Multi-mode on Multi-core: Making the best of both worlds with Omni

Robert Gifford Linh Thi Xuan Phan
University of Pennsylvania

***Abstract*—When scheduling multi-mode real-time systems on multi-core platforms, a key question is how to dynamically adjust shared resources, such as cache and memory bandwidth, when resource demands change, without jeopardizing schedulability during mode changes. This paper presents Omni, a first end-to-end solution to this problem. Omni consists of 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. Omni's resource allocation leverages the platform's concurrency and the diversity of the tasks' demands to minimize overload during mode transitions; it does so by intelligently co-distributing tasks and resources across cores. Omni's schedulability test ensures predictable mode transitions, and it takes into account mode-change effects on the resource demands on different cores, so as to best match their dynamic needs using the available resources.**

We have implemented a prototype of Omni, and we have evaluated it using randomly generated multi-mode systems with several real-world benchmarks as the workload. Our results show that Omni has low overhead, and that it is substantially more effective in improving schedulability than the state of the art.

I. 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 multi-core platforms in many critical areas, such as automotive [20] and avionics [22, 32], 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 multi-core 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 [10, 31, 34] 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 multi-core resource allocation [39], 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 multi-core 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 [12], 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 multi-core resource allocation methods, such as the holistic allocation from [39], 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 paper, we present Omni, a first integrated end-to-end solution to the multi-resource co-allocation problem of multi-mode systems on multi-core 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 IV);
- a resource-aware multi-mode schedulability analysis (Section V); and
- a prototype implementation of Omni that supports general mode-change semantics and dynamic resource allocation (Section VI)

We have evaluated Omni using multi-mode systems generated from PARSEC [5], SPLASH2x [37], DIS [24], and Isolbench [38] 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.

II. 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., [2, 7, 8, 16, 23, 28, 29, 30, 36]), and 2) mode change protocols for ensuring schedulability during mode transitions (e.g., see [6, 35] and references therein). Tools for systematic design exploration and evaluation of MCPs have also been developed [31]. Most existing solutions target uniprocessors, however.

Multi-mode scheduling and analysis have recently been extended towards multiprocessors [1, 4, 12, 17, 26, 33]. While the majority still considers CPU only, some recent work has begun to consider shared resources. For instance, Negrean [25] develops an analysis for multi-mode applications on AUTOSAR conform multi-core 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 [13]. Closely related to our work, Kwon [19] 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 multi-core platforms in response to mode changes, as Omni does.

Several multi-resource co-allocation techniques have been developed. For example, CaM [40] and its virtualization extension [39] 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 [15] 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. [27] develops an implementation of multi-mode monitors. Azim et. al. [3] proposes an implementation in LITMUS^{RT} that utilizes checkpoints and rollback-based mode-change mechanisms for efficient mode changes. Chen et. al. [10] 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 [21]. Unlike our Omni prototype, none of these platforms supports dynamic shared resource allocations.

III. System model and goal

Platform. We consider a multi-core 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[1]. 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 $\mathcal{M}$, an initial mode $\mathsf{m}_0 \in \mathcal{M}$, a set of mode transitions $\mathcal{R} \subseteq \mathcal{M} \times \mathcal{M}$, and a set of deadline-constrained periodic tasks $\mathcal{T}$ that need to execute in the modes. Each mode m has a set of tasks $\mathcal{T}^m \subseteq \mathcal{T}$ 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.

[1] 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 [40], 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 $\tau \in \mathcal{T}^m$, 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:[2]

- If a task is assigned to a different core in the new mode, its existing job will be 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 will be (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, $\Pi_\tau^m \in [1,K]$ represents the core on which τ is mapped to in mode m, and C_k^m and B_k^m 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 $C_{\max}$ ($B_{\max}$).

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 II) to our setting as baseline solutions. The first extends the multi-modal task partitioning technique in [12], 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) [11] schedulability test used in [12] with our EDF-based schedulability test in Section V.) The second baseline uses the static holistic resource allocation algorithm from [40] 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

[2] Systematic design exploration of novel resource-aware multi-core mode change protocols, e.g., a resource-aware extension of SafeMC [10] is an interesting future direction.

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.

IV. 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.

A. 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 1.

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 V. 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 2) aims to bring the system out of an unschedulable state by moving or swapping tasks between cores.[3] 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

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

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.

