Flexible Cache Partitioning for Multi-Mode Real-Time Systems

Ohchul Kwon*, Gero Schwäricke*, Tomasz Kloda*, Denis Hoornaert*, Giovani Gracioli† and Marco Caccamo*
*Technical University of Munich †Federal University of Santa Catarina
{ohchul.kwon, gero.schwaericke, tomasz.kloda, denis.hoornaert, mcaccamo}@tum.de, giovani@lisha.ufsc.br

***Abstract*—Cache partitioning is a well-studied technique that mitigates the inter-processor cache interference in multiprocessor systems. The resulting optimization problem involves allocating portions of the cache to individual processors. In multi-mode applications (e.g., flight control system that runs in take-off, cruise, or landing mode), the cache memory requirement can change over time, making runtime cache repartitioning necessary.**

This paper presents a cache partition allocation framework enabling flexible cache partitioning for multi-mode real-time systems. The main objective is to guarantee timing predictability in the steady states and during mode changes. We evaluate the effectiveness of our approach for multiple embedded benchmarks with different ranges of cache size sensitivity. The results show increased schedulability compared to static partitioning approaches.

***Index Terms*—Cache Partitioning, Mode Change, Real-time**

I. Introduction

In multiprocessor systems where cache memory is shared among the processors (e.g., last-level cache), tasks running concurrently on different processors may interfere with each other and evict each other's data (i.e., cross-processor eviction), leading to power and timing degradation. Cache partitioning is commonly employed to mitigate inter-processor interference by assigning an exclusive cache partition to each processor or task. The cache partitioning can be either implemented in hardware [10], using a cache lockdown feature available in many multiprocessor platforms (e.g., Intel's Cache Allocation Technology, ARM's Lockdown by Master or SiFive's Way Masking) or in software using page coloring [10].

A wide range of mathematical frameworks leverage cache partitioning to optimize the execution time or energy efficiency of tasks. Most assume a static cache partitioning where the partitions cannot be reassigned during the execution [4], [13], [15], [18], [26]. However, applications that exhibit a multi-mode behavior might change their cache memory requirements when the system state and the environment evolve. For such situations, a static partitioning approach is inefficient, and a cache repartitioning is required.

An example of a multi-mode application, where memory requirements change at runtime according to the mode, is depicted in Fig. 1. It consists of an unmanned aerial vehicle (UAV), featuring a low power System On Chip, that collects and analyses camera footage of the ground for agricultural purposes (e.g., [7]). Take-off (M_1) and landing (M_5) require a specialized control task that integrates an image processing algorithm to accurately track the runway. Simultaneously, an object detection algorithm runs in parallel to identify hazardous objects on the runway. This requires an approximately equal

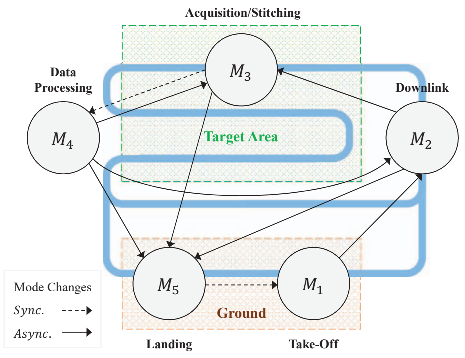


Fig. 1: Directed graph representing the mode changes of a UAV for agricultural monitoring.

split of the system's shared cache memory. While in flight ($M_2 - M_4$), the control algorithm is replaced by a way-point tracking with low cache memory demand, and different, more demanding applications are run in parallel. Therefore, the cache memory can be split unevenly according to the demand, also incurring in runtime cache memory repartitioning.

Unfortunately, runtime cache repartitioning can introduce prohibitive overheads. For instance, cache coloring introduces high repartitioning costs [20] due to excessive data copying. This overhead can be reduced for applications with low memory usage (please refer to [14] for a detailed information on overheads). On Multiprocessor Systems on Chip with an integrated block of programmable logic, zero-copy re-coloring mechanisms can be realized [23]. When hardware support for repartitioning is available, the cache controller can be reconfigured directly to reduce the repartitioning overhead [6].

It is worth noticing that the optimization problem for multi-mode systems cannot be reduced to the well-studied single-mode scenario. As in other multi-mode systems, meeting the requirements in every mode does not automatically imply that the requirements are met during the transient phase between modes [24]. In this paper, we make the following contributions: i) extend the multiprocessor mode change protocols [19] to handle cache repartitioning, ii) propose a cache partition allocation algorithm for multi-mode real-time multiprocessor systems, iii) evaluate the effectiveness of our approach with generic benchmarks based on the memory profiles obtained from modern embedded applications. The experimental results show that for varying target task set utilization, cache size, and size of mode change scenario, the proposed approach consistently achieves a higher allocation success ratio compared to existing static partitioning approaches.

