNA or DADNA run; this includes both the time to run the algorithms themselves and the time to change the cache and/or bandwidth allocations.

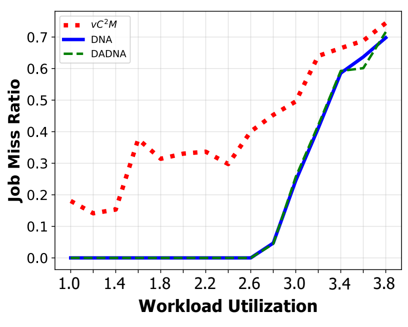
Table 4.1 shows our results. On average, both algorithms take about 16μs to run, so, if they are invoked every millisecond, the overhead is about 1.6%. (Notice that our implementation is unoptimized, and that the overhead could be further reduced by running DNA/DADNA only at phase change points, rather than periodically, as discussed in Section 4.6.) The 99th percentile and the maximum are higher. This is because adjusting resource allocations is expensive: changes to both the COS registers and the performance counters involve writing a model-specific register, which can take thousands of cycles on our platform. Normally, there are few changes, so these costs add at most a few microseconds, but in rare cases, most or all of the allocations have to be changed, which results in a higher cost. Overall, the run-time overhead of DNA/DADNA is reasonably small, and this overhead is already factored into their performance benefits reported in the following subsections.

4.7.3 Schedulability and Deadline Miss Ratio

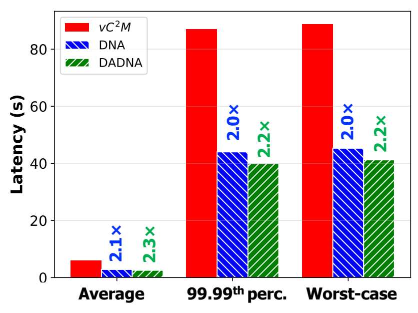
By dynamically giving resources to the tasks that can (currently) make the best use of them, DNA and DADNA should be able to improve the throughput and reduce latency, relative to a static allocation. Thus,



(a) Schedulability.

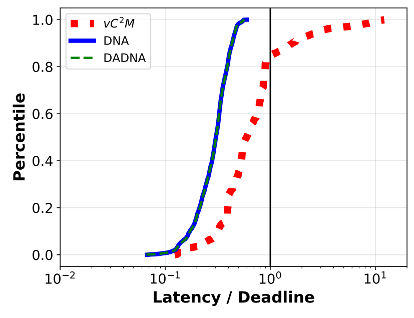


(b) Job miss ratio.

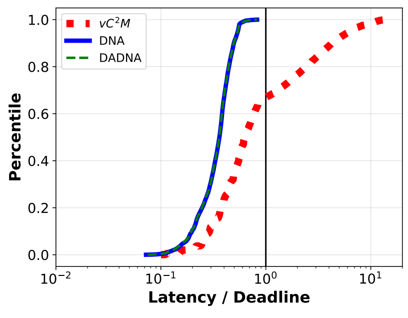


(c) Latency.

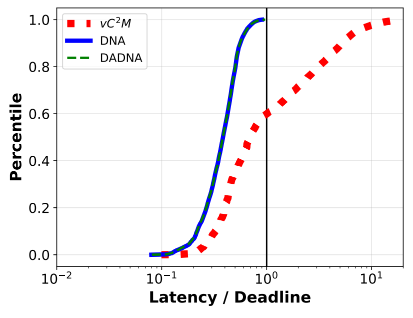
Figure 4.5: Performance of DNA, DADNA, and vC^{2}M across all workloads.



(a) Workload utilization 1.4.



(b) Workload utilization 2.0.



(c) Workload utilization 2.6.

Figure 4.6: CDF of normalized latencies.

the system should be able to schedule bigger workloads. Our first experiment tests that hypothesis. We ran experiments with DNA, DADNA, and vC^{2}M, using workloads with different utilizations, and we measured the fraction of tasksets that were empirically schedulable – i.e., tasksets whose jobs *all* meet their deadlines during our experiment.

Figure 4.5(a) shows our results. As expected, as the workload utilization increases, the fraction of schedulable tasksets also decreases for all algorithms. However, we observe that DNA and DADNA are able to schedule much larger workloads than vC^{2}M, due to their more effective use of the available resources. For instance, at a workload utilization of 2.6, vC^{2}M was able to schedule only 6.67% of the tasksets, whereas DNA and DADNA were able to schedule all tasksets (a 15× improvement). Hence, the latter is a clear improvement over the former in terms of schedulability.

Figure 4.5(b) shows how the job miss rate varies with the workload utilization. vC^{2}M experiences a substantial miss rate from very early on: even at a utilization of 1.6, it already incurs more than 30% miss rate. In contrast, DNA's and DADNA's miss rates remain zero for all utilizations up to 2.6. At a utilization of 3.0 (i.e., when the system is overloaded, as discussed in the workload generation above), DNA's and DADNA's miss rates are only half of vC^{2}M's, and the former remains strictly below the latter as the system becomes increasingly overloaded. Thus, the advantage of DNA/DADNA over vC^{2}M is beyond just the gain in schedulability: with a substantial lower deadline miss ratio, they deliver much better QoS than vC^{2}M, and this matters in a soft real-time context.

4.7.4 Latency

Another potential benefit of DNA and DADNA is that, due to the higher throughput, the latency of the jobs is potentially lower. Our next experiment is designed to examine this. We ran the same workloads as before, using vC^{2}M, DNA, and DADNA, but this time we measured the latency of each job. We then computed the average, 99.99th percentile, and worst-case latencies for each algorithm across all workload utilizations.

Figure 4.5(c) shows our results. The numbers above the columns show DNA's and DADNA's latency reduction factors relative to vC^{2}M. Again, DNA and DADNA substantially outperform vC^{2}M: they cut the average, the 99.99th percentile, and the worst-case latencies by more than half. This is expected: it is well known that EDF produces increasing latencies under overload conditions, since jobs tend to "back up" for some time once a deadline is missed, causing cascades of additional deadline misses along the way, and this effect increases with utilization. (This is also why the numbers are so high in absolute terms.) Thus, by making the best use out of the resources, DNA and DADNA can effectively reduce not only the average but also the tail and worst-case latencies.

Figure 4.6 shows these results in more detail; it contains CDFs of the normalized latency (that is, the ratio of latency to deadline) for different workload utilizations: 1.4 (a), 2.0 (b), and 2.6 (c). The results reinforce the earlier point that DNA's and DADNA's more efficient use of the available resources improves latency substantially, relative to vC^{2}M. In this experiment, there is little difference between DNA and DADNA because the behavior of the two differs only very close to a deadline.

The results also show that DNA's and DADNA's improvement factors depend on the system loads. Figure 4.7 shows their overall latency reduction factors, relative to vC^{2}M, at three different workload utilizations (1.0,

	Utilization = 1.0			Utilization = 2.0			Utilization = 3.6		
	Avg	99.99th	Max	Avg	99.99th	Max	Avg	99.99th	Max
DNA	4.5	14.3	13.9	3.2	5.3	5.0	1.7	2.0	2.0
DADNA	4.5	14.8	14.8	3.3	5.6	5.6	2.2	2.4	2.4

Figure 4.7: Latency reduction factors relative to vC^{2}M.

2.0 and 3.6) that represent light load, medium load and heavily overload scenarios. Again, DADNA and DNA consistently outperform vC^{2}M by a significant factor. For example, DADNA and DNA reduce the average latency by more than 4.5× at light load, 3.2× at medium load, and 1.7× at heavily overload scenarios, compared to vC^{2}M. The reduction factors for the 99.99th percentile and worst-case latencies are even more significant: DADNA and DNA cut latencies by more than 14×, 5.3× and 2.0× at light load, medium load and heavily overload scenarios, respectively.

4.7.5 DADNA vs. DNA

As shown in Figure 4.5(c) and Figure 4.7, DADNA is more effective than DNA in reducing latencies. However, their difference in schedulability and deadline miss ratio in Figures 4.5(a-b) appears to be somewhat small. This is because both algorithms allocate resources dynamically and efficiently, and because EDF is used in both cases; the only difference is that DADNA can handle an (important) corner case in which a job is not schedulable in the usual way but can be pushed over the edge with a greater resource allocation. This was observed in our results for bimodal tasksets, where DADNA offers up to 1.5× improvement over DNA in schedulability (and latencies); due to space constraints, we omit the details.

4.7.6 Summary

By taking phases into account and by reallocating resources dynamically, both DNA and DADNA can use the available resources more effectively than a static allocation method, such as vC^{2}M. This results in better schedulability, lower job deadline miss ratio, and lower latencies. In terms of overall performance, the two algorithms are similar, but DADNA is noticeably better than DNA at reducing latencies.

4.8 Related Work

Sharing-aware analysis: One way to achieve timing guarantees in the presence of shared resources is to factor the sharing-related overhead into the timing analysis, as is done for memory, e.g., in [171, 168, 172, 101, 221, 50], and for caches, e.g., in [208]. However, without isolation, it is difficult to obtain tight bounds because one generally has no choice but to assume worst-case interference from other tasks or cores, which leads to a high latency overhead.

Resource partitioning: Another approach is to explicitly divide up the shared resources among the cores or tasks, and to strictly enforce this allocation at runtime. Main memory interference is often split into two distinct categories: spatial and temporal interference. Spacial interference occurs when two cores access memory locations that fall within the same physical DRAM bank, creating additional latencies from repeatedly loading and storing bank rows. Existing work mitigates this by allocating physical memory to threads such that no two threads share the same bank, or such that banks are shared by threads that have complimentary resource usage patterns [132, 102]. Temporal interference on the other hand, arises when cores make memory requests too frequently and must wait on the DRAM controller to schedule their commands. We focus only on temporal interference in this work and leave spacial interference management as a future contribution. For memory bandwidth temporal interference, hardware-based techniques, such as [232, 85, 86, 83, 116] [67, 235], can provide fine-grained control, but they are not available in most commodity processors; software-based solutions, such as [223, 224, 5, 222], typically leverage existing hardware features, such as the processor's performance monitoring unit. On the cache side, software-only approaches – such as page coloring [103, 126, 218] or compiler-based [131] techniques – are more limited, but fortunately, modern processors increasingly have explicit support for cache partitioning, e.g., in the form of Intel's CAT [95] or the Lockdown-by-Master (LbM) technology in ARM processors [10]. These techniques enforce a given allocation but cannot decide *how* to best allocate the resources among the tasks, which is the focus of the present chapter. However, we rely on two of them – MemGuard [223] and Intel's CAT – to enforce DNA's and DADNA's allocations (see Section 4.2.1 for additional details).

Multiple resources: The solution we propose is able to 1) support latency-sensitive and soft real-time workloads, 2) take into account the intertwined relationship among three different resources (CPU, cache space, and memory bandwidth), and 3) consider the dynamic behavior of the workload, in the form of phases. There are several other systems that can provide some subset of these properties, but, to our knowledge, there is none that can provide all three. Several systems are able to allocate more than one resource at the same time; for instance, [122] allocates cache space and memory banks; DRF [74] allocates CPU and memory; Quasar [57] allocates nodes, cores, memory, and storage; and [24, 203, 113, 173, 186, 142] allocate cache space and memory bandwidth. However, these systems focus on throughput (or, in the case of CoPart [142] and DRF [74], on fairness) and do not consider worst-case latencies or deadlines. Other systems do consider timing constraints but their resource allocations only take into account various combinations of *two* resources – such as CPU and memory bandwidth [217, 5, 144, 128], cache and memory banks [194, 105, 104, 42], or cache and CPU [190, 209]– but not all three. We are aware of only a handful of systems that can both support real-time workloads and consider all three resources dynamically. MARACAS [219] and CaM/vC^{2}M [213, 212] both use static allocations and do not change the allocation based on dynamic behavior (i.e., the execution phases) while [77, 75] both focus on multimodal systems and update resources only at the boundary of mode transitions.

Prior work has considered dynamic allocation, but focuses primarily on individual resources. For instance, [188] changes allocations based on marginal gain, but a) it focuses on HPC workloads without deadlines, and b) it measures the marginal gain at runtime, using counters, which works fine for one resource but would be difficult for, e.g., both memory bandwidth and cache. vCAT [214] introduces an abstraction for virtualizing Intel's CAT and a way to control it at runtime, but requires the programmer to manually insert system calls at phase boundaries to adjust the partitions; in contrast, our solution finds the phases without developer input, and it uses DNA/DADNA to make allocation decisions automatically.

Recently, newer work in this area has begun to use learning based techniques for resource allocation. Of note, Xu and colleagues [215] use deep reinforcement learning to create a holistic offline task schedule and shared resource allocation using both last level cache and shared memory bandwidth. While their work considers all three resources, it doesn't do so in a dynamic online fashion. Zhao et al. [230] use a learning technique to intelligently allocate tasks to cores such that cache interference is minimized, however the authors never partition the cache itself.

Program types and phases: The insight that programs can have different interactions with resources is not new; previous work has classified the behavior using different colors [117], based on marginal utility [153], using miss models [37], via analytical modeling [20], or with animal types [210]. Zhuravlev et al. [234] compared some of these schemes and designed a (non-real-time) scheduling algorithm that uses them. Other work has used dependencies between resource usage and program inputs for scheduling, e.g., to save energy [73, 92]. The observation that the same program can go through different *phases* goes back to a paper by Sherwood and Calder [178], and since then, a number of techniques for identifying the phases have been developed, including ones based on k-means clustering [176], threshold clustering [60, 61, 177], visual inspection [47], pattern matching [108, 174], or wavelets [44, 45, 93, 174]. DNA and DADNA could leverage these techniques instead of the approach we used in our prototype (see Section 4.3). Phase-based workload characterization has also been exploited to build accurate energy models [84] and execution time prediction [158]. Recently, Chen et al. [40] used course-grain phase behavior for the purpose of identifying timely progress on black box applications under a variety of inputs, however this work doesn't provide fine-grain phase information on resource usage and moment-to-moment rates of execution. To our knowledge, DADNA is the first algorithm to use fine-grain phases for the multicore resource allocation of latency-sensitive and real-time workloads.

4.9 Conclusion

Our results suggest that it makes sense for schedulers and resource allocators to "look a bit more closely" at the tasks in their workloads. By leveraging an observation from the architecture community – namely that many programs go through multiple phases with distinct characteristics – DNA and DADNA are able to improve the performance of a system *without* adding more resources, by allocating the existing resources more effectively to the tasks that can benefit the most. Compared to prior work, this results in substantially better schedulability and a factor-of-two latency reduction. And yet, the phase analysis we used is relatively simple; to us, it seems likely that there is a lot more information about the needs and behaviors of tasks that could be extracted – e.g., through profiling or static analysis – and used profitably to improve scheduling. This could be an interesting future work. Another interesting direction is to develop a close-formed schedulability test for DNA and DADNA to bring their benefits to hard real-time systems.

CHAPTER 5

MULTI-MODE ON MULTI-CORE: MAKING THE BEST OF BOTH WORLDS WITH OMNI

In the last chapter, we saw that adapting allocations to task phases can improve efficiency and timeliness within a fixed task set. But real-world systems rarely operate under a single, steady-state workload. Tasks enter and leave the system, periods shift, and long running tasks may have their deadlines change over time. These changes mean that resource demands and interference patterns can fluctuate significantly, creating pressure points that static allocations cannot anticipate.

This chapter expands upon the previous one by exploring how dynamic allocation can be extended to handle systems where tasks and tasksets vary over time. In Section 5.1 we fully motivate the need for a multi-mode abstraction in modern adaptive real-time systems. Section 5.2 explores more specific realted work while Section 5.3 covers this chapter's multi-mode system model. Section 5.4 discusses the Omni algorithm in detail and Sections 5.6 and 5.7 cover prototype and evaluation.

This chapter is based on work that first appeared as: Robert Gifford and Linh Thi Xuan Phan. Multi-mode on multicore: Making the best of both worlds with omni. In *RTSS*, 2022.

5.1 Introduction

The trend towards full autonomy has transformed the design of cyber-physical systems (CPS) in two fundamental ways. On the software side, CPS are becoming increasingly adaptive: for example, a self-driving vehicle may need to dynamically execute different subsets of software features at runtime depending on the road conditions or detected obstacles, and/or it may switch between different controllers, such as an advanced MPC controller and a standard PID controller, to optimize performance and safety. On the hardware side, CPS increasingly use modern multicore platforms in many critical areas, such as automotive [112] and avionics [151, 124], where they not only provide enhanced capabilities and performance, but also lower costs. These developments have introduced a new real-time challenge: How to ensure predictable run-time adaptation on multicore platforms while maximizing resource use efficiency?

Recent research in real-time systems has already developed two important technologies that can help to address this challenge. First, multi-mode theories [148, 39, 156] can be used to model and formally reason about adaptive systems. In this formalism, the system is represented as a multi-mode system, whose modes correspond to different configurations and whose transitions correspond to configuration changes in response to events. Each mode is associated with a set of tasks that are active when the system is in that mode. Multi-mode analysis can then be used to analyze the timing behavior of the system, both within a mode and during a mode transition. The second technology is multicore resource allocation [212], which offers a way to reduce interference among concurrent executions, enable better isolation among tasks, and consequentially achieve better schedulability. However, these two technologies are generally disconnected from one another: the former either focuses on single core or completely ignores shared resources such as cache and memory bandwidth, while the latter typically assumes a static system with a fixed taskset that does not change at runtime. To the best of our knowledge, none of the existing solutions considers *both* multi-mode systems scheduling *and* shared multicore multi-resource allocation.

Unfortunately, there is no easy way to extend one of the two technologies to cover the entire scenario. One possible approach would be to integrate a multi-mode scheduling technique, such as [55], with an even partitioning of shared resources. This approach can ensure timing guarantees for multi-mode systems, but—as our evaluation shows—it suffers from poor schedulability and resource utilization because it cannot adapt to the dynamic changes in resource demands, which happen naturally whenever the system transitions between modes. Another approach would be to apply existing multicore resource allocation methods, such as the holistic allocation from [212], to each individual mode. This approach produces a unique resource configuration for each mode that closely matches each mode's resource demand, but, without considering mode transitions, it yields low schedulability and cannot provide any guarantees.

In this chapter, we present Omni, a first integrated end-to-end solution to the multi-resource co-allocation problem of multi-mode systems on multicore platforms. Omni consists of (1) a multi-mode resource allocation algorithm that holistically maps tasks and allocates cache and memory bandwidth resources to cores, and (2) a resource-aware schedulability test for the system. Our key insight is to concurrently (i) adapt resource and task allocation dynamically as the system transitions between modes, to closely match the changes in resource demands at mode changes, and to (ii) take into account the effect of mode changes on execution demands, which in turn guide our allocation decisions. Omni's resource allocation algorithm leverages the platform concurrency and the diversity of tasks' demands to minimize overload during mode transitions. This is achieved by strategically redistributing tasks and resources in tandem across cores. In doing so, Omni aims to maximize schedulability in each mode while minimizing the mode-change overhead by bringing the modes' allocations as close to each other as possible. Omni further separates new tasks that appear only in the new mode from existing tasks as much as possible; this is to avoid potential overload during the transition, when new jobs of new tasks (which are often released immediately at the mode-change instant) co-exist on the same core with unfinished jobs from the old mode. In summary, we make the following contributions:

- a multi-mode resource allocation algorithm that dynamically adapts task mapping and shared resource allocation to changes in resource demands at mode transitions (Section 5.4);
- a resource-aware multi-mode schedulability analysis (Section 5.5); and
- a prototype implementation of Omni that supports general mode-change semantics and dynamic resource allocation (Section 5.6)

We have evaluated Omni using multi-mode systems generated from PARSEC [23], SPLASH2x [185], DIS [133], and Isolbench [199] benchmarks. Our results show that Omni can be implemented with low overhead, and that it substantially outperforms state-of-the-art techniques in terms of schedulability and resource-use efficiency.

5.2 Related Work

Multi-mode systems have been studied extensively in real-time literature. Existing solutions focus on two key areas: 1) new models and timing analysis techniques for supporting multi-mode behaviors (see e.g., [29, 30, 78, 8, 150, 180, 146, 147, 127]), and 2) mode change protocols for ensuring schedulability during mode transitions (e.g., see [28, 156] and references therein). Tools for systematic design exploration and evaluation of MCPs have also been developed [148]. Most existing solutions target uniprocessors, however.

Multi-mode scheduling and analysis have recently been extended towards multiprocessors [79, 135, 55, 16, 155, 6]. While the majority still considers CPU only, some recent work has begun to consider shared resources. For instance, Negrean [134] develops an analysis for multi-mode applications on AUTOSAR conform multicore platforms; it considers resource sharing protocols such as Priority Ceiling Protocol (PCP) and spinlock-based mechanisms. Methods for mapping multi-mode applications on NoC platforms have also been studied [63]. Closely related to our work, Kwon [107] recently proposes a cache allocation technique that adapts cache allocation as the system mode changes; however, it focuses exclusively on cache allocation, and it considers neither memory bandwidth nor task mapping. To the best of our knowledge, none of the existing solutions for multi-mode systems are able to perform adaptive co-allocation of tasks and multiple shared resources (cache and memory bandwidth) on multicore platforms in response to mode changes, as Omni does.

Several multi-resource co-allocation techniques have been developed. For example, CaM [213] and its virtualization extension [212] propose holistic resource allocation techniques that find the assignments of tasks, cache and memory bandwidth to cores in an integrated fashion. Unlike Omni, these techniques are static and do not consider multi-mode behaviors. It is highly non-trivial to extend such a solution to the multi-mode setting–as our evaluation results show, a simple extension of CaM suffers from very poor performance.

Dynamic co-allocation of multiple resources has also been explored. For example, DNA/DADNA [76] adapts the resource allocation at run time based on program phases. Like existing work in this space, DNA/DADNA assumes single-mode systems. It is also designed for *soft* real-time and does not have a theoretical analysis. As DNA uses average rates of execution for its allocations, a worst-case schedulability test would require fundamental changes to the algorithm itself. Extending fine-grained adaptive solutions like DNA to multi-mode hard real-time systems is an interesting future direction.

Several platforms have been developed to support multi-mode behaviors. For instance, Neukirchner et al. [136] develops an implementation of multi-mode monitors. Azim et. al. [15] proposes an implementation in LITMUS^{RT} that utilizes checkpoints and rollback-based mode-change mechanisms for efficient mode changes. Chen et. al. [39] provides a Xen-based system called SafeMC for experimental exploration and evaluation of MCPs. There also exist multi-mode virtualization platforms that support multi-mode systems, such as M2-Xen [114]. Unlike our Omni prototype, none of these platforms supports dynamic shared resource allocations.

5.3 System Model and Goal

Platform. We consider a multicore platform consisting of K identical cores, and a shared cache and memory bus that are accessible by all cores. The cache is divided into $C_{\max}$ equal-size cache partitions, using an existing cache partitioning technique such as Intel's CAT. Similarly, the memory bandwidth (referred to as 'bandwidth' hereafter) is divided into $B_{\max}$ equal-size bandwidth partitions, using an existing technique such as MemGuard. Cache and bandwidth are allocated dynamically at the core level: at run time, each core is allocated a set of cache and memory bandwidth partitions, which are available to any task currently running on the core. To minimize run-time overhead, we restrict cache and memory reallocations to only during a mode change, i.e., when resource demands change the most.

Within each mode, tasks are assigned to fixed cores. Tasks on a core are scheduled using the partitioned Earliest Deadline First (EDF) policy[4]. However, during a mode change, a task may be assigned to a different core in the new mode to improve schedulability. Our choice of limiting migrations and resource reallocations to only during mode changes seeks to balance the tradeoff between run-time efficiency and schedulability.

Multi-mode system model. We consider a multi-mode system that is defined by a set of operating modes $\mathscr{M}$, an initial mode $\mathsf{m}_0 \in \mathscr{M}$, a set of mode transitions $\mathscr{R} \subseteq \mathscr{M} \times \mathscr{M}$, and a set of deadline-constrained periodic tasks $\mathscr{T}$ that need to execute in the modes. Each mode m has a set of tasks $\mathscr{T}^m \subseteq \mathscr{T}$ that are active when the system is in this mode. Each transition is triggered by a mode-change request event (MCR), which for simplicity is assumed to be unique for each transition. Initially, the system begins in the initial mode m_0. At runtime, whenever an MCR associated with an outgoing transition arrives, the system will perform the mode change, according to a given mode change protocol, to move to the destination mode. As in most existing work, we assume mode changes do not overlap – the system processes MCRs in a first-come-first-served basis, rejecting MCRs while performing the mode change until all mode change actions complete.

[4]We use EDF due to its high resource utilization bound; it should be possible to extend to other scheduling policies.

Resource-aware task model. We follow a resource-aware real-time task model based on [213], where each task is characterized by three per-mode timing attributes: a period, a deadline, and a resource-aware worst-case execution time (WCET) that specifies the task's WCET depending on the resources it is given. For each mode m and each task $\tau \in \mathscr{T}^m$, we denote by p_τ^m and d_τ^m the period and deadline of τ in mode m, respectively, and by $e_\tau^m(c,b)$ the WCET of τ in mode m when it is allocated c cache partitions and b bandwidth partitions ($1 \leq c \leq C_{\max}$ and $1 \leq b \leq B_{\max}$). As in existing work, the task's resource-aware WCET can be obtained by formal analysis or measurement; we followed the latter for our experiments.

Mode-change protocol (MCP). An MCP describes the execution behavior of a multi-mode system during a mode transition – that is, from the instant the associated MCR arrives until the instant where all new attributes associated with the new (destination) mode are in effect. Before specifying the MCP, we first distinguish the different types of tasks during a mode transition from mode m' to mode m:

- Old tasks: Active in m' but *not* in m.
- New tasks: Active in m but *not* in m'.
- Existing tasks: Active in both modes. These tasks consist of *unchanged* tasks, which maintain the same timing attributes (period, deadline, WCET) in the new mode, and *changed* tasks, which have at least one of their timing parameters modified in the new mode.

We further categorize jobs during a transition into two types: *existing jobs* are unfinished jobs that are released before the MCR instant but have deadlines after the MCR instant, and *new jobs* are jobs that are released at or after the MCR instant.

For analysis purposes, we assume the following mode change semantics, as it can maintain periodicity while minimizing mode change latencies and system loads. However, our analysis should extend to other MCPs as well.

- Old tasks are dropped immediately (including existing jobs) and there are no new releases in the new mode. Since old tasks are no longer needed, dropping them immediately helps avoid unnecessary overload.

- Unchanged tasks release their jobs as before without being affected by the mode change, to maintain their periodicity.
- Changed tasks release their first new jobs based on the old period (i.e., as if there were no mode transition), and release all subsequent jobs based on the new period. Again, this strategy aims to maintain the tasks' periodicity as much as possible.
- New tasks release their first jobs on their assigned cores immediately, to enable a prompt transition to the new mode.
- All new job releases follow their tasks' periods and deadlines associated with the new mode. Existing jobs, however, maintain their existing deadlines.

During a mode transition, tasks may migrate among cores and the resources allocated to each core may also change. Therefore, we need to extend the above protocol to consider migration and reallocation semantics, as follows:[5]

- If a task is assigned to a different core in the new mode, its existing job is migrated to the new core.
- All new jobs of active tasks will be released on the cores they are mapped to in the new mode.
- Cache and bandwidth reallocation will take effect at the MCR instant; consequently, existing jobs is (re)allocated the resources assigned to their cores in the new mode.

Goal. Given the above setting, our goal is perform task and resource co-allocation for the multi-mode system to maximize its schedulability. Specifically, for each mode m of the system, we seek to holistically compute the task-to-core mapping Π^m and the cache and bandwidth configurations (C^m, B^m) for each core when the system is in mode m. Here, $\Pi^m_\tau \in [1, K]$ represents the core on which τ is mapped to in mode m, and C^m_k and B^m_k represent the number of cache partitions and bandwidth partitions allocated to core k in mode m, respectively, for all $1 \leq k \leq K$. We require that the total number of cache (bandwidth) partitions allocated to all cores is no more than $C_{\max}$ ($B_{\max}$).

[5]Systematic design exploration of novel resource-aware multicore mode change protocols, e.g., a resource-aware extension of SafeMC [39] is an interesting future direction.

Challenges. One common challenge in scheduling multi-mode systems is to ensure timing guarantees during mode transitions, because of the potential overload caused by the co-existence of existing jobs and new jobs. In our setting, schedulability of a transition has two additional challenges: (1) Existing jobs on a core during a mode transition may come from a different core in the old mode, so their (remaining) execution times depend on not only the resource configurations and executions of their old cores but also those of their new cores. This leads to extra overhead and complicates resource allocation. (2) The task-to-core mapping and resource allocation are interdependent: a poor mapping in a mode can make it difficult to efficiently allocate resources, and vice versa.

Baseline solutions. Since there exists no existing work that considers shared resource allocation for multi-mode systems, we extend the two most closely-related hard real-time solutions (c.f. Section 5.2) to our setting as baseline solutions. The first extends the multi-modal task partitioning technique in [55], which computes a static mapping of tasks to cores that takes into account mode change effects but it ignores shared resources. Our extension, referred to as MM-Static, applies the same partitioning strategy, except that it statically divides the cache and bandwidth as evenly as possible among the cores, and then uses the resulting WCETs obtained under that resource configuration for the analysis. (For consistency, we also replace the zero-slack rate-monotonic (ZSRM) [54] schedulability test used in [55] with our EDF-based schedulability test in Section 5.5.) The second baseline uses the static holistic resource allocation algorithm from [213] to compute the task mapping and resource allocation for each individual mode in the system. We refer to this baseline as CaM.

One can observe that neither baseline solution is ideal: the first cannot handle cases where the set of active tasks or their timing attributes change drastically during a mode transition, since it relies on a single static mapping for all modes. In contrast, the second completely ignores mode transitions, which can lead to substantial mode change overhead (as the mappings may differ substantially) and poor schedulability during mode transitions. We next discuss the Omni algorithm and how it overcomes these challenges.

5.4 Omni Resource Allocation Algorithm

Basic ideas. Omni aims to co-allocate tasks and resources to cores in each mode such that the resources given to each core best match the demands generated by the tasks mapped to the core. Towards this, it uses a combination of four key insights:

Insight #1. Omni performs reallocation at *mode transitions*, since these are time points when resource demands may change significantly.

Insight #2. For each mode, the task mapping and the (cache and bandwidth) allocation are computed in a tightly integrated fashion to account for their interdependence and their combined effects on tasks' WCETs and cores' utilizations.

Insight #3. Omni reduces the worst-case overload and mode change overhead during any mode transition by considering all incoming transitions when computing an allocation for a mode. The idea is to make the target mode's allocation as similar to the source mode's allocation as possible across all possible incoming transitions, thus reducing task migrations and mode change latency. The implication is that existing tasks should be kept on the same cores across as many mode transitions as possible, without risking overall schedulability.

Insight #4. New tasks should share cores with existing tasks as little as possible to reduce the chance of them overloading the system during a mode transition, since they may co-exist with existing jobs.

Omni works by exploring the multi-mode system structure to compute new task and resource allocations for modes, followed by a redistribution of tasks (and resources) for unschedulable modes (if needed). It repeats this process over multiple rounds, until either the system is schedulable and a solution is found, or a given maximum number of task redistributions *R* is reached.

Our algorithm relies on two pre-configured parameters: *R*, the maximum number of task redistribution attempts *after each round*; and *r*, the maximum number of task redistributions performed *when computing each mode's allocation* during a round. These parameters can be configured based on, e.g., the maximum number of tasks per mode divided by the average task utilization. Intuitively, the smaller the average task utilization, the more tasks need to be redistributed to have an impact on core schedulability, and vice versa.

We next describe an overview of our allocation algorithm.

5.4.1 Algorithm Overview

Omni begins by initializing each mode with a base allocation, which consists of a task-to-core mapping and a per-core cache and bandwidth configuration. (For our experiments, we used the static allocation given by MM-Static; however, Omni can work with any base allocation, though the result may vary.) It then selects a mode, $\mathsf{m}_{\mathsf{start}}$, to be the starting mode for the multi-mode exploration; this could be the initial mode m_0.

Phase 1: Multi-mode exploration and allocation. Starting with $\mathsf{m}_{\mathsf{start}}$, Omni performs a round of iterative exploration of the multi-mode system structure (e.g., in a breadth-first-search manner). At each reachable mode m, it computes a new resource and task allocation for m based on m's current allocation and the allocations of its (direct) predecessor modes (i.e., modes with an incoming transition to m). This computation is done by a 'folding' procedure that aims to (i) keep existing tasks on the same cores across as many of these modes as possible (to reduce migrations), (ii) limit core sharing as much as possible between new tasks and existing tasks (to reduce overload during mode transitions), and (iii) balance loads among cores (to maximize schedulability). This is shown in the FOLD() function in Algorithm 3.

Phase 2: System-wide schedulability analysis. After all modes have been explored, Omni analyzes the schedulability of the system under the new task and resource allocation, using the schedulability analysis in Section 5.5. If the system is schedulable, Omni terminates and outputs the new allocation as a solution. Otherwise, it computes a new *load score* – defined as the sum of tasks' utilizations under the new allocation across all modes in the system. There are two cases:

a) If the new load score reduces the old load score (computed in the previous round) by more than some configurable threshold: Omni saves the new allocation and load score for the current round, and then continues to the next round of multi-mode exploration (Phase 1), starting with $\mathsf{m}_{\mathsf{start}}$ as before.

b) Otherwise: Omni reverts to the old allocation (from the previous round). It will then attempt to perform "task redistribution" (Phase 3) for unschedulable modes and transitions.

If the system is unschedulable and Omni has reached the maximum number of task redistribution attempts R, it simply reports unschedulable and outputs the old allocation.

Phase 3: Task redistribution. The task redistribution procedure (c.f. TASKREDISTRIBUTE() function, Algorithm 4) aims to bring the system out of an unschedulable state by moving or swapping tasks between cores.[6] Specifically, beginning with a mode m^* that is unschedulable or that has an unschedulable incoming mode transition, our task redistribution tries to move a task in m^* from an unschedulable core to a schedulable core. If this is not possible, it will consider swapping tasks between cores. (Task swapping is useful, e.g., when the algorithm falls into a local optima and cannot further reduce the system load sufficiently.) To determine whether a task move/swap is feasible, Omni checks whether the schedulable target core will remain schedulable after the move/swap *and* a resource redistribution. If a move/swap is feasible, Omni performs the move/swap, followed by a resource redistribution for m^*. It then saves the allocation given by the task redistribution, along with its resulting load score, for the current round. It will then repeat the multi-mode exploration (Phase 1), but starting with mode m^* (i.e., $\mathsf{m}_{\mathsf{start}} = m^*$). If neither a move nor a swap is feasible for m^*, Omni tries with the next unschedulable mode or transition. This is important because, even if an earlier explored mode or mode transition remains unschedulable, a task redistribution in a later mode can lead to a change in the allocations of the whole multi-mode system, thus potentially enabling schedulability of the earlier mode as well.

