Segment Streaming for the Three-Phase Execution Model: Design and Implementation

Muhammad R. Soliman*, Giovani Gracioli**, Rohan Tabish***, Rodolfo Pellizzoni*, and Marco Caccamo
*University of Waterloo, Canada, {mrefaat, rpellizz}@uwaterloo.ca
**Technical University of Munich, Germany and UFSC, Brazil, giovani@lisha.ufsc.br
***University of Illinois at Urbana-Champaign, USA, rtabish@illinois.edu
Technical University of Munich, Germany, mcaccamo@tum.de

***Abstract*—Scheduling tasks using the three-phase execution model (load-execute-unload) can effectively reduce the contention on shared resources in real-time systems. Due to system and program constraints, a task is generally segmented and executed over multiple intervals. Several works showed that co-scheduling memory (unload-load) and computation phases can improve the system schedulability by hiding the memory transfer time. However, this is limited to segments of different tasks and hence executing segments of the same task back-to-back is not allowed. In this paper, we propose a new streaming model to allow overlapping the memory and execution phases of segments of the same task. This is accomplished by a segmentation framework implemented within an LLVM-based compiler-level tool along with a Real-Time Operating System (RTOS) API to handle load/unload requests. Memory phases are processed by a DMA engine that loads/unloads the task content into ScratchPad Memory (SPM). We provide a schedulability analysis of the proposed model under fixed priority partitioned scheme and an RTOS implementation of the API on a latest-generation Multiprocessor System-on-Chip (MPSoC).**

***Index Terms*—Real-time systems; compiler segmentation; real-time operating systems; MPSoC**

I. Introduction

The increasing interest for emerging technologies, like autonomous driving cars, unmanned aerial vehicles, and smart manufacturing, has changed the type of workload considered "real-time" [1], [2]. These workloads demand both high performance and timing guarantees, and usually present high usage of I/O devices (such as cameras and LIDAR sensors) and memory requirements. Current multiprocessor Systems-on-Chip (MPSoC) platforms are the hardware manufacturers' answer for these new emerging applications [3]. MPSoCs provide a feature-rich environment, composed of multiple processing elements, high-bandwidth I/O devices and memories, and Direct Memory Access (DMA) engines.

As in Commercial-Off-The-Shelf (COTS) multicore processors, MPSoCs have shared resources that represent bottlenecks in terms of performance and predictability. In particular, the main memory shared by the heterogeneous processing elements constitutes a significant source of unpredictability [4], [5]. Memory-aware models, such as the PRedictable Execution Model (PREM) [6] and its extensions [7]–[13], have attempted to solve the memory contention problem by dividing the execution of each task in three phases[1]: memory load (which loads data/instruction from main memory into local memory, either a cache or scratchpad), execution phase (where the processor executes the task from the local memory), and memory unload (copies modified data back to main memory). Contention at the main memory level is avoided by sequentializing the load and unload phases of different processors. Furthermore, the latency of memory accesses can be hidden by employing a DMA engine: using a double-buffering technique [7], [10], [14], [15], a processor executes one task while the DMA loads the memory of the next task.

A key complexity related to the implementation of the three-phase model is program segmentation: due to either limited size of local memory, conditional program execution, dynamic memory usage, or scheduling constraints, a task must be divided into multiple segments, each consisting of load-execution-unload phases. Recently, we have introduced an automated framework [16], based on the LLVM compiler, that segments a set of real-time tasks with the objective of improving the system's schedulability. However, our existing framework, as well as related work in multitasking scheduling for the three-phase model [7], [17]–[21], suffers from two main limitations: (i) it does not provide an implementation for the interface between the application and the Real-Time Operating System (RTOS). Since the DMA is under the RTOS control, the task must communicate to the RTOS the data to load/unload; and (ii) it can only hide memory latency by overlapping the memory and execution phases for segments of different tasks. The main reason is that in general, it might not be possible to predict the data accessed in a segment until the previous segment of the same job has completed. Unfortunately, this leads to schedulability degradation for systems that run few, or even a single task on each processor; a situation we find common as the number of processors in an MPSoC increases.

In this work, we propose and implement a new streaming execution model for three-phase real-time tasks. Whenever the data used by a segment can be determined ahead of time, we allow it to be streamed into local memory while the previous segment of the same job executes. In this way, segments of the same task can execute back-to-back, without suffering unnecessary blocking time as in previous works. In details, our main contributions are:

1) We introduce the new streaming execution model to allow overlapping of memory and execution phases of segments of the same task. We further show that the

[1]The original PREM model had only two phases: memory and execution. The considered model is also referred to as three-phase model or acquisition-execution-replication model in related work.

schedulability of the proposed model can be assessed with limited changes to existing schedulability analyses for multitasking systems under the three-phase model.
2) We extend the compiler framework introduced in [16] to automatically segment streaming tasks.
3) We propose an OS-independent application programming interface (API) to realize the scheduling and segmentation decisions. We implement the API in Erika [22], [23], a commercial open-source RTOS and certified with the OSEK/VDX standard, and evaluate it in terms of memory footprint and execution time using a latest-generation MPSoC platform.

The rest of this paper is organized as follows. Section II discusses the related work. Section III presents background on scheduling three-phase tasks and motivates the proposed streaming model. Section IV introduces the system model and assumptions. Section V introduces the streaming execution model. Section VI presents the proposed OS-level API. Section VII discusses the schedulability analysis and the program segmentation framework. Section VIII presents the evaluation of the schedulability test and the API implementation on top of Erika RTOS. Finally, Section IX concludes the paper.

II. Related Work

Due to its increased predictability, the three-phase model has recently received significant attention from the real-time community [3], [7]–[13], [15], [17]–[21], [24]–[30]. While the primary goal of all such works is to avoid contention in main memory by scheduling a single memory phase at a time, the employed memory scheduling algorithm differs, ranging from round-robin [10] or TDMA [3], [7], [15], [20] arbitration among processors, to static [11]–[13], [25], [28], [29] or priority-based [17]–[19], [21], [30] schedule among tasks. Since our goal in this paper is to analyze each processor in isolation, we assume a TDMA schedule among processors.

Regarding implemented approaches [3], [8]–[11], [13], [15], [24], [26], [28], [30], the main difference is in the targeted processor type - general-purpose or GPU - and the targeted local memory - cache or ScratchPad Memory (SPM). Platforms that employ caches must use a processor to execute memory phases. On the other hand, platforms that employ SPM can use a separate DMA engine to load/unload data to SPM. The usage of a DMA component allows to not only achieve predictability in accessing main memory, but also to hide the access latency by overlapping processor execution with DMA transfers. In this paper, we are concerned with executing a set of sequential programs on a general-purpose processor, and we assume that memory phases are executed by a DMA engine.