II. Related Work

Many different mode-change protocols have been proposed in the literature, from simple uniprocessor synchronous [25] and asynchronous protocols, both for *Fixed Priority* [22], [24] and *Earliest Deadline First* [5], [16], to multiprocessor protocols [19]. These studies focus solely on schedulability analysis under mode change and do not consider resource allocation. Our work extends the minimum single offset mode change protocol (without periodicity) [19] to handle the shared cache memory allocation in partitioned multiprocessor scheduling.

The allocation of cache segments to different real-time tasks on a single processor in early works [15] aimed to alleviate the cache-related preemption cost. Bui et al. [4] showed that the cache partitioning problem on multiprocessor systems is NP-hard and proposed a genetic algorithm. A holistic approach [26] for cache and memory bandwidth partitioning in multiprocessor systems was adapted to minimize the usage of these two resources while ensuring timing guarantees. However, these studies do not consider multi-mode systems and the challenges introduced by the transition between modes. In [11] global scheduling policies are used for dynamic cache allocation to schedule tasks when the cache partition with the minimal necessary size for the task execution is available. While in this work the cache partitioning is assumed to be known, the main focus of our work is on solving the cache partition allocation problem under mode change. Predictability, alongside energy and thermal constraints management, can also be achieved through a holistic observe-decide-adapt loop as showed in [18].

To the best of our knowledge, the closest research to this work is in [12]. The authors propose an adaptive scratchpad memory allocation mechanism where the scratchpad's exact partitioning changes from one execution phase to another. However, the proposed mechanism aims solely at mitigating the power consumption, thus, the timing predictability and schedulability are overlooked.

III. System Model and Assumptions

A. Platform Model

We assume a multi-processor platform that consists of $|P|$ processors and a shared cache of size S. The cache can be divided into non-overlapping partitions that are a multiple of q memory blocks (there are $B = \frac{S}{q}$ allocatable cache blocks) and can be assigned to a specific processor $p \in P$ for its exclusive use. We assume that the cache partition assignment can be changed at runtime when the system changes the mode. The size of a cache partition for processor p in mode m is denoted as s_m^p. Table I summarizes the notation used in this work.

B. Modes, Tasks, and Mode Changes

A mode m is one of the $|M|$ steady states of the system and the environment that requires a specific task set per processor to be executed. Per mode m and processor p, a finite set $\mathcal{T}_m^p$ of real-time tasks is executed by partitioned scheduling. Each task set $\mathcal{T}_m^p$ consists of tasks with a given minimum inter-arrival time (period) T and relative deadline D, with $D \leq T$. The Worst-Case Execution Time (WCET) $c(s)$ of a task is given as a non-increasing function with respect to the cache partition size.

TABLE I: Notation used in this work.

Notation	Description
P	Set of processors.
S	Total size of partitionable cache.
R_m	Size of the unallocated cache partition in mode m.
q	Allocation granularity or minimal cache partition size.
B	Number of allocatable cache blocks ($B = \frac{S}{q}$).
M	Set of modes.
s_m^p	Size of a cache partition for processor p in mode m.
$\Delta_{m\to n}^p$	Duration between a mode change request (MCR) in mode m and the beginning of mode n on processor p.
$\delta_m^p(s)$	The time it takes for processor p in mode m to release its cache partition of size s after a MCR.
$\mathcal{T}_m^p$	Task set running on processor p in mode m consisting of one or more tasks.
ω_m^p	Initial jitter (mode specific delay) on processor p when changing to mode m.
$\Gamma_m^p(\mathcal{T}, \omega, s)$	Checks the schedulability of task set $\mathcal{T}$ with initial jitter ω and cache memory size s on processor p in mode m
$\mathcal{L}_m^p(\mathcal{T}, \omega)$	Computes the minimum cache partition size for which the task set $\mathcal{T}_m^p$ with initial jitter ω is schedulable.

The system can change between modes in a transient phase called mode change, started by a mode change request (MCR) event, that is synchronously received by all processors. A mode change is called *synchronous* if it does not release tasks of the next mode until all processors complete all tasks of the last mode, and *asynchronous* otherwise [22]. MCRs can only be issued once a previous mode change has been completed.

Mode changes are described by a directed graph [9]. Each node represents a mode, and each edge represents a possible mode change, as shown in Fig. 1. In a mode change graph, an edge $m \to n$ is parameterized by the mode change duration $\Delta_{m\to n}^p$, which describes the duration for processor p between the reception of an MCR in mode m and the beginning of the next mode n and its task set $\mathcal{T}_n^p$ [21]. Parallel edges are not permitted.