Termination condition. The algorithm terminates when (i) the system is schedulable at the end of the current round of multi-mode exploration, or (ii) task redistribution is not feasible for any of the unschedulable modes or mode transitions, or (iii) the maximum number of task redistribution attempts R has been reached, whichever is earlier. Termination is guaranteed because (1) either the system is schedulable, or (2) the algorithm terminates because it has performed R task redistribution attempts or because task redistribution is not feasible for any of the unschedulable modes and mode transitions.

[6]This procedure is also used internally by the FOLD() function in computing the new allocation for a mode during Phase 1.

5.4.2 Details of Key Procedures

Next, we describe the folding and task redistribution procedures in detail. Due to space constraints, we omit other simpler or less critical functions.

Folding procedure. Algorithm 3 shows the folding procedure for computing a new allocation for a current mode M_{cur}. The procedure works by "folding" the current allocations of the predecessor modes M_{prevs} and the current mode M_{cur}; the goal is to keep existing tasks on the same cores across as many mode transitions as possible. For this, Omni first filters out all tasks of the predecessors that are not active in M_{cur} from their allocations (Lines 2–3). It then assigns tasks in mode M_{cur} into *tiers* (Line 4). Specifically, a task τ is in tier i if, under the current allocation, i is mapped onto the same core for a maximum of i modes in $M_{prevs} \cup \{M_{cur}\}$. We refer to such a core as a *best core* for τ.

Intuitively, a task in a higher tier shares the same core across more modes than a task in a lower tier; hence, we should keep it on its best core if possible. In contrast, a task that belongs to the lowest tier either shares no common core across the modes or is a new task for any incoming mode transition (i.e., not active in M_{cur}'s predecessors). In the former case, the task will inevitably experience migration for all but at most one incoming mode transition, regardless of which core it is mapped to in M_{cur}. Therefore, Omni prioritizes minimizing worst-case utilization in selecting a core for such tasks to improve schedulability.[7] In the latter case, Omni aims to keep this new task on a different core from those of existing tasks to avoid overloading unfinished jobs.

Based on the above insight, the algorithm works by iterating through the tiers, in decreasing tier number until tier 1 (Line 6). For each task τ in a chosen tier, it assigns τ to its best core (Lines 7–8), keeping track of each core that contains an existing task (*carryoverCores* in Line 10). Between task assignments, Omni performs *resource redistribution* for M_{cur} to balance the loads across cores (Line 9).

Resource redistribution is done incrementally one partition at a time, by first selecting the least-loaded (lowest-utilization) core and most-loaded (highest-utilization) core, and then moving one resource partition

[7]The current *worst-case utilization* of a core is the total utilization of the tasks that have been assigned to the core so far, assuming that each such task has the largest WCET among its WCETs in $M_{prevs} \cup \{M_{cur}\}$.

Algorithm 3 Computation of a new allocation for a mode

```
1: function FOLD($M_{cur}$, $M_{prevs}$, $cores$, $r$)                                   ▷
       $M_{cur}$: current mode,
       $M_{prevs}$: previous modes that can transistion into $M_{cur}$,
       $r$: number of attempts for task redistribution
2:     for $M_{prev} \in M_{prevs}$ do
3:         FilterAllocation($M_{cur}$, $M_{prev}.cores$)
4:     tierList =
       CreateCoreTier($M_{cur}.T$, $M_{cur} \cup M_{prevs}$, $|M_{cur}| + |M_{prevs}|$)
5:     $carryoverCores = \emptyset$
6:     for $i = |\text{tierList}| \dots 1$ do                               ▷ Map tasks to cores until tier 1
7:         for $\tau \in$ tierList[$i$] do                               ▷ Assign to most popular core
8:             assignTask($M_{cur}$, $\tau$, $\tau.bestCore$)
9:             ResRedistribute($M_{cur}.cores$)
10:            carryoverCores $\cup$ $\tau.bestCore$                        ▷ Note carry-over task here
11:
12:    delaySet = $\emptyset$
13:    for $\tau \in$ tierList[0] do                                   ▷ Iterate through tier 1 tasks
14:        if $\tau \in M_{prevs}$ == false then
15:            delaySet $\cup$ $\tau$
16:            continue
17:        didAlloc = false
18:        $lcore$ = GetLowestUtil(carryoverCores)
19:        while $lcore$! = NULL && didAlloc == false do
20:            if $U[lcore] + U[\tau] > 1$ then
21:                $lcore$ = GetLowestUtil(carryoverCores)
22:                continue
23:            assignTask($M_{cur}$, $\tau$, $lcore$)
24:            didAlloc = true
25:
26:        if didAlloc == false then
27:            $lcore$ = GetNextLowestUtil($M_{prevs}.cores$)
28:            assignTask($M_{cur}$, $\tau$, $lcore$)
29:
30:        ResRedistribute($M_{cur}.cores$)
31:
32:    for $\tau \in delaySet$ do                                      ▷ Iterate through $M_{cur}$ only tasks
33:        didAlloc = false
34:        $lcore$ = GetLowestUtil($cores$ $\cap$ carryoverCores)
35:        while $lcore$! = NULL && didAlloc == false do
36:            if $U[lcore] + U[\tau] > 1$ then
37:                $lcore$ = GetLowestUtil(carryoverCores)
38:                continue
39:            assignTask($M_{cur}$, $\tau$, $lcore$)
40:            didAlloc = true
41:
42:        if didAlloc == false then
43:            $lcore$ = GetNextLowestUtil($M_{prevs}.cores$)
44:            assignTask($M_{cur}$, $\tau$, $lcore$)
45:        ResRedistribute($M_{cur}.cores$)
46:
47:    for $i = 0 \dots r$ do
48:        if ModeSchedulable($M_{cur}$) == true then
49:            return
50:        if !TaskRedistribute($M_{prevs}$, $M_{cur}.cores$, true) then
51:            TaskRedistribute($M_{prevs}$, $M_{cur}.cores$, false)
52:        ResRedistribute($M_{cur}.cores$)
```

from the former to the latter to reduce their utilization difference. If it is not possible to move a partition from the least loaded core to the most loaded core without creating a swap in their utilization (i.e., the former becomes more loaded than the latter), we move to the second least-loaded core, and so on. Resource redistribution completes when it is no longer beneficial to move partitions between cores. When that happens, Omni proceeds with the next task in the tier and repeats the same core assignment and resource redistribution procedure, until there is no task left.

Omni then proceeds to the lowest tier (*tierList[0]*), which contains tasks that do not share cores across modes. It finds an allocation for all existing tasks in this tier first (Lines 12–30), delaying new tasks to the final stage (Lines 32–51). For each existing task τ, Omni attempts to assign τ to the lowest-utilization core *lcore* among the *carryoverCores* set (containing only existing tasks so far) that can fit the task (Lines 18–24). If no core in *carryoverCores* can accommodate the task, Omni assigns τ to a core that has the smallest worst-case utilization. The worst-case utilization considers only tasks that have been assigned so far in the current round, and assuming that those tasks inherit their largest WCETs from any of the modes in M_{prevs} that they belong to (Lines 26–28).

Finally, Omni proceeds to the new tasks of M_{cur}. The allocation for a new task works similar to that of an existing task, except that Omni tries to assign the new task to cores that are not in the *carryoverCores* set (Lines 33–40). This is to separate new tasks from old tasks as much as possible to avoid overloads during mode changes. If no such core exists, it assigns the task to the core with the smallest worst-case utilization. By assigning tasks in this order, Omni simultaneously minimizes migrating carryover tasks, while still keeping new mode tasks separate to avoid mode change overloads.

If the mode is unschedulable after folding all tasks together, the algorithm attempts to redistribute tasks and resources for a configurable number of times, *r* (Lines 47–52). Task redistribution can be invoked to move either a new-mode task or a carry-over task from an unschedulable core to a schedulable core. We prioritize moving new mode tasks (Line 50) first, as these tasks do not experience mode change overheads that carry-over tasks potentially experience due to core migration or resource reallocation. If no new-mode task can be moved, we try again with a carry-over task (Line 51). Since task redistribution only moves a single task at a time, during heavy load situations, the mode may continue to be unschedulable after several redistributions.

Algorithm 4 Omni Task Redistribution

```
1: function TASKREDISTRIBUTE($M_{prevs}$, $cores$, nmTask)                          ▷
          $M_{prevs}$: Previous modes that transistion into $M_{cur}$,
          nmTask: true for new mode, false for carry over
2:     $hcore$ = GetHighestUtilCore($cores$)
3:     while $hcore$ do
4:         lastCore = false
5:         $lcore = NULL$
6:         $\tau$ = SelectTask($hcore$, $M_{prevs}$, nmTask)
7:         while $Util[hcore] - Util[\tau] > 1$ do
8:             $\tau$ = SelectTask($hcore$, $M_{prevs}$, nmTask)
9:         if $\tau$ == NULL then return ERR
10:
11:        if nmTask then
12:                                      ▷ Find next lowest util core with no carry over task
13:            $lcore$ =NextLowestUtilNoCOCore($cores$, $lcore$)
14:        else
15:                                      ▷ Find next lowest util core with carry over task
16:            $lcore$ =NextLowestUtilCOCore($cores$, $lcore$)
17:        if $lcore$ == NULL then
18:            $lcore$ = GetLowestUtilCore($cores$)
19:            lastCore = true
20:
21:        if $Util[\text{TestResRedistribute}(lcore, \tau)] \leq 1$ then
22:            MoveTask($hcore$, $lcore$, $\tau$)
23:            ResRedistribute($cores$)
24:            return OK                                      ▷ Successfully moved a task
25:
26:                                                           ▷ Try to swap tasks
27:        $\tau'$ = GetTaskWithUtilLessThan($lcore$, $1 - Util[lcore]$)
28:        if $\tau'$ == NULL && lastCore == false then
29:            go to 11                                       ▷ Try with next lowest core
30:        else if $\tau'$ == NULL && lastCore == true then
31:            return ERR
32:
33:        if $Util[\text{TestResRedistribute}(lcore, \tau')] \leq 1$ then
34:            SwapTasks($hcore$, $lcore$, $\tau$, $\tau'$)
35:            ResRedistribute($cores$)
36:            return OK
37:        $hcore$ = NextHighestUtilCore($cores$, $hcore$)
38:    return ERR
```

Task redistribution. Algorithm 4 shows the task redistribution algorithm for a mode m^* that is unschedulable or that has an unschedulable incoming transition. As discussed above, task redistribution is performed for new-mode tasks first and then for carry-over tasks in m^*. The algorithm takes as input a flag, *nmTask*, which is true if new-mode tasks are considered, and false otherwise. To redistribute tasks, Omni checks whether moving a task from an unschedulable core (in decreasing core utilization) to a schedulable core (in increasing core utilization) and redistributing resources afterwards will make the former core schedulable without making the latter unschedulable (Lines 2–24). If a move is not possible, Omni will attempt to swap tasks (Lines 27–36).

Starting with the highest-utilization core *hcore*, Omni selects a candidate task τ (i.e., a new task if $nmTask = 1$, and a carry-over task otherwise) on the core for moving (Line 6 and 8). It then searches for a destination core for τ among schedulable (low-utilization) cores, in increasing order of core utilization (Lines 11–19). For each candidate core *lcore*, Omni first checks whether the core would remain schedulable with the extra task τ after a resource redistribution (Lines 21–24). If such a move is not possible, Omni will check whether a task swap between the two cores, *hcore* and *lcore*, followed by a resource redistribution, would be feasible (Lines 27–36). If a move (swap) is feasible, Omni performs the move (swap), followed by the resource redistribution. (When there are multiple candidate tasks for moving (swapping) that meet our criteria, we prioritize the task(s) that would lead to the greatest decrease in the utilization difference between the two cores after task moving/swapping and resources redistribution.)

Complexity. As Algorithm 3 is the primary execution loop for Omni, it is useful to understand its runtime complexity. It takes $O(n \cdot K \cdot E)$ and $O(K \cdot \log K)$ to compute the tier list and to sort cores, respectively, where $n =$ maximum number of tasks per mode, $K =$ number of cores, and $E =$ maximum number of incoming transitions per mode. A resource redistribution takes $O(K \cdot C_{\max} \cdot B_{\max})$, and a task redistribution takes $O(n^2 \cdot K \cdot C_{\max} \cdot B_{\max})$, where $C_{\max}$ and $B_{\max}$ are the maximum number of cache and bandwidth partitions, respectively. Thus, Algorithm 3 takes $O(n \cdot E \cdot K + K \cdot \log K + r \cdot n^2 \cdot K \cdot C_{\max} \cdot B_{\max})$. Since E, K, B_{max} and C_{max} are (typically small) constants, we arrive at $O(r \cdot n^2)$.

5.5 Schedulability Analysis

We now present a schedulability analysis for our multi-mode system model under a given task and resource allocation, which is used by Omni during its allocation.

Consider a multi-mode system and mode change protocol defined by our model from Section 5.3 where tasks are scheduled on their cores using EDF. The system is schedulable under an allocation (Π^m, C^m, B^m) iff it is schedulable in each mode $m \in \mathscr{M}$ and during each mode transition $r \in \mathscr{R}$ under this allocation. Before establishing the conditions for each case, let us define some notation. We denote by $\mathscr{T}_k^m$ the set of tasks that are mapped onto core k in mode m, i.e., $\mathscr{T}_k^m = \{\tau \in \mathscr{T}^m \mid \Pi_\tau^m = k\}$. In addition, D_k^m denotes the maximum deadline of all tasks on core k in mode m, i.e., $D_k^m = \max_{\tau \in \mathscr{T}_k^m} \{d_\tau^m\}$. Finally, E_τ^m and U_τ^m denote the WCET and utilization of τ in mode m, respectively. That is, $E_\tau^m = e_\tau^m(C_k^m, B_k^m)$ where $k = \Pi_\tau^m$, and $U_\tau^m = E_\tau^m / p_\tau^m$.

Mode schedulability. The schedulability of a mode m can be determined using an existing EDF schedulability test, except that each task's WCET corresponds to the resources assigned to its core in this mode. The demand bound function (DBF) of a task τ on core k in mode m is given by

$$\forall t > 0 : \mathsf{dbf}_\tau^m(t) = \lfloor \frac{t - d_\tau^m + p_\tau^m}{p_\tau^m} \rfloor E_\tau^m. \tag{5.1}$$

The DBF of core k in mode m is thus given by and

$$\mathsf{dbf}_k^m(t) = \sum_{\tau \in \mathscr{T}_k^m} \mathsf{dbf}_\tau^m(t). \tag{5.2}$$

The next theorem states the schedulability condition for mode m. Its proof follows directly from the analysis in [64].

Theorem 5.1. *The system is schedulable in mode m if for all* $1 \leq k \leq K$: $U_k^m = \sum_{\tau \in \mathscr{T}_k^m} U_\tau^m \leq 1$ *and* $\forall t < L_k^m, \mathsf{dbf}_k^m(t) \leq t$, *where*

$$L_k^m = \max\Big\{D_k^m, \frac{1}{1 - U_k^m} \times \sum_{\tau \in \mathscr{T}_k^m} (p_\tau^m - d_\tau^m) U_\tau^m\Big\}.$$

Mode transition schedulability. To determine the system schedulability during a mode transition, we first assume that the system is schedulable in each individual mode (i.e., Theorem 5.1 holds for all modes). Consider a mode transition $m' \to m$, and suppose t_0 is the instant when the transition is triggered (i.e., when its MCR arrives). Since the system is assumed to be schedulable in the old mode m', it is schedulable during the mode transition from m' to m if every core k is schedulable from time t_0 onwards. Consider any interval $[t_1, t_2]$ of length t that begins at or after the MCR instant t_0. If $t_1 > t_0$, then all jobs whose release times and deadlines are both within $[t_1, t_2]$ could only be new jobs of the tasks in m. Hence, the total demand on core k during this interval is bounded by $\mathsf{dbf}_k^m(t)$, which is at most t since the system is assumed to be schedulable in mode m. Thus, we only need to consider the case where $t_1 = t_0$, i.e., the demand generated by jobs during the interval $I_t = [t_0, t_0 + t]$ for $t > 0$.

Because all old tasks of the mode transition are dropped at the MCR instant, only tasks that are active in the new mode m contribute to the mode transition demand. Let τ be a task on core k in mode m (i.e., $\tau \in \mathscr{T}_k^m$). The demand generated by τ during the interval I_t depends on its type:

Case 1) If τ is a new task (i.e., $\tau \notin \mathscr{T}^{m'}$), then its demand can be computed using the standard DBF function, which is given by $\mathsf{dbf}_\tau^m(t)$ in Eq. (5.1).

Case 2) If τ is an existing task (i.e., $\tau \in \mathscr{T}^{m'}$), then its demand consists of (i) the demand generated by its new jobs, which are released and have (new mode) deadlines in $[t_0, t_0 + t]$, and (ii) the carry-in demand from at most one (unfinished) existing job, which was released prior to t_0 and has deadline in the interval $(t_0, t_0 + t]$, if it exists. We call such a job the *carry-in* job of the task τ.[8]

To bound the demand of existing tasks, we first consider each existing task individually and compute the total demand generated by both its carry-in job and its new jobs based on a worst-case execution scenario. We further tighten the analysis by considering the total carry-in demand of carry-in jobs that came from the same core in the old mode altogether.

A worst-case scenario that generates the maximum demand by an existing task τ is given by Lemma 5.1. An example that illustrates this scenario is shown in Figure 5.1.

[8]There can be at most one carry-in job per existing task, since the system is schedulable in the old mode and a task's deadline is no more than its period. Note also that the deadline of the carry-in job remains unchanged across a mode transition; only new jobs' deadlines follow the new mode's.

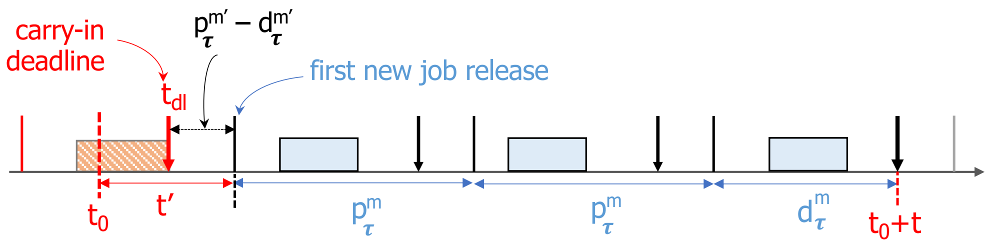


Figure 5.1: Worst-case demand scenario of an existing task.

Lemma 5.1. *The worst-case demand of an existing task τ in the interval $[t_0, t_0+t]$ happens when (i) there exists a job of τ with a deadline at t_0+t, (ii) all new jobs of τ are released as soon as possible, and (iii) the carry-in job of τ, if exists, is executed as late as possible in the old mode.*

The proof follows similar arguments as in existing EDF demand analysis; due to space constraints, we omit it here.

Under the worst-case scenario illustrated in Figure 5.1, the maximum demand of new jobs of τ in I_t is bounded by

$$\mathsf{dbf}_\tau^m(t) = \lfloor \frac{t - d_\tau^m + p_\tau^m}{p_\tau^m} \rfloor E_\tau^m \tag{5.3}$$

where p_τ^m, d_τ^m and E_τ^m are the period, deadline, and WCET of τ in the new mode m.

To compute the carry-in demand for τ, we first recall that the new resource allocation of the new mode takes effect immediately after the MCR instant; thus, the maximum time that τ's carry-in job needs to execute in the new mode m is no more than its WCET in the new mode, which is E_τ^m. This worst-case scenario can happen if τ's carry-in job is not executed at all prior to the MCR instant. In addition, under the assumption that the system is schedulable in each mode in isolation, τ's carry-in job is guaranteed to meet its deadline if the mode transition does not occur. Therefore, as Figure 5.1 illustrates, the maximum remaining execution time of τ's carry-in job if it is continued to be given the same resource allocation as in the old mode is bounded by $t_{\mathsf{dl}} - t_0 \le t' - (p_\tau^{m'} - d_\tau^{m'})$, where $p_\tau^{m'}$ and $d_\tau^{m'}$ are the period and deadline of τ in the old mode. Here, t' is the time from the MCR instant to τ's first new job release, if there exists at least one such job within the interval I_t under the worst-case scenario described above (i.e., if $t \ge d_\tau^m$); otherwise, t' is the same as t. In other words, $t' = (t - d_\tau^m) \mod p_\tau^m$ if $t \ge d_\tau^m$, and $t' = t$ otherwise.

However, since the resources allocated to τ's carry-in job may change in the new mode, its WCET can become larger than its old mode's WCET. This typically happens if τ is assigned fewer cache/bandwidth partitions in m than in m'. In that case, τ's carry-in job might need to execute an extra amount of at most $E_\tau^m - E_\tau^{m'}$ execution time units, i.e., the difference between τ's new WCET and its old WCET. Thus, depending on whether there exists a new job release for τ in the interval I_t, the maximum remaining execution time of τ's carry-in job under the new mode's allocation can be computed as follows:

Case 1) If $t \geq d_\tau^m$, then $t' = (t - d_\tau^m) \mod p_\tau^m$, and

$$E_\tau^{(m',m)} = \min\{E_\tau^m, \max\{0, t' - (p_\tau^{m'} - d_\tau^{m'})\} + \max\{0, E_\tau^m - E_\tau^{m'}\}\}.$$

Case 2) If $t < d_\tau^m$, then

$$E_\tau^{(m',m)} = \min\{E_\tau^m, t + \max\{0, E_\tau^m - E_\tau^{m'}\}\}.$$

Since the mode transition demand of an existing task τ is the sum of its new jobs' demand and its carry-in demand, its demand during the mode transition is bounded by

$$\mathsf{dbf}_\tau^{(m',m)}(t) = \lfloor \frac{t - d_\tau^m + p_\tau^m}{p_\tau^m} \rfloor E_\tau^m + E_\tau^{(m',m)} \tag{5.4}$$

By combining the demands of existing tasks and new tasks, we derive Lemma 5.2.

Lemma 5.2. *The maximum demand of a core k during a mode transition $m' \to m$ is bounded by*

$$\mathsf{dbf}_k^{(m',m)}(t) = \sum_{\tau \in \mathscr{T}_k^m \setminus \mathscr{T}^{m'}} \mathsf{dbf}_\tau^m(t) + \sum_{\tau \in \mathscr{T}_k^m \cap \mathscr{T}^{m'}} \mathsf{dbf}_\tau^{(m',m)}(t)$$

for all $t > 0$, where $\mathsf{dbf}_\tau^m(t)$ and $\mathsf{dbf}_\tau^{(m',m)}(t)$ are defined in Eqs. (5.1) *and* (5.4)*, respectively.*

The above lemma gives a means to check for schedulability during a mode transition. However, the DBF function $\mathsf{dbf}_k^{(m',m)}(t)$ so far only considers each task individually, which can be conservative. Since each mode is schedulable individually, it is possible to bound the carry-in demand of multiple tasks altogether to tighten the analysis.

Tightening the total carry-in demand. First, we observe that existing tasks on a core k in the new mode may be migrated from different cores in the old mode. We first divide the set of existing tasks on core k in the new mode m into different groups that correspond to the cores they were executing on in the old mode m'. Specifically, let $S_{i,k}^{(m',m)}$ denote the set of existing tasks $\tau \in \mathscr{T}_k^m$ such that τ was assigned to core i in mode m' (i.e., $\Pi_\tau^{m'} = i$), for all $1 \leq i \leq K$.

Then, the total execution demand of all the carry-in jobs in $S_{i,k}^{(m',m)}$ in an interval I_t, assuming that they continue with the old mode's resource allocation, must be bounded above both by the maximum of their old mode's deadlines and by t. If either condition is violated, then at least one of the carry-in jobs would miss its deadline on core i in the old mode in the absence of a mode change, which contradicts our assumption that each mode is always schedulable in isolation. The maximum extra execution time that the carry-in job might need to execute due to a change in the resource allocation in the new mode is at most $\max\{0, E_\tau^m - E_\tau^{m'}\}$. Hence, the total demand of all carry-in jobs in $S_{i,k}^{(m',m)}$ in I_t is bounded by

$$\min\{t, \max_{\tau \in S_{i,k}^{(m',m)}} d_\tau^{m'}\} + \sum_{\tau \in S_{i,k}^{(m',m)}} \max\{0, E_\tau^m - E_\tau^{m'}\}).$$

By combining the above with the DBF of new jobs of a task in $S_{i,k}^{(m',m)}$, given by Eq. (5.3), we imply that the total demand of all tasks in $S_{i,k}^{(m',m)}$ in the interval I_t is bounded by

$$\begin{aligned}\mathsf{dbf}_{k,i}^{(m',m)}(t) = \min\{t, \max_{\tau \in S_{i,k}^{(m',m)}} d_\tau^{m'}\} + \sum_{\tau \in S_{i,k}^{(m',m)}} \max\{0, E_\tau^m - E_\tau^{m'}\} \\ + \sum_{\tau \in S_{i,k}^{(m',m)}} \lfloor \frac{t - d_\tau^m + p_\tau^m}{p_\tau^m} \rfloor E_\tau^m.\end{aligned}$$

Thus, the total demand of all existing tasks in mode m on core k is bounded by the sum of the demand from all the groups i, that is $\sum_{i=1}^K \mathsf{dbf}_{k,i}^{(m',m)}(t)$. As a result, we can bound the demand of a core using the next lemma.

Lemma 5.3. *The maximum demand of a core k during a mode transition from m' to m is bounded by*

$$\overline{\mathsf{dbf}}_k^{(m',m)}(t) = \sum_{\tau \in \mathscr{T}_k^m \wedge \tau \notin \mathscr{T}^{m'}} \mathsf{dbf}_\tau^m(t) + \sum_{i=1}^{K} \mathsf{dbf}_{k,i}^{(m',m)}(t)$$

for all $t > 0$.

$$\mathsf{dbf}_{k,i}^{(m',m)}(t) = \min\{t, \max_{\tau \in S_{i,k}^{(m',m)}} d_\tau^{m'}\} + \sum_{\tau \in S_{i,k}^{(m',m)}} \max\{0, E_\tau^m - E_\tau^{m'}\}$$
$$+ \sum_{\tau \in S_{i,k}^{(m',m)}} \lfloor \frac{t - d_\tau^m + p_\tau^m}{p_\tau^m} \rfloor E_\tau^m.$$

The following theorem states the schedulability condition for a mode transition from m' to m. Its proof comes directly from Lemmas 5.2 and 5.3.

Theorem 5.2. *The system is schedulable during a mode transition from m' to m if*

(1) the system is schedulable in modes m' and m according to Theorem 5.1, and

(2) for all $1 \leq k \leq K$*:* $\min\{\overline{\mathsf{dbf}}_k^{(m',m)}(t), \mathsf{dbf}_k^{(m',m)}(t)\} \leq t$ *for all* $t > 0$.

Thus, the multi-mode system is schedulable under a given allocation if all of its mode transitions are schedulable under that allocation.

Overheads. The above analysis assumes that run-time overheads, such as task migration and resource allocation overheads, have been accounted for by inflating WCETs and DBFs. Briefly, we can account for overheads by modifying the mode schedulability test as follows: 1) adding the overhead for performing mode change actions (stop old tasks, release new tasks, modify task-to-core mapping, modify resource allocations, etc.) on each core into the core's DBF, and 2) for each task that may carry-over to an outgoing mode, inflating its WCET with the cost for reloading its working set to account for the impact of migration or resource allocation. Due to space limitations, we omit the detailed discussion.

5.6 Prototype

To evaluate Omni and to show its utility in practice, we built a prototype of Omni on top of LITMUS^{RT} [34]. In this section, we describe the key aspects of the prototype and some of the challenges we faced in our implementation.

Scheduler extension. We implemented our prototype as a scheduler plugin within the LITMUS^{RT} extension for the Linux 4.9.30 Kernel. LITMUS^{RT} modifies the Linux kernel to support various real-time task models and modular scheduler plugins. To execute a task as a real-time task on top of LITMUS^{RT}, the user can add special system calls to the target application's source code. These system calls enable the task to transition from Linux's default scheduler to the user's scheduler of choice within LITMUS^{RT}, such as EDF. The user can then loop the desired application code, where each iteration of the loop is considered as a single job instance by LITMUS^{RT}. During runtime, the plugin is responsible for keeping track of each task's real-time meta data, such as task ordering within its own run queues, execution time tracking, and relative deadline.

LITMUS^{RT} also provides other common mechanisms that our prototype utilizes, including, e.g., non-mode-change task preemptions, task migrations between Linux's run queues, and concurrency control. However, it does not natively support multi-mode execution. Hence, we extended LITMUS^{RT} to incorporate multi-mode system structures, mode-dependent task structures, and functions for executing mode change actions (such as forcefully aborting, adding, or modifying tasks).

Mode change handler. We extended LITMUS^{RT}'s system call interface to enable mode change information to be passed to our scheduler plugin and to trigger mode transitions at runtime from user space. For this, we dedicate one core to Linux for executing Linux system-related tasks and our mode change handler; we refer to this core as the management core. The remaining cores can be used to execute Litmus real-time tasks. This approach is not only essential for Linux to function but also needed to isolate the real-time tasks from Linux's potential interference. Further, to avoid task excessive creation overhead during mode transitions, we map each real-time Litmus task its own single Linux task, irrelevant of how many modes the real-time task is in.

When a mode transition occurs during a multi-mode execution, the management core preempts all Litmus cores, grabbing each of their scheduling locks to process mode change actions on each Litmus real-time task.

Using the mode change information passed in previously, along with the current mode transition number, our mode change handler determines if a task should be aborted, be added back to a core as a new mode task, have its runtime parameters updated and possibly migrated, or have its current state left unchanged. Although LITMUS^{RT} already offers existing interfaces for adding a task to and removing a task from its appropriate scheduling queues, these actions can only happen during specific times of a task's lifetime. For example, existing scheduler implementations only permit removing tasks from the ready queue (i.e., a queue used for tasks that have had a job already released and is ready for execution). In contrast, since an MCR can arrive at any time, we need to perform mode change actions whenever the MCR arrives. This asynchronous nature of MCRs created many new race conditions that we needed to handle carefully. One example is when a task finished its previous job and committed the next job to be released in the future, there is no existing mechanism in LITMUS^{RT} to cancel this pending release until after the task is put into the ready queue. Our mode change handler is able to avoid the race conditions involved with modifying a task's state regardless of which state the task is currently in (e.g., even if the task might have just been setup for a new release and will only be moved to the ready queue much later). To accomplish this with minimal overheads and with minimal modifications to the existing LITMUS^{RT} system, we employed a lazy removal technique in which tasks that are unable to be modified or removed from our scheduler plugin at mode change instant are flagged as pending for removal. When the task is eventually moved into the ready queue upon release, we check its pending removal flag and remove the task at that point if the removal flag is set.

Job reset. Another limitation of LITMUS^{RT} is that it has no way to control the progress of the user-level task that is executing for each job loop. Ideally, when a task is scheduled, it will finish its workload and return to the top of the loop where it will sleep until LITMUS^{RT} schedules its next job release. If a task misses its deadline, however, LITMUS^{RT} has no mechanism to reset the task back to the top of the loop such that its next job release will begin at the start of the application. We remedied this by adding a system call at the beginning of each application, immediately before the job loop. This system call takes a snapshot of each task's registers, which is reloaded at the beginning of each job loop iteration. This enables us to perform a reset of the task's state, even when its previous job never finished (e.g., because the job missed its deadline, or because it was aborted in a previous mode change). In total, our scheduler plugin contains about 2000 lines of code, including our mode change handler and all LITMUS^{RT} modifications.

Resource control. After performing mode change actions on each real-time task, the mode change handler releases each Litmus core's scheduling plugin lock. Each core immediately applies its resource allocation for the new mode, before picking the next task to execute. For cache and memory bandwidth partitioning, we integrated Intel's Cache Allocation Technology (CAT) [95] and Memguard [224] into our extension of LITMUS^{RT}, respectively. CAT divides the last-level cache into partitions, which can be allocated to cores using special model-specific registers (MSRs) on certain Intel CPUs. The number and size of these partitions are CPU specific and are based on the number of set-associative ways. Using bit masks, a user can set a class of service register (COS) to have a specific mapping of contiguous partitions. Cores are associated with a specific COS through the use of another per-core register (PQR) that specifies which COS should be used for that core. CAT enforces the property that all new last-level cache allocations from a logical core are made only to a way specified in the bitmask of that core's COS.

Memguard is a software-based technique to control the amount of memory requests a core can make within a configurable time window. Through a kernel module, a core is allocated a total bandwidth in terms of MB/s. If too many requests are made before the core's budget is refreshed, which is tracked through a CPU performance counter, then the core is preempted and a spinning thread is scheduled to throttle the core until the next replenishment period. Memguard does not work with LITMUS^{RT} out of the box because the throttling thread runs on Linux's normal scheduler, which has lower priority than any of Litmus's plugins. This means that if a real-time task is executing and Memguard wakes up, the throttle thread will never be chosen to execute. We fixed this by making the throttle thread a "real-time task" that is controlled by our scheduler. When too many memory requests are made, Memguard preempts our scheduler, which sees a per-core bit signaling that the core should be throttled. We then manually schedule the throttle thread until another interrupt disables the bit at the next Memguard replenishment time.

Note that CAT and Memguard provide the mechanisms for controlling shared cache and memory bandwidth throughput only; they do not solve the question of how to achieve an effective shared resource allocation, which our resource allocation algorithm focuses on.

5.7 Evaluation

To evaluate the effectiveness and applicability of Omni, we conducted an extensive set of experiments, both numerically and experimentally on real multicore hardware. Our key questions were: (1) What is the run-time overhead of Omni? (2) Can Omni indeed improve schedulability and resource use efficiency compared to state-of-the-art solutions? And (3) How well does Omni scale to the system size?

5.7.1 Experimental Setup

Algorithms for comparison: Since we were not aware of any existing solution for the shared cache, bandwidth, and task allocation for multi-mode systems, we compared Omni against two baseline solutions, MM-Static and CaM, described in Section 5.3. MM-Static extends the task-to-core partitioning method for multi-mode systems from [55] with a static, even distribution of cache and bandwidth to cores. (As we wanted to evaluate overloaded scenarios as well, we made a slight modification to the original algorithm in [55] in our implementation: if a taskset is unschedulable, instead of reporting failure, MM-Static schedules the task on the core with the lowest utilization.) CaM applies the holistic multicore resource allocation technique from [213] to compute the task mapping and resource allocation for each mode individually. These two baseline solutions are representative of the state of the art in multicore multi-mode system scheduling and holistic multicore resource allocation, respectively.