None of the previous three-phase scheduling schemes for multitasking DMA-based systems [7], [15], [16], [18] allow overlapping memory phases of a task with the execution of the task itself. The work in [10], [14] (for GPU kernels) and [31] (for general purpose processors) allow executing consecutive segments of the same task in sequence, but it does not support context-switching between multiple tasks. In this sense, *our work is the first real-time framework to generalize previous DMA-based approaches by allowing overlapping memory with the execution of either the same task, or a different task.* Outside the real-time community, DMA has been utilized to hide the memory latency and hence enhance the performance in SPM-based single-processor and multi-processor systems [32]–[34]; however, all such work focuses on a single application or computational kernel rather than multitasking systems.

Automated program segmentation for the three-phase model has been discussed in [10], [14] for GPU kernels, and in [28] and [16] for general-purpose processors. They rely on the LLVM compiler infrastructure, and employ loop splitting and tiling [35] to break loops that are too large to fit in local memory. The focus in [10], [14], [28] is on single-task execution; they use a greedy technique that maximizes the segment length to fit in the available SPM size. As we show in [16], this leads to excessive blocking time in multi-tasking systems; instead, we optimize segmentation decisions for the entire task set to improve its schedulability. Furthermore, only [16] targets DMA-based systems. Therefore, in this work we rely on the segmentation framework we introduced in [16], but extend it to allow streaming over tiled loops. However, our previous work does not discuss how the program interfaces with the OS to communicate which data must be loaded/unloaded by the DMA. In practice, this is far from trivial, and *a major contribution of this paper is the design and full implementation of the OS programming interface* in Section VI.

The closest related work has been recently presented in [30]. Specifically, the authors implemented the three phase model for a cache-based architecture in a research RTOS, and provided an API to specify the data to load/unload. They further propose to execute memory phases on a separate master processor, allowing segments of the same task to execute consecutively, similarly to our streaming model. However, both the OS implementation, as well as the provided schedulability analysis, is limited to a single task per core. Furthermore, no automated segmentation strategy is discussed, which increases the complexity of porting the code, also given that the API does not directly support loop tiling (see Section VI-A).

III. Background and Motivation

To provide required background on the scheduling of three-phase tasks and to motivate our new streaming model, we next discuss how tasks are scheduled in multi-tasking approaches for DMA-based systems [7], [15], [16], [18]. In essence, one can pipeline computation and memory phases using a double-buffering technique [7], [10], [14], [15], at the cost of halving the available local memory space. Figure 1 depicts the basic approach in [7], [15], [16], designed to schedule a set of partitioned fixed-priority, sporadic tasks.

Each processor has a dedicated SPM divided into two partitions. Figure 1 shows a schedule example on one processor, where the task under analysis executes a sequence of three segments s^1, s^3 and s^6, while segments s^2, s^4 and s^5 belong to other tasks. The schedule consists of a sequence of scheduling intervals. During each interval, a segment of a task (ex: s^2 in interval ②) executes using data and instructions in one partition. At the same time, the DMA unloads the content of the previous segment (s^1) and loads the next segment (s^3) in

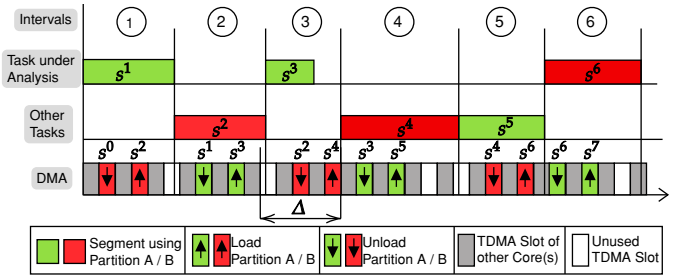


Fig. 1: Example: TDMA memory schedule with $M = 2$ cores.

the other partition. Note that the time required to complete the DMA memory operations depends on the number of processors in the system, as the DMA follows a TDMA arbitration where slots are assigned to processors. Furthermore, the length of each scheduling interval is the maximum of the execution time for the corresponding segment, and the DMA time. In the figure, interval ③ is bounded by the DMA time (the interval finishes when the load of s^4 completes), while all other intervals are bounded by the execution time of the segment.

Note that in the described approach, segments are scheduled non-preemptively; this ensures that two SPM partitions are sufficient to overlap execution and memory phases. The works in [19]–[21] propose to adopt preemptive scheduling, but they require a number of local memory partitions equal to the number of tasks, which we find excessively wasteful. The key issue with non-preemptive scheduling is blocking due to lower priority tasks, which is exacerbated when a task comprises multiple segments. The described approaches do not allow segments of the same task to run back-to-back: in general, the data required by a segment cannot be determined until the previous segment completes. Furthermore, consecutive segments typically share data; hence, we cannot load them in different partitions. Therefore, at least one interval must be inserted between consecutive segments of the same task. Unfortunately, no higher priority task may be ready during such interval in the worst case; hence, the interval could be used to execute a segment of a lower priority task, or the processor could idle while the DMA loads/unloads the task under analysis. This effectively adds to the response time of the task a number of blocking intervals proportional to its number of segments. Motivated by such issue, in this paper we develop a streaming execution model that allows segments of the same task to execute back-to-back, hence greatly reducing the blocking time suffered by each task.

Finally, we discuss two important implementation considerations. First, two models have been proposed to bound the length of DMA operations. The implementations in [15], [26] use a coarse-grained TDMA approach, where the size of each slot is sufficient to load or unload half the SPM (one partition). If we let σ to denote the size of the slot, and M the number of processors, the worst-case memory time [15] is then $\Delta = \sigma \cdot (2M + 1)$; as again shown in interval ③, the previous interval can finish right after the beginning of a TDMA slot assigned to the core under analysis, forcing that slot to be wasted. Under such *fixed-time* DMA model, the schedulability analysis can abstract away the details of the DMA model, and only use Δ as the fixed memory time for each interval. The implementations in [3], [7] instead rely on a fine-grained TDMA model, such that the length of DMA operations is proportional to the amount of data to load/unload. The schedulability analysis for such *variable-time* DMA model [7] is more complex since it needs to consider the DMA times of individual segments. Due to space limitations, and since our focus is on introducing the streaming model and discussing the OS interface, in this paper we will only employ the simpler fixed-time model. However, we are also able to extend the variable-time analysis [36] to handle our streaming model. Second, note that we cannot predict whether a task will execute from Partition A or B at run-time. Hence, all tasks must be relocatable in the SPM. To this end, the implementation in [15] forces all tasks to use only relative addresses, but such solution is not portable, and does not work with pointer-based data structures. Instead, the implementations in [3], [7] use address remapping in hardware, either through a dedicated MMU unit, or the standard virtual memory paging mechanism.