The cache partition release time $\delta_m^p(s)$ describes how long it takes for processor p on a mode change $m \to n$ to finish or terminate the active tasks of $\mathcal{T}_m^p$ (depending on the scheduling algorithm used) and release its cache partition of size s. Since the release time can be dependent on the tasks' WCETs, it is required to be a non-increasing function with respect to the cache partition size s. For each processor, the cache partition has to be released before the beginning of the next mode, independently of the cache partition size, i.e., $s \geq q$, thus $\forall m \to n, \forall p \in P : \delta_m^p(q) \leq \Delta_{m\to n}^p$.

When allocating cache partitions, the allocation algorithm is allowed to delay the allocation of a cache partition by an initial jitter ω_n^p. This initial jitter is added if and only if the required amount of memory exceeds the amount currently available (e.g., instant t_4 in Fig. 2). In that case, the task set is still released at $\Delta_{m\to n}^p$, albeit it is blocked until the cache partition is allocated at $\Delta_{m\to n}^p + \omega_n^p$. The allocation algorithm has to ensure the feasibility of the schedule, given this initial jitter. Fig. 2 shows a sample allocation during a mode change that demonstrates the use of the notations s_m^p, ω_n^p, $\delta_m^p(s_m^p)$, and $\Delta_{m\to n}^p$. At runtime, the precomputed cache partitions s_m^p and initial jitter ω_m^p are used in a time-triggered repartitioning, where in mode m, on an MCR to mode n, the cache partition s_n^p is allocated at $\Delta_{m\to n}^p + \omega_n^p$.

C. Dynamic Partitioned Scheduling

A task set is said to be schedulable under a given scheduling policy if all jobs of every task executed in the order given by this policy always meet their deadlines. Our framework accepts a wide range of scheduling policies fulfilling the following conditions: i) *sustainability* with respect to the WCET and initial jitter (i.e., a task set, deemed schedulable with given WCETs and initial jitter, remains schedulable when the WCETs or the initial jitter are reduced), and ii) accounting for or eliminating WCET deterioration caused by cache evictions due to task preemptions and context switching overhead. An example of a scheduling policy fulfilling both requirements is non-preemptive fixed-priority for synchronous task sets [8].

A predicate function $\Gamma_m^p(\mathcal{T}, \omega, s)$ is used to determine whether a task set $\mathcal{T}$ experiencing an initial jitter of ω and running on a cache partition of size s is schedulable (the function returns 1 if schedulable and 0 otherwise). Based on $\Gamma_m^p(\mathcal{T}, \omega, s)$, we define the function $\mathcal{L}_m^p(\mathcal{T}, \omega)$, which computes the minimum cache partition size for which the task set $\mathcal{T}$ with initial jitter ω is schedulable. This information can be obtained using a sensitivity analysis [2]. The function is non-increasing in ω, i.e., increasing the initial jitter can negatively affect the system schedulability, thus requiring more cache memory to compensate and meet the deadlines.

D. Formal Definition of the Allocation Problem

Given the constraints of the system model, we can formulate the allocation problem as a search problem. The problem is additionally constrained on mode changes by the previous mode's unallocated cache partition of size $R_m = S - \sum_{p\in\mathcal{P}} s_m^p$ as well as the changing available cache due to cache partition release and allocation events:

$$\begin{aligned}
\text{find}: \; & s_m^p,\, \omega_m^p, && \forall p \in P,\, \forall m \in M \\
\text{s.t.}: \; & \Gamma_m^p(T_m^p, \omega_m^p, s_m^p) = 1, && \forall p \in P,\, \forall m \in M \\
R_m + & \sum_{p\in P} H(t - \delta_m^p(s_m^p)) \cdot s_m^p \geq \sum_{p\in P} H(t - \Delta_{m\to n}^p - \omega_n^p) \cdot s_n^p, \\
& \forall m \to n,\, \forall t \in \mathbb{R}^+,\, 0 \leq t \leq \max_{p\in P}(\Delta_{m\to n}^p + \omega_n^p)
\end{aligned}$$

where $H(x)$ is the Heaviside step function, whose value is 0 for $x < 0$ and 1 for $x > 0$.

An optimal exhaustive search algorithm results in an exponential time complexity of $\mathcal{O}(C\,^\wedge|M|)$, where C is a P-combination with repetitions for the number of allocatable cache blocks B and $|M|$ is the number of modes. The complexity depends on the number of allocatable cache blocks B but not on the assignments of initial jitter ω_m^p, since, given a partition allocation s_m^p, the initial jitter can be found using the schedulability test Γ_m^p.