Experimental platform. Our prototype ran on a CAT-capable Intel Xeon E5-2683 v4 processor with 16 cores and a 40MB 20-way set-associate L3 cache that is divided into 20 partitions ($C_{\max} = 20$). The machine also has three single-channel 16GB PC-2400 DDR4 DRAM sticks. Using the method from [224] we measured a maximum guaranteed bandwidth of 1.4 GB/s, which we divided into 20 partitions of 70MB/s each ($B_{\max} = 20$). While this is lower than the peak bandwidth that the platform supports, it results in much better isolation between the cores. To avoid nondeterministic timing, we disabled Intel SpeedStep, hyperthreading and hardware prefetching. We performed both the WCET profiling and experimental evaluation on this platform. For our experiments, we configured the platform to use 4 cores, 12 cache partitions, and 12 bandwidth partitions, as this is a common core and cache configuration for CAT-enabled CPUs. Our numerical evaluation additionally considered a larger platform, with 8 cores and 20 cache/bandwidth partitions, to evaluate the effect of platform configurations on schedulability performance.

Workload. Since real traces for multi-mode multicore systems are usually proprietary and we are not aware of any public ones, we combined real applications (from benchmarks) with synthetic utilizations/periods for our evaluation workload.

We created a generation tool that can produce synthetic multi-mode systems with many different configurable parameters. For example, we can specify: the number of modes, the probability of mode transitions, the distribution of task utilizations, the probability of a task being an existing, new, or changed task across a mode transition, etc.

Following the approach outlined in [212], we randomly picked tasks (programs) from several different benchmarks, including PARSEC, SPLASH2x, DIS and IsolBench). We obtained a total of 11 different benchmark tasks, thus forming a diverse set of tasks with varied resource requirements. All these benchmarks support tasks with a single threaded execution mode, which we used in our evaluation.

We profiled each task program in the Omni prototype running on our experimental platform, where we ran the task under all possible cache and bandwidth configurations ($20 \times 20 = 400$ configurations in total). The collected WCET values were then used for our analysis. We used the WCET when the task is allocated the entire cache and the entire memory bus as its *reference WCET* for our taskset generation.

We generated tasksets with taskset (reference) utilizations ranging between 1.0 and 4.0, at steps of 0.1. (Note that, since a taskset's reference utilization assumes that each task is given the entire cache and memory bus, it corresponds to a much larger actual utilization when cache and bandwidth resources are partitioned among tasks.) For each taskset utilization, we generated 400 independent tasksets per mode, for a total of 24,800 tasksets per experiment in a system with 2 modes. Tasks' utilizations fall within the range of either [0.01, 0.4] or [0.4, 0.9]. The probability of selecting one over the other is determined by one of three different distributions: $[\frac{8}{9}, \frac{1}{9}]$, $[\frac{6}{9}, \frac{3}{9}]$ and $[\frac{4}{9}, \frac{5}{9}]$. We refer to first, second, and third distributions as light, medium, and heavy distributions, respectively. Each task's period (deadline) was set to be the ratio of its reference WCET to its utilization.

5.7.2 Run-Time Overhead

To evaluate the overheads introduced by our prototype, we collected micro-benchmarks for the tasksets that we generated above. We ran each taskset in our prototype on our experimental platform. We used the timestamp counter to track the time for a variety of functions and saved the timestamps into memory to be reported after an experiment had finished. In total, we collected micro-benchmarks of over 500 mode changes for the following functions:

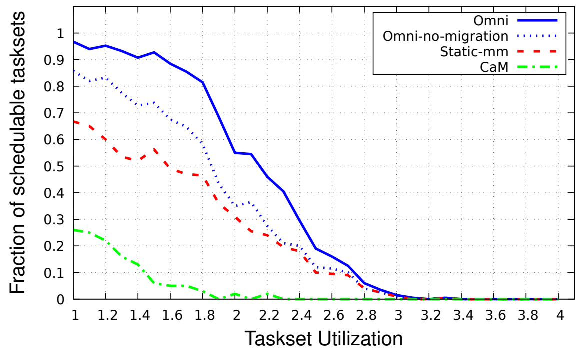
- `sched()`, the main scheduling function;
- `cache()`, setting CAT model-specific registers to configure cache allocation;
- `membw()`, setting Memguard bandwidth;
- `remove()`, mode change action to remove a single old mode task;
- `insert()`, mode change action to insert a single new mode task;
- `update()`, mode change action to update a single changed mode task;
- `MCR()`, full mode change handler for all tasks; and
- `Reset()`, resets task registers for a new job release after old mode removal or missed deadline.

We took measurements over the course of 4 hours (the time it took to run our experimental platform through 500 mode changes) and report the average.

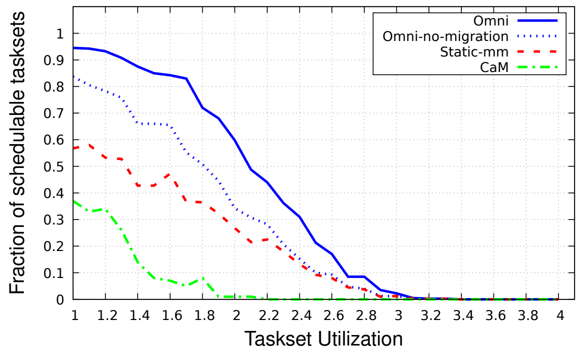
The overhead results are summarized in Table 5.1, where all times are averaged across the measured values and rounded to the nearest nanosecond. The results show that Omni incurs negligible scheduling overhead, and that it introduces only a small overhead for the mode change actions and cache/bandwidth allocations. The full mode change handler for all task (MCR) overhead has the largest overhead, which is expected since it covers the complete handling of a mode change.

Sched	Cache	Membw	Remove	Insert	Update	MCR	Reset
179	2362	7736	364	665	775	46615	772

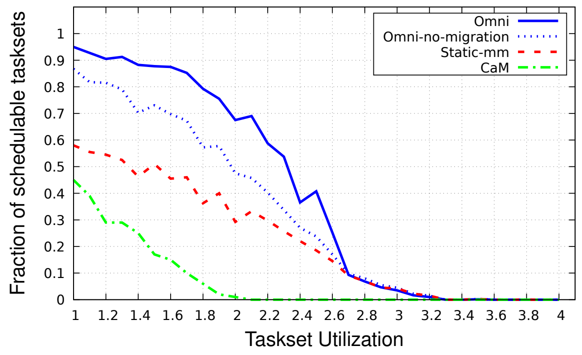
Table 5.1: Scheduling and mode-change overheads (in ns).



(a) Bimodal Light



(b) Bimodal Medium



(c) Bimodal Heavy

Figure 5.2: System schedulability under different taskset distributions (2 modes, 4 cores).

5.7.3 Numerical Evaluation

Schedulability performance. In this first experiment, we used a platform with 4 cores and 12 cache/bandwidth partitions, which resembles our configured experimental platform. We generated multi-mode systems with 2 modes, with a 20% probability of tasks migrating from one mode to the next, and a 50% probability of a migrated task having its parameters updated to a new value within the same utilization distribution. Each mode is associated with a taskset taken from the generated tasksets, as described earlier.

For each multi-mode system, we performed resource allocations and schedulability analysis under 4 different algorithms: 1) Omni, our proposed resource allocation algorithm (Section 5.4); 2) Omni-no-migration, a variant of Omni resource allocation algorithm but with only resource redistribution implemented, to evaluate how much migrations and resource distributions help separately; 3) MM-Static; and 4) CaM. For both Omni and Omni-no-migration, we set $R = 10$, $r = 30$, and the load score threshold to be 0.01.[9]

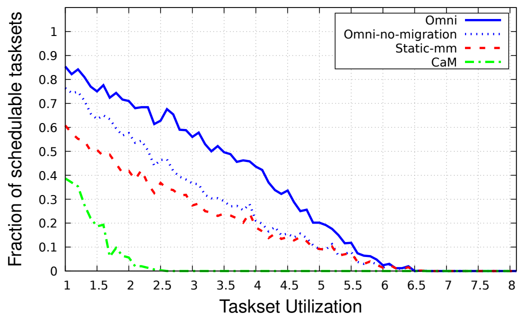


Figure 5.3: Schedulability of 8-core systems (2 modes).

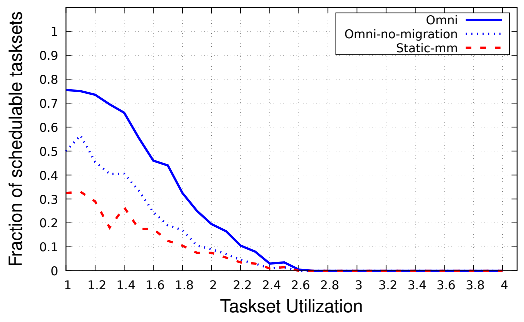


Figure 5.4: Schedulability of 4-mode systems (2 cores).

Results. Figure 5.2 shows the fraction of schedulable tasksets with respect to different taskset reference utilizations for each of the three task utilization distributions. We can make the following observations:

- CaM performs extremely poorly across all three utilization distributions. At reference utilization 1.0, it fails to schedule the majority of the tasksets. For example, only 26% of the tasksets are schedulable under light distribution and 44% of the tasksets are schedulable under heavy utilization distribution. This further confirms the importance of considering mode transitions when computing resource allocation.

- MM-Static performs consistently better, but it is still much worse compared to Omni and its simplified version Omni-no-migration.

- Both Omni and its simplified version without migration outperform the two baseline solutions by a significant factor. In particular, Omni can schedule up to 2× more tasksets compared to MM-Static, which is the better performed state-of-the-art solution.

- Omni also consistently performs better than Omni-no-migration, which shows that task migration does help substantially in improving schedulability. In addition, the difference between Omni-no-migration and the baseline solutions also show that task Omni's redistribution approach alone can already improve performance compared to the state of the art.

[9]This threshold value is small enough to capture the minimum amount of improvement that Omni could yield by redistributing a single resource partition or by a single task move/swap. A smaller value would lead to little improvement, whereas a larger one would make the algorithm stop too soon.

Impact of platform configurations. To evaluate how well Omni scales to the platform size, our next experiment considered a larger platform configuration with 8 cores and 20 cache/bandwidth partitions. We generated multi-mode systems as before, except that the taskset utilization ranges from 1.0 to 8.0 (with steps of 0.1). We performed resource allocation and schedulability analysis for all algorithms.

Figure 5.3 shows the schedulability results of the three algorithms Omni, Omni-no-migration, and MM-Static. We can observe that, as we double the number of cores and increase the number of partitions, Omni maintains a similar performance improvement factor over Omni-no-migration and MM-Static. The results also demonstrate the same relative performance among the three algorithms, as well as the positive impacts of both task redistribution and task migrations on schedulability.

Impact of multi-mode system size. Our last experiment evaluated the algorithms as we increased the number of modes. We considered the same platform configuration as in the first experiment (4 cores, 12 partitions), but generated multi-mode systems with twice as many modes (4 modes). Each mode always has a mode transition to the next immediate mode (Mode 0 $\rightarrow$ Mode 1 $\rightarrow$ Mode 2 $\rightarrow$ Mode 3 $\rightarrow$ Mode 0), as well as an additional 50% probability of having a mode transition to any other mode. We chose to exclude CaM from this analysis since it significantly under-performed in our schedulability evaluation with just 2 modes.

The results are shown in Figure 5.4. We observe a drop in schedulability across all three algorithms compared to that of the two-mode systems (c.f. Figure 5.2); this is expected, because when the number of modes and mode transitions increase, the system will also become harder to schedule. However, both Omni and Omni-no-migration continue to perform substantially better than the state-of-the-art algorithm MM-Static, and their improvement factors also increase with more modes.

Summary. Our evaluation demonstrates that the insights and strategies Omni employs are highly effective in improving schedulability and resource use compared to the state of the art. This schedulability improvement factor also maintains as we scale the platform and multi-mode system size.

5.7.4 Experimental Evaluation

For our experimental evaluation, we ran a subset of the generated multi-mode systems from the first numerical evaluation on our prototype. We configured our experimental platform to have 4 cores and 12 cache/bandwidth partitions. For each multi-mode system, the amount of time we waited between each mode transition is a random value that falls in the range of the maximum deadline in the whole taskset plus 20%-80%. Each core used for our experiment was fully isolated from the Linux kernel to the best of our ability, and every core not involved in running an experiment received a small amount of memory bandwidth and its own isolated cache partition.

To begin an experiment, we passed the multi-mode system to LITMUS^{RT} through a custom system call, so that LITMUS^{RT} knows a task's parameters for all modes. We then launched our experiment from a management program that was pinned to the management core (unused by LITMUS^{RT} for running real-time tasks). This management program forks a unique instance of each task in the multi-mode system, which is then transformed into a real-time LITMUS^{RT} task before executing its workload. After all tasks have transitioned to our LITMUS^{RT} scheduler, we launched a special setup mode transition which will abort any tasks that do not belong to our initial mode, while applying the proper task parameters to tasks that do. Once this initial mode transition completes from our management program, the experiment is considered to have started and the management task will sleep itself until the next mode transition. For each multi-mode system, we repeated the same experiment using two different resource allocation configurations: the resource allocation solution produced by Omni, and the one produced by MM-Static. We omit the experiment for CaM, as it performs poorly compared to all other algorithms.

Results. Our measurement results further confirm the relative performance between Omni and MM-Static. Specifically, the results show that for the same taskset utilization, Omni is able to improve both (observed) schedulability and reduce job deadline miss ratio substantially. For instance, at taskset utilization of 2.0, out of 36 tasksets we ran, MM-Static experienced 3.79$\times$ more jobs missing their deadlines, and 1.53$\times$ more multi-mode systems missing their deadlines compared to Omni. The results demonstrate that Omni can arrive at a much better resource and task allocation than existing solutions.

5.8 Conclusion

In this chapter, we have introduced Omni, the first end-to-end multi-mode real-time resource allocation algorithm that is able to dynamically adjust shared resources and task allocation to more optimally ensure schedulability during mode transitions. Omni contributes a novel multi-mode resource allocation algorithm and a resource-aware schedulability test that supports general mode-change semantics as well as dynamic cache and bandwidth resource allocation. Through our prototype and analysis implementations, we have demonstrated that Omni is able to outperform existing state-of-the-art solutions both numerically and on a real platform by a significant factor.

CHAPTER 6

DECNTR: OPTIMIZING SAFETY AND SCHEDULABILITY WITH MULTI-MODE CONTROL AND RESOURCE ALLOCATION CO-DESIGN

Building on the system-level adaptation of Chapter 5, we now turn to cyber-physical systems where timing guarantees alone are not enough: controllers must ensure that the physical plant remains stable, even as workloads shift and modes change. Our previous chapters assumed fixed controller (task) implementations and then attempted to schedule them. In practice, however, a variety of controllers can be synthesized to serve as valid implementations of a single task.

This chapter explores how resource allocation and control design can be co-optimized to ensure not just system schedulability but also to maximize system *robustness*. It presents the DECNTR framework, which allows controllers to expose multiple viable implementations (e.g., different sampling periods) and jointly selects both controller variants and resource budgets. A new mode-change protocol ensures that transitions respect not only schedulability constraints but also the safety conditions of the physical system. By combining robustness from control with flexibility in allocation, DECNTR demonstrates that safe and predictable execution can be preserved even under dynamic workloads.

In Section 6.1, we motivate the interplay between scheduling, allocation, and robustness in cyber-physical systems. Section 6.2 briefly covers related work, while Section 6.3 introduces our model and mode-change protocol assumptions. Sections 6.4 and 6.5 then present our co-design approaches to controller design and resource allocation, respectively. The final sections evaluate DECNTR 's ability to maintain safety, timeliness, and robustness under dynamic workloads through schedulability analysis and a case study.

This chapter is based on work that first appeared as: Robert Gifford, Felipe Galarza-Jimenez, Linh Thi Xuan Phan, and Majid Zamani. DECNTR: Optimizing safety and schedulability with multi-mode control and resource allocation co-design. In *RTAS*, 2024.

6.1 Introduction

Modern cyber-physical systems (CPS), such as self-driving vehicles and autonomous marine systems, are inherently adaptive. A self-driving car, for instance, needs to execute different control tasks depending on detected obstacles, road conditions, or potential hardware/software faults. A standard way to model and analyze such systems is to use *multi-mode formalism*, where each mode represents a system configuration (composed of tasks that are active in the mode), and each transition represents a switch from one configuration to another in response to a mode-change event (such as a detected obstacle).

Guaranteeing the safety, robustness, and timeliness of multi-mode CPS on multicore platforms is highly challenging. On the one hand, the control design must ensure that the output (or state) of the plant controlled by each task is inside a safe region; on the other hand, the scheduling and resource allocation of the control tasks must ensure schedulability in each mode and during each mode transition. Achieving these two (contradictory) goals concurrently is difficult, especially when the set of control tasks – and thus the system load – changes as the system moves from one mode to another.

One reason for this difficulty is that, to enable implementation on a platform, the controller (implemented as a control task) is designed for a particular sampling period (which is used as the task period). The controller guarantees the safety constraint only at the sampling instances, whereas the behavior at inter-sampling intervals might be unsafe due to plant dynamics or external disturbance. Fig. 6.1 illustrates this situation: the inner rectangle denotes the safe region X_S, the shaded rectangle denotes all possible plant states X, the curve represents the state during a simulation time interval, and the dots on the curve represent the state at sampling times. As highlighted in this figure, while the state at sampling times is always inside the safe region X_S, certain parts of the curve (highlighted in red) are outside X_S. Therefore, a goal of our work is also to increase the *robustness* of the controlled system – namely, the system's ability to remain safe even in the presence of slight disturbance affecting the state measurements at the sampling instant. In other words, the more robust a controller is, the more likely that the state is also inside X_S during inter-sampling intervals.

One way to increase robustness is to reduce the size of X_S or to impose a bound on the possible external disturbances, but this also reduces the controller's capability. For this reason, reducing the sampling period to

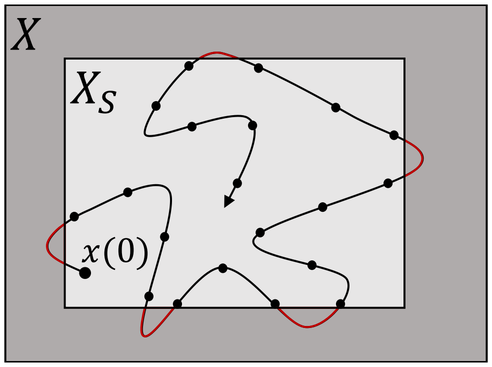


Figure 6.1: The plant is connected to a feedback controller that guarantees the invariance of X_S at sampling time (denoted by dots on the curve) but not during inter-sampling intervals.

reduce the uncertainty about the plant's inter-sample behaviors is a common choice (see [195] and references therein). However, a smaller period also makes a task harder to schedule. During a mode transition, this problem is exacerbated because one needs to schedule not only jobs released in the new mode but also unfinished jobs from the old mode. Unfinished jobs of tasks that continue in the new mode are hard to handle. In the worst case, they need to be executed immediately to meet their deadlines, at the cost of delaying the execution of new tasks, which can lead to deadline misses.

Multi-mode systems have been studied extensively in the real-time community, but the focus is mostly on schedulability analysis (particularly during mode transitions), and existing work generally considers a single core. In the last chapter, Omni [77] presented an end-to-end task mapping and resource allocation solution for multi-mode systems on multicore platforms. By dynamically adjusting the mapping of tasks to cores and the allocation of shared resources (such as cache and memory bandwidth) to cores at mode transitions, it can improve schedulability substantially. However, Omni *completely ignores control aspects* and assumes instead that the sampling period for each task in each mode is given a priori. Thus, if the task periods need to be small to avoid safety violations during inter-sampling times (to enhance robustness). it may not be able to schedule the tasks. In addition, to minimize the number of re-allocations during transitions, Omni works by finding an allocation for each mode that is as similar to one another as possible. Consequently, its allocation may not be optimal for any individual mode.

In this chapter, we present a co-design approach to achieving safety, robustness, and timeliness for multi-mode CPS on multicore platforms. Our insights are twofold: First, given a control task, it is possible to design safety conditions for switching between different controllers (implementations), which may have different sampling periods. Having multiple implementations under different sampling periods enables the resource allocation to determine the 'best' period for each task in each mode to maximize schedulability and robustness, while still guaranteeing safety. Second, it is possible to 'delay' the deadline of an unfinished job (or even to kill them) without compromising safety, as long as the extended deadline is within one of the safe periods of the controller used. In other words, we can relax the *periodicity* requirement for continuing tasks, often imposed in existing work (including Omni). This relaxation helps reduce the load during mode transitions, which not only increases schedulability but also enables us to use allocations that are ideal for each mode individually, thus further improving resource efficiency.

To realize this approach, we introduce DECNTR, a concrete co-design method for co-optimizing safety, robustness, and schedulability. At the core, DECNTR consists of a control design technique for developing safe controllers that can switch between different implementations (sampling periods) and that can accommodate a certain delay of unfinished jobs during a switch. DECNTR resource allocation algorithm intelligently exploits these properties to determine, for each mode, a sampling period for each task, a mapping of tasks to cores, and an allocation of cache and memory bandwidth to each core so as to maximize schedulability while minimizing periods (to increase robustness). This is done in tandem with delaying unfinished jobs, if necessary, to improve schedulability during mode transitions. By co-designing the controller and resource allocation this way, DECNTR substantially increases schedulability and robustness compared to the state of the art, while guaranteeing safety. As a side benefit, DECNTR resource allocation can be used for optimizing real-time performance by adapting the best periods, or allowing graceful degradation of service as well.

In summary, this chapter makes the following contributions:

- A co-design approach for jointly developing controllers and resource allocation algorithms to maximize safety, robustness and schedulability.
- A set of controllers that guarantees the CPS's safety under known sampling-time variations and transitions between two different control implementations (Section 6.4).
- A novel multicore task and resource allocation algorithm that guarantees safety while maximizing robustness and resource efficiency (Section 6.5).

Our evaluation using a CPS case study and real-time benchmarks shows that DECNTR can substantially improve robustness while increasing schedulability by up to $11\times$ compared to the state of the art.

6.2 Related Work

There is a large body of work on multi-mode systems. Prior work in this area often focuses on one of two key areas: 1) new models and timing analysis techniques (see e.g., [29, 30, 78, 8, 150, 180, 146, 147, 127]), and 2) mode-change protocols for ensuring schedulability during mode transitions (e.g., see [28, 156] and references therein). Recently, multi-mode scheduling and analysis have been extended towards multiprocessors [79, 135, 55, 16, 155, 6]. The majority only considers CPU, but some recent work, like Omni [77], considers shared resources [134, 107].

Several multi-resource and task co-allocation techniques have been developed. For example [213, 212] propose holistic resource allocation techniques that find the assignments of tasks, cache and memory bandwidth to cores. DNA/DADNA [76] does the same dynamically at run time for soft real-time tasks but also does not consider multiple modes. These techniques, however, focus on single-mode systems.

Safety controllers are extensively studied for CPS applications. One particular approach is the design of so-called barrier functions [152, 7, 98]. This method is applicable to systems with nonlinear dynamics and circumvents the direct computation of the *controlled invariant set*, which delineates the region within which the system can maintain safety. Nonetheless, a level-set of barrier functions, which constitutes the controlled invariant set, can be overly conservative due to the inherent conservatism of barrier functions. Other research concentrates on systems with linear dynamics [21, 25, 26], which can compute the maximal controlled invariant set via operations over sets.

Lastly, there is extensive research in control and scheduling co-design. The result in [12] is one of the first to introduce the concept of modifying a control task's sampling period for the sake of improving CPU efficiency. Since then, there has been a large collection of techniques [71, 170, 160, 167, 31, 233, 181, 46, 169, 81, 49, 183, 184] that apply co-design principles to resource allocation, to scheduling, or to multi-mode systems. However, to the best of our knowledge, no prior work has considered a holistic co-design of multi-mode controllers with multi-mode allocations of tasks and shared resources (such as cache and memory bandwidth) on multicore platforms.

6.3 System Modeling and Mode-Change Protocol

6.3.1 Multi-Mode System and Platform Modeling

Platform. The system is deployed on a multicore platform consisting of r identical cores that can access a shared cache and shared memory bandwidth (BW). The cache and BW are divided into $C_{\max}$ and $W_{\max}$ equal-size partitions, respectively. At run time, distinct sets of cache and BW partitions are distributed to cores, and each core has exclusive access to its allocated partitions. Cache and BW allocation can be performed using existing mechanisms, such as Intel's CAT [95] for cache and MemGuard [224] for BW.

System model. We consider a multi-mode system defined by $\{\mathscr{M}, \mathscr{R}, m_0, \mathscr{T}\}$, where $\mathscr{M}$ is the set of modes, $\mathscr{R} \subseteq \mathscr{M} \times \mathscr{M}$ is the set of transitions, $m_0 \in \mathscr{M}$ is the initial mode, and $\mathscr{T}$ is the set of control tasks. Each mode m is associated with a set of periodic tasks $\mathscr{T}^m \subseteq \mathscr{T}$ that are active in m. Each transition (m', m) is triggered by a mode-change request event (MCR), upon which the system moves from mode m' to mode m by executing a mode-change protocol (described below). We assume that when an MCR occurs, another MCR can only occur after the system has completely moved to the new mode. Within a mode, each active task is mapped onto a core, and it remains on the core while the system is in this mode. Tasks on the same core are scheduled under the Earliest Deadline First (EDF) policy.

Task model. Each task τ_i in the system controls a particular physical plant of the CPS. It is associated with a set of q_i controllers (implementations) $\mathscr{K}_i = \left\{\mathscr{K}_{i,j} \mid 1 \leq j \leq q_i\right\}$, from which it can pick to execute in a mode. Each controller $\mathscr{K}_{i,j}$ has a safe range of sampling periods, $[\rho_{i,j}, \rho_{i,j}]$, and the task can switch between any two sampling periods in this range that are multiples of ρ without compromising safety, where $\rho > 0$ is a configurable control parameter. We denote by $\rho_{i,\min} = \min_{j=1}^{q_i} \rho_{i,j}$ and $\rho_{i,\max} = \max_{j=1}^{q_i} \rho_{i,j}$ the minimum and maximum safe period of τ_i, respectively. In each mode m where τ_i is active, our resource allocation algorithm will assign a specific controller $\mathscr{K}_i^m \in \mathscr{K}_i$ and a period value p_i^m in the period range of $\mathscr{K}_i^m$ for execution. For simplicity, we assume the task's deadline is the same as its assigned period; it should be straightforward to extend our algorithm to the case where deadlines are smaller than periods.

We consider a resource-aware task model where a task's worst-case execution time (WCET) depends on the resource it is given. Specifically, $e_i(c, w)$ represents the WCET of τ_i when it is given c cache partitions and w BW partitions, for all $1 \leq c \leq C_{max}$ and $1 \leq w \leq W_{max}$.

Mode-change protocol design. To define the execution behavior during a mode transition from m' to m, we distinguish three types of tasks associated with the transition: (i) *old* tasks are active in m' but not in m; (ii) *new* tasks are active in m but not in m'; and (iii) *carry-over* tasks are active in both m and m'. If a carry-over task has an unfinished job, we call the job a carry-over job during the mode transition.

We drop old tasks (and their unfinished jobs) at the MCR instant. However, the mode-change semantics for carry-over and new tasks are designed to leverage the capability of our controllers in tolerating certain delay for the next actuation (sampling) time during a mode transition and in allowing more than one sampling period.

Formally, for each transition (m',m), we determine for each task τ_i in mode m whether we need to delay the deadline (within a safe range) of its first job to complete after the MCR (to improve schedulability). This could either be its carry-over job (if exists) or the first job that τ_i releases in mode m. If we do, we mark the task as *delayable* by setting $\mathsf{del}_i(m',m) = 1$, and compute a corresponding 'delayed' deadline. If τ_i has a carry-over job, then the delayed deadline must be a safe period value for $\mathscr{K}_i^{m'}$ (the controller used in the old mode); otherwise, the delayed deadline must be a safe period value of $\mathscr{K}_i^{m}$. Hence, we will associate with each task τ_i in each transition $(m',m) \in \mathscr{R}$ a delayed deadline $d_i(m',m)$ that may be applied to the first job τ_i releases in mode m, for all $\tau_i \in \mathscr{T}^m$. In addition, we also compute a delayed deadline $d_i^c(m',m)$ for the carry-over job of each carry-over task τ_i of the transition.

Mode-change semantics. When a mode transition (m',m) is triggered, we execute the following mode-change actions:

Step 0) Drop all old tasks, including their unfinished jobs.

Step 1) For each new task, release its first job immediately.

Step 2) For each carry-over task that has no carry-over job, release the next job at p_i^m time units after its last job release.

Step 3) For each task τ_i in m that is delayable, if it has a carry-over job, we set the deadline of the carry-over job to be $d_i^c(m',m)$; otherwise, we set the deadline of its first job in m to be $d_i(m',m)$. All subsequent jobs are released and assigned deadlines according to the task's assigned period p_i^m. For all tasks τ_i that are not delayable, all jobs will follow the periods determined by the mode in which they are released.

In the new mode m, the tasks and resources allocated to each core may change. We assume that any such change take effect immediately, including for carry-over jobs.

6.3.2 Controller Design

As discussed earlier, each task in a mode implements a feedback controller that controls a physical plant. We now discuss our control model, starting with some necessary notation.

Given sets X and Y, the projection map of X onto Y is denoted by $\Pi_Y(X)$. As usual, $\mathbb{N}, \mathbb{Z}, \mathbb{Z}_{\geq 0}, \mathscr{R}, \mathscr{R}_{>0}$ and $\mathscr{R}_{\geq 0}$ denote the set of natural, integer, nonnegative integer, real, positive, and nonnegative real numbers, respectively. Notations $[a,b]$, $]a,b[$, $[a,b[$ and $]a,b]$ denote closed, open, and half-open sets in $\mathscr{R}$. Likewise, $[a;b]$, $]a;b[$, $[a;b[$ and $]a;b]$ denote closed, open, and half-open sets in $\mathbb{Z}$. Thus, $[a,b] \cap \mathbb{Z} = [a;b]$.

Given sets A and B, $f : A \rightrightarrows B$ denotes a *set-valued map*, whereas $f : A \rightarrow B$ denotes a *single-valued map* (i.e., a *function*). We denote the identity map by $\mathscr{I}$. The set of maps from Y to X is denoted by X^Y. The set of all signals with image on X defined on intervals $[0;T[$ is denoted by $X^{[0;T[}$. Finally, $X^\infty = \bigcup_{T \in \mathbb{Z}_{\geq 0} \cup \{\infty\}} X^{[0;T[}$.

We consider plants as linear control systems evolving in continuous-time, as defined below.

Definition 6.1 (ct-LTI). *A linear control system is described by:*

$$\dot{\xi}(t) = A_c \xi(t) + B_c \nu(t), \tag{6.1}$$

where $A_c \in \mathscr{R}^{n \times n}, B_c \in \mathscr{R}^{n \times d}$, $\xi(t) \in \bar{X} \subseteq \mathscr{R}^n$, and $\nu(t) \in \bar{U} \subseteq \mathscr{R}^d$ for all $t \in \mathscr{R}_{\geq 0}$. ξ and ν are called state and input trajectories of the system, respectively. Given $\rho \in \mathscr{R}_{>0}$ and interval $I \subseteq [0, \rho]$, a solution of (6.1) *on I under input ν is defined as an absolutely continuous function $\xi : I \rightarrow \bar{X}$ whose time derivative $\dot{\xi}(t)$ satisfies* (6.1) *for almost every $t \in I$.*

To enable implementation on a digital platform, we consider a sampled-and-hold version of the plant's model for the sake of design as introduced below.

Definition 6.2 (dt-LTI)**.** *A discrete-time linear control system associated with the control system in* (6.1) *and sampling time* $\rho > 0$ *is a tuple*

$$\mathscr{S} = (X, X_0, U, A, B, \rho), \tag{6.2}$$

where $X = \bar{X}$ *is the state set,* $U = \bar{U}$ *is the input set,* $X_0 \subseteq X$ *is a set of initial states, and the evolution of the system is described as:*

$$x(t+1) = Ax(t) + Bu(t), \tag{6.3}$$

with $A = e^{A_c \rho}$ *and* $B = \int_0^\rho e^{A_c t} dt \cdot B_c$*.*

A tuple $(\mathbf{u}, \mathbf{x}) \in U^{[0;T[} \times X^{[0;T[}$ is a *solution* of the system in (6.2) over $[0;T[$ if for $T \in \mathbb{N} \cup \{\infty\}$, (6.3) holds $\forall t \in [0;T-1[$ and $\mathbf{x}(0) \in X_0$.

Here, for a given safety set $X_S \subseteq X$, we are interested in designing feedback controllers $\mathscr{K} : D \rightrightarrows U$, for some $D \subseteq X_S$, forcing solutions of (6.3) to evolve within $U^\infty \times D^\infty$, i.e. $x(t) \in D$ for all $t \in \mathbb{N} \cup \{\infty\}$, where $u(t) \in \mathscr{K}(x(t))$. In particular, we denote the set of such controllers by $\bar{\mathscr{K}}(U)$. Such a family of controllers is characterized by the maximal controlled invariant set contained in X [21, 25, 26, 201], formally defined as follows.

Definition 6.3. *Consider a safety set* $X_S \subseteq X$*. A set* $R \subseteq X_S$ *is called controlled invariant w.r.t.* (6.3) *and* U*, if there exists a feedback controller* $\mathscr{K} \in \bar{\mathscr{K}}(U)$ *so that every solution* $(\mathbf{u}, \mathbf{x})$ *of* (6.3) *with initial state* $x(0) \in R$ *and* $\mathbf{u} \in \mathscr{K}(\mathbf{x})$*, evolves in* $U^\infty \times R^\infty$*. Moreover, we denote the maximal controlled invariant set of* (6.3) *and* U *as* $R(X_S)$*.*

We use the following iteration for the computation of $R(X_S)$ as proposed in [21]:

$$R_0 = X_S, \quad R_{i+1} = \text{pre}(R_i) \cap X_S, \tag{6.4}$$

where $\text{pre}(R) = \{x \in X \mid \exists\, u \in U \text{ s.t. } Ax + Bu \in R\}$. If it exists, the fixed point of the iteration is the set $R(X_S)$.

6.3.3 Problem Formulation

Our first goal is to design a set of safety controllers (see Fig. 6.2) for each plant. Specifically, given a plant as in Definition 6.1, for a given $\rho > 0$ and dt-LTI $\mathscr{S} = (X, X_0, U, A, B, \rho)$ associated to the plant, and a *safety* set $X_S \subseteq X$, we aim to develop a set of feedback controllers of the form $\mathscr{K} : R(X_S) \rightrightarrows U$ guaranteeing that for any $\mathbf{u} \in \mathscr{K}(\mathbf{x})$, $(\mathbf{u}, \mathbf{x}) \in U^\infty \times X_S^\infty$ (see Definition 6.3). Specifically, we will develop for each task τ_i a set of $q_i \in \mathbb{N}$ *safety controllers* $(\mathscr{K}_{i,j}, p_{i,j})$, where $\mathscr{K}_{i,j} : R(X_{Si}) \rightrightarrows U_i$ and respective sampling period $p_{i,j}$ with $j \in \{1, \ldots, q_i\}$, which guarantee the safety even under switching implementations (see Fig. 6.2).