B. 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 1 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.[4] 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).

Algorithm 1 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 $i = |\text{tierList}| \ldots 1$ 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 $i = 0 \ldots r$ 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($M_{cur}.cores$)
```

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 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.

[4] 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 $M_{prevs} \cup \{M_{cur}\}$.

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.

Task redistribution. Algorithm 2 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.)

Algorithm 2 Omni Task Redistribution

```
 1: function TaskRedistribute($M_{prevs}$, cores, nmTask)                      ▷
            $M_{prevs}$: Previous modes that transition 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 $Util[$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 $Util[$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
```

Complexity. As Algorithm 1 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 1 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 $O(r \cdot n^2)$.

V. 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 III 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 $m \in \mathcal{M}$ and during each mode transition $r \in \mathcal{R}$ under this allocation. Before establishing the conditions for each case, let us define some notation. We denote by $\mathcal{T}_k^m$ the set of tasks that are mapped onto core k in mode m, i.e., $\mathcal{T}_k^m = \{\tau \in \mathcal{T}^m \mid \Pi_\tau^m = k\}$. In addition, D_k^m denotes the maximum deadline of all tasks on core k in mode m, i.e., $D_k^m = \max_{\tau \in \mathcal{T}_k^m} \{d_\tau^m\}$. 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

$$\forall t > 0 : \mathsf{dbf}_\tau^m(t) = \lfloor \frac{t - d_\tau^m + p_\tau^m}{p_\tau^m} \rfloor E_\tau^m. \tag{1}$$

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

$$\mathsf{dbf}_k^m(t) = \sum_{\tau \in \mathcal{T}_k^m} \mathsf{dbf}_\tau^m(t). \tag{2}$$

The next theorem states the schedulability condition for mode m. Its proof follows directly from the analysis in [14].

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

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

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 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., $\tau \in \mathcal{T}_k^m$). The demand generated by τ during the interval I_t depends on its type:

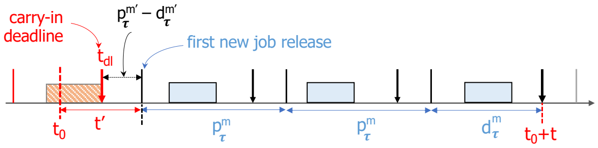


Fig. 1: Worst-case demand scenario of an existing task.

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

Case 2) If τ is an existing task (i.e., $\tau \in \mathcal{T}^{m'}$), 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.[5]

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 2. An example that illustrates this scenario is shown in Figure 1.

Lemma 2. *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 1, the maximum demand of new jobs of τ in I_t is bounded by

$$\mathsf{dbf}_\tau^m(t) = \lfloor \frac{t - d_\tau^m + p_\tau^m}{p_\tau^m} \rfloor E_\tau^m \tag{3}$$

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 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

[5] 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.

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 $t \geq d^m_\tau$); otherwise, t' is the same as t. In other words, $t' = (t - d^m_\tau) \mod p^m_\tau$ if $t \geq d^m_\tau$, 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 $E^m_\tau - E^{m'}_\tau$ 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 $t \geq d^m_\tau$, then $t' = (t - d^m_\tau) \mod p^m_\tau$, and