IV. Allocation Algorithm

We propose an offline heuristic to find a solution to the allocation problem. The algorithm is iterative and traverses the edges of the mode change graph in reverse post-order until it converges on a stable set of allocations or an allocation fails.

The algorithm starts by allocating the minimum required cache partition size $s_1^p = \mathcal{L}(\mathcal{T}_1^p, 0)$ per processor p for the initial mode $m = 1$. Since all cache blocks are immediately available, an initial jitter is not required. If the sum of the allocated cache partitions exceeds the total size of the partitionable cache memory S, then there is no feasible solution and the algorithm terminates. The same allocation strategy is used for all synchronous mode changes $m \to n$ to allocate the cache partitions s_n^p, because all partitions s_m^p are released before the synchronous allocation for mode n.

The allocation for the remaining asynchronous mode changes is performed according to the steps shown in Alg. 1. For each mode change $m \to n$, the algorithm allocates a cache partition s_n^p and computes an initial jitter ω_n^p per processor p for the next mode n. For that, it uses an event queue of (cache partition) releases $\delta_m^p(s_m^p)$ from the last mode m and (cache partition) requests $\Delta_{m\to n}^p + \omega_n^p$ for the next mode n (line 6 in Alg. 1). These events correspond to the time points t_1 to t_6 in the example in Fig. 2. Then, the algorithm iterates over the event queue while handling the following conditions:

- If the event is a release (e.g., t_1, t_3, t_5 in Fig. 2), then its partition size s_m^p is added to the available cache size R (line 9).
- If instead the event is a request (e.g., t_2) and a cache partition of size s, determined by the allocation function $\mathcal{L}(\mathcal{T}_n^p, \omega_n^p)$ (line 11), is allocatable, then the cache partition s_n^p is assigned, and the available cache size R is reduced (line 13). To cope with several mode changes with the same target mode n, the maximum of the previous allocation size s_n^p and the current size s is used (line 11). Since all scheduling methods are required to be sustainable, an increase in s_n^p cannot invalidate the

Algorithm 1: Proposed allocation heuristic.

```
1  $\forall p \in P, \forall m \in M : last(s_m^p, \omega_m^p) = (-1, -1), (s_m^p, \omega_m^p) = (0, 0)$
2  while $\forall p \forall m : last(s_m^p, \omega_m^p) \neq (s_m^p, \omega_m^p)$ do
3      $\forall p \forall m : last(s_m^p, \omega_m^p) := (s_m^p, \omega_m^p),\ \omega_m^p := 0$
4      for $\forall m \to n$ do
5          $R := R_m$
6          $events := ordered\_inc(\forall p : \delta_m^p(s_m^p) \cup (\Delta_{m\to n}^p + \omega_n^p))$
7          for $e = next(events)$ do
8              if $e\ is\ \delta_m^p(s_m^p)$ then
9                  $R := R + s_m^p$
10             else if $e\ is\ (\Delta_{m\to n}^p + \omega_n^p)$ then
11                 $s = max(s_n^p, \mathcal{L}_n^p(\mathcal{T}_n^p, \omega_n^p))$
12                 if $s \leq R$ then
13                     $s_n^p := s,\ R := R - s_n^p$
14                 else if $\exists\, \delta_m^x(s_m^x) \in events,\ \delta_m^x(s_m^x) > (\Delta_{m\to n}^p + \omega_n^p)$ then
15                     $\omega_n^p := \delta_m^x(s_m^x) - \Delta_{m\to n}^p$
16                     move $(\Delta_{m\to n}^p + \omega_n^p)$ after $\delta_m^x(s_m^x)$ in $events$
17                 else if $R_m > 0 \wedge \exists\, \delta_m^l(s_m^l)\ in\ visited(events)$ then
18                     $s_m^l := s_m^l + q,\ \omega_n^p := 0$
19                     go to 4
20                 else
21                     Allocation failed
```

schedules already deemed feasible for other mode changes. The potentially reduced release time $\delta_n^p(s_n^p)$ can change the event order for other mode changes, but it cannot invalidate already existing allocations (see problem definition in Sec. III-D, the Heaviside step function is non-decreasing).