IV. System Model

We consider scheduling sequential, sporadic three-phase tasks on an MPSoC platform comprising multiple processors. Tasks are partitioned to processors. Each processor has a dedicated SPM, which is used to execute its assigned tasks. A DMA engine is used to execute memory phases and shared among all processors. We consider a fixed-time DMA model, so that the time required to complete all memory phases in each scheduling interval is constant and equal to Δ. We further assume that a SPM-based OS is used to control the schedule of task execution segments and memory phases, similarly to related work [3], [15], and that virtual memory (paging) is used for task relocation.

We let $\Gamma = \{\tau_1, \ldots, \tau_N\}$ to denote the set of sporadic tasks executed on a given processor. We use T_i to denote the period (or minimum inter-arrival time) of task τ_i, and D_i for its relative deadline. We assume constrained deadline: $D_i \leq T_i$. At run-time, each job of task τ_i executes a sequence of task segments. While the sequence can vary between jobs of the same task, we assume a unique initial segment s_i^{begin} and final segment s_i^{end}, as we consider programs with a single entry and exit point. A job of τ_i that arrives at time t is thus feasible if s_i^{end} completes execution no later than $t + D_i$ [2].

To describe the possible segment sequences for jobs of τ_i, we employ the conditional three-phase model introduced in [16]: τ_i is characterized by a Direct Acyclic Graph (DAG) of segments $G_i = (S_i, E_i)$, where S_i is a set of nodes representing segments, and E_i is a set of directed edges representing precedence constraints between segments. To simplify notation, we will

[2]Note that some previous related work [15] required the unload phase of s_i^{end}, rather than its execution phase, to complete by $t + D_i$. The use of either definition depends on platform-related assumptions, i.e., whether output operations are performed during the execution of the task, or during the unload phase. Since our model guarantees that the unload of a segment completes in the scheduling interval after s_i^{end}, we could meet the requirements of [15] by simply reducing the relative deadline by the memory time Δ.

write $P \in G_i$ to denote a maximal path P of DAG G_i, that is, an ordered sequence of segments that starts with s_i^{begin} and ends with s_i^{end}. The execution of any job of τ_i can then be represented by a maximal path in its DAG G_i. In practice, as we discuss in Section VII-A, the compiler framework creates the DAG by analyzing the Control Flow Graph (CFG) of the program and dividing CFG blocks among segments; segments that have multiple outgoing edges in the DAG represent branches in the CFG. As an example, Figure 2 shows a DAG with three maximal paths: $P = (s^0, s^1, s^2, s^3, s^4, s^9)$, $P' = (s^0, s^5, s^6, s^9)$, and $P'' = (s^0, s^5, s^7, s^8, s^9)$. At run-time, a job of τ_i always executes $s^0 = s_i^{begin}$ first. While executing s^0, it decides whether to branch to s^1 (path P) or to s^5; if the latter, then while executing s^5 it decides to branch to s^6 (path P') or s^7 (path P''). The job then executes all segments in the path until it completes $s^9 = s_i^{end}$.

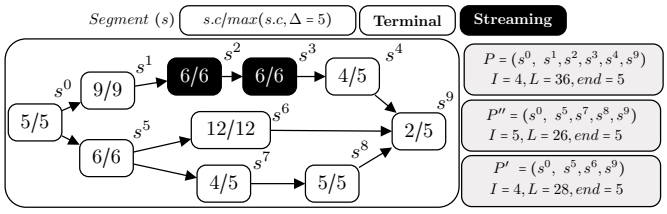


Fig. 2: Example segment DAG (s^0 is s^{begin} and s^9 is s^{end}).

We next introduce some additional segment-related notation that will be used throughout the paper, including in Figure 2. Each segment s is either a terminal or a streaming segment: a segment s^j is a streaming segment if s^j has a unique immediate successor segment s^k, and s^k can be executed in the scheduling interval after the one where s^j executes; otherwise, s^j is terminal. We detail the conditions that allow s^j to stream into s^k in Section V. As explained in Section III, related work [7], [15], [16] assumed that all task segments are terminal. We use $s.c$ to denote the worst-case execution time of s. We further define the length $s.l$ of a segment to be the maximum length of any scheduling interval where s is executed. Since the memory time is Δ, and the length of an interval is the maximum of the segment execution time and the memory time, we have $s.l = \max(s.c, \Delta)$. Finally, for a path P, we use $P.I$ to denote the number of terminal segments in the path, $P.L$ for the sum of the lengths of all segments in the path, and $P.end$ for the length of the last segment, that is $P.end = s_i^{end}.l$; Figure 2 shows the corresponding values for the three maximal paths in the example.

V. Streaming Execution Model

We now introduce our new streaming model for conditional three-phase tasks. As in previous approaches discussed in Section III, the code and data of each segment must be loaded in the local SPM of the processor under analysis before the segment starts execution. However, we do not statically divide the SPM into two partitions. Instead, we simply require that the footprint of each segment - the total number of virtual memory pages occupied by the code and data of the segment - is no larger than half the SPM size. Since virtual memory allows us to allocate a page of a segment to any frame in the SPM, this assumption still guarantees that the code and data of two different segments can simultaneously fit in the SPM.

The DMA operations performed during a scheduling interval depend on the type of segments executed during the previous, current, and following interval. Terminal segments behave in the same way as previous work, requiring DMA load/unload operations. We mark a segment s^j of task τ_i as streaming if the footprint of the next segment s^k of τ_i can be loaded in parallel with the execution of s^j. Note that this effectively requires the footprint of s^k to be known while the segment of τ_i executed before s^j, call it s^l, is running. Hence, s^k must be the only immediate successor of s^j (no conditional segment path decision can be taken during s^j). Also note that to allow s^j and s^k to execute back-to-back, data/code shared between the two segments should not be loaded in parallel with s^j, since it is already in the SPM; hence, the DMA only loads a subset of the footprint of s^k (the pages which do not also belong to s^j). To differentiate such operation from the case where the entire footprint of a segment is loaded, we call it a *swap-in* operation. Similarly, if the previous segment s^l was also streaming, then data/code shared between s^l and s^j must not be unloaded; instead, only the data/code of s^l that is not used by s^j is *swapped-out*.