$$E^{(m',m)}_\tau = \min\{E^m_\tau, \max\{0, t' - (p^{m'}_\tau - d^{m'}_\tau)\} + \max\{0, E^m_\tau - E^{m'}_\tau\}\}.$$

Case 2) If $t < d^m_\tau$, then

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

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

$$\mathsf{dbf}^{(m',m)}_\tau(t) = \lfloor \frac{t - d^m_\tau + p^m_\tau}{p^m_\tau} \rfloor E^m_\tau + E^{(m',m)}_\tau \quad (4)$$

By combining the demands of existing tasks and new tasks, we derive Lemma 3.

Lemma 3. *The maximum demand of a core k during a mode transition $m' \to m$ is bounded by*

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

for all $t > 0$, where $\mathsf{dbf}^m_\tau(t)$ and $\mathsf{dbf}^{(m',m)}_\tau(t)$ are defined in Eqs. (1) *and* (4)*, respectively.*

The above lemma gives a means to check for schedulability during a mode transition. However, the DBF function $\mathsf{dbf}^{(m',m)}_k(t)$ 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 $S^{(m',m)}_{i,k}$ denote the set of existing tasks $\tau \in \mathcal{T}^m_k$ such that τ was assigned to core i in mode m' (i.e., $\Pi^{m'}_\tau = i$), for all $1 \leq i \leq K$.

Then, the total execution demand of all the carry-in jobs in $S^{(m',m)}_{i,k}$ 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 $\max\{0, E^m_\tau - E^{m'}_\tau\}$. Hence, the total demand of all carry-in jobs in $S^{(m',m)}_{i,k}$ in I_t is bounded by

$$\min\{t, \max_{\tau \in S^{(m',m)}_{i,k}} d^{m'}_\tau\} + \sum_{\tau \in S^{(m',m)}_{i,k}} \max\{0, E^m_\tau - E^{m'}_\tau\}).$$

By combining the above with the DBF of new jobs of a task in $S^{(m',m)}_{i,k}$, given by Eq. (3), we imply that the total demand of all tasks in $S^{(m',m)}_{i,k}$ in the interval I_t is bounded by

$$\mathsf{dbf}^{(m',m)}_{k,i}(t) = \min\{t, \max_{\tau \in S^{(m',m)}_{i,k}} d^{m'}_\tau\} + \sum_{\tau \in S^{(m',m)}_{i,k}} \max\{0, E^m_\tau - E^{m'}_\tau\} + \sum_{\tau \in S^{(m',m)}_{i,k}} \lfloor \frac{t - d^m_\tau + p^m_\tau}{p^m_\tau} \rfloor E^m_\tau.$$

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 $\sum_{i=1}^K \mathsf{dbf}^{(m',m)}_{k,i}(t)$. As a result, we can bound the demand of a core using the next lemma.

Lemma 4. *The maximum demand of a core k during a mode transition from m' to m is bounded by*

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

for all $t > 0$.

$$\mathsf{dbf}^{(m',m)}_{k,i}(t) = \min\{t, \max_{\tau \in S^{(m',m)}_{i,k}} d^{m'}_\tau\} + \sum_{\tau \in S^{(m',m)}_{i,k}} \max\{0, E^m_\tau - E^{m'}_\tau\} + \sum_{\tau \in S^{(m',m)}_{i,k}} \lfloor \frac{t - d^m_\tau + p^m_\tau}{p^m_\tau} \rfloor E^m_\tau.$$

The following theorem states the schedulability condition for a mode transition from m' to m. Its proof comes directly from Lemmas 3 and 4.

Theorem 5. *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 1, and*
(2) *for all $1 \leq k \leq K$: $\min\{\overline{\mathsf{dbf}}^{(m',m)}_k(t), \mathsf{dbf}^{(m',m)}_k(t)\} \leq t$ 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.

VI. Prototype

To evaluate Omni and to show its utility in practice, we built a prototype of Omni on top of LITMUS^{RT} [9]. 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) [18] and Memguard [41] 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.

VII. Evaluation

To evaluate the effectiveness and applicability of Omni, we conducted an extensive set of experiments, both numerically and experimentally on real multi-core 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?

A. 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 III. MM-Static extends the task-to-core partitioning method for multi-mode systems from [12] 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 [12] 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 multi-core resource allocation technique from [40] 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 multi-core multi-mode system scheduling and holistic multi-core 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 [41] 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 multi-core 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 [39], 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: [$\frac{8}{9}$, $\frac{1}{9}$], [$\frac{6}{9}$,