- If the requested size s is not available (e.g., t_4), then the request is delayed until right after the next release event $\delta_m^x(s_m^x)$, and the request's release jitter ω_n^p is increased by the delay duration (line 16). Increasing ω_n^p can invalidate the schedules for $\mathcal{T}_n^p$, that were already deemed feasible on other mode changes. However, the schedulability of $\mathcal{T}_n^p$ with ω_n^p is tested again, and an allocation, delayed by an increase in ω_n^p, cannot invalidate the allocations for the other processors (analog to reduced $\delta_n^p(s_n^p)$, due to non-decreasing Heaviside step function).
- If there are no further releases in the event queue, but the last mode m has unallocated cache blocks, then the algorithm increases the cache partition size of the last released cache partition s_m^l for the last mode m by the minimal amount q and retries the allocation for this mode change from the beginning (line 19). The increase of s_m^l reduces the cache partition release time $\delta_m^l(s_m^l)$, which subsequently reduces the release jitter ω_n^p caused by $\delta_m^l(s_m^l)$ in line 16. The smaller jitter reduces the cache partition demand from $\mathcal{L}(\mathcal{T}_n^p, \omega)$ in line 11.
- If there are no more cache partition releases and the last mode has no remaining cache blocks, then the allocation fails.

A solution is found if, for all modes m, neither s_m^p nor ω_m^p change in two consecutive iterations (`while` in line 2) of the algorithm. The allocation is bound to converge or fail because cache partitions can only be increased in size (due to the maximum function in line 11), and the total size of the partitionable cache is limited.

Our heuristic has a polynomial worst-case time complexity of $\mathcal{O}(|P|^3 \, log(P) \cdot B^3 \, log(B) \cdot |M| \cdot E)$, where $|P|$ is the number of processors, B is the number of allocatable cache blocks, $|M|$ is the number of modes, and E is the number of mode changes. The time complexities of the schedulability tests $\Gamma_m^p(\mathcal{T}, \omega, s)$ are not included, because they are application dependent.

V. Evaluation

In this section, we discuss how task requirements and parameters are generated (Sec. V-A), describe the evaluation setup (Sec. V-B), and discuss the results (Sec. V-C).

A. Cache Size Sensitivity

We used *Cachegrind*[1] to acquire the cache sensitivity of a set of benchmark applications. Then, we use the cache sensitivity obtained from the benchmarks to generate synthetic task sets to evaluate our algorithm (details are in the next section). We define the cache sensitivity as the change of execution time given a change in cache size. It correlates with a task's WCET $c(s)$ given a cache partition of size s. The cache sensitivity is derived from the cache miss ratio over the size of the cache for each power of two from 1 KB up to 8 MB. Our experiments cover 77 embedded applications[2], including

[1]https://valgrind.org/info/tools.html#cachegrind

[2]Available at: https://github.com/ockwon9/FlexibleCachePartitioning

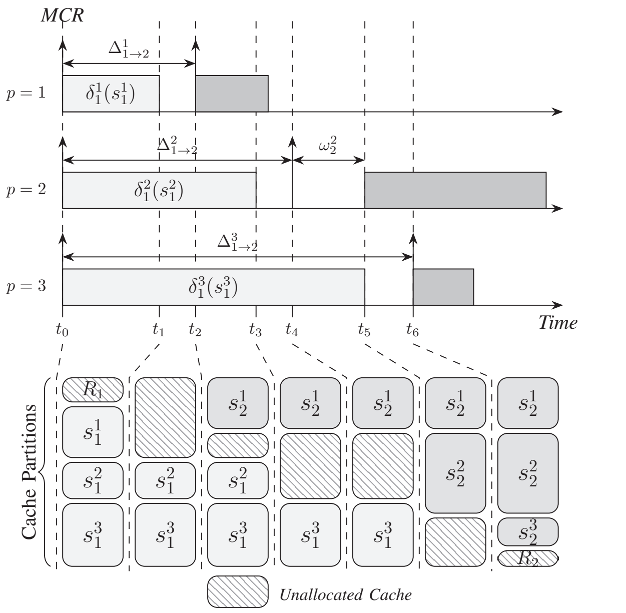


Fig. 2: An example of an asynchronous mode change on three processors and the evolution of the cache partitioning.