We next present a comprehensive scheduling example in Figure 3 for a task under analysis over 11 scheduling intervals. The task under analysis has six segments: s^1 and s^9 are terminal segments while s^3, s^4, s^5, and s^8 are streaming segments. Segments s^2, s^6 and s^7 are terminal and belong to other tasks.

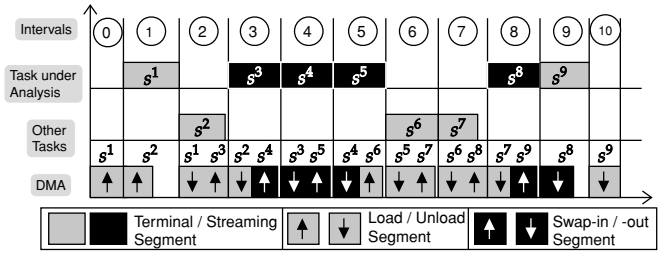


Fig. 3: Streaming Execution Model.

The schedule in Intervals ⓪, ① and ② follows the same scheme as in previous work on the three-phase model, since it involves terminal segments. Assume that initially no task is ready, the SPM is empty, and that the task under analysis arrives at the beginning of Interval ⓪. Then the scheduler first loads s^1, so that it can be executed in Interval ① after the load operation finishes. Since s^1 is terminal, it cannot be followed by a segment of the task under analysis; instead, assuming that a segment s^2 of another task is ready, s^2 is loaded during Interval ① to be executed in Interval ②. As s^3 is the first streaming segment after a terminal segment, it is completely loaded during Interval ② after s^1 is unloaded. Intervals ③, ④, and ⑤ depict the case of multiple streaming segments executed in sequence. First, s^4 is swapped-in the SPM during Interval ③ and executed in Interval ④. Then, s^5 is swapped-in during Interval ④ in exchange for swapping-out

s^3, then executed in Interval ⑤. Although s^5 can be streamed into s^8, we assume that the stream is preempted by segments s^6 and s^7 from other tasks with higher priority than the task under analysis. To execute s^6 during Interval ⑥, s^4 must be swapped-out and s^6 must be loaded during Interval ⑤, while the preempted streaming segment s^5 executes. To load s^7, s^5 has to be completely unloaded from the SPM during Interval ⑥. As the task under analysis can resume execution in Interval ⑧, s^8 is loaded in Interval ⑦. Due to the preemption of the stream, all the code and data for s^8 has to be loaded. While s^8 executes in Interval ⑧, we swap-in s^9 in exchange for unloading s^7. Finally, assuming no other task is ready, s^8 is swapped-out in Interval ⑨ and s^9 is unloaded in Interval ⑩.

We can now summarize the scheduling rules for our model, where $\tau(s^i)$ denotes the task to whom segment s^i belongs.

1) The schedule comprises a sequence of scheduling intervals. During each Interval$_i$, at most one segment s^i is executed in parallel with at most one unload or swap-out memory operation, and one load or swap-in operation. The interval ends when both the segment execution (if any) and the memory operations (if any) have completed.
2) If a segment s^i executes during Interval$_i$, or if there is a ready task at the end of Interval$_i$, then Interval$_{i+1}$ starts immediately after Interval$_i$ ends. Otherwise, Interval$_{i+1}$ starts when a task becomes ready.
3) Given two segments s^i and s^{i+1} executed in successive scheduling intervals, if $\tau(s^i) = \tau(s^{i+1})$ then s^i must be a streaming segment.
4) Scheduling decisions are only taken at the beginning of a scheduling interval; namely, at the beginning of Interval$_i$, the scheduler decides which segment s^{i+1} (if any is ready and it does not violate Rule 3) to execute in the following Interval$_{i+1}$.
5) Consider memory operations performed during Interval$_i$ executing segment s^i (if any). If Interval$_{i-1}$ executed a segment s^{i-1}, then: if $\tau(s^{i-1}) = \tau(s^i)$, s^{i-1} is swapped-out; otherwise, s^{i-1} is unloaded. If Interval$_{i+1}$ will execute a segment s^{i+1} (based on the decision in Rule 4), then: if $\tau(s^{i+1}) = \tau(s^i)$, s^{i+1} is swapped-in; otherwise, s^{i+1} is loaded.

We now make a few observations based on the rules. First, Rule 4 does not specify how to choose the next scheduled segment; to construct a schedulability analysis, in the rest of the paper we will assume a fixed per-task priority assignment. Second, assuming that the footprint of each segment is no larger than half the SPM guarantees that segments can always be allocated in the SPM: by Rule 5, a segment s^{i-1} is always unloaded or swapped-out during Interval$_i$. Hence, after the unload/swap-out operation in Interval$_i$, only the content of segment s^i (if any) remains in the SPM, and the other half of the SPM can be used to either load or swap-in the next segment s^{i+1}. For the same reason, the maximum amount of memory that needs to be transferred by the DMA during a scheduling interval is equal to the size of the SPM; this allows us to bound the memory time Δ under the fixed-size model.

One final key observation is related to virtual memory and the segment footprint. Assume that two consecutive streaming segments s^j, s^k share a data object a. Since SPM memory allocation can only be performed at the granularity of a memory page, the entire page(s) containing a must be allocated in the SPM while the segments execute, even if a is much smaller than the 4 KB page size and the segments themselves are not using the remaining data in the page(s). When the shared data changes from pair to pair of consecutive segments, such internal fragmentation could lead to a large increase in the footprint of the segments, hence wasting SPM space and memory bandwidth. For this reason, in this paper we consider a more restrictive streaming implementation at the program code level; namely, we assume that for a sequence of streaming segments, the footprint of each segment can be divided into two regions: a region containing data/code [3] that is shared among all segments in the sequence, and a region containing data that is used only by that specific segment. Each region can then be laid out sequentially in virtual memory so that internal fragmentation is limited to 4 pages. That is, the code/data in each region is aligned to the beginning of a virtual page and objects/buffers are allocated sequentially, hence only the last page may be under-utilized. As our model has two tasks in the SPM with fragmentation of 1 page/region, the maximum fragmentation is 4 pages. In the next section, we refer to the code and data in the first region as 'objects' while we refer to the code and data in the second region as 'buffers'.