Our second goal is to design a resource allocation algorithm that exploits the designed controllers to optimize schedulability and robustness. Specifically, for each mode $m \in \mathscr{M}$ and each task $\tau_i \in \mathscr{T}^m$, we need to determine:

1. c_k^m and w_k^m, the numbers of cache and BW partitions allocated to each core k $(0 \leq k < r)$ in mode m;
2. core_i^m, a core assigned to τ_i in m; and
3. $\mathscr{K}_i^m$ and p_i^m, a controller and a period assigned to τ_i in m, where $\mathscr{K}_i^m \in \mathscr{K}_i$ and p_i^m is a period of $\mathscr{K}_i^m$.

In addition, for each incoming mode transition $(m', m) \in \mathscr{R}$, we need to determine

- $\mathsf{del}_i(m', m)$, a boolean variable indicating whether we should delay a job of τ_i during the transition (m', m);
- $d_i^c(m', m)$, a delayed deadline for the carry-over job of τ_i, if τ_i is a carry-over task; and
- $d_i(m', m)$, a delayed deadline for the first job of τ_i released in mode m. At run time, if τ_i has a carry-over job, then we only delay the deadline of its carry-over job (using $d_i^c(m', m)$) but not its first new job in m.

Our objective is to maximize schedulability, while minimizing the assigned periods and delayed-deadlines of all tasks in all modes and mode transitions.

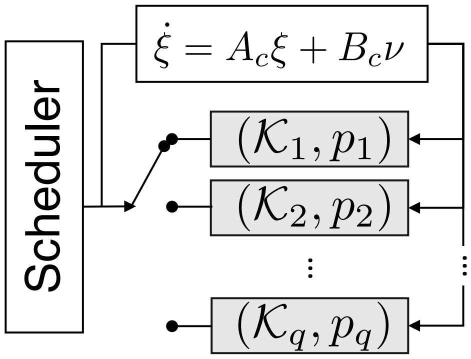


Figure 6.2: The scheduler selects one of the controllers (implementation) $\mathcal{K}_j$ with period p_j. Task index i is omitted for clarity.

6.4 Controller Design

In this section, we focus on the controller design for just one plant; thus, we drop the task index i and only use τ for the sake of simple presentation. First, we discuss the design of an isolated safety feedback controller with a constant sampling period. We then present a switching condition for a task τ that guarantees safety while switching between different controllers with different sampling periods, which will be used by our resource allocation algorithm (Section 6.5).

6.4.1 Safety Feedback Controller With Constant Sampling Period

We consider a dt-LTI $\mathcal{S}$ for a given plant and the following regularity assumptions, which are for the *tractable computation* of the maximal controlled invariant set [25].

Assumption 6.1. *Assume the pair (A,B) is controllable, i.e., the controllability matrix $\mathcal{C}(A,B) = [B\ AB\ A^2B\ \dots\ A^{n-1}B]$ is full rank.*

Assumption 6.2. *Sets X_S and U are polytopes. Namely, there are matrices H_x, H_u and vectors h_x, h_u such that $X_S = \{x \in \mathscr{R}^n \mid H_x x \leq h_x\}$ and $U := \{u \in \mathscr{R}^d \mid H_u u \leq h_u\}$ are compact sets.*

Moreover, we enforce the following condition on the sampling period of the controllers we aim to design for a given task τ.

Assumption 6.3. *The sampling period of every controller is a natural multiple of ρ, i.e., for each $j \in \{1,\dots,q\}$, there is $\alpha_j \in \mathbb{N}$ such that $p_j = \rho_{\alpha_j} = \alpha_j\rho$.*

Assumption 6.3 does not restrict practical applications since the sampling period of the controller is related to the sampling period of the sensor from which the controller receives the feedback signal.

Now, let us consider only one controller j, and let Assumptions 6.1-6.3 hold. Then, for $\rho_\alpha = \alpha\rho$, the value of $u(t)$ applied for α consecutive instants must ensure that if $x(t) \in X_S$, then $x(t+\alpha) \in X_S$. One can compute the evolution of the system in this interval as follows:

$$\begin{aligned} x(t+1) &= Ax(t) + Bu(t) \\ x(t+2) &= A^2x(t) + (A+\mathscr{I})Bu(t) \\ \vdots \quad &= \quad \vdots \\ x(t+\alpha) &= A^\alpha x(t) + (A^{\alpha-1} + A^{\alpha-2} + \ldots A + \mathscr{I})Bu(t). \end{aligned}$$

We abuse notation and denote the "α"-maximal controlled invariant set as $R^\alpha(X_S)$, and the set $\text{pre}^\alpha(R) = \{x \in \mathscr{R}^n \mid \exists\, u \in U \text{ s.t. } A^\alpha x + A_{\alpha-1}Bu \in R\}$, where $A_\gamma := \sum_{i=0}^{\gamma} A^i$, and $A_0 = \mathscr{I}$. In particular, the iteration in (6.4) is initialized as follows:

$$\begin{aligned} &\text{pre}^\alpha(R_0) = \\ &\Pi_{X_S}\left(\left\{(x,u) \,\middle|\, \begin{bmatrix} H_xA^\alpha & H_xA_{\alpha-1}B \\ 0 & H_u \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} \leq \begin{bmatrix} h_x \\ h_u \end{bmatrix}\right\}\right), \end{aligned} \tag{6.5}$$

where $R_0 = X_S$ and $R_{i+1} = \text{pre}^\alpha(R_i)$. Assuming the iteration admits a fixed point in a finite iteration, the fixed point is the polytope $R^\alpha(X_S) := \{x \in X \mid H_{R^\alpha}x \leq h_{R^\alpha}\}$ when it is not empty [26].

Remark. *The linearity of the dt-LTI and Assumption 6.2 provide an LMI in* (6.5) *which simplifies the computation of* Π_{X_S}. *However, these assumptions are insufficient to guarantee the finite convergence of the iteration. Modifications to the iteration can be made to allow the computation of outer- and inner-approximations of* $R^\alpha(X)$, *ensuring the finite-time convergence of the iteration (see [162] and references therein). These modifications are omitted for the sake of clarity.*

Note that $R^\alpha(X_S)$ is the domain of only one safety feedback controller with a *fixed* period $p = \rho_\alpha$.

6.4.2 Safety Feedback Controllers with Multiple Sampling Time

We now consider the condition that allows the platform scheduler to switch among multiple (q) safety feedback controllers, each of which has a set of sampling periods that guarantee safety. This has two implications: 1) The scheduler may change the period of a chosen controller, namely, a control task τ implements a controller $(\mathscr{K}_j, p_j)$, where $p_j \in [\rho_j, \rho_j]$ with $\rho_j \leq \rho_j < \infty$. However, in the previous section, we designed $(\mathscr{K}_j, \rho_j)$ assuming the control input $u(t)$ is applied at time instants $\rho_j t$ with $t \in \mathbb{Z}_{\geq 0}$. Hence, for any $p_j \neq \rho_j$, we cannot claim safety satisfaction; 2) The scheduler may change the controller of a given task, and the transition must be feasible and ensure the safety property. This means that if the scheduler executes controller $(\mathscr{K}_{j'}, p_{j'})$ after controller $(\mathscr{K}_j, p_j)$, then it should hold that $x(t+\alpha_j) \in R^{\alpha_{j'}}(X_S)$. Otherwise, $\mathscr{K}_{j'}(x(t+\alpha_j))$ might be empty, and one cannot ensure the safety of the closed-loop system.

To address the aforementioned issues, we consider a maximum delay value between executions. Without loss of generality, we consider $(\mathscr{K}_j, \rho_j)$, and $(\mathscr{K}_{j'}, \rho_{j'})$ with $j, j' \in \{1, 2, \ldots, q\}$, and the maximum delay between two executions $\rho_{jj'}$ as in Fig. 6.3. We provide $\rho_{jj'}$ for every combination of j and j' even for $j = j'$; thus, $p_j \in [\rho_j, \rho_j]$ with $\rho_j = \rho_j + \max_{j' \in \{1,\ldots,q\}} \rho_{jj'}$.

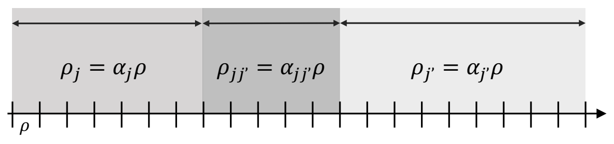

$\rho_j = \alpha_j \rho$
$\rho_{jj'} = \alpha_{jj'} \rho$
$\rho_{j'} = \alpha_{j'} \rho$
ρ

Figure 6.3: Switching between two (different) controllers may be delayed in case the next implementation, $(\mathscr{K}_{j'}, \rho_{j'})$ in our case, cannot be allocated immediately after the final execution of $(\mathscr{K}_j, \rho_j)$.

Similar to Assumption 6.3, we assume the following condition for the maximum delay time.

Assumption 6.4. *For all j, j' with $j, j' \in \{1, \ldots, q\}$, the maximum delay is either zero or a natural multiple of ρ, namely, $\rho_{jj'} = \alpha_{jj'} \rho$ where $\alpha_{jj'} \in \mathbb{Z}_{\geq 0}$.*

The value of $\rho_{jj'}$ is a design parameter that allows $p_j \in [\alpha_j\rho; (\alpha_j + \alpha_{jj'})\rho]$, providing flexibility for the scheduler to delay a task in the event of an overloaded mode transition. For a given dt-LTI $\mathscr{S}$, a safe set X_S, and sampling times ρ_j, $\rho_{j'}$, with maximum delay $\rho_{jj'}$, we define the following inequality:

$$\begin{bmatrix} H_x A^{\alpha_j+\alpha_{jj'}} & H_x A_{\alpha_j+\alpha_{jj'}-1} B \\ H_x A^{\alpha_j+\alpha_{jj'}-1} & H_x A_{\alpha_j+\alpha_{jj'}-2} B \\ \vdots & \vdots \\ H_x A^{\alpha_j+1} & H_x A_{\alpha_j} B \\ H_x A^{\alpha_j} & H_x A_{\alpha_j-1} B \\ 0 & H_u \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} \leq \begin{bmatrix} h_x \\ h_x \\ \vdots \\ h_x \\ h_x \\ h_u \end{bmatrix}. \tag{6.6}$$

The design of each controller must ensure the safety property under any switching of controllers – that is, we need to provide a controlled invariant set $R^{\cap\alpha}$ such that for all $j, j' \in \{1, \ldots, q\}$, the inequality in (6.6) is satisfied. Under Assumptions 6.1-6.4, we modify iteration (6.4) as:

$$\begin{aligned} \text{pre}(R_0) = \Pi_{X_S} \big(\{ (x,u) \in X_S \times U \mid \\ \forall j, j' \in \{1, \ldots, q\} \text{ inequality (6.6) holds} \} \big). \end{aligned} \tag{6.7}$$

Assuming (6.7) converges in finite time to a not empty set, we have that the controlled invariant set is given by $R^{\cap\alpha} = \{x \in X_S \mid H_{R^{\cap\alpha}} x \leq h_{R^{\cap\alpha}}\}$.

Now, we provide the main result of this section.

Theorem 6.5. *Consider a ct-LTI as in Definition 6.1, its associated dt-LTI $\mathscr{S}$ with sampling period $\rho > 0$, and a safe set $X_S \subseteq X$. Let Assumptions 6.1-6.4 hold and $\bar{\alpha}_{jj'} := \max_{j'}\{\alpha_{jj'}\}$. The safety controllers $(\mathscr{K}_j, p_j)$ with $j \in \{1, \ldots, q\}$ have the following form:*

$$\mathscr{K}_j(x) = \{u \in U \mid \tag{6.8}$$
$$\begin{bmatrix} H_{R^{\cap\alpha}} A_{\alpha_j+\bar{\alpha}_{jj'}-1} B \\ \vdots \\ H_{R^{\cap\alpha}} A_{\alpha_j} B \\ H_{R^{\cap\alpha}} A_{\alpha_j-1} B \end{bmatrix} u \leq \begin{bmatrix} h_{R^{\cap\alpha}} - H_{R^{\cap\alpha}} A^{\alpha_j+\bar{\alpha}_{jj'}} x \\ \vdots \\ h_{R^{\cap\alpha}} - H_{R^{\cap\alpha}} A^{\alpha_j+1} x \\ h_{R^{\cap\alpha}} - H_{R^{\cap\alpha}} A^{\alpha_j} x \\ h_{R^{\cap\alpha}} \end{bmatrix} \Bigg\}.$$

Note that the computation of $R^{\cap\alpha}(X_S)$ is done offline. Given a state measurement x, the controller consists of an algorithm that returns a subset of U fulfilling the Linear Matrix Inequality (LMI) in (6.8).

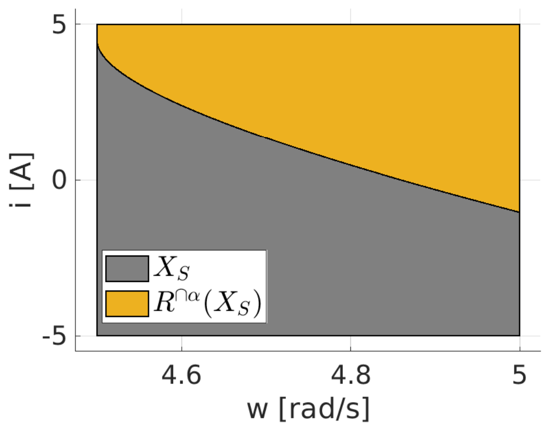


Figure 6.4: Safe set and controlled invariant set for the DC Motor example.

6.4.3 Example

For the sake of illustration, consider a DC motor [226], with states (w, i) representing, respectively, the rotational speed and electric current and DC voltage V as the input. The ct-LTI model is given by:

$$\begin{bmatrix} \dot{w} \\ \dot{i} \end{bmatrix} = \begin{bmatrix} -b/J & K/J \\ -K/L & -R/L \end{bmatrix} \begin{bmatrix} w \\ i \end{bmatrix} + \begin{bmatrix} 0 \\ 1/L \end{bmatrix} V,$$

with $J = 0.25$, $b = 0.05$, $K = 0.05$, $R = 0.1$, and $L = 0.1$. Consider the safe set $X = X_S = [4.5, 5] \times [-5, 5]$, input set $U = [-1, 1]$, and $\rho = 0.01s$. We consider three controllers with $p_1 = 12\rho$, $p_2 = 7\rho$, and $p_3 = 5\rho$. We implement the iteration in (6.4) with $\mathrm{pre}^{\alpha}(R)$ as in (6.5), and obtain the results as in Fig. 6.4.

Next, we introduce an algorithm for task and resource allocation that leverages possible values of ρ and $p \in [\alpha_j \rho, (\alpha_j + \max_{j'}\{\alpha_{jj'}\})\rho] = [\rho_j, \rho_j]$ to improve the resource efficiency, schedulability, and robustness of the system.

6.5 DECNTR Resource Allocation Algorithm

6.5.1 Key Ideas and Algorithm Overview

Key ideas. The DECNTR algorithm allocates tasks and resources to cores in a way that optimizes each mode individually. By tightening task periods and delaying job deadlines during mode transitions if necessary, it achieves safety and schedulability while improving system robustness. We highlight three key insights of DECNTR that make this possible:

Insight #1: Group tasks with similar cache and BW requirements on the same cores to best consolidate resources to tasks that need them the most. If we instead place tasks with high resource requirements on many separate cores, there may not be sufficient resources for such cores to meet their demands.

Insight #2: Tightening each task's period to better improve system robustness. As discussed in Section 6.4, shorter task periods lead to measurable improvements in system robustness. Our allocation minimizes task periods whenever possible (i.e., as long as the system is still schedulable).

Insight #3: Delaying deadlines of some jobs within a safe interval during mode transitions to reduce transient overloads and improve transition schedulability. During a mode transition, the combination of both new jobs and carry-over jobs, with the latter often having very little slack time in the worst case, can lead to an overload in resource demands right after the MCR. We leverage the ability to safely delay controller actuation time to reduce this transient overload, making the transition easier to schedule.

Overview. Based on the above insights, DECNTR first computes an initial allocation for each mode. The initial mode allocation aims to (1) maximize resource efficiency, (2) minimize task periods (i.e., to improve controller robustness), and (3) guarantee each mode's schedulability. DECNTR achieves this by grouping tasks with similar resource needs onto the same cores. During this step, the algorithm assumes each task has the smallest possible period and increases it gradually – after having attempted to migrate tasks between cores of similar resource needs – until the mode becomes schedulable. The outcome of the initial allocation is a mapping of tasks and resources to cores, as well as a controller and a period assigned to each task, for each mode such that each mode is schedulable in isolation. If mode schedulability cannot be guaranteed, the system is deemed unschedulable and the algorithm terminates.

Following the initial allocation, DECNTR makes sure that each mode transition is schedulable. For this, it uses breath-first search to iterate over all mode transitions. For each transition (m', m), it checks whether the system is schedulable during the transition under the current mode allocation. If not, it adds a minimum delay – as small as possible within the safe duration allowed by the controller – to the deadline of the first job to be completed for a task on a core that is unschedulable, until either the transition becomes schedulable or no further delaying is possible. In the former case, it saves the delayed deadlines for this transition and moves to the next transition. In the latter case, it will attempt to increase the task periods in the new mode and re-check if the transition is schedulable. If so, it saves the assigned periods and moves to the next transition. If the periods have been increased up to their maximum sampling periods allowed by the controllers and the transition remains unschedulable, DECNTR reports system unschedulability and returns the current allocations. The algorithm outputs schedulability and the corresponding allocation if all transitions are schedulable.

We next discuss the key procedures for allocation.

Algorithm 5 Mode Allocation

```
 1: function ModeAlloc($m$)                                                ▷ m: mode
 2:     InitTasks($\mathscr{T}^{\mathsf{m}}$)
 3:     SortByResourceSensitivity($\mathscr{T}^{\mathsf{m}}$)
 4:     reset = true
 5:     while (true) do
 6:         if reset then
 7:             ResetCoreAlloc(cores)
 8:             cores[0].cache = $C_{max}$
 9:             cores[0].bw = $W_{max}$
10:             cores[0].tasks = $\mathscr{T}^m$
11:
12:         for $i = 0$; $i < r$ and IsSchedulable(cores[i]); $i$++ do
13:         if $i \geq r$ then return SCHEDULABLE
14:         for $j = i + 1$; $j < r$ and cores[j].util $\geq 1$; $j$++ do;
15:         if $j < r$ then
16:             splitPoint = GetSplitPoint(cores[i].tasks)
17:             SplitCore(cores[i], cores[j], splitPoint))
18:             ResRedistrubte(cores)
19:             reset = false
20:             continue
21:         success = MIGRATE($m$)
22:         if success then return SCHEDULABLE
23:
24:         if Inflate($m$) == false then break
25:         reset = true
26:     return UNSCHEDULABLE
```

6.5.2 Mode Allocation

Algorithm 5 shows the pseudo-code for finding an allocation for a mode m. DECNTR initializes each task τ_i in m by assigning its period to be its minimum safe period, i.e., $p_i^m = \rho_{i,\min}$. In addition, it sets $\mathscr{K}_i^m$ to be the controller $\mathscr{K}_{i,j}$ with the highest maximum period $\rho_{i,j}$ where $\rho_{i,j} = p_i^m$. It then sorts all of m's tasks in non-increasing order of resource sensitivity (Line 3), where the resource sensitivity of a task τ_i in m is defined as:

$$\gamma_{i,m} = \frac{e_i(1,1) - e_i(C_{max}, W_{max})}{e_i(1,1)}. \tag{6.9}$$

Intuitively, the resource sensitivity of a task is a metric that depicts how much a task's WCET will be reduced when provided more resources. Here, we capture it simply as the relative difference in WCETs when provided minimum and maximum resources, since this already works well; however, a different estimation can also be used.

Algorithm 6 Task Migration

```
1: function MIGRATE($m$)                                    ▷ m: mode
2:     while (true) do
3:         for ($i = 0$; $i < r$ and IsSchedulable(cores[i]); $i$++) do
4:         if $i \geq r$ then return true
5:         for (dist = 1; dist $< r$; dist++;) do
6:             if $i -$ dist $\geq 0$ and
7:           TaskResRedist($m$, i, i $-$ dist) then break
8:             if $i +$ dist $< r$ and
9:           IsSchedulable(cores[i $+$ dist]) and
10:          TaskResRedist($m$, i, i $+$ dist) then break
11:        if dist $\geq$ r then return false
```

After sorting, DECNTR begins its first iteration where it resets all core allocations (Line 7), assigns all resources to core 0 (Lines 8-9), and assigns all the sorted tasks to core 0 (Line 10). DECNTR then begins checking for the schedulability of each core in increasing order of core index (Line 12), from 0 to $r-1$ (where r is the number of cores), using the schedulability analysis in Section 6.6. If all cores are schedulable, DECNTR returns with the current allocation for m (Line 13), and it will move on to find an allocation for the next mode. If a core is found to be unschedulable, DECNTR will attempt to *split* the core's taskset (Lines 14-17).

To split, we first find the closest higher-index core j whose utilization is less than 1 (Line 14) and a *split point*, defined as the index of the task with the highest difference in resource sensitivity compared to the next task's on this core (tasks are sorted in non-increasing order of resource sensitivity). All tasks after the split point are moved onto core j, keeping their current ordering (Lines 16-17). This way, cores are automatically ordered in non-increasing resource sensitivity – a lower-index core contains tasks that benefit more from having additional resources than a higher-index core does.

After splitting, DECNTR redistributes resources onto all cores (Line 18). This is done by reassigning them one partition at a time, going from lower-utilization cores to higher-utilization cores to balance utilizations across cores. It then continues with rechecking schedulability for all cores (Line 20 then Line 12) and performs splitting of tasks on unschedulable cores as before. If we run out of a valid core to split onto, DECNTR attempts to migrate tasks between cores (Line 21).

Algorithm 7 Task Inflation

```
1: function INFLATE($m$)                                    ▷ m: mode
2:     $i = -1$
3:     $\mathsf{dist} = 0$
4:     for $j \in \mathsf{T}^m$ do
5:         if $\rho_{j,max} - p_j^m > \mathsf{dist}$ then
6:             $\mathsf{dist} = \rho_{j,max} - p_j^m$
7:             $i = j$
8:     if $i < 0$ then return false
9:     while $p_i^m \leq \rho_{i,max}$ do
10:        $p_i^m \mathrel{+}= \rho$                            ▷ Update period
11:        $\mathscr{K}_i^m = \mathsf{GetController}(\mathscr{K}_{(i)}, p_i^m)$   ▷ Find the controller
12:        if $\mathscr{K}_i^m$ then return true                ▷ Eventual guarantee
```

Migration. Algorithm 6 shows the migration function, which performs a more fine-grained movement of tasks from unschedulable cores to some *nearby* schedulable cores to best leverage our resource sensitivity groupings. In each while-loop iteration, it first finds the first core i that is unschedulable, checking in increasing core index order (Line 3). If none exists, the function returns true (Line 4). Otherwise, it attempts to migrate tasks from core i to a neighboring core, checking cores in increasing distance from itself (Lines 5-10). For each distance value, dist, it checks whether a migration to the core $(i - \mathsf{dist})$ is successful (Lines 6-7) before checking the core $(i + \mathsf{dist})$ (Lines 8-10). Since all cores with index less than i are already schedulable, we only need to check schedulability for the higher-index core before attempting migration (Line 9). If a migration is successful, we continue to the next iteration of the while loop and repeats the process (Line 7 and Line 10). If it is not feasible to migrate tasks to any of the cores, the function returns false (Line 11).

To migrate tasks, we attempt to move a single task to the chosen target core, followed by a resource redistribution. As long as the target core remains schedulable after this redistribution, we keep the migration. Otherwise, we revert to the previous resource allocation and attempt to migrate another task. To select a candidate task, we iterate over the sorted list of tasks in the same *direction* of the target core. For migrating to a higher-index core, we consider tasks in reverse order of $\gamma_{i,m}$, and vice versa. Intuitively, we move less (resp. more) resource-sensitive tasks onto less (resp. more) resource-sensitive cores.

Inflation. The goal of task inflation is to increment a task's current assigned period by ρ (the control parameter) and then find the controller with the largest ρ such that the increased period falls within its range. Algorithm 7 shows how this is done. We select the task τ_i whose current assigned period p_i^m is farthest away from its maximum safe period $\rho_{i,max}$ for inflation (Lines 2-7). If no such task exists, the function returns false. Otherwise, in Lines 9-12, we increase the period and search for a (potentially) new controller for τ_i. If there is no controller in $\mathscr{K}_i$ that has a safe period equal to the updated p_i^m, we increment p_i^m again until a controller is found. Note that a controller is *guaranteed* to be found eventually, since the p_i^m will eventually reach the maximum safe period defined by some controller. Once a controller is found, we save it as $\mathscr{K}_i^m$ and keep the updated p_i^m, then return true.

6.5.3 Mode Transition Allocation

After obtaining a feasible allocation for every mode, DECNTR proceeds to check whether the system is schedulable during every mode transition, using the transition schedulability test in Section 6.6. If a transition (m',m) is not schedulable, then there exists at least one core in mode m that is unschedulable, and this happens because of the additional load from carry-over jobs from mode m' (since m is schedulable in isolation). Leveraging our controllers' ability to tolerate some delay between sampling time points and to allow changes of sampling periods, we can reduce the transition load by extending the deadlines of carry-over jobs and/or increasing the task periods in mode m. The former strategy has only a transient effect on robustness, since it delays the actuation time of at most one job per task during the transition. In contrast, the latter strategy affects all jobs released in the new mode, and may affect robustness for a longer duration. To minimize potential negative impact on robustness, we prioritize extending the deadlines of carry-over jobs over increasing task periods.

Algorithm 8 shows the pseudo-code for achieving schedulability for a mode transition. DECNTR invokes this function for each mode transition $(m',m) \in \mathscr{R}$ in the system. It first initializes $d_i(m',m) = p_i^m$ for each task τ_i in m, and $d_i^c(m',m) = p_i^{m'}$ for each carry-over task τ_i (Line 3). It then checks if the transition is schedulable (Lines 6-7). Here, $\mathsf{McpAnalysis}(m',m)$ implements the transition schedulability test in Section 6.6; it returns 0 if the transition is schedulable, and the index of an unschedulable core ($\mathsf{failedCore}$) otherwise. If the transition is schedulabe, the function returns schedulable and DECNTR moves to the next transition. Otherwise, we call

Algorithm 8 Transition Allocation

```
1: function TRANSITION($m'$, $m$)                              ▷ m: new mode, m': old mode
2:     jobType = 0                                             ▷ 0: carry-over job, 1: first new job
3:     InitDelays($\mathscr{T}^{m'}$, $\mathscr{T}^{m}$)
4:     while true do
5:         delayed = false
6:         failedCore = McpAnalysis($m'$, $m$)
7:         if !failedCore then return SCHEDULABLE
8:         for ; jobType $\leq$ 1; jobType++ do
9:             i = MaxSlackTask(jobType, $m$, $m'$, failedCore)
10:            $j = \mathscr{K}_i^{m'}$
11:            if !jobType and $d_i^c(m',m) < \rho_{i,j}$ then
12:                $d_i^c(m',m)$ += $\rho$                             ▷ Delay
13:                delayed = true
14:            $j = \mathscr{K}_i^{m}$
15:            if $d_i(m',m) < \rho_{i,j}$ then
16:                $d_i(m',m)$ += $\rho$                               ▷ Delay
17:                delayed = true
18:                break
19:        if delayed then continue
20:        if !INFLATE($m$) then break
21:        jobType = 0
22:        InitDelays($\mathscr{T}^{m'}$, $\mathscr{T}^{m}$)
23:    return UNSCHEDULABLE
```

MaxSlackTask(...) to retrieve the task i that has the largest difference between the maximum period of its assigned controller and its assigned period. If jobType $= 0$ (computing the delayed deadline for a carry-over job from m'), the assigned controller is $\mathscr{K}_i^{m'}$; otherwise, the assigned controller is $\mathscr{K}_i^{m}$. We then extend this task's delayed deadline(s) by the minimum amount, which is the configurable parameter ρ (Lines 12,16). Algorithm 8 attempts to delay the deadlines of all carry-over tasks before delaying deadlines of new tasks. After any increment, we repeat the process and recheck mode transition schedulability. If the transition is still unschedulable after extending the first deadlines of new tasks, we will attempt to inflate task periods for the target mode m using the same procedure (Algorithm 7) as in the single mode case (Line 20). If inflation is possible, we restart the process in a new iteration, checking carry-over tasks first again (Line 21) as inflating a task can change its controller; otherwise, the function returns with the failed allocation.

6.6 Schedulability Analysis

This section presents the schedulability analysis used by DECNTR during its allocation. Given a multi-mode system with the mode-change protocol as defined in Section 6.3. Let $\mathscr{A}$ be an allocation of the system, which specifies

$$\left\{c_k^m, w_k^m, \mathsf{core}_i^m, \mathscr{K}_i^m, p_i^m, \mathsf{del}_i(m',m), d_i^c(m',m), d_i(m',m)\right\}$$

for all $m \in \mathscr{M}$, $0 \le k < r$, $\tau_i \in \mathscr{T}^m$, $\mathscr{K}_i^m \in \mathscr{K}_{(i)}$, p_i^m is a valid period of $\mathscr{K}_i^m$, and $(m',m) \in \mathscr{R}$. Then, the system is schedulable under $\mathscr{A}$ iff it is schedulable in each mode $m \in \mathscr{M}$ and in each transition $(m',m) \in \mathscr{R}$.

Notation. We begin by recalling the notation used in $\mathscr{A}$. First, c_k^m and w_k^m are the numbers of cache and bandwidth partitions assigned to core k in mode m, respectively. Second, core_i^m, $\mathscr{K}_i^m$ and p_i^m are the core, controller and sampling period assigned to task τ_i in mode m. Finally, $\mathsf{del}_i(m',m)$ indicates whether we will delay the deadline of a job of τ_i during the transition (m',m); if so, $d_i^c(m',m)$ is the delayed deadline of the carry-over job of τ_i (if τ_i has a carry-over job) and $d_i(m',m)$ is the delayed deadline of the first job of τ_i released in m. We denote by e_i^m the WCET of τ_i in mode m, i.e., $e_i^m = e_i(c_k^m, w_k^m)$ where $k = \mathsf{core}_i^m$. Let $\mathscr{T}_k^m$ denote the set of tasks that are mapped onto core k in mode m under the allocation $\mathscr{A}$. Then, the utilization of a core k is $U_k^m = \sum_{\tau_i \in \mathscr{T}_k^m} \frac{e_i^m}{p_i^m}$.

6.6.1 Mode Schedulability

The schedulability of a mode m in isolation can be determined using the standard demand-based EDF schedulability test, except that the WCET e_i^m of each task τ_i in mode m is dependent on the allocation. Specifically, the demand bound function (DBF) of a task τ_i in mode m is $\forall t \ge 0 : \mathsf{dbf}_{\tau_i}^m(t) = \lfloor \frac{t}{p_i^m} \rfloor e_i^m$. Since tasks on each core are scheduled under EDF, the DBF of a core k in mode m is given by

$$\forall t \ge 0 : \mathsf{dbf}_k^m(t) = \sum_{\tau_i \in \mathscr{T}_k^m} \mathsf{dbf}_{\tau_i}^m(t). \tag{6.10}$$

The next theorem states the schedulability condition for mode m. Its proof follows directly from the existing analysis in [64].

Theorem 6.6. *The system is schedulable in mode m if for all* $0 \leq k < r$, $U_k^m \leq 1$ *and for all* $t < L_k^m$, $\mathsf{dbf}_k^m(t) \leq t$, *where*

$$L_k^m = \max\left\{D_k^m, \frac{1}{1-U_k^m}\right\}.$$

and D_k^m *denotes the maximum of the assigned periods (deadlines) of the tasks in* $\mathscr{T}_k^m$.

6.6.2 Mode Transition Schedulability

The mode transition schedulability can be established using a similar analysis as in Omni [77]. However, since Omni does not delay jobs during a transition, we extend its analysis slightly to handle delayed jobs. Below, we sketch key schedulability conditions and accompanying arguments.

Intuitively, for each core, we compute each task's worst-case demand for an interval t during a mode transition; if their total demand of all tasks on a core is no more than t, for all $t > 0$, then the core is schedulable. Towards this, we treat each carry-over job as a new job released at the MCR instant, with its WCET equal to the carry-over job's maximum remaining execution time and its absolute deadline the same as the carry-over absolute deadline (if the task is not delayable) or the carry-over delayed deadline (otherwise). As the worst-case demand of a core for any interval that begins *after* the MCR instant will only include demands from jobs released in the new mode, we can compute its demand using the same method as in an isolated mode m. Thus, we only need to consider the worst-case demands for intervals that begin at the MCR instant (referred to as transition demands).

Like in Omni [77], a task's worst-case demand occurs when (i) it has a job with deadline at the end of the interval; (ii) new jobs are released as soon as possible; and (iii) the carry-over (unfinished) job was executed as late as possible. Based on these conditions, we can bound the transition demands of the jobs for each task type, using conventional demand-bound analysis, while taking into consideration that 1) a job's WCET depends on the current resource allocation (of the new mode), and 2) a task may be delayable, in which case either their carry-over job or their first job will be delayed, and thus the delayed deadline is used for computing the demand instead of the original deadline.

More concretely, to establish the schedulability of a mode transition, we derive the worst-case demands of all jobs on a core during the transition. The carry-over demand during a mode transition (m',m) consists of (i) the demands of carry-over jobs of carry-over tasks, and (ii) the demands of new jobs of both carry-over and new tasks.

Transition demands of new tasks. Since jobs of a new task are only released in the new mode, the task's transition demand can be computed similarly to the task's demand in an isolated mode. The only exception is that, if the task is delayable, then the delayed deadline is used for its first job.

Lemma 6.1. *The DBF of a* **new** *task τ_i during a transition (m',m) is given by*

$$\mathsf{dbf}_{i,N}^{m',m}(t) = \lfloor \frac{t - d_i(m',m) + p_i^m}{p_i^m} \rfloor e_i^m \tag{6.11}$$

where e_i^m and p_i^m are the WCET and sampling period of τ_i in mode m, respectively. Further, $d_i(m',m)$ is the delayed deadline of τ_i's first job if it is a delayble task (that is, $\mathsf{del}_i(m',m) = 1$), and $d_i(m',m) = p_i^m$ otherwise.