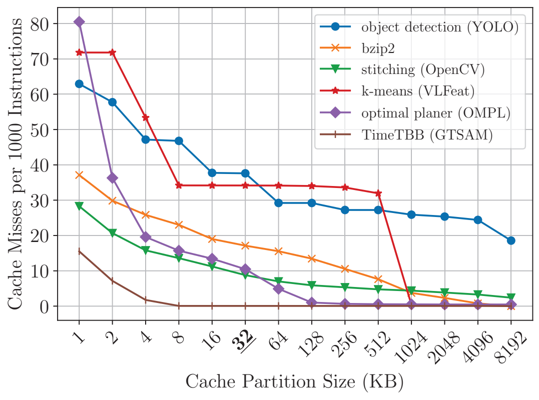


Fig. 3: Evolution of the cache miss rate per 1000 instructions depending on the cache partition size for selected benchmarks.

solvers (*SoPlex*), object-detection (*YOLO*), sorting, compression, WCET analysis tool (*Mälardalen WCET Benchmarks*), computer vision (*OpenCV*, *VLFeat*), and motion planning algorithms (*OMPL*). Selected results are shown in Fig. 3. The starting value and the slope of a task determine a task's cache sensitivity. To classify this observation, we create a separation criterion by computing the average difference between the largest measured cache size (8 MB) and the minimal relevant amount (32 KB) for all benchmarks. We chose 32 KB for this distinction, to match a reasonably high granularity, which can be achieved with cache coloring (4 KB pages, 8-way cache). Thereafter, we assign the benchmarks to a cache sensitivity class by comparing them to the separation criterion.

We call tasks that show only a minimal change in WCET as *weakly cache size sensitive* (e.g., TimeTBB in *GTSAM*). *Strongly cache size sensitive* tasks (e.g., optimal planner in *OMPL*) show a falling curve over the considered range. In the following evaluation, we use an even split of strongly and weakly cache sensitive tasks per task set.

B. Experimental Setup

We assume a non-preemptive fixed-priority partitioned scheduling [8] with *Rate Monotonic* priority assignment. To evaluate our cache partition allocation algorithm, we generate random task sets, with two tasks each. The tasks are based on the benchmark applications used in Sec. V-A. Each task set consists of two randomly selected tasks: one strongly cache size sensitive task and one weakly cache size sensitive task. We use the procedure from [4] to generate the task WCET based on the number of instructions, the number of memory accesses, and the cache miss ratio obtained from the corresponding benchmark. The instruction duration, data cache hit, and data cache miss delays are assumed to be respectively 1, 50, and 150 cycles [17]. We define a base setup where the task set is assigned the whole cache and select a target task set utilization for this base setup. Then, we use *UUnifast* [3] to randomly generate task utilizations from a uniform distribution based on the target task set utilization. According to the generated task utilizations, we assign the task periods using the WCET for the base setup. The period is not altered in the experiments, but the WCET changes, due to its dependency on the assigned cache partition. Accordingly, the actual task utilization is not fixed and always larger than or equal to the target utilization for the base setup. Task deadlines are set equal to the periods.

In the experiments, we use a mode change scenario consisting of 5 modes, because a larger number of modes impacts the comparative approaches more severely and renders the comparison biased (see Fig. 4c). The modes are connected in sequence with either synchronous or asynchronous mode changes. For asynchronous mode changes, the mode change durations $\Delta^p_{m \to n} = \sum_{\tau}^{\mathcal{T}^p_m} T_\tau$ are assumed, which enable a last execution of all tasks after the MCR.

To evaluate our approach, we compare our algorithm to a *Fair Allocation* and an *Unfair Allocation* on the synchronous and the asynchronous scenario. The comparison is done with the same task sets. In our experiments, *Fair Allocation* refers to a static allocation policy, where the cache is evenly split amongst the processors. The *Unfair Allocation* policy tries to find an uneven static allocation, for which the task sets of all processors in all modes are schedulable. Provided that the sum of the required cache by all the processors does not exceed the size available, we consider the allocation as successful. The results for the *Fair* and the *Unfair Allocation* do not change depending on the scenario because the static allocation restricts the policy to allocate a cache partition of at most the size of the one from the last mode. Thus the cache partition release and allocation times on the other processors are irrelevant.

We consider a system with four homogeneous processors ($P = 4$) and a 2 MB cache (unless stated otherwise) with 8 ways (as offered by the ARM CoreLink Level 2 Cache [1]). We assume cache coloring, which provides a higher partitioning granularity than way-based approaches. For our system, cache coloring provides 64 colors (i.e., allocatable cache blocks B), each having a size of 32 KB (i.e., the cache partitioning granularity q). The target utilization has been set to 20% (unless stated otherwise), as from our experiments, higher values often lead to an unschedulable system. Note that the actual task set utilization after dividing the available cache size is much higher. All experiments are run 1,000 times for each approach.