VI. OS Programming Interface

In this section, we propose an OS-level API to be inserted in the code of a task to partition it into segments and to manage the SPM content. Table I presents a short description for each API function.

The SPM allocated to a task is partitioned into a set of buffers and objects that are allocated/deallocated using API calls. A buffer is used for segment streaming such that the content of the buffer can be swapped during execution. We refer to other SPM allocations that are not buffers as objects. An object/buffer can be a 1D or a 2D memory block. A 1D object/buffer represents a 1D (sub-)array or any linear structure such that all its content is contiguous in the main memory. Hence, a 1D object/buffer has a single length `size`. A 2D object/buffer represents a sub-array of a 2D array in the main memory. A 2D transfer depends on `spitch, dpitch, height` and `width` as in Figure 4: `spitch` is the width of the source 2D array in main memory, `dpitch` is the width of the destination 2D array in the SPM, `height` and `width` represent the number of rows and the number of bytes per row to be copied. Each object/buffer has a usage attribute `attr` that indicates if the data in the object/buffer is write-only (`WO`), read-only (`RO`), or read-write (`RW`).

Objects are allocated (deallocated) in terminal segments only using `allocate/allocate2d` (`deallocate`) functions. Buffers are allocated before the first streaming segment using `allocate_buffer` and deallocated using `deallocate_buffer` at the end of the segment stream. The content of a buffer can be modified during the execution of streaming segments using `swap_buffer` and `swap2d_buffer` functions.

[3] We load the code for the whole task in this work.

TABLE I: Proposed OS-level API to support the streaming model.

`int allocate(uint64_t *src, uint64_t *dst, int size, int attr)` / `int allocate2d(uint64_t* src, uint64_t* dst, int width, int height, int spitch, int dpitch, int attr)` : Allocate an object at `dst` and copy 1D/2D array from `src` if `attr` is `RO/RW` → return the ID assigned to the object.
`void deallocate(int id)` : Release the object with ID `id` and write-back the data if the object is `WO/RW`.
`int allocate_buffer(uint64_t *dst, int attr)` : Allocate a buffer at `dst` for data streaming → return the buffer ID.
`void swap_buffer(int id, uint64_t *src, int size)` / `void swap2d_buffer(int id, uint64_t *src, int width, int height, int spitch, int dpitch)` : Swap the 1D/2D data in the buffer with ID `id` by writing-back the current data for `WO/RW` buffer and copying data from `src` for `RO/RW` buffer.
`void deallocate_buffer(int id)` : Release the buffer with ID `id` and write-back the data if the buffer is `WO/RW`.
`void dispatch()` : Force all buffer DMA requests to move from waiting queue to dispatch queue.
`void end_segment()` : End segment execution.

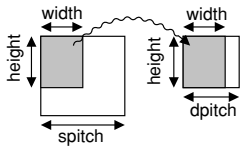


Fig. 4: 2D transfer example.

TABLE II: Data fields of an entry in the Three-Phase Table (3PT) and Streaming Table (ST).

Data Structure Field	Description
`attr`	read-only, write-only, or read-write.
`size`	Size for 1D object/buffer.
`width`, `height`, `spitch`, `dpitch`	Information for a 2D object/buffer.
`src`	Pointer to the address in main memory.
`dst`	Pointer to the SPM address.

A swap implies that the current data will not be needed by the next segment, and that new data should be loaded in the buffer before the end of the next segment of the task; note that the swap-out and swap-in operations defined in Section V correspond to reading / writing the content of buffers from / to main memory.

The `end_segment` function informs the OS that the execution of the current segment has finished. If there are still pending memory transfers in the current interval, the OS suspends execution until all transfers have completed. Then, a new interval starts according to Rule 2, and the scheduler is invoked according to Rule 4. Finally, the `dispatch` function informs the OS that the buffers allocated so far will be needed by the next segment. This function is only used in the terminal segment that precedes a segment stream since all buffers are allocated in this segment.

The proposed API represents a general OS interface for the three-phase model resembling other memory management interfaces like CUDA. The API calls can be inserted manually or automatically during the program compilation as we propose in this work. Although the proposed API represents system calls to the OS, lightweight RTOSes do not have a separation between user/kernel spaces. This means that there is no system-call in the standard way. In Erika, OS-service calls are embedded into the application code and hence do not cause significant overhead, as we show in the evaluation. For an OS with significant mode-switch overheads, the allocation/swapping functions could be implemented to construct one data structure for multiple objects/buffers to be passed to the OS using a single system call at the end of the segment.

A. API Implementation

We now discuss how one can implement the API in an RTOS and show how it works with an example. The OS tracks the objects and buffers used by each task using two tables, *Three-Phase Table (3PT)* and *Streaming Table (ST)*. An entry in the 3PT or ST includes the fields in Table II. An entry is created in the 3PT when either `allocate` or `allocate2d` is called and the entry ID is returned. When `deallocate` is called with the entry ID, the entry is removed from the table. Similarly, an entry is created in the ST using `allocate_buffer` and is removed using `deallocate_buffer`.

The memory transfers of a task are managed by the OS using three different queues: *Three-Phase Queue (3PQ)* for objects and *Streaming Wait Queue (SWQ)*/*Streaming Dispatch Queue (SDQ)* for buffers. Each queue contains a list of DMA transfer requests: either reading an object/buffer from main memory to the SPM, or writing back an object/buffer from the SPM to main memory. Allocating a read-only/read-write object adds a read request to the 3PQ, and deallocating a write-only/read-write object adds a write request to the 3PQ. Swapping a buffer adds a write request for the current data to the SWQ if the buffer is write-only/read-write, and a read request for the new data if the buffer is read-only/read-write. Note that the first swap of a buffer does not add a write request as the buffer is empty. Deallocating a buffer adds a write request for the current data to the SWQ. Using two queues for the streaming buffers is necessary: if a single queue is used, the current segment may call `swap_buffer`/`swap2d_buffer`/`deallocate_buffer` and hence add new read/write requests to the queue while the requests for the next segment are processed. Using two queues avoids this issue by distinguishing between the requests for the next segment, which are processed during the current segment from the SDQ, and requests added to the SWQ by the buffer swap/deallocation. Moving requests from the SWQ to the SDQ is done by the scheduler, with the exception of the explicit usage of `dispatch` function before the segment stream starts.