Proof. First, suppose τ_i is a non-delayable new task. Then, its worst-case demand during the transition is identical to its worst-case demand in mode m in isolation, since all jobs are released and have deadlines in the new mode and following their corresponding new-mode periods. Thus, $\mathsf{dbf}_{i,N}^{m',m}(t) = \lfloor \frac{t}{p_i^m} \rfloor e_i^m$, which is equivalent to Eq. (6.11) since $d_i(m',m) = p_i^m$ in this case.

Next, consider the case where τ_i is a delayable task. Then, the (delayed) deadline for its first new job is $d_i(m',m)$, whereas the deadline of a subsequent job is the same as the period p_i^m. Further, according to our mode change protocol (c.f. Section 6.3), τ_i releases its first job immediately after the MCR arrives, and each subsequent job is released at the absolutely deadline of the previous job. Hence, the maximum demand of τ_i in the transition interval of length t starting from the MCR instant is at most $\lfloor \frac{t-d_i(m',m)+p_i^m}{p_i^m} \rfloor e_i^m$. In other words, Eq. 6.11 holds. □

Transition demands of carry-over tasks. For the demand of a carry-over task τ_i however, we need to consider two cases: 1) τ_i has a carry-over job and its carry-over job has a deadline of $d_i^c(m',m)$, and 2) τ_i has no carry-over job and its first job has a deadline of $d_i(m',m) > p_i^m$. We can easily show that the worst-case demand of τ_i in the first case is always higher than the demand in the second case, for all time intervals t. Hence, we can derive the maximum demand for τ_i based on the worst-case scenario with the carry-over job. Let $d_i^{m'}$ be the deadline of the carry-over job. Then, $d_i^{m'} = p_i^{m'}$ if τ_i is not delayable and $d_i^{m'} = d_i^c(m',m)$ if it is. With this, we can compute the DBF of τ_i during a transition (m',m) as follows:

Lemma 6.2. *The DBF of a* **carry-over** *task τ_i during a transition (m',m) is given by*

$$\mathsf{dbf}_{i,CO}^{m',m}(t) = \lfloor \frac{t}{p_i^m} \rfloor e_i^m + E_i^{m',m}, \tag{6.12}$$

where $E_i^{m',m}$ is τ_i's maximum carry-over demand, defined by

(a) If $t \le d_i^{m'} - p_i^{m'}$, then $E_i^{m',m} = 0$.

(b) If $d_i^{m'} - p_i^{m'} < t < p_i^m$, then

$$E_i^{m',m} = \min\{e_i^m, t + \max\{0, e_i^m - e_i^{m'}\}\}.$$

(c) Otherwise, $t' = (t - p_i^m) \mod p_i^m$, and

$$E_i^{m',m} = \min\{e_i^m, \max\{0, t'\} + \max\{0, e_i^m - e_i^{m'}\}\}.$$

Proof sketch. The demand of a carry-over task during the transition consists of (1) the demand from all new job releases of τ_i in mode m, which is given by $\lfloor \frac{t}{p_i^m} \rfloor e_i^m$ (the first term in the RHS of Eq. (6.12)); and (2) the demand from the carry-over job of τ_i itself. Thus, to prove the lemma, we need to show that $E_i^{m',m}$ correctly bounds this carry-over demand.

Recall that the carry-over job has a (delayed) deadline of $d_i^{m'}$. Thus, the amount of time that its deadline is delayed is equal to $d_i^{m'} - p_i^{m'}$, since $p_i^{m'}$ is its original deadline (since it is released in the old mode m'). There are two scenarios:

If $t \leq d_i^{m'} - p_i^{m'}$, then the length of the time interval we are considering, t, is no more than the amount of time we can delay the carry-over job. Thus, the carry-over job imposes no demand, since its deadline falls outside the interval. Thus, $E_i^{m',m} = 0$ and the case (a) of the lemma holds.

Otherwise, the maximum carry-over demand can be conservatively computed using the analysis from Omni. This is possible, because Omni does not delay carry-over jobs and thus the worst-case carry-over demand under Omni is at least equal to or larger than the carry-over demand under DECNTR. The two cases (b) and (c) of the lemma come directly from the results in Omni [77] (see Lemma 2 and Lemma 3 in [77]), modified appropriately to reflect our different notation. □

The next lemma gives the mode change demand of a core during a transition (m', m). Its proof is established based directly on Eq. (6.11) and Eq. (6.12).

Lemma 6.3. *The maximum demand of a core k during a mode transition* (m', m) *is bounded by*

$$\mathsf{dbf}_k^{m',m}(t) = \sum_{\tau_i \in \mathcal{T}_k^N} \mathsf{dbf}_{i,N}^{m',m}(t) + \sum_{\tau_i \in \mathcal{T}_k^{CO}} \mathsf{dbf}_{i,CO}^{m',m}(t)$$

where $\mathcal{T}_k^N$ *and* $\mathcal{T}_k^{CO}$ *are the set of new tasks, and carry-over tasks that are mapped onto core k in mode m.*

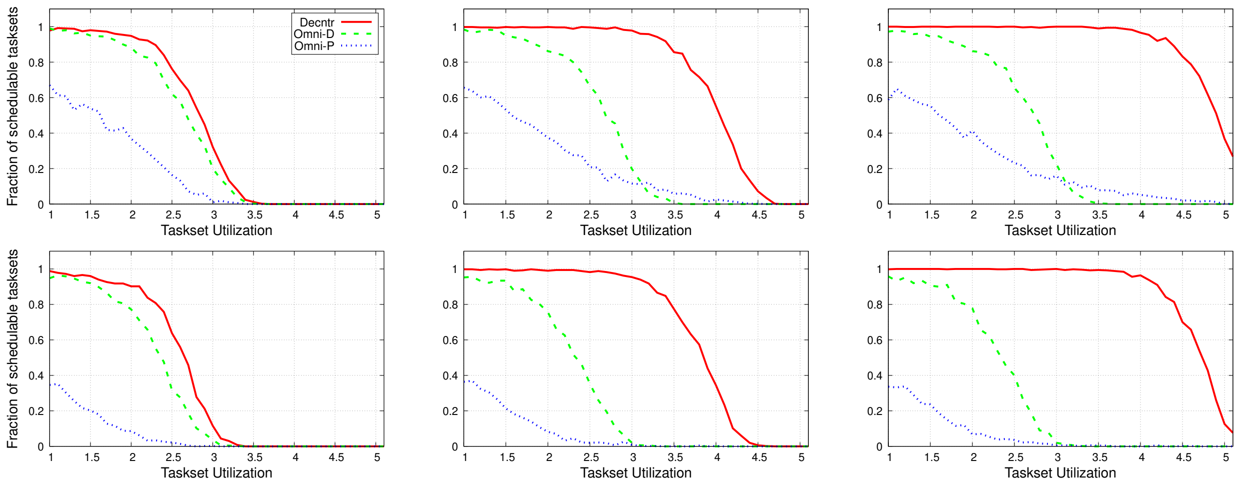
Based on Lemma 6.3 and the mode schedulability, we can directly imply the next theorem.

Theorem 6.7. *The system is schedulable during a mode transition from* m' *to m if (1) it is schedulable in modes* m' *and m (Theorem 6.6), and (2) for all* $0 \geq k < r$*, for all* $t > 0$, $\mathsf{dbf}_k^{m',m}(t) \leq t$.

6.7 Evaluation

To evaluate the effectiveness of DECNTR, we conducted a series of experiments using real-time benchmarks and an automotive case study. Our goal was to evaluate (1) how much DECNTR improves schedulability over the state of the art, and (2) how effective DECNTR is in increasing system robustness.

Algorithms. We compare DECNTR against Omni [77], the state-of-the-art technique for multi-mode, multicore task and resource allocation. Like DECNTR, Omni computes an allocation of tasks, cache and



(a) MaxPeriodFactor = 1.0 (b) MaxPeriodFactor = 1.3 (c) MaxPeriodFactor = 1.5

Figure 6.5: System schedulability under different max period factors. Top row: 2 modes, Bottom row: 4 modes.

memory bandwidth to cores in each mode to ensure the multi-mode system is schedulable. However, Omni assumes the tasks' periods are given a priori and jobs' deadlines cannot be delayed. To enable a more direct comparison for Omni, we develop two extensions of Omni (which performed strictly better than Omni in our evaluation):

1) Omni-P: we set task periods to be the periods assigned by DECNTR, then use Omni to find an allocation.

2) Omni-D: we use Omni to find an allocation assuming tasks are assigned their reference periods (defined below), but extend it to support delaying job deadlines as DECNTR does.

6.7.1 Schedulability Evaluation with Benchmarks

Workload. To evaluate DECNTR's effectiveness across a range of loads and timing parameters, we used a collection of 11 benchmarks from the PARSEC [23], SPLASH2x [206], DIS [133], and Isolbench [199] suites as workloads. To obtain their WCETs, we ran each benchmark under all possible cache and BW configurations on an Intel Xeon E5-2683 v4 processor with 16 cores and a 40MB L3 cache. The cache is divided into $C_{\max}$ = 20 equal partitions using Intel's CAT. Using the method from [224], we measured a maximum guaranteed bandwidth of 1.4 GB/s, which we divided into $W_{\max}$ = 20 partitions of 70MB/s each using MemGuard. We disabled any hardware feature that can lead to nondeterministic timings and used a single-threaded execution mode.

We then generated tasksets using a similar approach to [77] by randomly picking tasks from our benchmarks to fill a target taskset utilization. Each individual task's utilization was selected from a bimodal distribution with a $\frac{6}{9}$ likelihood to fall within [0.01, 0.4] and a $\frac{3}{9}$ likelihood to fall within [0.4, 0.9].

Each task τ_i was assigned a *reference period* (p_i^{ref}), defined as the ratio of its reference WCET, $e_i(C_{\max}, W_{\max})$, to its utilization, rounded to the nearest multiple of ρ. We set $\rho = 1$ms in all experiments. We then defined the period range $[\rho_i, \rho_i]$ for τ_i based on the reference period, where the minimum period ρ_i is the smallest multiple of ρ that is above the task's reference WCET, and the maximum period $\rho_i = k_{\max} \times p_i^{\text{ref}}$ for some configurable value $k_{\max}$ called the *max period factor* (MPF). The task's period can be any value $k\rho$ in the range $[\rho_i, \rho_i]$, where k is a positive integer. Note that Omni-D uses the reference period as the task's period, whereas DECNTR assigns the task's period based on its period range.

We generated tasksets with (reference) utilizations in the range [1.0, 5.0] at steps of 0.1. For each utilization step, we generated 500 independent tasksets per mode, for a total of 41,000 tasksets in a 2 mode system (82,000 tasksets with 4 modes). We generated 10 experiments (half with 2 modes and half with 4), with varying MPFs ($1.0 - 1.5$). The generated tasks had a 20% probability of being a carry-over task.

Results. Fig. 6.5 shows the fraction of schedulable tasksets under the three algorithms for tasksets with MPFs being 1.0, 1.3 and 1.5. The results show that DECNTR is significantly better at scheduling tasksets compared to Omni when they use the same assigned periods (DECNTR VS. Omni-P). Our improvement factor at a medium load of 2.0 reference utilization ranges from 2.4$\times$ (2-mode systems, MPF = 1.0) to 11$\times$ (4-mode system, MPF = 1.5). Note that as the MPF increases from 1.0 to 1.5, both DECNTR and Omni-P can schedule more tasks. This is expected, since DECNTR can assign larger periods to tasks in all modes with larger MPF. Interesting, this trend holds for DECNTR in both 2- and 4-mode systems, whereas Omni-P only achieves improved schedulability under 2 modes, highlighting that DECNTR scales better with the number of modes. We also see that when the MPF is limited to 1.0, DECNTR not only schedules more tasksets than Omni-D but also reduces task periods by 5-21% on average.

Runtime. Tables 6.1 and 6.2 reports the running time of each algorithm for the 2-mode systems (MPF = 1.3) in our experiments. We observe that DECNTR is consistently more efficient than both Omni variants (3.4-8.2$\times$ faster on average).

	1.00 Taskset utilization				2.00 Taskset utilization			
	min	max	**avg.**	99th	min	max	**avg.**	99th
DECNTR	0.02	3.18	**0.06**	0.35	0.11	4.39	**0.24**	1.68
Omni-P	0.03	24.3	**0.24**	2.28	0.13	88.0	**0.96**	12.8
Omni-D	0.03	33.7	**0.29**	3.42	0.13	102	**1.71**	30.2

Table 6.1: Runtime of the three algorithms in seconds (Part 1: Low utilization).

	3.00 Taskset utilization				4.00 Taskset utilization			
	min	max	**avg.**	99th	min	max	**avg.**	99th
DECNTR	0.14	5.11	**0.51**	3.11	0.24	12.5	**0.80**	6.72
Omni-P	0.21	57.5	**1.71**	19.9	0.28	193	**3.37**	54.0
Omni-D	0.20	85.9	**4.21**	54.7	0.30	64.2	**5.24**	38.6

Table 6.2: Runtime of the three algorithms in seconds (Part 2: High utilization).

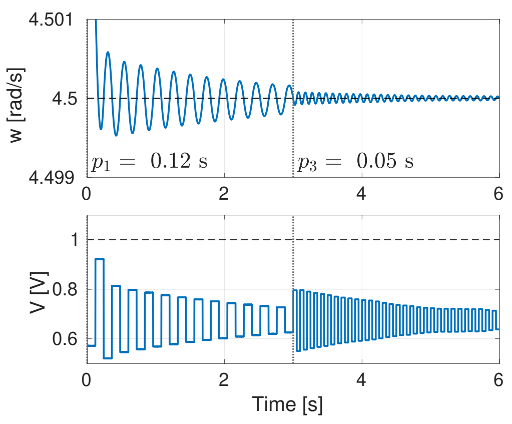
6.7.2 CPS Case Study

To evaluate the benefits of our co-design approach in a real-world CPS, we conducted a case study of an electric vehicle system. The system contains 5 different control tasks: battery [41], DC motor [226], active suspension, [143], automatic cruise control [82], and lane-keeping assistant [231]. We consider both robustness and schedulability in our evaluation.

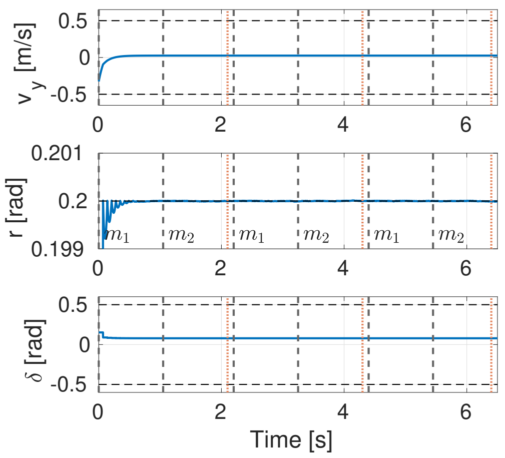
Robustness. For each plant, we designed multiple safety controllers $(\mathscr{K}_j, \rho_j)$ as in (6.8), and simulated the closed-loop performance under additive Gaussian noise in the control input. Due to space constraints, we present three of the five simulations to show the effect of the increase in sampling rate on the safety of CPSs.

DC Motor: We use the model and safe set presented in the example of Section 6.4.3. The results are presented in Fig. 6.6a for the state evolution of w under the input sequence V. Our safety property states that $w(t) \in [4.5, 5]$ and $V(t) \in [-1, 1]$ for all $t \geq 0$. The figure demonstrates that w drifts into the unsafe region for longer intervals when p is larger.

Automatic Cruise Control (ACC): We use the model and safety conditions from [82], which maximizes speed while keeping a safe distance from a forward vehicle. The input is acceleration, a, and the states are the two vehicle velocities and the distance from the forward car (v, v_f, d). We implemented the ct-LTI given by $\dot{v} = a$, $\dot{v}_f = 0$, and $\dot{d} = v_f - v$ and we impose the safe set given by the conditions $a = [-1, 1]$, $v \in [-5, 5]$, $v_f \in [-5, 5]$, $d \in [5, 20]$, $v \leq \frac{d}{t_{safe}}$, where $t_{safe} = 2s$, to represent a minimum time-gap that allows the first vehicle to react in an emergency. Fig. 6.7 shows the results of the control reducing the distance between



(a) DC Motor



(b) LKA with modes m_1, m_2

Figure 6.6: Control simulations for DC Motor and LKA demonstrating robustness as sampling rates change

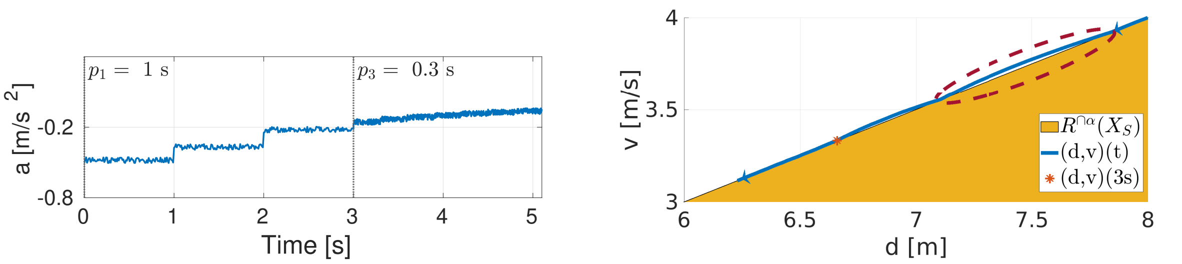


Figure 6.7: ACC control simulation demonstrating robustness as sampling rates change

the vehicles while trying to keep the minimum time-gap with initial state values $(4.4, 3, 9) \in R^{\cap\alpha}(X_S)$. The controller with $p = 1s$ cannot satisfy the safety property due to significant inter-sampling time (highlighted with a dashed red ellipsoid). After $3s$ we switch to a controller with $p = 0.3$ with no safety issues.

Lane Keeping Assistant (LKA): The LKA control steers the horizontal speed and the yaw angle rate of the vehicle (v_y, r), respectively, provided changes in the steering wheel angle δ as input. We define $\rho = 0.01s$, and the safe set given by $v_y \in [-0.5, 0.5]$, $r \in [-0.2, 0.2]$, and $\delta \in [-0.5, 0.5]$. We repeatedly run the controller in two modes m_1, m_2 and simulate periods provided by DECNTR. Fig. 6.6b shows the result where the periods are $p_1 = 7\rho$ in m_1 and $p_2 = 7\rho$ in m_2. Notably, DECNTR's allocation requires the LKA task to have a delayed job release $(p = 10\rho)$ for every transition (m_2, m_1). The result confirms that even when extending jobs' deadlines during mode transitions, the system under DECNTR allocation remains safe.

Schedulability. We generated tasksets with the controllers in our case study, using the methodology in

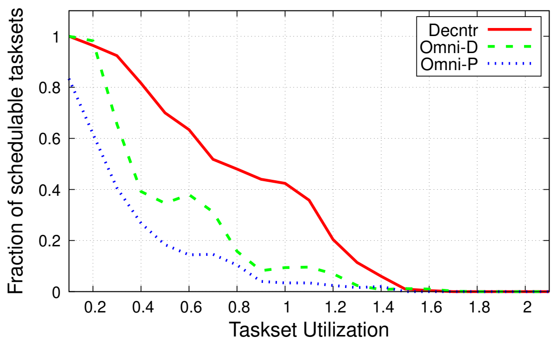


Figure 6.8: Case study schedulability, 2 modes.

Section 6.7.1. To generate a taskset at a target utilization, we randomly selected a single task until the taskset utilization reaches the target utilization, where each task's reference utilization is computed as the reference WCET over the *max* period: $\frac{e_i(C_{max},W_{max})}{\rho_{i,max}}$. Since the reference utilization is small compared to the utilization when assigned resources and a period from DECNTR, we generate tasksets with utilization range of [0.1,4]. For WCETs, we profiled the controllers on resource constrained hardware relevant for our case study: a Raspberry Pi 3 Model B+ with 512 KB of shared cache and a guaranteed BW throughput of 100 MB/s (20 partitions of 5 MB/s each).

Fig. 6.8 shows the schedulability results for systems with 2 modes and 500 tasksets per utilization step. As expected, DECNTR outperforms both Omni variants. For example, at reference utilization of 0.5, DECNTR can schedule 3.5$\times$ more tasksets than Omni-P and 1.8$\times$ than Omni-D. Notably, DECNTR achieves this schedulability improvement even with much smaller task periods (43% on average) than Omni-D, and thus it also provides much higher robustness.

6.8 Conclusion

In this chapter, we introduced DECNTR, along with a controller-switching co-design approach, which draws inspiration from the domains of multi-mode resource allocation and control theory. Our solution demonstrates remarkable performance in terms of safety, robustness, and timeliness for multi-mode CPS deployed on multicore platforms. DECNTR accomplishes this by harnessing multiple control implementations to dynamically adjust task periods, thereby enhancing system safety and significantly improving schedulability when compared to existing state-of-the-art solutions. As part of our future endeavors, we aim to expand our techniques to encompass plants with nonlinear dynamics.

CHAPTER 7

RASCO: RESOURCE ALLOCATION AND SCHEDULING CO-DESIGN FOR DAG APPLICATIONS ON MULTICORE

As our final technical thread, we now consider applications whose tasks are not independent but instead interact through precedence constraints. Many real-time workloads take the form of directed acyclic graphs (DAGs), where upstream delays propagate downstream and end-to-end guarantees depend jointly on execution times and precedence relations. In such settings, it is not clear how to easily apply our previous dynamic resource allocation techniques as intricate task dependencies complicate the resource needs of a single isolated task.

This chapter addresses this challenge with the RASCO algorithm, which unifies resource allocation and scheduling for DAG-structured workloads. RASCO models resource-dependent execution rates, detects phase structure, and coordinates both allocation and placement so that end-to-end deadlines are met.

This chapter is based on work that first appeared as: Abigail Eisenklam, Robert Gifford, Georgiy A Bondar, Yifan Cai, Tushar Sial, Linh Thi Xuan Phan, and Abhishek Halder. RASCO: Resource allocation and scheduling co-design for dag applications on multicore. *ACM Trans. Embed. Comput. Syst.*, 2025. My contribution to this work is substantial and includes co-design of the RASCO algorithm, the initial prototype implementation for the numerical evaluation as well as the complete system implementation for our experimental evaluation.

In Section 7.1, we motivate and discuss the unique challenges of DAG-structured workloads on multicore platforms. Section 7.2 covers DAG specific related work and Section 7.3 covers our new multi-phase model, expanded from Chapter 4. Section 7.4 discusses the details of RASCO and Section 7.5 numerically evaluates RASCO 's performance on representative DAG workloads. Lastly Section 7.6 covers our prototype, microbenchmarks and our real-world evaluation.

7.1 Introduction

Modern real-time embedded applications are inherently complex, comprising interconnected tasks that depend on one another through input/output data constraints. For example, a video processing application often processes the raw video input through several stages—such as decoding, filtering, motion detection, compression, and encoding—where each stage relies on the output of one or more preceding stages. In this example, decoding must be completed before motion detection can begin, and compression occurs only after filtering and detection have been completed. Similarly, tasks in autonomous vehicle systems are interdependent. These applications can therefore be naturally modeled as *taskgraphs*—i.e., directed acyclic graphs, or DAG tasks, as they are commonly known—in which nodes represent sequential task execution and edges represent precedence constraints [200].

As their resource demands increase, 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 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, such as reducing inter-core communication [179], accounting for memory contention [36], exploiting cache recency [230], or co-locating tasks to reduce execution time [196]. 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 [76] has shown that the resource requirements of a task vary significantly throughout its execution, particularly for data-intensive tasks, such as those in video processing or autonomous driving applications. It also demonstrates 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 minimize deadline misses [76]. 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 shared (cache and memory bandwidth) resources 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. Importantly, the model must also support WCET analysis under dynamic budget changes to enable schedulability analysis. Such a model does not currently exist. Second, 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 [165], which in turn depend on their allocated resources. A larger task deadline allows for a smaller resource budget, freeing resources for other concurrent tasks, but it also tightens deadlines for successor tasks and thus increases their resource demands. 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. Additionally, to facilitate resource re-allocation, 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 which tasks, among those scheduled concurrently, would benefit the most from additional resources. To demonstrate its utility in scheduling and

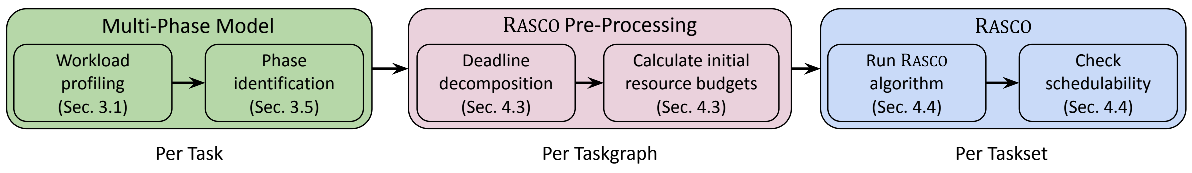

Multi-Phase Model
Workload profiling (Sec. 3.1)
Phase identification (Sec. 3.5)
RASCO Pre-Processing
Deadline decomposition (Sec. 4.3)
Calculate initial resource budgets (Sec. 4.3)
RASCO
Run RASCO algorithm (Sec. 4.4)
Check schedulability (Sec. 4.4)
Per Task
Per Taskgraph
Per Taskset

Figure 7.1: Workflow of our resource allocation and scheduling co-design algorithm for DAG-based applications.

resource allocation, we present a measurement-based method—a common WCET approach in real-time multicore systems [76, 2, 213]—for constructing the multi-phase model.[10] However, our multi-phase model can be constructed using other phase identification and WCET analysis techniques (see Sec. 7.3.5).

To leverage our proposed model, we introduce RASCO, the first fine-grained dynamic resource allocation and scheduling co-design method for periodic taskgraphs with schedulability guarantees. RASCO allocates resources and schedules tasks in a tightly coupled fashion. It uses deadline decomposition to transform each taskgraph into independent tasks with release offsets and deadlines, which are scheduled globally under Earliest Deadline First (EDF) on the cores. Unlike existing deadline decomposition techniques, however, we jointly compute the deadlines for tasks and their resource budgets to effectively utilize available resources and improve execution progress, all while meeting end-to-end deadlines. Starting with an initial base resource allocation and deadline assignment, RASCO iteratively redistributes available resources to the tasks that they most benefit. As execution times decrease due to the additional resources, deadlines are tightened, which in turn affects which tasks are selected to run on the cores. The output of RASCO is a set of tasks with resource-aware release offsets and deadlines. Under global EDF, this provides a schedule of tasks on the cores and their allocated resources. By co-allocating resources and deadlines to tasks and leveraging tasks' multi-phase models, RASCO effectively utilizes shared resources to improve worst-case execution times, reduce end-to-end latency, and maximize schedulability.

Fig. 7.1 shows the workflow for our approach. In summary, we make the following contributions:

- based on empirical studies, 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 timing model from task execution profiles;

[10]We note that 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.

- we introduce RASCO, a co-design algorithm for fine-grained resource allocation and scheduling of DAG applications that leverages the phase information in the tasks' timing models to maximize worst-case instruction rates and resource-use efficiency;
- we present a numerical evaluation of our technique based on real benchmarks;
- 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.

7.2 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 [200] and references therein). Prior work in this area often focuses on two key directions: (1) schedulability and response time analysis methods for a given scheduling algorithm (e.g., global and partitioned EDF with various degrees of preemptions) to improve resource augmentation bounds [18, 99] and tighten worst-case response time [88]; and (2) parallel scheduling algorithms that aim to effectively utilize cores, reduce run- time overheads, and improve schedulability (e.g., [89, 229, 38, 228]). Many techniques use deadline decomposition [165] and static priority assignment computed offline to maximize parallelization. For instance, Zhao et al. [228] models DAG tasks as workload distributions and computes an offline priority and core assignment that accumulates the workload distributions to avoid inter-DAG interference. Sun et al. [191] relies 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, [179] proposes a scheduling technique that combines tasks of a DAG into execution groups to reduce inter-core communication [179]. Casini et al. [36] incorporates the potential overhead due to memory contention in the schedulability analysis. The work in [196] co-locates tasks on the same core to reduce cache overhead and improve run-time performance. In [230], 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 [213, 212] propose holistic resource allocation techniques that find the assignments of tasks, cache and memory bandwidth to cores. Chapter 4 discusses DNA/DADNA [76] in detail which 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 [215] 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 Sec. 7.3.3). 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*.

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

Towards a more accurate timing model for real-time tasks on multicore platforms, we first present an empirical study of real workloads under different budgets of shared resources. Based on insights from this study, we present a multi-phase model that succinctly captures a task's resource-dependent execution behavior. This model can be constructed using existing techniques for phase identification and worst-case timing analysis. To demonstrate its utility, we present a measurement-based approach for constructing estimates of this model from task execution profiles. Once obtained, the model can be used for scheduling and resource allocation, as we will discuss in Section 7.4.

7.3.1 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 different resource allocations.

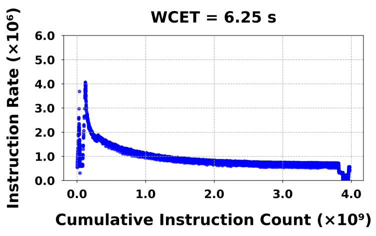
Setup. Our benchmarks consisted of programs and inputs taken from the PARSEC [23] and SPLASH2x [206] benchmark suites. These benchmarks have often been used as workloads by prior work on resource allocation (e.g., [76, 103, 224]). We used Ubuntu 20.04.6 running on an Intel Xeon CPU E5-2618L v3 machine, which supports Intel's Cache Allocation Technology (CAT) [95] for cache allocation and MemGuard [224] for memory bandwidth allocation (see Sec. 7.6 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 $N_{\mathsf{ca}} = 20$ cache partitions. We divided the maximum guaranteed bandwidth of 1.4 GB/s (measured on our machine) into $N_{\mathsf{bw}} = 20$ 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 $\beta = (\beta_{\mathsf{ca}}, \beta_{\mathsf{bw}})$ 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.

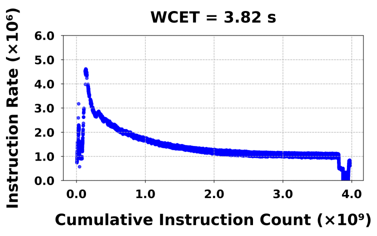
We recorded these metrics for 100 runs of each task under each possible resource budget. By repeating the measurement process for all possible allocated budgets β (i.e., a total of $(N_{\mathsf{ca}} - 1) \times N_{\mathsf{bw}} = 19 \times 20 = 380$ different configurations of β), we obtained the complete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

7.3.2 Results and Discussions: The Case for Fine-Grained Phase-Based Resource Allocation

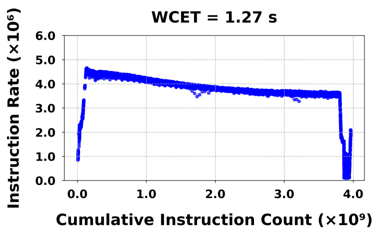
Fig. 7.2 shows the profiling results for two example tasks, *canneal* and *fft*. From these figures, we can identify several key observations. First, a task's WCET depends on its resource budget. As shown in the top horizontal axes of Fig. 7.2a and Fig. 7.2c, *canneal*'s WCET improves from 6.25 to 1.27 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 (Fig. 7.2d and Fig. 7.2f). This decrease in execution time is directly linked to the increase in instruction rate,



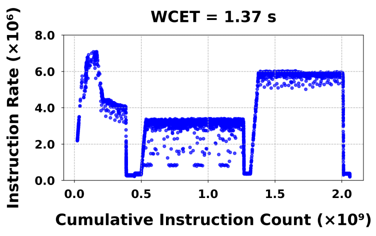
(a) *canneal* $(2_{ca}, 2_{bw})$



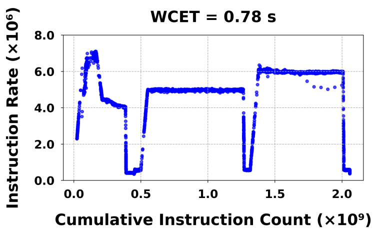
(b) *canneal* $(3_{ca}, 3_{bw})$



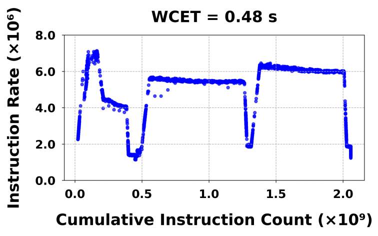
(c) *canneal* $(10_{ca}, 10_{bw})$



(d) *fft* $(2_{ca}, 2_{bw})$



(e) *fft* $(3_{ca}, 3_{bw})$



(f) *fft* $(10_{ca}, 10_{bw})$

Figure 7.2: 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.

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.*

Our third key observation is that 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 variation in instruction rate during execution differs across tasks.*

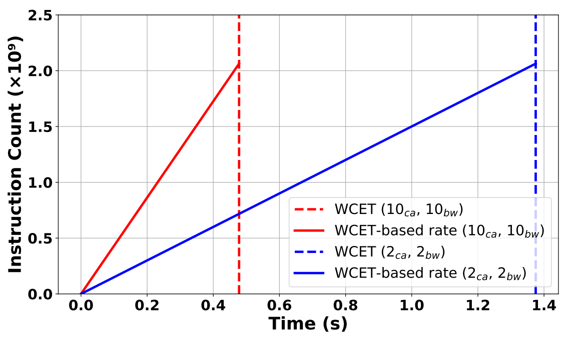
Our final observation is that the instruction rate of a task does not typically increase linearly with the number of resource partitions allocated. Instead, many tasks experience a significant increase in instruction rate only after a certain critical threshold of resource budget (e.g., an amount sufficient to hold their working set). We observed this behavior in *canneal* and *fft*. For *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.*

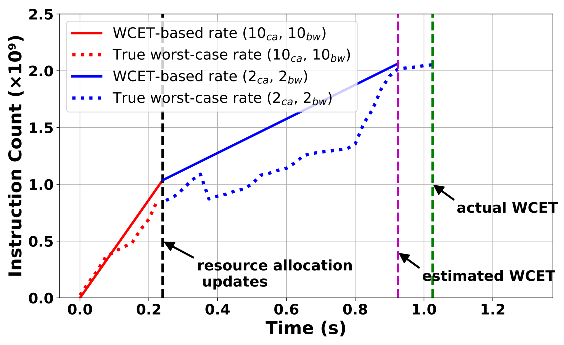
These insights underscore the need for a phase-based model that accurately captures a task's complex timing behavior and a fine-grained dynamic resource allocation method that adapts resources for tasks during their job execution based on the observed impact on instruction rates.