C. Experiments Evaluation

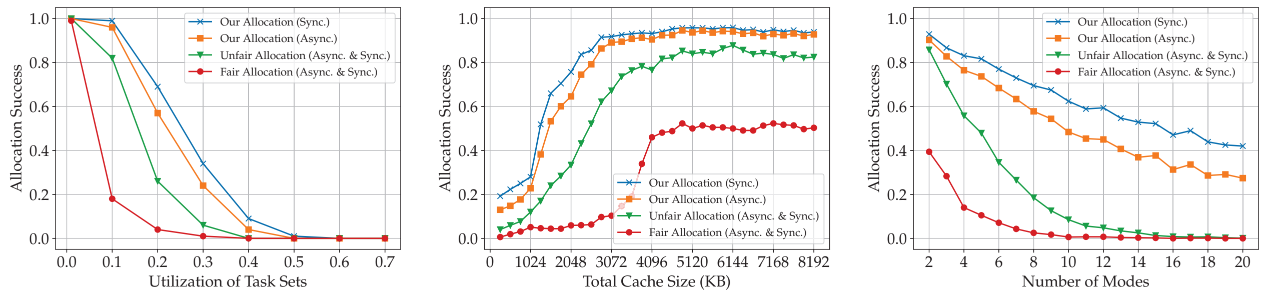
In our experiments, we vary the task sets target utilization (Fig. 4a), the total size of the cache (Fig. 4b) and the length of the mode change sequence (Fig. 4c).

From the experiments shown in Fig. 4, we can derive a strict ranking of the allocation policies. The *Fair Allocation* policy always yields the worst results due to its inflexible static allocation, since it depends on the cache size and the number of processors. The *Unfair Allocation* policy is aware of the memory requirements of each processor and derives the best possible static allocation, which increases the success ratio compared to the *Fair Allocation*. Nevertheless, this policy is static and lacks the flexibility required to cope with the stronger requirements of task sets with higher utilization and longer sequences of mode changes. These drawbacks are addressed by the two proposed allocation policies, which systematically outperform the *Fair Allocation* and the *Unfair Allocation* policies in our experiments in all cases. *Our Allocation* generally performs better for the synchronous mode change scenario because the overlap of cache partition releases and requests in the asynchronous scenario can create cases that are impossible to solve or for which a solution is not found by the heuristic.

In Fig. 4a, the proposed allocation policy deals better with higher target utilization. As soon as the target task set utilization is larger than 30%, *Fair Allocation* fails to find successful allocations. The *Unfair Allocation* follows at 40% target task set utilization. The discrepancy between *Our Allocation (Sync.)* and *Our Allocation (Async.)* is 3.88% on average. Note that the target task set utilization is derived for a base setup with 2 MB of cache and one processor, and increases for smaller cache partition sizes. The actual task set utilization is not directly comparable, since the *Unfair Allocation* and our proposed allocation policies do not distribute the remaining cache size and thus have a higher actual task set utilization for the feasible schedules, but also a higher allocation success rate.

As depicted in Fig. 4b, the proposed allocation policies outperform both the *Fair* and the *Unfair Allocation*, especially for small to medium cache sizes. In the interval of many real-time platforms, between 256 KB and 2048 KB, *Fair Allocation* has a 31.81% lower allocation success ratio, and *Unfair Allocation* has a 19.01% lower allocation success ratio in comparison to *Our Allocation (Async.)*.

Fig. 4c shows the variance in the allocation success ratio depending on the number of modes in a mode change sequence. With the increasing number of modes, the proposed policy shows high resilience in the synchronous and asynchronous scenario, with *Our Allocation (Sync.)* and *Our Allocation (Async.)* finding respectively 42% and 27.4% of successful



(a) Variable target utilization. (b) Variable cache size. (c) Variable number of modes.

Fig. 4: Allocation success ratio for the four different policies.

allocation for a sequence of 20 modes. In comparison, the *Unfair Allocation* falls below 40% allocation success ratio after only 5 modes, while the *Fair Allocation* starts with 39.4% for two modes and quickly drops to below 5% for 7 modes.

VI. Conclusion and Future Work

In this paper, we propose a cache partition allocation framework enabling flexible cache partitioning for multi-mode real-time systems. To guarantee feasible schedules in all modes and preserve timing predictability in the transient phase between modes, the proposed approach allocates partitions in the shared cache memory, considering synchronous and asynchronous mode changes of multiple processors. Through the experimental results, we show that our approach achieves a higher schedulability for varying target task set utilization, cache size, and length of mode change scenario compared to existing allocation methods. As future work, we plan to extend our framework to co-allocate memory bandwidth to provide better memory access predictability in multiprocessor systems.

Acknowledgment

Marco Caccamo was supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research.

References

[1] ARM, "Corelink level 2 cache controller l2c-310 technical reference manual," Tech. Rep. [Online]. Available: https://developer.arm.com/documentation/ddi0246/h

[2] E. Bini, M. Di Natale, and G. Buttazzo, "Sensitivity analysis for fixed-priority real-time systems," in *18th Euromicro Conference on Real-Time Systems (ECRTS'06)*, 2006, pp. 10 pp.–22.