Note that the OS keeps separate tables and queues for each task as attributes of the Task Control Block (TCB). We provide an evaluation in terms of memory footprint and execution time of the proposed implementation in Section VIII. Since the data required for the first segment in the task has to be allocated in

the SPM before executing the segment, the TCB also contains the allocation state for the first segment, which the OS copies in the 3PT when a new job of the task starts.

Besides allocation/swapping/deallocation, DMA requests are managed based on the scheduling decision at the beginning of each interval. Algorithm 1 shows the steps taken by the scheduler for a given processor at the beginning of Interval$_i$. The scheduler starts by determining the segment s^{i+1} to be executed in Interval$_{i+1}$ (Rule 4). Then, it dispatches buffer DMA requests (if any) from SWQ to SDQ of task $\tau(s^i)$. Finally, it determines the memory operations to be carried out in Interval$_i$ based on Rule 5. In the case of a swap-out/swap-in operation, the write/read requests for the previous/next segment in the SDQ are sent to the DMA. For an unload operation, write requests for modified (write-only or read-write) objects in the 3PT of $\tau(s^{i-1})$ are added to its 3PQ. Note that if an object is in 3PT, this means that it has not been deallocated yet and thus needs to be reloaded when the task resumes execution. Therefore, read requests are also added to 3PQ for all objects in 3PT[4]. Then, write requests in the 3PT are sent to the DMA. For buffers, all write requests in SWQ are sent to the DMA even though they have not dispatched to SDQ. This is necessary as all the modified data has to be written back to main memory. For a load operation, if $\tau(s^{i+1})$ has not started yet, the initial state of the 3PT is copied from the TCB of $\tau(s^{i+1})$ and read requests are added to the 3PQ for read-only/read-write objects. Then, all read requests in 3PQ and SDQ are sent to the DMA.

Finally, we assume that the RTOS has access to a platform-specific DMA driver, which it uses to send requests to the DMA by writing DMA descriptors (or a pointer to each descriptor) to a shared memory location. TDMA arbitration among processors can either be implemented by the DMA hardware through multiple channels, or in software by the driver. For 2D transfers, the data to be read/written is not contiguous; however, standard scatter-gather DMA capability can be employed to transfer the object/buffer in a single DMA transaction. Note that when the DMA finishes transferring a request, it must generate an interrupt to informs the OS to remove the request from the appropriate queue (either 3PQ, SDQ or SWQ for $\tau(s^{i-1})$/$\tau(s^{i+1})$).

Example. We describe how the schedule from Figure 3 is accomplished using the proposed API and implementation with an example program. Figure 5a shows the source code of `histogram` function that uses two arrays h and a. It starts with an initialization of all elements of h in `init(h, 128)`, then it iterates over elements of a masking their values and then incrementing the corresponding histogram bin in `hist(h, a, 500)`. Our goal is to do the initialization of h in a segment and break the histogram loop into 5 segments such that each segment processes 100 element of array a. The code in Figure 5b represents the segmented function after adding the API calls[5] and Figure 6 shows the OS tables and queues for the task; for each scheduling interval in Figure 3, we show

[4]Even if the object is write-only, as the object might have been modified before it is written to main memory.

[5]We unrolled the outer loop that iterates over segments to illustrate the details of the execution.

Algorithm 1 Scheduler logic in Interval$_i$

Determine segment s^{i+1} (if any)
if a segment s^i is scheduled in Interval$_i$ **then**
  $\tau(s^i)$: Dispatch requests from SWQ to SDQ.
if a segment s^{i-1} was scheduled in Interval$_{i-1}$ **then**
  if $\tau(s^{i-1}) = \tau(s^i)$ **then** (swap-out)
    $\tau(s^{i-1})$: Send write requests in SDQ to DMA.
  else(unload)
    $\tau(s^{i-1})$: Add read/write requests to 3PQ for objects in the 3PT.
    $\tau(s^{i-1})$: Send write requests in 3PQ to DMA.
    $\tau(s^{i-1})$: Send write requests in SWQ to DMA.
if a segment s^{i+1} is scheduled in Interval$_{i+1}$ **then**
  if $\tau(s^{i+1}) = \tau(s^i)$ **then** (swap-in)
    $\tau(s^{i+1})$: Send read requests in SDQ to DMA.
  else(load)
    if $\tau(s^{i+1})$ has not started **then**
      $\tau(s^{i+1})$: Copy initial state of 3PT from TCB.
      $\tau(s^{i+1})$: Add read requests to 3PQ for objects in 3PT.
    $\tau(s^{i+1})$: Send read requests in 3PQ to DMA.
    $\tau(s^{i+1})$: Send read requests in SDQ to DMA.

the content for the tables and queues after the scheduler logic and the code of the segment is executed, but before the DMA interrupt removes any request.

```
int h[128];
char a[500];

void init(int* x, int n) {
  for(int i = 0; i < n; i++)
      x[i] = 0;
}

void hist(int *x, int *y, int n){
  for(int i = 0; i < n; i++) {
      y[i] = y[i] & 127;
      x[y[i]] += 1;
  }
}

void histogram() {
   init(h, 128);
   hist(h, a, 500);
}
```

(a) Original code.

```
void histogram() {
  // 3PT is initialized to allocate
  // h to h_obj with ID (O1)
  init(h_obj, 128)
  B1 = allocate_buffer(a_buf1, RW);
  B2 = allocate_buffer(a_buf2, RW);
  swap_buffer(B1, a, 100)
  dispatch();
  swap_buffer(B2, a+100, 100)
  end_segment();                          s^1
  hist(h_obj, a_buf1, 100)
  swap_buffer(B1, a+200, 100);
  end_segment();                          s^3
  hist(h_obj, a_buf2, 100)
  swap_buffer(B2, a+300, 100);
  end_segment();                          s^4
  hist(h_obj, a_buf1, 100)
  swap_buffer(B1, a+400, 100);
  end_segment();                          s^5
  hist(h_obj, a_buf2, 100)
  deallocate_buffer(B2);
  end_segment();                          s^8
  hist(h_obj, a_buf1, 100)
  deallocate(O1);
  deallocate_buffer(B1);
  wait();                                 s^9
}
```

(b) Segmented code.

Fig. 5: API usage example.