7.3.3 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 [76]. Our goal is to enable such allocation with *worst-case guarantees*. This is highly challenging: without a fine-grained timing model, estimating a task's WCET under dynamic resource budget changes is difficult. While we can estimate a task's WCET under a fixed budget through measurements, profiling all possible budget changes at every execution point is impractical. 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



(a) Constant rates of *fft* based on WCETs

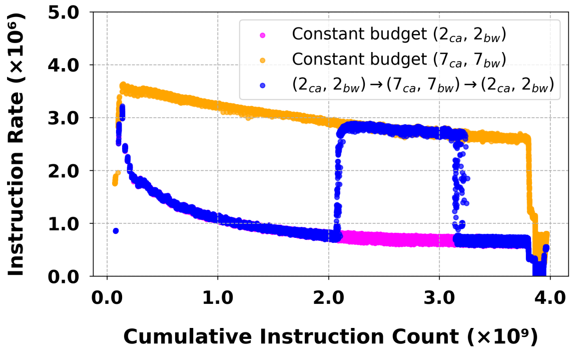


(b) Rate change with dynamic resource allocation

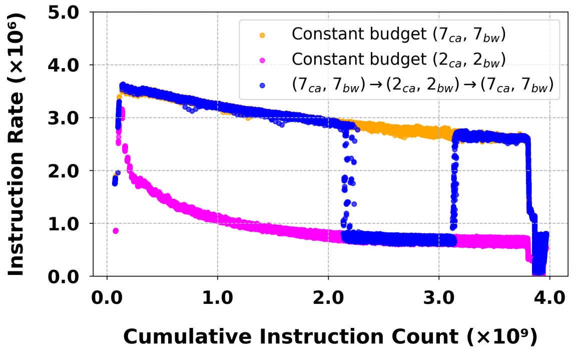
Figure 7.3: Unsafe WCET estimation when assuming a constant instruction rate under each resource budget

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, Fig. 7.3 shows an actual execution of the *fft* benchmark program, where we initially allocated to it a resource budget of $(10_{ca}, 10_{bw})$ and later changed its budget to $(2_{ca}, 2_{bw})$. In Fig. 7.3a, the red and blue solid lines represent the cumulative instruction counts assuming constant instruction rates under $(10_{ca}, 10_{bw})$ and $(2_{ca}, 2_{bw})$, respectively. The dashed vertical lines indicate when the task completes its execution under these resource budgets (i.e., corresponding to their measured WCETs). In Fig. 7.3b, the *actual time-varying* instruction count of the task under our given allocation strategy is shown as the dotted curve, where the red portion corresponds to the initial execution under $(10_{ca}, 10_{bw})$ and the blue corresponds to the subsequent execution after the resource change. 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 resource budget that 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 (Fig. 7.3a), which is much larger than the measured WCET with dynamic resource allocation (just above 1 second, in Fig. 7.3b). Thus, to avoid overly-conservative WCETs, we need to consider the worst-case rates of a task under different resource budgets and their time-varying nature (as illustrated in Fig. 7.3).



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



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

Figure 7.4: Profiled instruction rates of *canneal* under dynamic (blue) vs. constant (yellow & pink) budgets.

Basic idea. We model a task's execution under a given resource budget as a series of phases, each with their 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, Fig. 7.4a 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 Fig. 7.4b.

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 Sec. 7.6 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.

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

We now present a resource-dependent multi-phase model for tasks to capture their dynamic timing behaviors under different resource budgets and to allow safe estimation of WCET under dynamic resource allocation. We use $\beta = (\beta_{\mathsf{ca}}, \beta_{\mathsf{bw}})$ to denote an allocated resource budget, which is a vector of the number of cache partitions and the number of bandwidth partitions. For example, $\beta = (2_{\mathsf{ca}}, 5_{\mathsf{bw}})$ 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

$$\Theta_{\tau|\beta} = (\theta_1, \theta_2, \ldots, \theta_k) \tag{7.1}$$

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 that expresses the potential to increase the worst-case instruction rate of the task in its current phase given a maximum remaining resource budget available for allocation. (More details on θ_i^Δ are in Sec. 7.3.5.) Next, we describe an algorithm for constructing the model from tasks execution profiles, such as the ones shown in Fig. 7.2.

7.3.5 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 by inspecting program behavior and explicitly setting phase boundaries [3], by profiling the worst-case execution path and applying clustering [76, 125], or using changepoint detection (as done in this work). Similarly, a wide range of

WCET analysis techniques exist, which broadly fall into three categories, each with its own tradeoffs: 1) static analysis, which uses software control flow graphs, implicit path enumeration, and often integer linear programming to determine the worst-case execution path through the program, but relies on abstract hardware timing models; 2) measurement-based timing analysis, which measures a program's execution time on the actual hardware, but relies on a constructed set of inputs to explore different execution paths; and 3) hybrid approaches, which combine static and measurement-based analysis (see [2] for a survey of each category). Regardless of the techniques employed, the worst-case instruction rate of each phase can be computed by dividing the number of instructions in the phase by its WCET.

In this work, we employ measurement-based timing analysis—commonly used in multicore real-time systems [2]—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. Note that 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 [2]). To use the multi-phase model for such programs, we must then verify that this path remains valid over all resource budgets. Since tasks often run faster than their worst-case rates, we must ensure that receiving a certain resource budget *early* does not *increase* the WCET under dynamic resource budget, which can occur if the worst-case *path* changes. This requirement can be verified using prior work that incorporates allocations of shared resources into multicore timing analysis [159, 80]. Once the WCEP is identified, our proposed measurement-based approach may be used to construct the multi-phase model. As we will show in Sec. 7.6, 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 Fig. 7.2). 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 [140], an approach commonly used to detect the time points where certain data 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 [9] from the `ruptures` [198] 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 i-th 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.

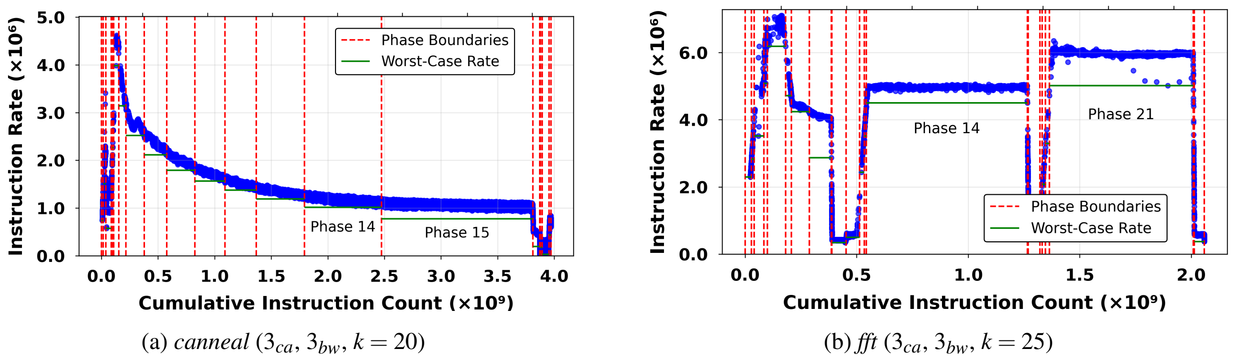


Figure 7.5: Results of changepoint detection for *canneal* and *fft* with k = 20 and k = 25, respectively. The worst case rate of each phase is shown in green and the boundary between each phase is shown in red.

Fig. 7.5 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:

$$\textit{WCET Amplification Ratio} = \frac{\textit{phase-based WCET}}{\textit{profiled WCET}} \tag{7.2}$$

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

$$\textit{phase-based WCET} = \sum_{i=1}^{k} \left(\frac{\theta_i^e - \theta_i^s}{\theta_i^r} \right). \tag{7.3}$$

Intuitively, for each phase θ_i of τ under resource budget β, we find the worst-case execution time 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 worst-case execution time 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 *fft* workload is shown in Fig. 7.6. The results show that the median WCET Amplificiation Ratio for *fft* 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.

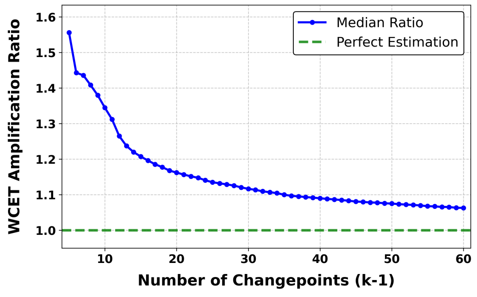


Figure 7.6: The median WCET amplification ratio over all possible resource budgets for *fft*.

Computing the rate increase lookup table. The final parameter of the multi-phase model is θ_i^Δ, which is used by our resource allocation and scheduling co-design algorithm, 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 (determined based on their current instruction counts).

A naïve heuristic for selecting this task τ^* would be to give the resource to whichever task whose worst-case rate improves the most from the additional resource. However, this approach has an important drawback: It 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 <u>one</u> 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 $R = (R_{\mathsf{ca}}, R_{\mathsf{bw}})$, 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 $\beta' \in [(0,0), (R_{\mathsf{ca}}, R_{\mathsf{bw}})]$.

7.4 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 growing interest in real-time embedded systems [230, 216, 191, 38, 164]. RASCO imposes no constraints on the number of taskgraphs or the number of tasks per taskgraph. As such, it naturally supports both a single taskgraph and multiple independent tasks (i.e., one task per taskgraph).

7.4.1 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 $A_i = \{(k-1) \cdot P_i \mid 1 \le k \le K_i\}$ 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 $\mathscr{J}$ 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 k^{th} job of task τ_j belonging to taskgraph G_i. Then, the system is schedulable if all jobs $J_{i,j,k} \in \mathscr{J}$ 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 worst-case execution path that holds for all resource budgets can be determined for each task's workload, and 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 Sec. 7.3.5.

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 last-level shared cache and the memory bandwidth, as described in Sec. 7.3. 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.

7.4.2 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 $\mathscr{J}$ in a hyper-period. Starting with the initial release time, deadline, and base resource budget β_J^{init} for each job J in $\mathscr{J}$, 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 worst-case execution times (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

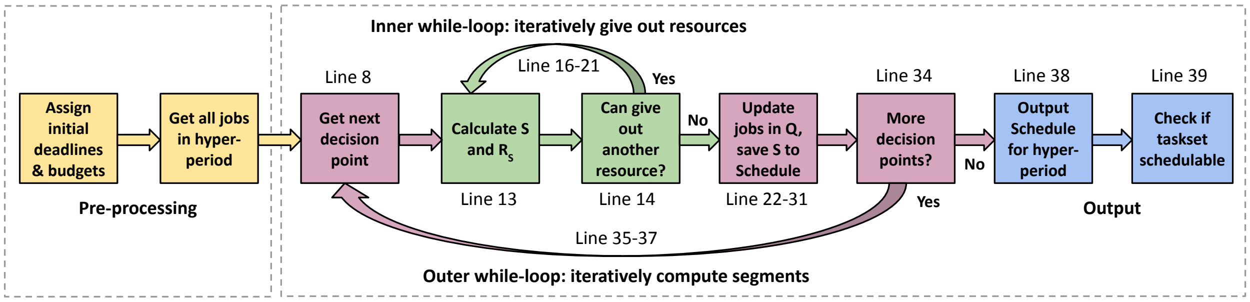

Inner while-loop: iteratively give out resources
Line 8
Line 16-21
Yes
Line 34
Line 38
Line 39
Assign initial deadlines & budgets
Get all jobs in hyper-period
Get next decision point
Calculate S and R_S
Can give out another resource?
No
Update jobs in Q, save S to Schedule
More decision points?
No
Output Schedule for hyper-period
Check if taskset schedulable
Pre-processing
Line 13
Line 14
Line 22-31
Yes
Output
Line 35-37
Outer while-loop: iteratively compute segments

Figure 7.7: Overview of RASCO.

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.

Fig. 7.7 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.

7.4.3 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 [99] which decomposes each taskgraph by placing the tasks into time segments, such that the load in each segment is minimized. We begin by setting 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 $\beta_J^{init} \geq (2_{\mathsf{ca}}, 2_{\mathsf{bw}})$

Algorithm 9 RASCO

```
1:  Input: $\mathscr{J}, A, \{r_J, d_J, \beta_J^{init}, \mathsf{maxIns}_J \mid J \in \mathscr{J}\}, \{\Theta_\tau \mid \tau \in T\}$    ▷ 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 $J \in \mathscr{J}$ do
7:      $\mathsf{ins}_J \leftarrow 0$; $\mathsf{done}_J \leftarrow$ false; $\beta_J \leftarrow \beta_J^{init}$; $c_J \leftarrow$ GETFINISH$(J, \beta_J, \mathsf{ins}_J, r_J, \infty)$    ▷ Initialize per-job variables
8:  $t \leftarrow 0$; $t_{\mathsf{next}} \leftarrow \min\{r_J > 0, c_J > 0 \mid J \in \mathscr{J}\}$    ▷ Current and next decision points
9:  $\mathscr{A} \leftarrow A \setminus \{0\}$, $Q \leftarrow \{J \in \mathscr{J} \mid r_J = 0\}$    ▷ $\mathscr{A}$: 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:         $S, R_s, t_{\mathsf{next}} \leftarrow$ GETSCHEDSET$(Q, \{\beta_J, \mathsf{ins}_J | J \in Q\}, I, R_{\max}, t_{\mathsf{next}}, m)$    ▷ Get $m$ jobs to schedule, budget used by $S$
14:         $(J, \delta_J) \leftarrow$ RESOURCEALLOC$(Q, \{\beta_J, \mathsf{ins}_J | J \in Q\}, S, R_s, t, t_{\mathsf{next}}, I, b)$    ▷ Select job $J$ to get resource $\delta_J$
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:         $d_J \leftarrow d_J - (c_J -$ GETFINISH$(J, \beta_J, \mathsf{ins}_J, t, t_{\mathsf{next}}))$; $c_J \leftarrow$ GETFINISH$(J, \beta_J, \mathsf{ins}_J, t, t_{\mathsf{next}})$    ▷ Update $d_J$ and $c_J$
18:         if $J \notin S$ then
19:             $J_{\max} \leftarrow \arg\max_{J' \in S}\ d_{J'}$    ▷ 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$(J, \beta_J, \mathsf{ins}_J, t_{\mathsf{next}}, \infty)$    ▷ Reset any unscheduled jobs
24:     for all $J \in S$ do    ▷ Update scheduled jobs and successors
25:         $\mathsf{ins}_J \leftarrow$ COMPUTEINS$(J, \beta_J, \mathsf{ins}_J, t, t_{\mathsf{next}})$
26:         if $\mathsf{ins}_J = \mathsf{maxIns}_{\tau \equiv J}$ then    ▷ $J$ finished
27:             $c_J \leftarrow t_{\mathsf{next}}$; $\mathsf{done}_J \leftarrow$ true    ▷ Check for any new job releases
28:             $\mathsf{ReadySucc} \leftarrow \{J_{\mathsf{succ}} \mid J_{\mathsf{succ}} \in \mathsf{successors}(J) \wedge \mathsf{done}_{J_{\mathsf{pred}}} =$ true $\forall J_{\mathsf{pred}} \in \mathsf{predecessors}(J_{\mathsf{succ}})\}$
29:             for all $J_{\mathsf{succ}} \in \mathsf{ReadySucc}$ do $r_{J_{\mathsf{succ}}} \leftarrow c_J$; $c_{J_{\mathsf{succ}}} \leftarrow$ GETFINISH$(J_{\mathsf{succ}}, \beta_{J_{\mathsf{succ}}}^{init}, 0, r_{J_{\mathsf{succ}}}, \infty)$
30:             $Q \leftarrow (Q \setminus \{J\}) \cup \mathsf{ReadySucc}$    ▷ Add any newly released jobs to $Q$, remove $J$
31:     $\mathsf{Schedule}[t] \leftarrow \{(J, \beta_J) \mid J \in S\}$    ▷ Save jobs and budgets for decision point $t$
32:     if $Q = \emptyset \wedge \mathscr{A} = \emptyset$ then break    ▷ No more jobs, complete algorithm
33:     if $Q = \emptyset$ then $t \leftarrow \min(\mathscr{A})$ else $t \leftarrow t_{\mathsf{next}}$    ▷ Prepare for next segment, update current decision point
34:     for all $(J \in \mathscr{J} \mid r_J = t \wedge \mathsf{parents}(J) = \emptyset)$ do
35:         $Q \leftarrow Q \cup \{J\}$    ▷ Release ready source jobs, add to $Q$
36:     $\mathscr{A} \leftarrow \mathscr{A} \setminus \{t\}$    ▷ Remove $t$ from set of future anchor points
37:     $t_{\mathsf{next}} \leftarrow \min\{\ \min(\mathscr{A}),\ \min_{J \in Q} c_J\ \}$    ▷ 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 $c_{J_{i,j,k}} \leq A_{i,k} + D_i\ \forall J_{i,j,k} \in \mathscr{J}$    ▷ Schedulability test: $J_{i,j,k}$ is the kth job of task $\tau_j$ in taskgraph $G_i$
```

Figure 7.8: Pseudocode for RASCO algorithm.

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.

7.4.4 RASCO Algorithm Details

Algorithm 9 shows the pseudocode for RASCO. Using the initial release time, deadline and base allocation, RASCO computes a static schedule, Schedule. Each element $\mathsf{Schedule}[t_i]$ 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 $\mathscr{J}$ 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 $J \in \mathscr{J}$. For its co-allocation, RASCO also takes as input the total number of instructions $\mathsf{maxIns}_{\tau \equiv J}$ 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., $R_{\max} = (N_{\mathsf{ca}}, N_{\mathsf{bw}})$ 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, $\mathscr{A}$ 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 $J \in \mathscr{J}$, 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 $\mathscr{J}$ that occurs after t. Next, RASCO initializes the set of future anchor points $\mathscr{A}$ and constructs the set of ready jobs Q to consider for scheduling and resource allocation at time t.

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 10). 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 11), which uses the multi-phase execution model $\Theta_{\tau \equiv J}$ to determine the job J in Q that 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 12.

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 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.

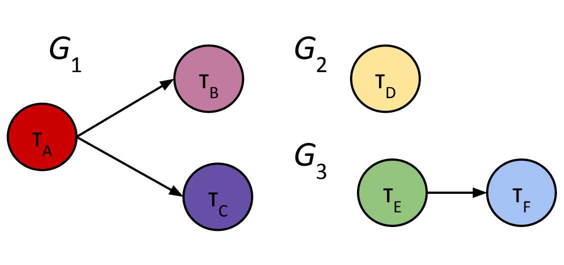
While there are more resources to give out, RASCO will repeat Step 1 (to update S and R_s) and Step 2 (to pick another job to receive a resource). When no more resources can be given, RASCO then moves to Step 3. Note that for RESOURCEALLOC to return null and for the inner while-loop to terminate, $R_{\max} - R_s$ 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 13) to compute the number of instructions ins_J that J would have completed by t_{next} (i.e., after executing in the segment $[t, t_{\mathsf{next}})$) 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), 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).

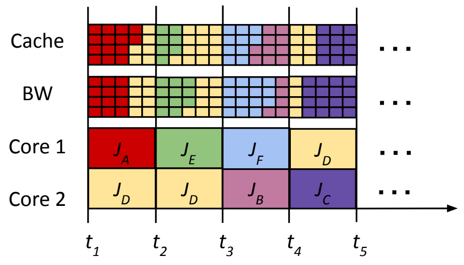
Finally, RASCO saves the current set of job-budget pairs in S to $\mathsf{Schedule}$ for decision point t (Line 31). If the ready set Q is empty and there is no future anchor point, then the static schedule is completed (Line 32) and the algorithm returns whether the taskset was schedulable (Line 39). 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 $\mathscr{A}$ (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 $[t, t_{\mathsf{next}})$ (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 $J_{i,j,k} \in \mathscr{J}$ 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 k^{th} 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 7.9 shows a simple taskset with three taskgraphs and the $\mathsf{Schedule}$ computed by RASCO.



(a) Example system with 3 taskgraphs



(b) Schedule with $N_{\mathsf{ca}} = 20$, $N_{\mathsf{bw}} = 20$, and $m = 2$

Figure 7.9: RASCO visualized. Note J_A represents the first job of task τ_A, etc.

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

GETSCHEDSET **(Algorithm 10):** 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 $c_J < t_{\mathsf{next}}$, 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

Algorithm 10 GETSCHEDSET

```
1: Input: $Q, \{\beta_J, \mathsf{ins}_J | J \in Q\}, I, R_{\max}, t_{\mathsf{next}}, m$
2: Output: $S, R_s, t_{\mathsf{next}}$          ▷ S: $m$ jobs with earliest deadlines, $R_s$: budget used by $S$
3: $S \leftarrow \arg\min_{Q' \subset Q,\, |Q'|=m} \sum_{J \in Q'} d_J$; $R_s \leftarrow \sum_{J \in S} \beta_J$
4: for $J \in S$ do
5:     if $c_J < t_{\mathsf{next}}$ then          ▷ Changing $\beta_J$ caused the segment boundary to change
6:         $t_{\mathsf{next}} \leftarrow c_J$          ▷ Update $t_{\mathsf{next}}$
7:         for all $J' \in Q \setminus \{J\}$ do          ▷ Reset all jobs in $Q$ aside from $J$
8:             $\beta_{J'} \leftarrow \beta_{J'}^{init}$; $d_{J'} \leftarrow d_{J'}^{init}$; $c_{J'} \leftarrow$ GETFINISH$(J', \beta_{J'}, \mathsf{ins}_{J'}, t, \infty)$
9:         $S \leftarrow \arg\min_{Q' \subset Q,\, |Q'|=m} \sum_{J' \in Q'} d_{J'}$; $R_s \leftarrow \sum_{J' \in S} \beta_{J'}$          ▷ Recompute $S$ and $R_s$
10: while $(R_s > R_{\max})$ do          ▷ Check if initial budgets exceed $R_{\max}$, if so take away resources
11:     $J_{\mathsf{save}} \leftarrow \arg\min_{J \in S | c_J = t_{\mathsf{next}}} d_J$          ▷ If any jobs define $t_{\mathsf{next}}$, save one of them from resource removal
12:     $J \leftarrow \arg\max_{J' \in S \,|\, J' \neq J_{\mathsf{save}}} d_{J'} - c_{J'}$          ▷ Job with largest slack (that does not specify $t_{\mathsf{next}}$)
13:     $\theta_i \leftarrow \Theta_{\tau \equiv J | \beta}[\mathsf{ins}_J]$          ▷ Current phase of $J$
14:     $\delta_J \leftarrow \arg\min_{e_j \in I \,|\, e_j R_s > e_j R_{\max}} \; \theta_i^{\Delta}[R_{\max} \cdot e_j - e_j]$          ▷ Unit vector for least impactful resource type
15:     $\beta_J \leftarrow \beta_J - \delta_J$; $c_J \leftarrow$ GETFINISH$(J, \beta_J, \mathsf{ins}_J, t, t_{\mathsf{next}})$          ▷ Take away this resource from $J$
16:     $R_s \leftarrow R_s - \delta_J$          ▷ Update $R_s$
17: Output: $S, R_s, t_{\mathsf{next}}$
```

Algorithm 11 RESOURCEALLOC

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

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 11):** 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 $R_{\mathsf{avail}} = R_{\max} - R_s$. 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

Algorithm 12 GETFINISH

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

Algorithm 13 COMPUTEINS

```
1:  Input: $J, \beta_J, \mathsf{ins}_J, t, t_{\mathsf{next}}$    ▷ Given that $J$ is at instruction count $\mathsf{ins}_J$ at time $t$ with resource budget $\beta_J$
2:  Output: $\mathsf{ins}_J$    ▷ Compute instruction count at $t_{\mathsf{next}}$
3:  $\theta_i \leftarrow \Theta_{\tau \equiv J|\beta_J}[\mathsf{ins}_J]$    ▷ Find current phase $\theta_i$ using $\mathsf{ins}_J$, where $J$ is a job of task $\tau$
4:  $t_{\mathsf{left}} \leftarrow (t_{\mathsf{next}} - t)$; $\mathsf{insRetired} \leftarrow 0$    ▷ Get time until $t_{\mathsf{next}}$ and initialize instructions retired
5:  while $t_{\mathsf{left}} > 0$ do
6:      $\mathsf{startIns} \leftarrow \max(\theta_i^s, \mathsf{ins}_J)$; $\mathsf{endIns} \leftarrow \theta_i^e$    ▷ Get number of instruction in current phase
7:      $t_{\mathsf{phase}} \leftarrow \min(t_{\mathsf{left}}, (\mathsf{endIns} - \mathsf{startIns})/\theta_i^r)$    ▷ Compute time that $J$ will spend in this phase
8:      $\mathsf{insRetired} \leftarrow \mathsf{insRetired} + (\theta_i^r \cdot t_{\mathsf{phase}})$    ▷ Update $\mathsf{insRetired}$
9:      $t_{\mathsf{left}} \leftarrow t_{\mathsf{left}} - t_{\mathsf{phase}}$    ▷ Subtract time spent in this phase from $t_{\mathsf{left}}$
10:     $i \leftarrow i + 1$    ▷ Go to next phase
11: $\mathsf{ins}_J \leftarrow \mathsf{ins}_J + \mathsf{insRetired}$
12: Output: $\mathsf{ins}_J$
```

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 12):** 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.

7.4.5 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 $2 \cdot |\mathscr{J}|$ times since each iteration corresponds to a decision point, and there are at most two unique decision points for every job in $\mathscr{J}$ (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 $\beta_J^{init} = \mathbf{0}\ \forall J \in Q$, 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., $c_J < t_{\mathsf{next}}$). 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 $O(|Q|^2 \cdot |\mathscr{J}|)$. 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).

7.5 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, following a similar approach to [229]. 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 [48] developed by [229] 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 [66]). 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 $\beta_{\mathsf{even}} = (N_{\mathsf{ca}}/m, N_{\mathsf{bw}}/m)$, obtained through measurements (as discussed in Section 7.3.1). 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 [229]). 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 7.3.5 to construct a multi-phase model for each benchmark from its execution profile, obtained via measurements (Sec. 7.3.1). 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.

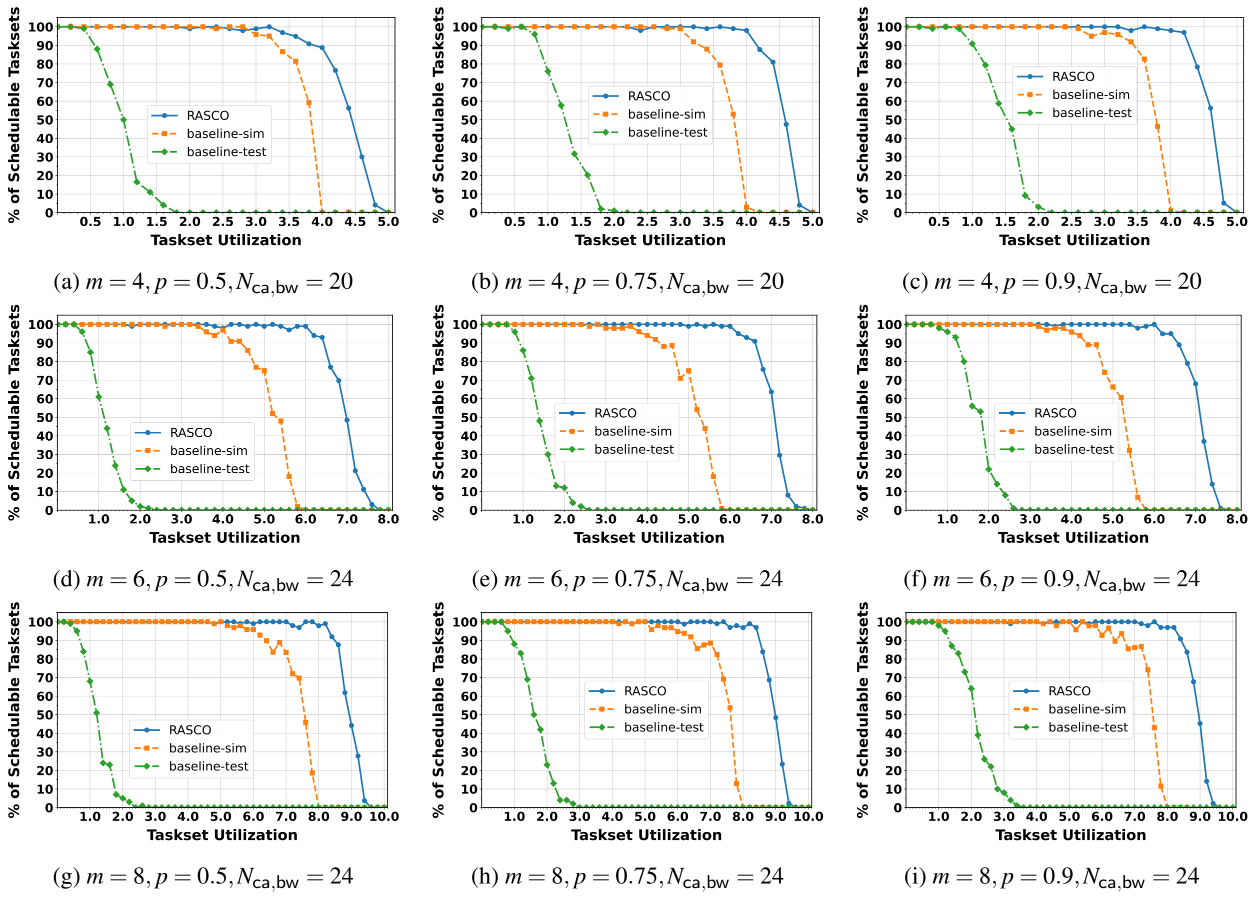
Implemented algorithms. The output of RASCO is a static schedule for a hyper-period which follows global EDF, where the job deadlines are computed based on the fine-grained resource allocation and deadline decomposition co-design. As we are not aware of any prior work on the co-design of resource allocation 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 $\beta_{\mathsf{even}} = (N_{\mathsf{ca}}/m, N_{\mathsf{bw}}/m)$ 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 [99] 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 [99]. 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.

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 LOCs in total).

Schedulability results. Fig. 7.10 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 $N_{\mathsf{ca}} = 20$ cache partitions and $N_{\mathsf{bw}} = 20$ bandwidth partitions. Therefore, both baselines have static resource partitions of size $\beta_{\mathsf{even}} = (5_{ca}, 5_{bw})$ for each task on each core. Figures 7.10a, 7.10b, and 7.10c show the schedulability results for $p = 0.50$, $p = 0.75$,



(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) $m = 8, p = 0.9, N_{\text{ca,bw}} = 24$

Figure 7.10: Percent of schedulable tasksets for increasing p values (L to R) and increasing cores (top to bottom).

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 Fig. 7.10c, 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 ($\beta_{\mathsf{even}} = (5_{ca}, 5_{bw})$) is relatively large when we 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 $N_{\mathsf{ca}} = 20$ and $N_{\mathsf{bw}} = 20$ partitions) has $m = 8$ cores. This means that an even split of these resources to cores would give $\beta_{\mathsf{even}} = (2.5_{ca}, 2.5_{bw})$, 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 $N_{\mathsf{ca}} = N_{\mathsf{bw}} = 24$ to have the same total amount of resources for all three algorithms when $m = 6$ and $m = 8$.

The second row of Fig. 7.10 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 $\beta_{\mathsf{even}} = (4_{ca}, 4_{bw})$ while RASCO is able to dynamically reallocate all $N_{\mathsf{ca}} = N_{\mathsf{bw}} = 24$ 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.

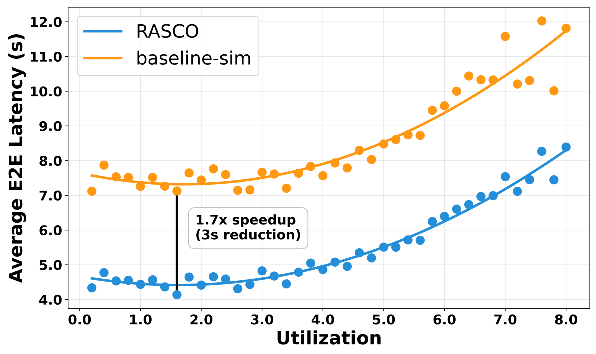
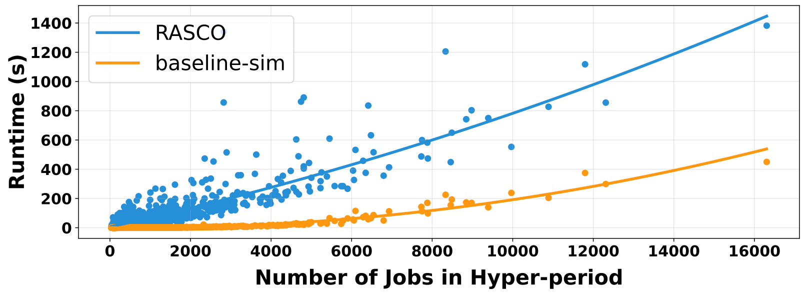


Figure 7.11: Latency of RASCO vs BASELINE-SIM ($m = 6$)

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*. Fig. 7.11 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.



Algorithm	Min. (s)	Median (s)	Max. (s)
RASCO	1.506	19.284	1382.011
BASELINE-SIM	0.004	0.112	450.137

Figure 7.12: Runtime data ($m = 6$)

Runtime comparison. Fig. 7.12 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 Sec. 7.4.5). 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 Fig. 7.12 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).

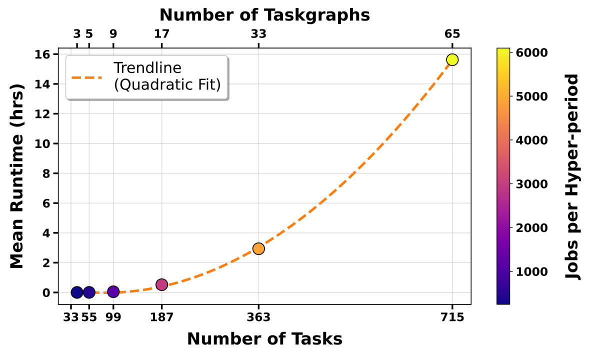


Figure 7.13: RASCO runtime vs. taskset size ($m = 8$)

Runtime scalability. To evaluate the scalability of RASCO, we performed an additional study with exponentially increasing taskset sizes. To clearly 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. Fig. 7.13 shows the results. The largest taskset size contained 65 taskgraphs with 715 tasks and had a mean runtime of $\sim$16 hours, which, although large, is feasible for an offline algorithm with known runtime complexity.

7.6 Prototype, Runtime overheads, and Evaluation

To evaluate the runtime overheads, safety, and practical utility of RASCO, we implemented a prototype of a RASCO runtime scheduler within a real-time operating system (RTOS) and measured its runtime overheads. We then presented a method for incorporating these overheads into RASCO, and experimentally evaluated the safety and utilization overhead of its overhead-aware schedules.