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

TABLE I: Scheduling and mode-change overheads (in ns).

$\frac{3}{9}$] 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.

B. 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 I, 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.

C. 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 IV); 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.[6]

Results. Figure 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:

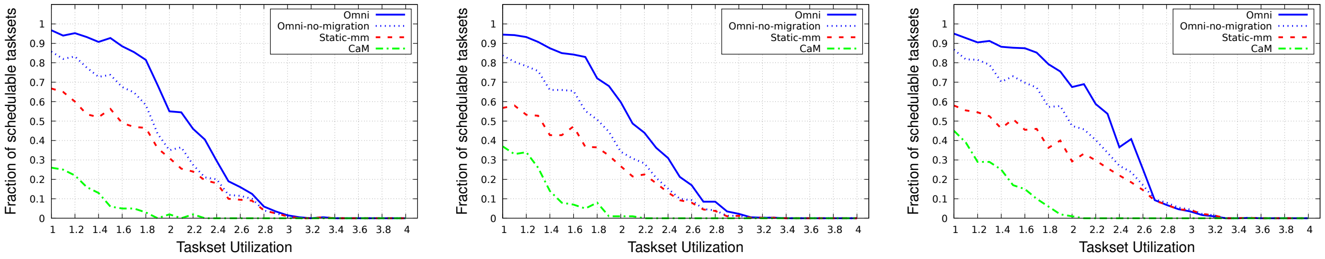
- CaM performs extremely poorly across all three utilization distributions. At reference utilization 1.0, it fails to schedule the majority of the tasksets. For example, only 26% of the tasksets are schedulable under light distribution and 44% of the tasksets are schedulable under heavy utilization distribution. 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.

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 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 → Mode 1 → Mode 2 → Mode 3 → 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

[6] 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.



(a) Bimodal Light

(b) Bimodal Medium

(c) Bimodal Heavy

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

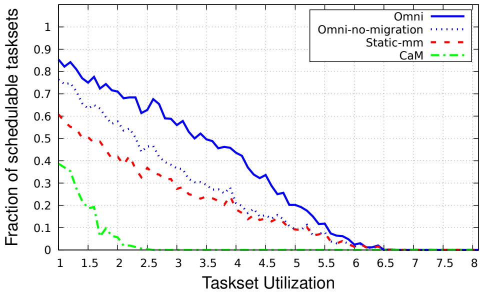


Fig. 3: Schedulabilty of 8-core systems (2 modes).

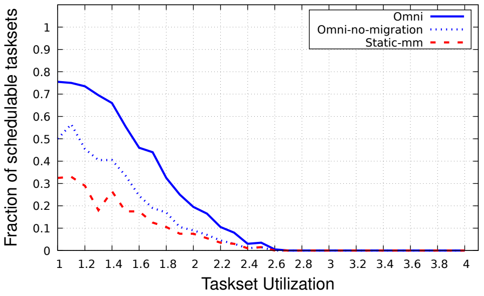


Fig. 4: Schedulabilty of 4-mode systems (2 cores).

since it significantly under-performed in our schedulability evaluation with just 2 modes.

The results are shown in Figure 4. We observe a drop in schedulability across all three algorithms compared to that of the two-mode systems (c.f. Figure 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.

D. *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× more jobs missing their deadlines, and 1.53× 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.

VIII. Conclusion

In this paper, 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.

Acknowledgments

This work was supported in part by NSF grants CNS-1703936, CNS-1750158, CNS-1955670, CNS-2111688, and by ONR N00014-20-1-2744.

References

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

[2] 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.

[3] A. Azim and S. Fischmeister. Efficient mode changes in multi-mode systems. In *ICCD*, 2016.

[4] H. Baek, K. G. Shin, and J. Lee. Response-time analysis for multi-mode tasks in real-time multiprocessor systems. *IEEE Access*, 8:86111–86129, 2020.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, 2008.

[6] A. Burns. System mode changes-general and criticality-based. In *WMC*, 2014.

[7] G. C. Buttazzo, G. Lipari, and L. Abeni. Elastic task model for adaptive rate control. In *RTSS*, 1999.

[8] 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.

[9] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. LITMUS^{RT}: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *RTSS*, 2006.

[10] T. Chen and L. T. X. Phan. SafeMC: A system for the design and evaluation of mode-change protocols. In *RTAS*, 2018.

[11] D. de Niz, K. Lakshmanan, and R. Rajkumar. On the Scheduling of Mixed-Criticality Real-Time Task Sets. In *RTSS*, 2009.

[12] D. de Niz and L. T. X. Phan. Partitioned Scheduling of Multi-Modal Mixed-Criticality Real-Time Systems on Multiprocessor Platforms. In *RTAS*, 2014.

[13] P. Dziurzanski, A. Singh, and L. Indrusiak. Multi-criteria resource allocation in modal hard real-time systems. In *J Embedded Systems*, 2017.