Since the first segment for the task s^1 uses array h, h has to be allocated in the SPM before s^1 starts. Accordingly, the 3PT is initialized with entry O1 as shown in Figure 5 corresponding to allocating h to h_obj with a write-only status. The OS processes this allocation in interval ⓪ before the segmented function starts. Note that O1 does not trigger a read request as the object is write-only. However, the code and other data, e.g. the stack, are transferred in this interval.

During s^1 execution in interval ①, h_obj is initialized and two read-write buffers B1 and B2 are allocated for array a. The first 100 bytes of a are swapped-in B1 and a read request B1↑ is added to the SWQ, then moved to the SDQ once `dispatch` function is called. After that, the next 100 bytes of a are swapped-in B2 and a load transfer B2↑ is added to the Wait

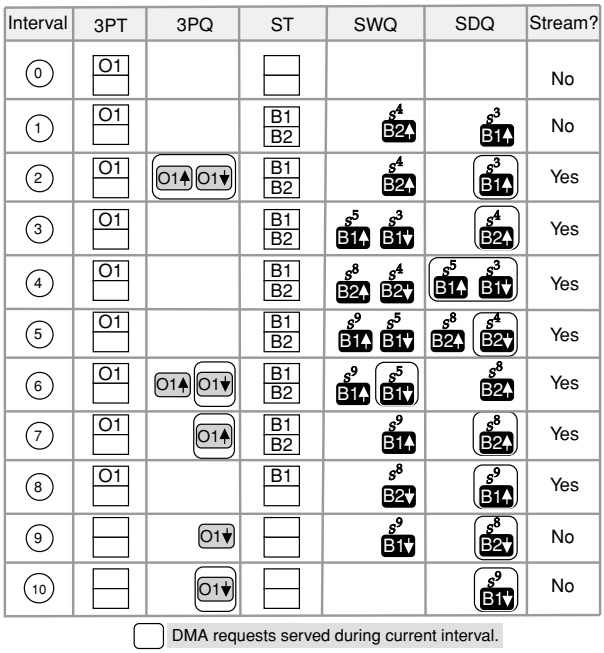


Fig. 6: SPM management example (s^x above DMA requests in SWQ/SDQ means data used in segment s^x).

Queue. Since s^1 is a terminal segment, a different task executes in interval ②. Hence, a write request O1↓ and read request O1↑ for O1 are added to the 3PQ by the scheduler. In this example, segment s^3 of the task is assumed to resume in interval ③. Therefore, both O1↓ and O1↑ are processed[6] during interval ② as well as B1↑. In interval ③, s^3 executes the histogram loop on the first 100 bytes of a from B1, then invokes a swap for B1 for the third 100 bytes of a. This adds two transfers in the SWQ to write back the current data B1↓ and then read the new data B1↑. The entry for B1 is modified to reflect the swap result. While s^3 is executing, B2 is filled with the second 100 bytes of a as B2↑ is processed from the SDQ. In interval ④, swapping B1 proceeds from the SDQ and a swap for B2 is added to the SWQ. At the beginning of ⑤, the OS schedules segment s^6 from another task to be executed in interval ⑥. Thus, s^4 is swapped-out of B2 in interval ⑤, but s^8 is not swapped-in and is kept in the SDQ. A swap for B1 is also added to the SWQ in interval ⑤. At the beginning of interval ⑥, write/read requests for O1 are added to the 3PQ. Then, B1↓ is processed from the SWQ, and O1↓ is processed from the 3PQ. The OS prepares the task in interval ⑦ to resume execution. So, B2↑ is processed from the SDQ along with O1↑ from the 3PQ. In interval ⑧, the task resumes executing s^8 while B1↑ is processed. Since s^8 is the last segment to use B2, B2 is deallocated and B2↓ is added to the SWQ. In interval ⑨, the segment stream is concluded with s^9 in which all the remaining objects and buffers are deallocated; and therefore the tables are cleared. The deallocation triggers B1↓ and O1↓ to copy back the modified data to main memory. These requests are processed in interval ⑩ as shown in Figure 5.

The OS identifies the execution mode (streaming or three-phase, as needed for Rule 3) based on the current mode and the API calls. That is, when B1 and B2 are allocated in the terminal segment s^1, the OS knows that it will switch to streaming mode in the next segment. The task remains in the streaming mode until all buffers are deallocated and the ST is empty. Note that the task ends with a call to the `wait` function instead of `end_segment` to inform the OS that this is the last segment of the current task activation; the OS then suspends the task until its next periodic activation.

VII. Schedulability Analysis and Program Segmentation

In this section, we discuss how to obtain a sufficient schedulability analysis for a set of streaming, conditional sporadic tasks scheduled on one processor according to fixed per-task priorities under the fixed-time DMA model; hence, without loss of generality assume that tasks in $\Gamma = \{\tau_1, \ldots, \tau_N\}$ are ordered by decreasing and distinct priorities. In particular, we show that we can employ the analysis in [16], which did not consider streaming (i.e. equivalently, all segments are terminal) but otherwise relied on the same scheduling assumptions, by simply changing the inter-segment blocking term.

The analysis in [16] computes a bound on the response time $R_i(P)$ of task under analysis τ_i from its arrival time until the time when its last segment starts executing, assuming that the task takes path P. The task is then schedulable if for all paths $P \in G_i$, it holds:

$$R_i(P) \leq D_i - P.end. \tag{1}$$

$R_i(P)$ can be computed based on response time iteration:

$$R_i(P) = P.L - P.end + \text{Inter}_i\big(R_i(P)\big) + B_i + (P.I - 1) \cdot l_i^{l\max}. \tag{2}$$

We next discuss the four terms in Equation 2. (i) Term $P.L - P.end$ captures the execution time of all segments of τ_i, excluding the last one. (ii) Term $\text{Inter}_i\big(R_i(P)\big)$ accounts for the interference of higher priority tasks, which is computed as:

$$\text{Inter}_i(t) = \sum_{j=1}^{i-1} \lceil t/T_j \rceil \cdot L_j^{max}, \tag{3}$$

where L_j^{max} represents the maximum length $P_j.L$ of any path P_j of τ_j. Note that the bound does not rely on the fact that successive segments of τ_j cannot run back-to-back; hence, it still applies to streaming execution. (iii) Term B_i represents the maximum blocking time suffered by the first segment of τ_i, or higher priority tasks in the busy interval. The analysis computes such bound conservatively as twice the maximum length of any interval executing either a lower priority segment, or no segment at all, which is:

$$l_i^{l\max} = \max(\Delta, \max_{j=i+1,N} \max_{s \in S_j} s.l)$$

Intuitively, two blocking intervals must be considered because the critical instant can happen just after the beginning of the first blocking interval, in which case the highest priority task must

[6]This case can be optimized in the implementation to avoid the read/write DMA transfers.

wait until the beginning of the second blocking interval to start being loaded. Again, the bound still applies whether consecutive segments of the same task execute back-to-back or not. (iv) The term $(P.I-1)\cdot l_i^{l\max}$, where in [16] $P.I$ denoted the total number of segments in P, represents further blocking suffered by each segment of τ_i except the last: without streaming, in the worst case an interval executing either a lower priority segment, or no segment at all, can be interleaved between any two segments of τ_i. The same bound applies to the streaming model, except that $P.I$ is now the number of terminal segments, as defined in Section IV: terminal segments (except the last segment) can still suffer blocking. However, that is not the case for streaming segments, since unless preempted by a higher priority task, a streaming segment will be followed by the next segment of the same task.