7.6.1 RASCO Runtime Scheduler Prototype

Our prototype was implemented in LITMUS^{RT} [34], 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 $\approx$ 880 LoCs.

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.[11] 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 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.[12] Once the current segment ends, the core's scheduler schedules jobs based on the scheduling table as before.

Resource allocation. Our prototype leverages Intel's CAT [95] for cache allocation and MemGuard [224] for memory bandwidth allocation. Using CAT, 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

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

[12]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.

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 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$^{\text{RT}}$, 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.

7.6.2 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 $N_{\mathsf{ca}} = N_{\mathsf{bw}} = 20$ 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 7.3.1, 7.3.5, and 7.5, 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.

7.6.3 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 7.1.

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 7.1: Measured direct runtime overheads of RASCO prototype over all utilizations

In Table 7.1, "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. 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. Fig. 7.4), 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 $(\beta_{\mathsf{ca}}, \beta_{\mathsf{bw}})$ 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

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

Table 7.2: Time to fill a 2MB L3 cache partition by memory bandwidth budget ($\beta_{\mathsf{bw}} \cdot 72$ MB/s)

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 7.2 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. Sec. 7.6.1). 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

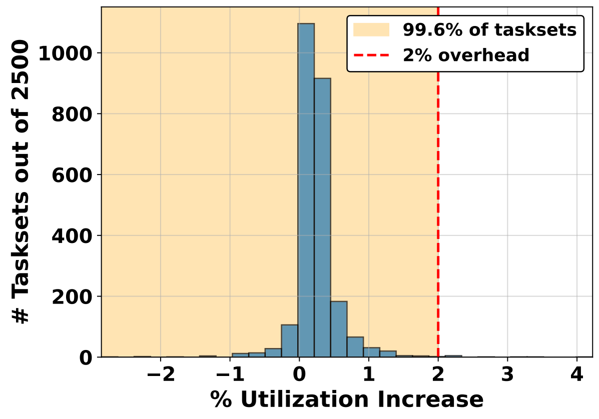


Figure 7.14: Percentage of utilization increase: overhead-aware vs. standard RASCO.

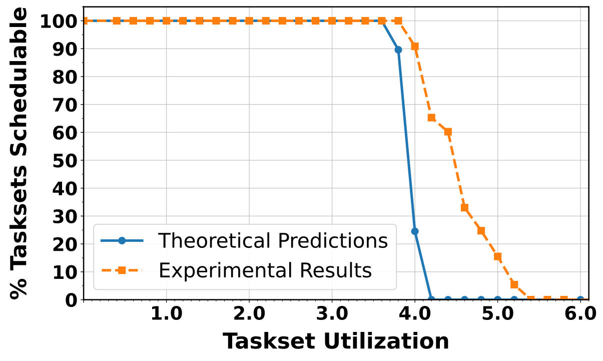


Figure 7.15: Schedulability for overhead-aware RASCO: experimental results vs. theoretical prediction.

a new overhead-aware schedule for each generated taskset in Sec. 7.6.2 and calculated the percentage increase in CPU utilization compared to the original schedule. Fig. 7.14 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.

7.6.4 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.

Fig. 7.15 shows the percentage of schedulable tasksets observed experimentally, compared to those predicted through numerical analysis, across all *actual* taskset utilizations. The actual utilization of 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.

7.7 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 RASCO 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.

CHAPTER 8

SYSTEMS INTEGRATION

This chapter explains how the ideas in Chapters 4, 5, and 7 were realized as running systems across three generations of platforms. It begins with the historical implementations—DNA on Xen, followed by OMNI and RASCO on LITMUS^{RT}—and motivates the move away from a hypervisor-centric design toward an in-kernel research OS. It then presents the completed integration on upstream Linux 6.12 with PREEMPT_RT, which unifies the enforcement mechanisms and experiment orchestration necessary to enable future research.

8.1 From Xen to LITMUS^{RT}

8.1.1 DNA on Xen

We initially implemented DNA on the Xen hypervisor [17, 1] because the then–state-of-the-art static resource allocator, vC^{2}M (see Chapter 4), was implemented on Xen. Building DNA in the same environment provided the most direct and defensible comparison. The competing approaches shared the same virtualization layer, scheduler hooks, and measurement harness, which avoided confounding effects and complexity from cross-platform porting. As a result, observed differences in throughput, deadline performance, or overhead could be attributed to policy rather than platform idiosyncrasies. After completing the head-to-head evaluation against vC^{2}M, we chose to move away from the hypervisor setting. A hypervisor-centric implementation would have constrained subsequent chapters to virtualized deployments and complicated integration with native Linux mechanisms such as CAT, cgroups v2, `SCHED_DEADLINE`, and modern tracing infrastructure. Furthermore, many of the mechanisms needed to control shared resources operate at the physical-core granularity, which Xen virtualizes to the OS as vCPUs. This forced a one-to-one mapping of vCPUs to physical CPUs, defeating the abstraction and undermining a central benefit of virtualization. Therefore, to avoid limiting the scope of the work to hypervisor environments—and to place enforcement where it actually occurs—we transitioned to an in-kernel path for the later systems.

8.1.2 Why LITMUS$^{\text{RT}}$

LITMUS$^{\text{RT}}$ [34] is an excellent kernel sandbox for developing and evaluating new real-time scheduling policies. It provides direct access to dispatching paths, precise control over preemption and interrupt threading, and an established methodology for measuring overheads in the presence of real-time workloads. Moving *into* the kernel addressed the granularity and timing issues encountered with Xen by allowing resource allocation to interact with enforcement mechanisms at the point they take effect, rather than through a hypervisor layer of indirection.

Equally important, LITMUS$^{\text{RT}}$ enabled rapid iteration. Because LITMUS$^{\text{RT}}$ already offered stable user–kernel interfaces and exemplar scheduling plugins, we could adapt those hooks and APIs rather than reimplementing them from scratch. As a result, mode-aware transitions and runtime hints could be introduced and evaluated without the engineering overhead of a full mainline integration. This, in turn, enabled the second and third systems—OMNI and RASCO—to grow their own runtime support while retaining a consistent experimental framework.

8.1.3 Omni and RASCO on LITMUS$^{\text{RT}}$

OMNI and RASCO were implemented on LITMUS$^{\text{RT}}$ to leverage its existing scheduler infrastructure and measurement toolkit. OMNI required a notion of global mode changes with bounded transition latency, so we extended the scheduler with a mode-aware wrapper that allows tasks to change real-time parameters and resource allocations atomically at well-defined instants. Achieving this behavior demanded substantial modifications to LITMUS$^{\text{RT}}$, which was not originally designed for this level of preemption and dynamic task reconfiguration. By contrast, RASCO's runtime was simpler: it used baseline LITMUS$^{\text{RT}}$ to execute a RASCO-computed scheduling table with lightweight hooks for node readiness and completion. We also added safeguards for a soft real-time variant in which long-running "runaway" tasks could cause divergence from the preset table, enabling the runtime to steer execution back toward feasibility without abandoning precedence constraints. For completeness and baseline comparison, we ported DNA to LITMUS$^{\text{RT}}$, which removed the need to maintain a parallel Xen codebase.

This phase demonstrated that the algorithms could run *in kernel time* with instrumentation precise enough to attribute costs to specific events such as resource allocations, throttle wakeups, and task migrations. At the same time, it exposed the long-term costs of evolving a research kernel: forward-porting patches across upstream changes, introducing multi-mode support that strained LITMUS^{RT}'s original APIs and abstractions, and limited interoperability with emerging mainline facilities such ascgroups v2. Over several years of additions, our LITMUS^{RT} branch diverged in multiple directions and lacked a unified framework that could be cleanly maintained. Compounding this, LITMUS^{RT} was no longer under active maintenance, leaving our platform increasingly out of date relative to upstream Linux.

8.2 Why Upstream: Linux 6.12 with PREEMPT_RT

Historically, stock Linux was ill-suited for hard or firm real-time work because large portions of the kernel executed non-preemptibly, interrupt service routines ran in hard-IRQ context with potentially long and unbounded handler chains, and substantial work was deferred into softirqs and bottom halves that could not be preempted by real-time tasks. These properties produced long and highly variable latencies in precisely the paths that enforcement and scheduling decisions depend on, which is why prior real-time research often relied on specialized frameworks such as LITMUS^{RT} to obtain predictable dispatch and measurement. PREEMPT_RT is a long-running effort to retrofit the mainline kernel with real-time semantics by making almost all in-kernel code preemptible, threading most interrupt handlers as kernel threads with real-time priorities, converting many spinlocks into priority-inheriting `rt_mutexes`, and moving softirq work into schedulable contexts. With Linux 6.12, PREEMPT_RT is included in mainline, making this a natural point to base the next integration on upstream Linux alone.

The next integration targets Linux 6.12 with full PREEMPT_RT support because it aligns with the needs of DNA, OMNI, and RASCO. First, PREEMPT_RT shortens non-preemptible sections and threads most IRQ handlers, which makes the timing of enforcement paths predictable and reduces the latency tails that directly affect throttle wakeups and resource allocations. Second, upstream Linux provides the mechanisms where real systems run: cache allocation via CAT, memory-bandwidth monitoring, cgroups v2 for placement and `SCHED_DEADLINE` configuration, and modern perf/eBPF tooling for measurement and profiling. Implementations that rely on these facilities better enable a deployable system and result in artifacts that are simpler to replicate. Third, platform breadth matters for external validity, and upstream kernels carry wider hardware support and driver coverage so that results that hold across multiple LLC geometries and microarchitectures are less likely to be artifacts of a narrow configuration. If future work expands resource allocation to new domains—e.g., heterogeneous platforms that include GPUs—modern Linux driver support is essential. Finally, minimizing carry-along patches and code-bases that included abstraction breaking code reduces the maintenance burden for future researchers and lowers the barriers for others to learn about, evaluate, and extend the work.

8.3 The Completed Integration on Linux 6.12

The present integration on Linux 6.12 implements DNA together with a new machine-learning phase classifier on top of the runtime and enforcement mechanisms described in this chapter. All of the supporting infrastructure is in place, including mechanisms for last-level cache and memory-bandwidth control, a unified kernel interface, a single trace and measurement pipeline, and experiment automation. The interface and tools are designed so that OMNI and RASCO can reuse the same mechanisms without reworking the lower layers. We additionally extended functionality to control CPU frequency on a per-CPU basis and to record power consumption at per-core granularity to enable future investigations into managing CPU speed as an allocation variable under power objectives.

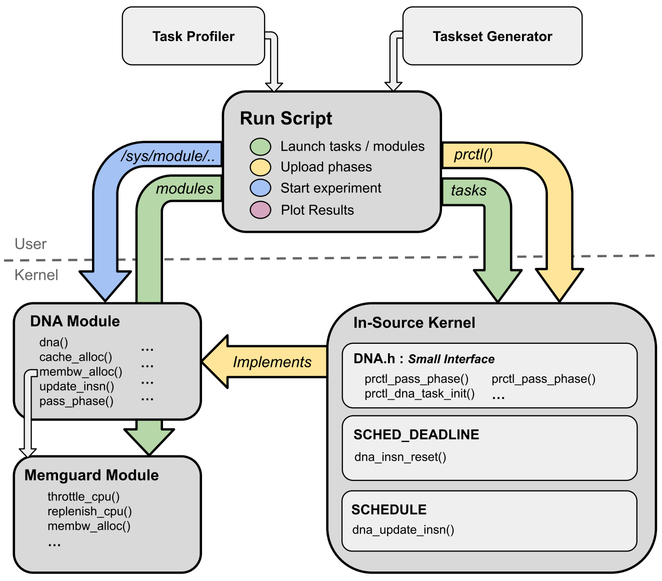


Figure 8.1: Design overview of DNA on Linux 6.12

8.3.1 Architecture and Interface

Figure 8.1 summarizes the Linux 6.12 implementation and anchors the discussion that follows. The leading design principle was to move all policy and most glue code *out of the Linux source tree* into a separate loadable module, keeping in-tree changes as small and auditable as possible. This yields several benefits: algorithm updates no longer require full kernel rebuilds or reboots, crashes are isolated to the module rather than destabilizing unrelated subsystems, forward-porting to new kernels is simpler because the in-tree delta stays small, A/B comparisons are easy because policies can be swapped by loading different modules (e.g., loading the ML-based policy module in place of the classical DNA module), and reproducibility improves because the kernel commit and module hashes can be version-pinned independently by the run script. It also reduces developer friction by enabling fast iteration cycles and by keeping the boundary clear between low-level mechanism (in-tree hooks) and policy (module code and run-time control).

At a high level, the implementation is organized into three layers: (i) a thin, in-tree interface and scheduler hooks that expose the necessary control points to policy; (ii) an out-of-tree DNA module that implements periodic progress tracking and resource allocation and programs the hardware controllers; and (iii) a user-space orchestration layer that launches workloads, configures cgroups, sets experiment metadata, and coordinates start barriers. The remainder of this subsection follows this structure.

Kernel-facing interface and hooks. To support the separation between mechanism and policy, the kernel includes a very small header and stub library (`DNA.h`) that exports a handful of symbols the scheduler and control paths may call. At build time these stubs satisfy references so the kernel compiles and boots even when the DNA module is not present, and at run time the module installs real implementations for the same symbols when it is loaded. This keeps the kernel free of policy code while preserving a typed, explicitly documented interface that the module must implement.

The upload path extends `prctl()`, Linux's per-thread control system call for configuring process attributes via compact numeric commands with typed arguments. Using `prctl()` avoids introducing a new device node or ioctl ABI, works naturally with cgroups, enforces well-defined permission checks, and lets a thread register its metadata early in initialization without extra file-descriptor plumbing. Our `PR_DNA_*` operations let a thread mark itself as DNA-managed, enable instruction-count tracking, and bind per-run workload identifiers needed by the policy.

Beyond the header stubs and control path, the in-tree changes are localized and small. First, we add a helper in the `SCHED_DEADLINE` path that resets the per-job "instruction sum" used by DNA whenever a new job is released. Second, we invoke instruction accounting from the low-level `context_switch()` routine. The `context_switch()` function is the architecture-independent handoff where the kernel finalizes the outgoing task and installs the incoming one, so attributing recently retired instructions at this point ensures progress is charged to the correct task at every preemption and voluntary yield with bounded overhead. Third, we add a small DNA metadata structure to the main Linux task struct to hold the instruction sum, the experiment workload type for the task, and cached policy state such as the next phase boundary. These hooks keep the scheduler maintainable while providing accurate accounting for the module.

DNA policy module and resource controllers. When the DNA module is loaded, it replaces the kernel stubs with working implementations and supplies the policy itself. It includes the DNA algorithm, the code that interprets per-thread `prctl()` settings and metadata, and high-resolution timers that periodically update instruction totals used in progress estimation. The module applies resource allocation by programming resctrl/CAT for last-level cache partitions, coordinating with MemGuard for memory-bandwidth limits, and setting optional per-CPU frequency caps so that an update produces a consistent configuration for the next epoch. MemGuard is itself a kernel module; because Linux 6.12 with PREEMPT_RT threads interrupt work and alters priority interactions, we made adjustments to keep throttling timely under real-time load.

MemGuard required targeted adjustments to operate correctly on a kernel where interrupt work is threaded and most kernel paths are preemptible. In prior kernels, MemGuard assumed that hard-IRQ work would run to completion immediately and that `SCHED_FIFO` throttle threads would be the highest-priority activity during enforcement windows. On PREEMPT_RT, IRQ work can run as kernel threads with real-time priorities, and `SCHED_DEADLINE` tasks may preempt `SCHED_FIFO` activity, which can delay throttle wakeups and degrade the timeliness of memory-bandwidth control—or even prevent it entirely. To restore timely enforcement, we implemented a narrow per-CPU priority gate that prevents `SCHED_DEADLINE` tasks from delaying throttle execution during the short window in which MemGuard applies throttling. We also configured MemGuard's high-resolution timers to execute in hard-IRQ context, preventing the work from being deferred to a scheduling thread and ensuring it is processed immediately. While this departs from PREEMPT_RT's general philosophy of minimizing unpreemptible kernel code, the MemGuard work performed in hard-IRQ context is strictly bounded. These changes keep memory-bandwidth control predictable while preserving PREEMPT_RT's preemptibility for normal operation.

Experiment orchestration. Experiments are declared as paths to directories that contain taskset descriptions grouped by total utilization, with each subdirectory holding the binaries, arguments, and CPU affinities for a single run. Per-run scripts specify policy parameters for DNA and pass them to the kernel through our `prctl()` extension so that each thread registers its metadata and experiment identifier during initialization. Initial cache and memory-bandwidth settings are distributed evenly across the configured experiment cores so that warm-up behavior is consistent before DNA begins adapting allocations. The run script then provisions the cgroups v2 layout, loads required modules, warms caches, launches all tasks in the selected taskset, captures a unified trace, and renders the figures referenced in the DNA chapter. This arrangement allows complete end-to-end reproduction by pointing the runner at a specific experiment directory and parameter configuration.

Workloads are launched under `SCHED_DEADLINE` using standard tooling (e.g., `chrt` or an equivalent wrapper) after per-thread metadata has been registered via `prctl()`. Because `SCHED_DEADLINE` is global by default, the runner places the workload into a dedicated cgroup v2 subtree and uses the `cpuset` controller to confine the tasks to the CPU subset specified by the experiment configuration (for example, CPUs `0-3`). This ensures that only the intended cores execute the workload and that cache, bandwidth, and frequency settings are applied to the same subset.

To achieve a simultaneous release, the runner freezes the workload cgroup using the cgroup v2 freezer before admitting tasks, which causes threads to block once they reach the runnable state. After modules are loaded, metadata is registered, and tasks are staged, the runner writes to a simple sysfs control point exported by the module at `/sys/module/dna_module/experiment_start` to mark the start of the experiment. When the flag is set, the module records a start timestamp and applies the first resource allocation, and the runner immediately unfreezes the cgroup so that all threads are released together at a well-defined instant. Aligning the barrier, the first allocation, and the unified trace's timestamps simplifies analysis and improves repeatability across runs.

8.3.2 Cache, Memory Bandwidth, and CPU-Frequency Control

Last-level cache control uses CAT, with way-mask changes coalesced per core to minimize MSR writes during an update step. Memory-bandwidth control is provided by the PREEMPT_RT-aware port of MemGuard that runs a throttle thread per core under `SCHED_FIFO` and uses high-resolution timers for refill and accounting, similar to previous implementations. CPU frequency can be set per core, and frequency changes are applied together with cache and bandwidth updates so that all settings reflect the same allocation decision. Coupling frequency with cache and bandwidth enables DNA to express energy–performance trade-offs at epoch boundaries. Future work is needed to determine the policy for how to best allocate CPU frequency with these other resources.

8.3.3 Machine-Learning Phase Classification for DNA

The machine-learning model (work done in collaboration with Professor Ricardo Sanfelice and colleagues at the University of California, Santa Cruz) replaces DNA's phase-detection pipeline by directly inferring a task's current execution rate from the same performance-counter inputs previously used for profiling and clustering. Instead of constructing phase clusters offline and then identifying the active cluster online, the model predicts the near-term rate of progress for the task at runtime, which the policy then translates into a resource allocation for the next epoch. This removes the need for explicit phase identification and eliminates the associated clustering step while preserving the intent of DNA's benefit-based allocation. A second motivation for the model is sensitivity to input-dependent behavior: by conditioning on recent measurements, it can detect shifts caused by different input paths in the benchmarks, which DNA's discrete phase detector could not reliably capture.

Training occurs offline using traces collected with the same instrumentation implemented to support the new Linux 6.12 port of DNA. At runtime, inference runs in the policy's loadable kernel module and can serve as input to DNA to inform resource allocation. In practice, we hope this design preserves DNA's low overhead while enabling faster adaptation to workload changes and greater robustness to input variability.

8.3.4 Status and Path for Future Systems

While this chapter evaluates DNA on Linux 6.12, the same interface is intended to support OMNI and RASCO. With an extension of the current design, both systems can publish their mode changes and DAG events through `prctl()` and mark mode transitions via the `/sys/module/` interface. This keeps all three systems on a common set of kernel hooks and controllers, simplifying future ports and comparative evaluation.

8.3.5 Artifacts

We release Linux 6.12 patches and loadable modules for last-level cache control (CAT), memory-bandwidth control (MemGuard with PREEMPT_RT adjustments), and per-CPU frequency control. The kernel–user interface is provided through a small family of `prctl()` commands (`PR_DNA_*`) together with a sysfs control point under `/sys/module/DNA/` for synchronized experiment release and status. The in-tree changes are intentionally minimal and include per-task metadata needed by DNA and support for recording instructions retired to track task progress with bounded overhead. We include a thin C library that wraps the `prctl()` commands and exposes helper routines to initialize task metadata. We provide both out-of-tree DNA policy modules—the implementation from Chapter 4 and the machine-learning–driven variant—along with example configuration files and module parameters. The overheads of our implementation are shown in Table 8.1. We include unified trace tools that record allocation timing, throttle activity, cache reprogram costs, per-core frequency settings, and energy samples, alongside parsers and the figure-rendering scripts used in the DNA chapter. We provide the experiment runner and the tools to generate complete tasksets organized by utilization, as well as basic stress tests for regression (MemGuard timing, DNA allocation updates, `prctl()` data paths, metadata initialization, and start-barrier correctness) and the scripts to execute them. We include a kernel configuration fragment for 6.12 with PREEMPT_RT options, build instructions, and a short guide for reproducing the DNA figures end-to-end on supported platforms.

Table 8.1: Runtime Overhead of DNA and Mechanisms in μs

Type	Count	Mean	90th	95th	99th	Max
DNA	28347	1.07	1.54	2.01	6.20	10.73
Res. Alloc. Total	124392	25.1	28.9	30.9	35.8	45.4
Cache Aloc.	124392	1.70	1.70	1.80	1.80	9.60
Membw. Aloc.	124392	0.20	0.20	0.30	1.40	6.90
Freq. Alloc.	124392	7.90	9.40	9.80	10.9	27.0

CHAPTER 9

CONCLUSION

9.1 Summary of Results

This dissertation investigated how to dynamically allocate shared-resources across a variety of settings, often incorporating co-design principles to deliver efficient and predictable real-time performance. It advanced the state of the art through four main thrusts that together form a progressively richer body of work: phase-aware resource allocation (DNA/DADNA), multi-mode resource allocation (Omni), control-and-allocation co-design for CPS (DECNTR), and DAG-aware runtime co-design (RASCO). Below we synthesize the principal findings of each thrust and the combined lessons they yield.

9.1.1 DNA and DADNA: Phase-Aware Dynamic Allocation for Soft Real-Time

We introduced DNA, which leverages task profiles to learn phase structure and resource sensitivities, and a deadline-aware extension DADNA that integrates slack awareness into allocation. DNA/DADNA couples an offline profiler (to learn execution phases and per-phase rate–vs–budget relations) with an online allocator that dynamically reassigns LLC capacity and memory bandwidth to the tasks with the highest current benefit. A Xen-based prototype with MemGuard bandwidth control and Intel CAT cache partitioning demonstrated low runtime overheads and substantial improvements in schedulability and tail latency relative to a static baseline. These results establish that fine-grain dynamic resource reallocation, even at 5ms granularity, can improve resource utilization and decrease job misses. This work strongly establishes that there is substantial performance to be gained from intelligently controlling resources over time and that these benefits outweigh reallocation overhead costs.

9.1.2 Omni: Multi-Mode on Multicore

We developed *Omni*, an end-to-end algorithm that combines mode-aware schedulability analysis with dynamic allocation of cache and memory bandwidth across modes. Omni selects resource budgets and processor affinities that remain feasible across mode changes and, at runtime, rebalances budgets to preserve deadlines during transitions. Prototype and numerical studies show that Omni improves observed schedulability and reduces deadline-miss ratios versus static multi-mode baselines, particularly at moderate to high utilizations. The key insight is that respecting mode-change semantics while allowing resource flexibility yields robust, high-utilization operation without sacrificing predictability.

9.1.3 DECNTR: Co-Designing Control Robustness and Real-Time Allocation

Recognizing that timing guarantees alone are insufficient in cyber-physical systems, DECNTR co-designs controller implementations and shared-resource budgets to jointly maximize both robustness and schedulability. Controllers expose multiple viable implementations, and the framework selects both controller variants and resource allocations while enforcing a safe mode-change protocol. Evaluations with canonical control workloads (e.g., DC motor, lane-keeping, adaptive cruise) show that DECNTR sustains safety margins across mode changes while preserving deadline constraints as workloads shift. This establishes a path to principled integration of control-theoretic robustness with OS-level resource management and opens an interesting design space of robustness and schedulability trade-offs for future research.

9.1.4 RASCO: Resource–Scheduling Co-Design for DAG Workloads

We proposed RASCO, which introduces an updated phase-based timing model and a co-design runtime for DAG-structured applications. The model captures per-phase worst-case instruction rates under varying cache and bandwidth budgets, enabling sound WCET aggregation under dynamic allocations. At runtime, RASCO uses a budget-aware heuristic to prioritize ready tasks and phases that cross critical resource thresholds, improving progress on the DAG's critical path while respecting real-time constraints. Empirically, phase-aware WCET estimation converges with modest runtime complexity, and dynamic co-design reduces end-to-end latency relative to static even-allocation baselines.

9.1.5 Combined Lessons

A central lesson of this dissertation is that *dynamic resource allocation is strongly worthwhile*. While phase behavior of tasks has long been recognized, it was historically thought that dynamic reallocation would incur prohibitive costs due to cache refills and allocator overhead. DNA showed that this assumption was overly pessimistic: static allocations leave significant efficiency untapped, and dynamic reallocation of cache and memory bandwidth can substantially improve schedulability and reduce latency.

Building on this, the subsequent techniques demonstrate the value of *holistic co-design*. By coupling allocation with additional aspects of the system—whether task deadlines, mode transitions, controller robustness, or DAG structure—stronger solutions emerge. Incorporating knowledge from control theory, multi-mode semantics, or DAG critical paths into resource-allocation decisions ensures both efficiency and safety. Together, these results show that real-time systems benefit most when allocation decisions are informed by and integrated with the broader system context.

Finally, *implementation matters*: isolating memory bandwidth and cache capacity, carefully measuring kernel overheads, and integrating with mature schedulers are essential to realize theoretical gains in practice.

9.2 Assumptions, Limitations, and Future Work

While our prototypes and studies span simulated and real platforms, several assumptions bound the present scope and suggest concrete next steps. We group them by platform, OS/runtime, modeling, and application domain.

9.2.1 Hardware Platforms

Our empirical results assume uniform memory access multicore platforms with uniform cores and OS-visible hardware support for partitioning for LLC (CAT)and memory bandwidth (performance counters). Future work should validate the techniques on NUMA systems, heterogeneous SoCs, and platforms with asymmetric cores (e.g., high and low power) or co-processors (e.g., GPUs) where long critical sections and migration costs differ. NUMA locality constraints may favor partitioned or clustered scheduling with memory-aware task placement. Heterogeneous cores invite controller- and phase-aware mapping that accounts for core microarchitectural differences or knowledge of co-processor usage.

9.2.2 OS/Runtime Integration

The first prototype targeted Xen with MemGuard and Intel CAT to control memory bandwidth and cache capacity. Later prototypes were built within Linux 4.19 with LITMUS^{RT} support. A key next step is a Linux 6.12 integration that (i) ports MemGuard and cache partitioning to contemporary kernels, (ii) exposes a low-latency user–kernel interface so learned phase predictions can steer allocation and scheduling, and (iii) provides repeatable profiling and tracing infrastructure to support learning-in-the-loop experiments. Our existing work has yet to explore threaded workloads in detail as this presents a unique challenge for profiling.

9.2.3 Deterministic Single Inputs

This dissertation assumes deterministic task behavior to ensure that profiles capture accurate phases. Because of this, we profile each task with only a single input. This is a major drawback as few real world tasks are limited by this constraint. This problem is partially handled by re-profiling the task with various inputs and generating phases for each in isolation, however profiling and phase generation takes time and often the size of the set of possible inputs is very large or unknown. Two on-going projects are looking to solve this problem through the use of learning techniques. The first looks to address the lengthy profiling process by using learning to generate profile data via multi-marginal schrodinger bridges. This technique requires only that we profile tasks with a few resource allocations, and then generate missing profiles with resource allocations that fall between the empirical ones. Doing so means that only a small fraction of real profiles need to be gathered and the model can generate the "gaps" between them, greatly reducing the time needed to produce a full set of profiles for DNA's phase generation. The second project looks to completely replace phase generation itself. The phase boundary techniques used in Chapter 4 (clustering) and Chapter 7 (changepoint detection) are simply used to identify points in the program that have similar hardware characteristics and the boundaries where this behavior changes. We can instead use a deep learning model, trained on empirical or generative profiles, to predict the rates of execution directly, no phase detection required. Alternatively, an online-learning approach could be used on an extra core to update any learned model for tasks where the compute input set is unknown.

9.2.4 Spatial Main Memory Interference

Our work considers allocating only main memory bandwidth and last-level cache. As briefly discussed in Chapter 2, there can also be interference on spatial main memory. To extend our work to account for this interference, one could naïvely expand the OS's memory allocator, as done in existing work, to ensure that no two threads share physical pages that map to the same dram bank. This will not serve as a complete solution, however. If the system has many threads or simply requires many pages, it will not always be possible to prevent simultaneous dram bank conflicts. More so, this type of naïve implementation does not leverage existing phase information to know when tasks are making large volumes of memory requests. Therefore, we envision a more tightly integrated solution that explores the tradeoff between dynamically changing the bank allocations of threads to the overhead that comes with copy main memory between different banks (since changing bank allocations involves copying memory to different DRAM locations). It is not directly clear if the time saved from reducing dram bank conflicts makes up for the large cost of copying memory.

9.2.5 Control Co-Design Implementations

The current version of DECNTR focuses on controllers that vary only their sampling period. This simplifies the design space but limits flexibility, as in practice different types of controller implementations—such as Model Predictive Control (MPC) versus Proportional–Integral–Derivative (PID)—offer distinct tradeoffs in robustness, runtime cost, and sensitivity to shared resources. Future work should therefore extend the framework to support multiple classes of controllers and explicitly model their computational demands and resource sensitivities. The allocation and selection algorithm should jointly decide not only the sampling period but also which controller implementation to employ, ensuring that both timing constraints and control objectives are satisfied under dynamic workloads.

9.2.6 Mixed-Criticality and Tooling

Investigating mixed-criticality scheduling and per-criticality dynamic allocation, could be a promising direction. A complete toolchain can be developed to bundle profilers, analyzers, and a reproducible benchmarking harness so others can evaluate new learning models and allocators on top of our algorithms for better reproducibility.

9.3 Closing Remarks

This dissertation shows that predictable real-time performance on modern multicores benefits from exposing phases, modes, and control robustness and from jointly managing time and shared resources. The resulting collection of algorithms offers a principled foundation for future real-time systems that are both adaptive and analyzable, and it charts a path toward robust, resource-aware autonomy on commodity platforms.

APPENDIX A

LIST OF PAPERS

This thesis is based on the following papers:

1. **Robert Gifford, Neeraj Gandhi, Linh Thi Xuan Phan, Andreas Haeberlen** "Dna: Dynamic resource allocation for soft real-time multicore systems," *RTAS*, 2021. Status: Published

2. **Robert Gifford, Linh Thi Xuan Phan** "Multi-mode on multi-core: Making the best of both worlds with omni," *RTSS*, 2022. Status: Published

3. **Robert Gifford, Felipe Galarza-Jimenez, Linh Thi Xuan Phan, Majid Zamani** "Decntr: Optimizing safety and schedulability with multi-mode control and resource allocation co-design," *RTAS*, 2024. Status: Published

4. **Abigail Eisenklam, Robert Gifford, Georgiy A. Bondar, Yifan Cai, Tushar Sial, Linh Thi Xuan Phan, Abhishek Halder** "Rasco: Resource Allocation and Scheduling Co-design for DAG Applications on Multicore," *EMSOFT*, 2025. Status: Published

Author's Contribution to Primary Work:

1. Lead author, DNA algorithm, and system design. System implementation.

2. Lead author, Omni algorithm, and system design. System implementation.

3. Lead author, Decntr algorithm, scheduling analysis, and numerical evaluation.

4. Co-lead author, co-design Rasco algorithm, and system design. System implementation.

Additional, but not primary, work leading up to this dissertation includes:

5. **Tian Yang, Robert Gifford, Andreas Haeberlen, Linh Thi Xuan Phan** "The synchronous data center," *HotOS*, 2019. Status: Published

6. **Meng Xu, Robert Gifford, Linh Thi Xuan Phan** "Holistic multi-resource allocation for multicore real-time virtualization," *DAC*, 2019. Status: Published

7. **Neeraj Gandhi, Edo Roth, Robert Gifford, Linh Thi Xuan Phan, Andreas Haeberlen** "Bounded-time recovery for distributed real-time systems," *RTAS*, 2020. Status: Published

8. **Georgiy A Bondar, Robert Gifford, Linh Thi Xuan Phan, Abhishek Halder** "Path structured multimarginal Schrödinger bridge for probabilistic learning of hardware resource usage by control software," *ACC*, 2024. Status: Published

9. **Georgiy A Bondar, Robert Gifford, Linh Thi Xuan Phan, Abhishek Halder** "Stochastic Learning of Computational Resource Usage as Graph-Structured Multimarginal Schrödinger Bridge," *IEEE Transactions on Control Systems Technology*, 2025. Status: Published

10. **Karan Newatia, Robert Gifford, Qingjie Lu, Andreas Haeberlen, Linh Thi Xuan Phan** "Running Distributed Systems Like Clockwork," *NINeS*, 2026. Status: Submitted

11. **Georgiy A. Bondar, Abigail Eisenklam, Yifan Cai, Robert Gifford, Tushar Sial, Linh Thi Xuan Phan, Abhishek Halder** "Generative Profiling for Soft Real-Time Systems and its Applications to Resource Allocation," *RTAS*, 2026. Status: Submitted

BIBLIOGRAPHY

[1] RT-Xen: Real-Time Virtualization Based on Compositional Scheduling. https://sites.google.com/site/realtimexen/.

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

[3] Bilge Acun, Kavitha Chandrasekar, and Laxmikant V. Kale. Fine-grained energy efficiency using per-core dvfs with an adaptive runtime system. In *IGSC*, pages 1–8, 2019.

[4] Homa Aghilinasab, Waqar Ali, Heechul Yun, and Rodolfo Pellizzoni. Dynamic memory bandwidth allocation for real-time gpu-based soc platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[5] Ankit Agrawal, Gerhard Fohler, Johannes Freitag, Jan Nowotsch, Sascha Uhrig, and Michael Paulitsch. Contention-Aware Dynamic Memory Bandwidth Isolation with Predictability in COTS Multicores: An Avionics Case Study. In *ECRTS*, 2017.