[14] A. Easwaran. Demand-based scheduling of mixed-criticality sporadic tasks on one processor. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 78–87, 2013.

[15] R. Gifford, N. Gandhi, L. T. X. Phan, and A. Haeberlen. Dna: Dynamic resource allocation for soft real-time multicore systems. In *RTAS*, 2021.

[16] S. Goddard and X. Liu. A variable rate execution model. In *ECRTS*, pages 135–143, 2004.

[17] J. Goossens and P. Richard. Partitioned scheduling of multimode multiprocessor real-time systems with temporal isolation. 2013.

[18] Intel. Improving real-time performance by utilizing cache allocation technology, Apr. 2015. White Paper.

[19] O. Kwon, G. Schwäricke, T. Kloda, D. Hoornaert, G. Gracioli, and M. Caccamo. Flexible cache partitioning for multi-mode real-time systems. In *DATE*, pages 1156–1161, 2021.

[20] P. Leteinturier, S. Brewerton, and K. Scheibert. Multicore benefits and challenges for automotive applications. In *SAE World Congress and Exhibition*. SAE International, apr 2008.

[21] H. Li, M. Xu, C. Li, C. Lu, C. Gill, L. Phan, I. Lee, and O. Sokolsky. Multi-mode virtualization for soft real-time systems. In *RTAS*, pages 117–128, 2018.

[22] T. M. Lovelly and A. D. George. Comparative Analysis of Present and Future Space-Grade Processors with Device Metrics. *J. Aerosp. Inf. Syst.*, 14(3):184–197, 2017.

[23] 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.

[24] J. Musmanno. Data intensive systems (dis) benchmark performance summary. page 144, 08 2003.

[25] M. Negrean, S. Klawitter, and R. Ernst. Timing analysis of multi-mode applications on autosar conform multicore systems. In *DATE*, 2013.

[26] V. Nelis, J. Goossens, and B. Andersson. Two protocols for scheduling multi-mode real-time systems upon identical multiprocessor platforms. In *ECRTS*, 2009.

[27] M. Neukirchner, K. Lampka, S. Quinton, and R. Ernst. Multi-mode monitoring for mixed-criticality real-time systems. In *CODES+ISSS*, 2013.

[28] L. Phan, S. Chakraborty, and P. Thiagarajan. A multi-mode real-time calculus. In *RTSS*, 2008.

[29] L. T. X. Phan, S. Chakraborty, and I. Lee. Timing analysis of mixed time/event-triggered multi-mode systems. In *RTSS*, 2009.

[30] L. T. X. Phan, I. Lee, and O. Sokolsky. Compositional analysis of multi-mode systems. In *ECRTS*, 2010.

[31] L. T. X. Phan, I. Lee, and O. Sokolsky. A semantic framework for multi-mode systems. In *RTAS*, 2011. Available from http://repository.upenn.edu/cgi/viewcontent.cgi?article=1495&context=cis_papers.

[32] W. Powell. High-Performance Spaceflight Computing (HPSC) Project Overview, Nov. 2018.

[33] P. Rattanatamrong and J. A. Fortes. Mode transition for online scheduling of adaptive real-time systems on multiprocessors. In *RTCSA*, volume 1, pages 25–32, 2011.

[34] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26:161–197, 2004.

[35] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26(2):161–197, 2004.

[36] Y. Shin, D. Kim, and K. Choi. Schedulability-driven performance analysis of multiple mode embedded real-time systems. In *DAC*, 2000.

[37] Splash2x benchmark. http://parsec.cs.princeton.edu/parsec3-doc.htm#splash2x.

[38] P. K. Valsan, H. Yun, and F. Farshchi. Taming non-

blocking caches to improve isolation in multicore real-time systems. In *RTAS*, pages 1–12, 2016.

[39] M. Xu, R. Gifford, and L. T. X. Phan. Holistic multi-resource allocation for multicore real-time virtualization. In *DAC*, page 168, 2019.

[40] M. Xu, L. T. X. Phan, H. Choi, Y. Lin, H. Li, C. Lu, and I. Lee. Holistic resource allocation for multicore real-time systems. In *RTAS*, 2019.

[41] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory bandwidth management for efficient performance isolation in multi-core platforms. *IEEE Transactions on Computers*, 65(2):562–576, Feb 2016.