[3] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1-2, pp. 129–154, 2005.

[4] B. D. Bui, M. Caccamo, L. Sha, and J. Martinez, "Impact of cache partitioning on multi-tasking real time embedded systems," in *2008 14th IEEE RTCSA*, Aug 2008, pp. 101–110.

[5] G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni, "Elastic scheduling for flexible workload management," *IEEE Transactions on Computers*, vol. 51, no. 3, pp. 289–302, 2002.

[6] D. Chiou, P. Jain, L. Rudolph, and S. Devadas, "Application-specific memory management for embedded systems using software-controlled caches," in *Proc. of the 37th DAC*, 2000, pp. 416–419.

[7] O. D. Dantsker, M. Theile, M. Caccamo, S. Yu, M. Vahora, and R. Mancuso, "Continued development and flight testing of a long-endurance solar-powered unmanned aircraft: Uiuc-tum solar flyer," in *AIAA Scitech 2020 Forum*, 2020, p. 0781.

[8] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, "Controller area network (can) schedulability analysis: Refuted, revisited and revised," *Real-Time Systems*, vol. 35, no. 3, pp. 239–272, Apr 2007.

[9] P. Ekberg, M. Stigge, N. Guan, and W. Yi, "State-based mode switching with applications to mixed criticality systems," *Proc. WMC, RTSS*, pp. 61–66, 2013.

[10] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A survey on cache management mechanisms for real-time embedded systems," *ACM Comput. Surv.*, vol. 48, no. 2, Nov. 2015.

[11] N. Guan, M. Stigge, W. Yi, and G. Yu, "Cache-aware scheduling and analysis for multicores," in *Prof. of the 7th EMSOFT*, 2009, p. 245–254.

[12] M. Kandemir, O. Ozturk, and M. Karakoy, "Dynamic on-chip memory management for chip multiprocessors," in *Prof. of CASES*, 2004, p. 14–23.

[13] H. Kim, A. Kandhalu, and R. Rajkumar, "A coordinated approach for practical os-level cache management in multi-core real-time systems," in *2013 25th ECRTS*, 2013, pp. 80–89.

[14] H. Kim and R. Rajkumar, "Real-time cache management for multi-core virtualization," in *2016 EMSOFT*. IEEE, 2016, pp. 1–10.

[15] D. B. Kirk, "Smart (strategic memory allocation for real-time) cache design," in *[1989] Proceedings. Real-Time Systems Symposium*, 1989, pp. 229–237.

[16] T. Kloda, B. d'Ausbourg, and L. Santinelli, "EDF schedulability test for the E-TDL time-triggered framework," in *2016 11th IEEE SIES*, 2016, pp. 1–10.

[17] T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna, "Deterministic memory hierarchy and virtualization for modern multi-core embedded systems," in *2019 IEEE RTAS*, 04 2019.

[18] T. Mück, A. A. Fröhlich, G. Gracioli, A. Rahmani, and N. Dutt, "Chips-ahoy: A predictable holistic cyber-physical hypervisor for mpsocs," in *Proc. of the 18th SAMOS*, 2018, p. 73–80.

[19] V. Nelis, J. Goossens, and B. Andersson, "Two protocols for scheduling multi-mode real-time systems upon identical multiprocessor platforms," in *2009 21st ECRTS*, 2009, pp. 151–160.

[20] L. T. X. Phan, M. Xu, and I. Lee, "Cache-aware interfaces for compositional real-time systems," *ACM SIGBED Review*, vol. 13, no. 3, pp. 52–55, 2016.

[21] J. Real and A. Crespo, "Offsets for scheduling mode changes," in *Proc. 13th ECRTS*, 2001, pp. 3–10.

[22] J. Real and A. Crespo, "Mode change protocols for real-time systems: A survey and a new proposal," *Real-time systems*, vol. 26, no. 2, pp. 161–197, 2004.

[23] S. Roozkhosh and R. Mancuso, "The potential of programmable logic in the middle: Cache bleaching," in *2020 IEEE RTAS*, 2020, pp. 296–309.

[24] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham, "Mode change protocols for priority-driven preemptive scheduling," *Real-Time Systems*, vol. 1, no. 3, pp. 243–264, Dec. 1989.

[25] K. Tindell and A. Alonso, "A very simple protocol for mode changes in priority preemptive systems," 1996.

[26] 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 *2019 IEEE RTAS*, 2019, pp. 345–356.