[6] Masud Ahmed and Nathan Fisher. Tractable schedulability analysis and resource allocation for real-time multimodal systems. *ACM Trans. Embed. Comput. Syst.*, 13(2s), jan 2014.

[7] Aaron D Ames, Xiangru Xu, Jessy W Grizzle, and Paulo Tabuada. Control barrier function based quadratic programs for safety critical systems. *IEEE Transactions on Automatic Control*, 62(8):3861–3876, 2016.

[8] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES: a tool for schedulability analysis and code generation of real-time systems. In *FORMATS*, 2003.

[9] Sylvain Arlot, Alain Celisse, and Zaid Harchaoui. A kernel multiple change-point algorithm via model selection. *Journal of machine learning research*, 20(162):1–56, 2019.

[10] ARM. PrimeCell level 2 cache controller (PL310) - technical reference manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0246c/index.html, 2015. Accessed: 2025-05-23.

[11] Arm Limited. Configuring mpam via resctrl file-system. https://developer.arm.com/documentation/108032/latest/A-closer-look-at-MPAM-software/Linux-MPAM-overview/Configuring-MPAM-via-resctrl-file-system, 2024. Accessed 2025-09-22.

[12] K.-E. Arzen, A. Cervin, J. Eker, and L. Sha. An introduction to control and scheduling co-design. In *Conference on Decision and Control (CDC)*, volume 5, pages 4865–4870 vol.5, 2000.

[13] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz,

Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 2009.

[14] Neil C Audsley, Alan Burns, MF Richardson, and AJ Wellings. Deadline monotonic scheduling theory. In *Real-Time Programming 1992*, pages 55–60. Elsevier, 1992.

[15] Akramul Azim and Sebastian Fischmeister. Efficient mode changes in multi-mode systems. In *ICCD*, 2016.

[16] Hyeongboo Baek, Kang G. Shin, and Jinkyu Lee. Response-time analysis for multi-mode tasks in real-time multiprocessor systems. *IEEE Access*, 8:86111–86129, 2020.

[17] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP*, 2003.

[18] Sanjoy Baruah. Improved multiprocessor global schedulability analysis of sporadic DAG task systems. In *ECRTS*, 2014.

[19] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *RTSS*, 1990.

[20] Soroush Bateni, Zhendong Wang, Yuankun Zhu, Yang Hu, and Cong Liu. Co-optimizing performance and memory footprint via integrated cpu/gpu memory management, an implementation on autonomous driving platform. In *RTAS*, 2020.

[21] Dimitri Bertsekas. Infinite time reachability of state-space regions by using feedback control. *IEEE Transactions on Automatic Control*, 17(5):604–613, 1972.

[22] Ashikahmed Bhuiyan, Mohammad Pivezhandi, Zhishan Guo, Jing Li, Venkata Prashant Modekurthy, and Abusayeed Saifullah. Precise scheduling of dag tasks with dynamic power management. In *ECRTS 2023*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

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

[24] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *MICRO*, 2008.

[25] Franco Blanchini. Set invariance in control. *Automatica*, 35(11):1747–1767, 1999.

[26] Franco Blanchini and Stefano Miani. *Set-theoretic methods in control*, volume 78. Springer, 2008.

[27] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 2011.

[28] Alan Burns. System mode changes-general and criticality-based. In *WMC*, 2014.

[29] G. C. Buttazzo, G. Lipari, and L. Abeni. Elastic task model for adaptive rate control. In *Real-Time Systems Symposium (RTSS)*, 1999.

[30] G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51(3):289–302, 2002.

[31] Giorgio Buttazzo, Manel Velasco, and Pau Marti. Quality-of-control management in overloaded real-time systems. *IEEE Transactions on Computers*, 56(2):253–266, 2007.

[32] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 3rd edition, 2011.

[33] Giorgio C Buttazzo, Marko Bertogna, and Gang Yao. Limited preemptive scheduling for real-time systems. a survey. *IEEE transactions on Industrial Informatics*, 2012.

[34] John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. LITMUS^{RT}: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *RTSS*, 2006.

[35] T. Caliński and J. Harabasz. A dendrite method for cluster analysis. *Communications in Statistics*, 3(1):1–27, 1974.

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

[37] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contenton on a chip multi-processor architecture. In *HPCA*, 2005.

[38] Nan Chen, Shuai Zhao, Ian Gray, Alan Burns, Siyuan Ji, and Wanli Chang. Precise response time analysis for multiple DAG tasks with intra-task priority assignment. In *RTAS*, 2023.

[39] T. Chen and L. T. X. Phan. Safemc: A system for the design and evaluation of mode-change protocols. In *RTAS*, 2018.

[40] Weifan Chen, Ivan Izhibirdeev, Denis Hoornaert, Shahin Roozkhosh, Patrick Carpanedo, Sanskriti Sharma, and Renato Mancuso. Low-overhead online assessment of timely progress as a system commodity. In *ECRTS*, 2023.

[41] Yixing Chen, Deqing Huang, Qiao Zhu, Weiqun Liu, Congzhi Liu, and Neng Xiong. A new state of charge estimation algorithm for lithium-ion batteries based on the fractional unscented kalman filter. *Energies*, 10(9):1313, 2017.

[42] Micaiah Chisholm, Ward Bryan C, Namhoon Kim, and James H. Anderson. Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In *RTSS*, 2015.

[43] Micaiah Chisholm, Namhoon Kim, Stephen Tang, Nathan Otterness, James H. Anderson, F. Donelson Smith, and Donald E. Porter. Supporting mode changes while providing hardware isolation in mixed-criticality multicore systems. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*. Association for Computing Machinery, 2017.

[44] C.-B. Cho and T. Li. Complexity-based program phase analysis and classification. In *PACT*, 2006.

[45] C.-B. Cho and T. Li. Using wavelet domain workload execution characteristics to improve accuracy, scalability, and robustness in program phase analysis. In *ISPASS*, 2007.

[46] Hoon Sung Chwa, Kang G Shin, and Jinkyu Lee. Closing the gap between stability and schedulability: A new task model for cyber-physical systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 327–337. IEEE, 2018.

[47] Cédric Courtaud, Julien Sopena, Gilles Muller, and Daniel Gracia Pérez. Improving prediction accuracy of memory interferences for multicore platforms. In *RTAS*, 2019.

[48] Xiaotian Dai. dag-gen-rnd: A randomized multi-DAG task generator for scheduling and allocation research, 2022.

[49] Xiaotian Dai, Shuai Zhao, Yu Jiang, Xun Jiao, Xiaobo Sharon Hu, and Wanli Chang. Fixed-priority scheduling and controller co-design for time-sensitive networks. In *Conference on Computer-Aided Design*, pages 1–9, 2020.

[50] Dakshina Dasari, Vincent Nelis, and Benny Akesson. A framework for memory contention analysis in multi-core platforms. *Real-Time Syst.*, 52(3):272–322, May 2016.

[51] D. L. Davies and D. W. Bouldin. A cluster separation measure. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(2):224–227, 1979.

[52] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 2011.

[53] Robert I. Davis and Liliana Cucu-Grosjean. A survey of probabilistic timing analysis techniques for real-time systems. *Leibniz Trans. Embed. Syst.*, 6(1):03:1–03:60, 2019.

[54] D. de Niz, K. Lakshmanan, and R. Rajkumar. On the Scheduling of Mixed-Criticality Real-Time Task Sets. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, 2009.

[55] D. de Niz and Linh T. X. Phan. Partitioned Scheduling of Multi-Modal Mixed-Criticality Real-Time Systems on Multiprocessor Platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[56] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 2013.

[57] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and QoS-aware cluster management. In *ASPLOS*, 2014.

[58] Michael L. Dertouzos. Control robotics: The procedural control of physical processes. In *IFIP Congress*, 1974.

[59] Sudarshan K. Dhall and C. L. Liu. On a real-time scheduling problem. *Oper. Res.*, 1978.

[60] A. Dhodapkar and J. E. Smith. Dyanmic microarchitecture adaptation via co-designed virtual machines. In *ISSCC*, 2002.

[61] A. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *ISCA*, 2002.

[62] Marco Di Natale and John Stankovic. Dynamic end-to-end guarantees in distributed real time systems. 1995.

[63] P. Dziurzanski, A. Singh, and L. Indrusiak. Multi-criteria resource allocation in modal hard real-time systems. In *J Embedded Systems*, 2017.

[64] Arvind Easwaran. Demand-based scheduling of mixed-criticality sporadic tasks on one processor. In *Real-Time Systems Symposium (RTSS)*, pages 78–87, 2013.

[65] Abigail Eisenklam, Robert Gifford, Georgiy A Bondar, Yifan Cai, Tushar Sial, Linh Thi Xuan Phan, and Abhishek Halder. Rasco: Resource allocation and scheduling co-design for dag applications on multicore. *ACM Trans. Embed. Comput. Syst.*, 2025.

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

[67] Giorgio Farina, Gautam Gala, Marcello Cinque, and Gerhard Fohler. Enabling memory access isolation in real-time cloud systems using intel's detection/regulation capabilities. *Journal of Systems Architecture*, 2023.

[68] Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-time systems*, 1999.

[69] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S. Wahby, and Keith Winstein. Salsify: Low-Latency network video through tighter integration between a video codec and a transport protocol. In *NSDI*. USENIX Association, 2018.

[70] FreeRTOS Developers. Freertos documentation overview. https://www.freertos.org/Documentation/00-Overview, 2025.

[71] M.E.M.B. Gaid, A. Cela, and Y. Hamam. Optimal integrated control and scheduling of networked

control systems with communication constraints: application to a car suspension system. *IEEE Transactions on Control Systems Technology*, 14(4):776–787, 2006.

[72] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1990.

[73] Stefan Valentin Gheorghita, Twan Basten, and Henk Corporaal. Intra-task scenario-aware voltage scheduling. In *CASES*, 2005.

[74] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[75] Robert Gifford, Felipe Galarza-Jimenez, Linh Thi Xuan Phan, and Majid Zamani. Decntr: Optimizing safety and schedulability with multi-mode control and resource allocation co-design. In *RTAS*, 2024.

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

[77] Robert Gifford and Linh Thi Xuan Phan. Multi-mode on multi-core: Making the best of both worlds with omni. In *RTSS*, 2022.

[78] S. Goddard and X. Liu. A variable rate execution model. In *ECRTS*, pages 135–143, 2004.

[79] Joël Goossens and Pascal Richard. Partitioned scheduling of multimode multiprocessor real-time systems with temporal isolation. In *Conference on Real-Time Networks and Systems (RTNS)*, pages 297–305, 2013.

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

[81] Arpan Gujarati, Mitra Nasri, and Björn B. Brandenburg. Quantifying the Resiliency of Fail-Operational Real-Time Networked Control Systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, volume 106, pages 16:1–16:24, 2018.

[82] George Gunter and Daniel Work. Safe driving with control barrier functions in mixed autonomy traffic when cut-ins occur. In *European Control Conference (ECC)*, pages 411–416. IEEE, 2022.

[83] Danlu Guo and Rodolfo Pellizzoni. A requests bundling dram controller for mixed-criticality systems. In *RTAS*, 2017.

[84] Marcus Hähnel and Till Smejkal. Modular energy modeling using energy/utility. In *ICPE*, 2018.

[85] Mohamed Hassan, Hiren Patel, and Rodolfo Pellizzoni. A framework for scheduling dram memory accesses for multi-core mixed-time critical systems. In *RTAS*, 2015.

[86] Mohamed Hassan, Hiren Patel, and Rodolfo Pellizzoni. PMC: A requirement-aware dram controller for multicore mixed criticality systems. *ACM Trans. Embed. Comput. Syst.*, 16(4):100:1–100:28, May 2017.

[87] Qingqiang He, Nan Guan, Zhe Jiang, and Mingsong Lv. On the degree of parallelism for parallel real-time tasks. *Journal of Systems Architecture*, 2024.

[88] Qingqiang He, Nan Guan, Mingsong Lv, Xu Jiang, and Wanli Chang. Bounding the response time of DAG tasks using long paths. In *RTSS*, 2022.

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

[90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Burlington, MA, 6th edition, 2019.

[91] Shashanka HOLLA, Azzedine Adam TOUZNI, and Srinivas RAMANA. Memory-system resource partitioning and monitoring (mpam) configuration using secure processor, 2025.

[92] Chung-Hsing Hsu, Ulrich Kremer, and Michael S. Hsiao. Compiler-directed dynamic voltage/frequency scheduling for energy reduction in microprocessors. In *ISLPED '01*, 2001.

[93] T. Huffmire and T. Sherwood. Wavelet-based phase classification. In *PACT*, 2006.

[94] Intel. x86: intel cache allocation technology support. http://lwn.net/Articles/622893/. Accessed: 2025-05-23.

[95] Intel. Improving real-time performance by utilizing cache allocation technology, April 2015. White Paper.

[96] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B, November 2020. Order No. 325462-073US, https://software.intel.com/content/dam/develop/external/us/en/documents-tps/325462-sdm-vol-1-2abcd-3abcd.pdf.

[97] Ivan Izhbirdeev, Denis Hoornaert, Weifan Chen, Alexander Zuepke, Youssef Hammad, Marco Caccamo, and Renato Mancuso. Coherence-aided memory bandwidth regulation. In *RTSS24*, 2024.

[98] Pushpak Jagtap, Sadegh Soudjani, and Majid Zamani. Formal synthesis of stochastic systems via control barrier certificates. *IEEE Transactions on Automatic Control*, 2020.

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

[100] Zhihao Jiang, Miroslav Pajic, and Rahul Mangharam. Cyber–physical modeling of implantable cardiac medical devices. 2012.

[101] Hyoseung Kim, Dionisio De Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *RTAS*, 2014.

[102] Hyoseung Kim, Dionisio De Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding and reducing memory interference in cots-based multi-core systems. *Real-Time Systems*, 2016.

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

[104] Namhoon Kim, Micaiah Chisholm, Nathan Otterness, James H. Anderson, and F. Donelson Smith. Allowing shared libraries while supporting hardware isolation in multicore real-time systems. In *RTAS*, 2017.

[105] Namhoon Kim, Bryan C. Ward, Micaiah Chisholm, Cheng-Yang Fu , James H. Anderson , and F. Donelson Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. In *RTAS*, 2016.

[106] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 2003.

[107] Ohchul Kwon, Gero Schwäricke, Tomasz Kloda, Denis Hoornaert, Giovani Gracioli, and Marco Caccamo. Flexible cache partitioning for multi-mode real-time systems. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1156–1161, 2021.

[108] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder. Motivation for variable length intervals and hierarchical phase behavior. In *ISPASS*, 2005.

[109] Edward A Lee. Cyber physical systems: Design challenges. In *ISORC*, 2008.

[110] Edward A Lee. Computing needs time. *Communications of the ACM*, 2009.

[111] Insup Lee, Oleg Sokolsky, Sanjian Chen, John Hatcliff, Eunkyoung Jee, BaekGyu Kim, Andrew King, Margaret Mullen-Fortino, Soojin Park, Alexander Roederer, and Krishna K. Venkatasubramanian. Challenges and research directions in medical cyber–physical systems. 2012.

[112] Patrick Leteinturier, Simon Brewerton, and Klaus Scheibert. Multicore benefits and challenges for automotive applications. In *SAE World Congress and Exhibition*. SAE International, apr 2008.

[113] Bin Li, Li Zhao, Ravi Iyer, Li-Shiuan Peh, Michael Leddige, Michael Espig, Seung Eun Lee, and Donald Newell. Coqos: Coordinating qos-aware shared resources in noc-based socs. *Journal of Parallel and Distributed Computing*, 71(5):700 – 713, 2011.

[114] Haoran Li, Meng Xu, Chong Li, Chenyang Lu, Christopher Gill, Linh Phan, Insup Lee, and Oleg Sokolsky. Multi-mode virtualization for soft real-time systems. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 117–128, 2018.

[115] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, 1995.

[116] Yonghui Li, Benny Akesson, and Kees Goossens. Architecture and analysis of a dynamically-scheduled real-time memory controller. *Real-Time Syst.*, 52(5):675–729, September 2016.

[117] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *HPCA*, 2008.

[118] Linux Kernel Documentation. User interface for resource control feature (resctrl). https://docs.kernel.org/filesystems/resctrl.html, 2025. Accessed 2025-09-22.

[119] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 1973.

[120] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20:46–61, January 1973.

[121] Jane W. S. Liu. *Real-time systems / by Jane W.S. Liu.* Prentice Hall, 2000.

[122] Lei Liu, Yong Li, Zehan Cui, Yungang Bao, Mingyu Chen, and Chengyong Wu. Going vertical in memory management: Handling multiplicity by multi-policy. In *ISCA*, 2014.

[123] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: Efficient deterministic multithreading. In *Proc. SOSP*, 2011.

[124] Tyler M. Lovelly and Alan D. George. Comparative Analysis of Present and Future Space-Grade Processors with Device Metrics. *J. Aerosp. Inf. Syst.*, 14(3):184–197, 2017.

[125] Erika Susana Alcorta Lozano and Andreas Gerstlauer. Learning-based phase-aware multi-core cpu workload forecasting. *ACM Trans. Des. Autom. Electron. Syst.*, 2022.

[126] Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. Real-time cache management framework for multi-core architectures. In *RTAS*, 2013.

[127] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, 46(3):219–254, 2003.

[128] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio Buttazzo. Memory-processor co-scheduling in fixed priority systems. In *RTNS*, New York, NY, USA,

2015. ACM.

[129] Garima Modi, Priyanka Singla, Neetu Jindal, Ayan Mandal, and Preeti Panda. Farre: Fairness aware request response arbitration in shared caches. *ACM Transactions on Embedded Computing Systems*, 2025.

[130] T. K. Moon. The expectation-maximization algorithm. *IEEE Signal Processing Magazine*, 13(6):47–60, 1996.

[131] Frank Mueller. Compiler support for software-based cache partitioning. In *LCTES*, 1995.

[132] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *MICRO*, 2011.

[133] Joseph Musmanno. Data intensive systems (dis) benchmark performance summary. *AFRL Technical Report AFRL-IF-RS-TR-2003-198*, 2003.

[134] Mircea Negrean, Sebastian Klawitter, and Rolf Ernst. Timing analysis of multi-mode applications on autosar conform multi-core systems. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2013.

[135] Vincent Nelis, Joël Goossens, and Björn Andersson. Two protocols for scheduling multi-mode real-time systems upon identical multiprocessor platforms. In *Euromicro Conference on Real-Time Systems*, 2009.

[136] Moritz Neukirchner, Kai Lampka, Sophie Quinton, and Rolf Ernst. Multi-mode monitoring for mixed-criticality real-time systems. In *CODES+ISSS*, 2013.

[137] Lanshun Nie, Chenghao Fan, Shuang Lin, Li Zhang, Yajuan Li, and Jing Li. Holistic resource allocation under federated scheduling for parallel real-time tasks. *ACM Trans. Embed. Comput. Syst.*, 2022.

[138] Jan Nowotsch and Michael Paulitsch. Leveraging multi-core computing architectures in avionics. In *Proceedings of the 2012 Ninth European Dependable Computing Conference*, EDCC '12. IEEE Computer Society, 2012.

[139] Rashidah Olanrewaju, Asifa Baba, Burhan Khan, Mashkuri Yaacob, Amelia Azman, and Mohammad Mir. A study on performance evaluation of conventional cache replacement algorithms: A review. 2016.

[140] Ewan S Page. Continuous inspection schemes. *Biometrika*, 41(1/2):100–115, 1954.

[141] Xing Pan and Frank Mueller. Numa-aware memory coloring for multicore real-time systems. *Journal of Systems Architecture*, 2021.

[142] Jinsu Park, Seongbeom Park, and Woongki Baek. CoPart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers. In *EuroSys*, 2019.

[143] Vyomkumar P Patel, Vijaykumar S Jatti, and Vinaykumar S Jatti. Design of quarter car model for active suspension system and control optimization. In *Optimization Methods for Structural Engineering*, pages 211–225. Springer, 2023.

[144] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for cots-based embedded systems. In *RTAS*, Washington, DC, USA, 2011. IEEE Computer Society.

[145] Ryan S. Peterson and Emin Gün Sirer. Antfarm: Efficient content distribution with managed swarms. In *NSDI*, 2009.

[146] L. T. X. Phan, S. Chakraborty, and I. Lee. Timing analysis of mixed time/event-triggered multi-mode systems. In *Real-Time Systems Symposium (RTSS)*, 2009.

[147] L. T. X. Phan, I. Lee, and O. Sokolsky. Compositional analysis of multi-mode systems. In *Euromicro Conference on Real-Time Systems*, 2010.

[148] L. T. X. Phan, I. Lee, and O. Sokolsky. A semantic framework for multi-mode systems. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011. Available from http://repository.upenn.edu/cgi/viewcontent.cgi?article=1495&context=cis_papers.

[149] Linh TX Phan, Insup Lee, and Oleg Sokolsky. A semantic framework for mode change protocols. In *RTAS*, 2011.

[150] L.T.X. Phan, S. Chakraborty, and P.S. Thiagarajan. A multi-mode real-time calculus. In *Real-Time Systems Symposium (RTSS)*, 2008.

[151] Wesley Powell. High-Performance Spaceflight Computing (HPSC) Project Overview, November 2018.

[152] Stephen Prajna and Ali Jadbabaie. Safety verification of hybrid systems using barrier certificates. In *International Workshop on Hybrid Systems: Computation and Control*, pages 477–492, 2004.

[153] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, 2006.

[154] Ragunathan Rajkumar, Insup Lee, Lui Sha, and John Stankovic. Cyber-physical systems: the next computing revolution. In *DAC*, 2010.

[155] Prapaporn Rattanatamrong and Jose A.B. Fortes. Mode transition for online scheduling of adaptive real-time systems on multiprocessors. In *International Conference on Embedded and Real-Time Computing Systems and Applications*, volume 1, pages 25–32, 2011.

[156] Jorge Real and Alfons Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26:161–197, 2004.

[157] Douglas A Reynolds. Gaussian mixture models. *Encyclopedia of biometrics*, 741, 2009.

[158] M. Roitzsch, S. Wächtler, and H. Härtig. Atlas: Look-ahead scheduling using workload metrics. In *RTAS*, 2013.

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

[160] Debayan Roy, Sumana Ghosh, Qi Zhu, Marco Caccamo, and Samarjit Chakraborty. Goodspread: Criticality-aware static scheduling of cps with multi-qos resources. In *Real-Time Systems Symposium (RTSS)*, pages 178–190. IEEE, 2020.

[161] Real-time deferrable server (rtds) scheduler. https://wiki.xenproject.org/wiki/RTDS-Based-Scheduler, 2015. Accessed: 2025-05-23.

[162] Matthias Rungger and Paulo Tabuada. Computing robust controlled invariant sets of linear systems. *IEEE Transactions on Automatic Control*, 62(7):3665–3670, 2017.

[163] Ahsan Saeed, Dakshina Dasari, Dirk Ziegenbein, Varun Rajasekaran, Falk Rehm, Michael Pressler, Arne Hamann, Daniel Mueller-Gritschneder, Andreas Gerstlauer, and Ulf Schlichtmann. Memory utilization-based dynamic bandwidth regulation for temporal isolation in multi-cores. In *RTAS22*, 2022.

[164] Abusayeed Saifullah, Sezana Fahmida, Venkata P. Modekurthy, Nathan Fisher, and Zhishan Guo. CPU Energy-Aware Parallel Real-Time Scheduling. In *ECRTS*, 2020.

[165] Abusayeed Saifullah, David Ferry, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D. Gill. Parallel real-time scheduling of dags. *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[166] Antonio Savino, Gautam Gala, Marcello Cinque, and Gerhard Fohler. Multicore dram bank-& row-conflict bomb for timing attacks in mixed-criticality systems, 2024.

[167] Lucas Scheuvens, Tom Hößler, André Noll Barreto, and Gerhard P. Fettweis. Wireless control communications co-design via application-adaptive resource management. In *5G World Forum (5GWF)*, pages 298–303, 2019.

[168] Simon Schliecker and Rolf Ernst. Real-time performance analysis of multiprocessor systems with shared memory. *ACM Trans. Embed. Comput. Syst.*, 10(2):22:1–22:27, January 2011.

[169] M.T. Schmitz, B.M. Al-Hashimi, and P. Eles. A co-design methodology for energy-efficient multi-mode embedded systems with consideration of mode execution probabilities. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 960–965, 2003.

[170] Reinhard Schneider, Dip Goswami, Sohaib Zafar, Martin Lukasiewycz, and Samarjit Chakraborty. Constraint-driven synthesis and tool-support for flexray-based automotive control systems. In *IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 139–148, 2011.

[171] Andreas Schranzhofer, Jian-Jia Chen, and Lothar Thiele. Timing analysis for TDMA arbitration in resource sharing systems. In *RTAS*, 2010.

[172] Andreas Schranzhofer, Rodolfo Pellizzoni, Jian-Jia Chen, Lothar Thiele, and Marco Caccamo. Timing analysis for resource access interference on adaptive resource arbiters. In *RTAS*, 2011.

[173] Akbar Sharifi, Shekhar Srikantaiah, Asit K. Mishra, Mahmut Kandemir, and Chita R. Das. Mete: Meeting end-to-end qos in multicores through system-wide resource management. *SIGMETRICS Perform. Eval. Rev.*, 39(1):13–24, June 2011.

[174] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *ASPLOS*, 2004.

[175] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT*, 2001.

[176] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS*, 2002.

[177] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *ISCA*, 2003.

[178] Timothy Sherwood and Brad Calder. Time varying behavior of programs. Technical Report CS99-630, UCSD, August 1999.

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

[180] Y. Shin, D. Kim, and K. Choi. Schedulability-driven performance analysis of multiple mode embedded real-time systems. In *DAC*, 2000.

[181] D. Simon, D. Robert, and O. Sename. Robust control/scheduling co-design: application to robot control. In *Real Time and Embedded Technology and Applications Symposium*, pages 118–127, 2005.

[182] Parul Sohal, Rohan Tabish, Ulrich Drepper, and Renato Mancuso. E-warp: A system-wide framework for memory bandwidth profiling and management. In *RTSS20*, 2020.

[183] Damoon Soudbakhsh, Linh T. X. Phan, Anuradha Annaswamy, and Oleg Sokolsky. Co-design of arbitrated network control systems with overrun strategies. *IEEE Transactions on Control of Network Systems*, 5(1):128–141, 2018.

[184] Damoon Soudbakhsh, Linh T. X. Phan, Anuradha Annaswamy, Oleg Sokolsky, and Insup Lee. Co-

design of control and platform with dropped signals. In *Proceedings of the ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, 2013.

[185] Splash2x benchmark. http://parsec.cs.princeton.edu/parsec3-doc.htm#splash2x, 1995. Accessed: 2025-05-23.

[186] Lavanya Subramanian. *Providing High and Controllable Performance in Multicore Systems Through Shared Resource Management*. PhD thesis, Carnegie Mellon University, 2015.

[187] Marion Sudvarg, Andrew Clark, and Chris Gill. Integrated real-time control and scheduling for safety critical cyber-physical systems. In *RTAS 2025*, 2025.

[188] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *Journal of Supercomputing*, 28(1):7–26, 2004.

[189] Connor Sullivan, Alex Manley, Mohammad Alian, and Heechul Yun. Per-bank bandwidth regulation of shared last-level cache for real-time systems. In *RTSS24*, 2024.

[190] Binqi Sun, Debayan Roy, Tomasz Kloda, Andrea Bastoni, Rodolfo Pellizzoni, and Marco Caccamo. Co-optimizing cache partitioning and multi-core task scheduling: Exploit cache sensitivity or not? In *RTSS*, 2023.

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

[192] Binqi Sun, Zhihang Wei, Andrea Bastoni, Debayan Roy, Mirco Theile, Tomasz Kloda, Rodolfo Pellizzoni, and Marco Caccamo. Multi-objective memory bandwidth regulation and cache partitioning for multicore real-time systems. In *ECRTS*, 2025.

[193] Herb Sutter et al. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

[194] Noriaki Suzuki, Hyoseung Kim, Dionisio de Niz, Bjorn Andersson, Lutz Wrage, Mark Klein, and Ragunathan (Raj) Rajkumar. Coordinated bank and cache coloring for temporal protection of memory accesses. In *ICESS*, 2013.

[195] Andrew J Taylor, Victor D Dorobantu, Ryan K Cosner, Yisong Yue, and Aaron D Ames. Safety of sampled-data systems with control barrier functions via approximate discrete time models. In *Conference on Decision and Control (CDC)*, pages 7127–7134. IEEE, 2022.

[196] Corey Tessler, Prashant Modekurthy, Nathan Fisher, Abusayeed Saifullah, and Alleyn Murphy. Co-located parallel scheduling of threads to optimize cache sharing. In *RTSS*, 2023.

[197] Zelin Tong, Shareef Ahmed, and James H. Anderson. Holistically budgeting processing graphs. In

RTSS 2023, 2023.

[198] Charles Truong, Laurent Oudre, and Nicolas Vayatis. Selective review of offline change point detection methods. *Signal Processing*, 167:107299, 2020.

[199] P. K. Valsan, H. Yun, and F. Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *RTAS*, 2016.

[200] Micaela Verucchi, Ignacio Sañudo Olmedo, and Marko Bertogna. A survey on real-time DAG scheduling, revisiting the global-partitioned infinity war. *Real Time Systems*, 59(3):479–530, 2023.

[201] Renée Vidal, Shawn Schaffert, John Lygeros, and Shankar Sastry. Controlled invariance of discrete time systems. In *International Workshop on Hybrid Systems: Computation and Control*, pages 437–451. Springer, 2000.

[202] Valeria Villani, Fabio Pini, Francesco Leali, and Cristian Secchi. Survey on human–robot collaboration in industrial settings: Safety, intuitive interfaces and applications. *Mechatronics*, 2018.

[203] Xiaodong Wang and Jose F. Martinez. Xchange: A market-based approach to scalable dynamic multi-resource allocation in multicore architectures. In *HPCA*, 2015.

[204] Bryan C Ward, Jonathan L Herman, Christopher J Kenna, and James H Anderson. Making shared caches more predictable on multicore platforms. In *2013 25th Euromicro Conference on Real-Time Systems*. IEEE, 2013.

[205] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 2008.

[206] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. ISCA '95, 1995.

[207] Yecheng Xiang and Hyoseung Kim. Pipelined data-parallel cpu/gpu scheduling for multi-dnn real-time inference. In *RTSS 2019*, 2019.

[208] Jun Xiao, Sebastian Altmeyer, and Andy D. Pimentel. Schedulability analysis of non-preemptive real-time scheduling for multicore processors with shared caches. In *RTSS*, 2017.

[209] Jun Xiao, Yixian Shen, and Andy D. Pimentel. Cache interference-aware task partitioning for non-preemptive real-time multi-core systems. *TECS*, 2022.

[210] Yuejian Xie and Gabriel H. Loh. Dynamic classification of program memory behaviors in CMPs. In *CMP-MSI*, 2008.

[211] Cong Xu, Karthick Rajamani, Alexandre Ferreira, Wesley Felter, Juan Rubio, and Yang Li. dcat: dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service. In *Proceedings of the Thirteenth EuroSys Conference*, page 14. ACM, 2018.

[212] Meng Xu, Robert Gifford, and Linh Thi Xuan Phan. Holistic multi-resource allocation for multicore real-time virtualization. In *DAC*, 2019.

[213] Meng Xu, Linh Thi Xuan Phan, H. Choi, Y. Lin, H. Li, C. Lu, and Insup Lee. Holistic resource allocation for multicore real-time systems. In *RTAS*, 2019.

[214] Meng Xu, Linh Thi Xuan Phan, Hyon-Young Choi, and Insup Lee. vcat: Dynamic cache management using cat virtualization. In *RTAS*, 2017.

[215] Zijin Xu, Yuanhai Zhang, Shuai Zhao, Gang Chen, Haoyu Luo, and Kai Huang. Drl-based task scheduling and shared resource allocation for multi-core real-time systems. In *ICITES*, 2023.

[216] Atsushi Yano and Takuya Azumi. Deadline miss early detection method for mixed timer-driven and event-driven dag tasks. In *ECRTS*, 2024.

[217] Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Heechul Yun, and Marco Caccamo. Global real-time memory-centric scheduling for multicore systems. *IEEE Transactions on Computers*, 65(9):2739–2751, Sept 2016.

[218] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. Coloris: A dynamic cache partitioning system using page coloring. In *PACT*, 2014.

[219] Ying Ye, Richard West, Jingyi Zhang, and Zhuoqun Cheng. Maracas: A real-time multicore vcpu scheduling framework. In *RTSS*, 2016.

[220] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2014.

[221] Heechul Yun, Rodolfo Pellizzoni, and Prathap Kumar Valsan. Parallelism-aware memory interference delay analysis for cots multicore systems. In *ECRTS*, 2015.

[222] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *ECRTS*, 2012.

[223] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *RTAS*, 2013.

[224] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory bandwidth management for efficient performance isolation in multi-core platforms. *IEEE Transactions on*

Computers, 65(2):562–576, Feb 2016.

[225] Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. A survey of autonomous driving: Common practices and emerging technologies. *IEEE access*, 2020.

[226] Luca Zaccarian. Dc motors: dynamic model and control techniques, 2012.

[227] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. EuroSys, 2009.

[228] Shuai Zhao, Xiaotian Dai, and Iain Bate. DAG scheduling and analysis on multi-core systems by modelling parallelism and dependency. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):4019–4038, 2022.

[229] Shuai Zhao, Xiaotian Dai, Iain Bate, Alan Burns, and Wanli Chang. DAG scheduling and analysis on multiprocessor systems: Exploitation of parallelism and dependency. In *RTSS*, 2020.

[230] Shuai Zhao, Xiaotian Dai, Benjamin Lesage, and Iain Bate. Cache-aware allocation of parallel jobs on multi-cores based on learned recency. In *RTNS*, 2023.

[231] Xingyu Zhou, Heran Shen, Zejiang Wang, and Junmin Wang. Individualizable vehicle lane keeping assistance system design: A linear-programming-based model predictive control approach. *IFAC-PapersOnLine*, 55(37):518–523, 2022.

[232] Yanqi Zhou and David Wentzlaff. MITTS: Memory inter-arrival time traffic shaping. In *ISCA*, 2016.

[233] Qi Zhu and Alberto Sangiovanni-Vincentelli. Codesign methodologies and tools for cyber–physical systems. *Proceedings of the IEEE*, 106(9):1484–1500, 2018.

[234] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contentionin multicore processors via scheduling. In *ASPLOS*, 2010.

[235] Alexander Zuepke, Andrea Bastoni, Weifan Chen, Marco Caccamo, and Renato Mancuso. Mempol: Policing core memory bandwidth from outside of the cores. In *RTAS*, 2023.