In summary, Equation 2 still applies to our new streaming model by properly defining $P.I$, which means that we can also reuse some key properties of the analysis proven in [16].

Definition 1. *Given two maximal paths P, P', we say that P' dominates (is worse than or equal to) P and write $P' \succeq P$ iff: $P'.L \geq P.L$ and $P'.I \geq P.I$ and $P'.end \leq P.end$. If neither $P' \succeq P$ nor $P \succeq P'$ holds, we say that the two paths are incomparable.*

Property 1. *Consider two paths P, P' with $P' \succeq P$. If Equation 1 holds for P', then it also holds for P.*

Since the $\succeq$ relation defines a partial order between maximal paths, based on Property 1 we can simply check Equation 1 over the Pareto frontier[7] of dominating paths of G_i, which we indicate as $G_i.C$, rather than all paths in G_i. For the example in Figure 2, we have $G_i.C = \{P, P''\}$, since $P \succeq P'$.

Property 2. *According to the analysis: (A) the schedulability of task τ_i depends on the maximum length $l_i^{l\max}$ of any segment of lower priority tasks $\tau_i+1, \ldots \tau_N$, but not on any other parameter of those tasks; (B) if τ_i is schedulable for a value l of $l_i^{l\max}$, then it is also schedulable for any other value $l' \leq l$.*

Based on Property 2, Algorithm 2 (Algorithm 4 in [16]) describes the combined schedulability analysis and program segmentation for task set Γ. The algorithm recursively explores the task set from the highest to the lowest priority task. At each step i, the value of $l^{\max}$ represents the maximum length of segments of tasks $\tau_i, ..., \tau_N$ under which the previously explored higher priority tasks $\tau_1, ..., \tau_{i-1}$ are schedulable. The algorithm first uses function SEGMENTTASK$(\tau_i, l^{\max})$ to segment task τ_i based on maximum segment length $l^{\max}$. The function invokes our compiler framework, which analyzes the code of τ_i and returns a set of candidate DAGs $\mathcal{G}_i$, representing different possible segmentations for the task. The schedulability analysis is then used to determine the maximum length $\overline{l_i^{l\max}}$ of lower priority segments that results in τ_i being schedulable[8], which is akin to computing the maximum blocking tolerance of a task in non-preemptive and limited-preemptive systems [38]. Finally, the algorithm moves to task τ_{i+1}, until the schedulability of the lowest priority task τ_N can be assessed. Note that at each step, $l^{\max}$ is updated as the minimum of its previous value and the computed $\overline{l_i^{l\max}}$; this ensures that segments in τ_{i+1} meet the length requirement of all higher priority tasks, and thus $\{\tau_1, \ldots, \tau_i\}$ remain schedulable.

Algorithm 2 Task Set Segmentation

Require: Task set Γ, source code for each task in Γ
1: SEGMENTTASKSET(Γ, $i, +\infty, \varnothing$)
2: Terminate with FAILURE
3: **function** SEGMENTTASKSET($\Gamma, i, l^{\max}, \{G_1, \ldots, G_{i-1}\}$)
4: Generate $\mathcal{G}_i$ = SEGMENTTASK($\tau_i, l^{\max}$)
5: **if** $i < N$ **then**
6: **for all** $G_i \in \mathcal{G}_i$ **do**
7: Compute the maximum value $\overline{l_i^{l\max}}$ of $l_i^{l\max}$ based on analysis
8: SEGMENTTASKSET(Γ, $i+1$, $\min\left(l^{\max}, \overline{l_i^{l\max}}\right)$, $\{G_1, \ldots, G_i\}$)
9: **else**
10: **for all** $G_N \in \mathcal{G}_i$ **do**
11: If analysis returns schedulable on $\{G_1, \ldots, G_N\}$, terminate with SUCCESS

Assuming that SEGMENTTASK$(\tau_i, l^{\max})$ returns all valid DAGs for τ_i with maximum segment length $l^{\max}$, in [16] Algorithm 2 is proven to be optimal with respect to the introduced schedulability analysis, in the sense that it will find a schedulable task set segmentation if any exists. However, an obvious downside of Algorithm 2 is that it tests all DAGs in $\mathcal{G}_i$. Hence, the algorithm is exponential in the number of tasks. Luckily, we can rely on one last property of the analysis.

Definition 2. *Given two segment DAGs G, G', we say that G' dominates (is worse than or equal to) G and write $G' \succeq G$ iff: $\forall P \in G.C, \exists P' \in G'.C : P' \succeq P$. If neither $G' \succeq G$ nor $G \succeq G'$ holds, the two DAGs are incomparable.*

Property 3. *Consider two DAGs G_j, G'_j for task τ_j where $1 \leq j \leq i$ and $G'_j \succeq G_j$. If τ_i is schedulable for G'_j according to the analysis, then it is also schedulable for G_j.*

Based on Property 3, if $\mathcal{G}_i$ contains two DAGs G_i, G'_i with $G'_i \succeq G_i$, then we can remove the dominating DAG G'_i from $\mathcal{G}_i$ without affecting the optimality of Algorithm 2. In practice, as we show in Section VIII, this allows us to drastically cut the number of DAGs, such that the algorithm can be applied to non-trivial number of tasks.

A. Program Segmentation

We next discuss how function SEGMENTTASK$(\tau_i, l^{\max})$ generates a set of candidate DAGs $\mathcal{G}_i$. Due to space limitations, we are unable to get into the details of the compiler framework; instead, here we summarize the behavior of the algorithm and the changes we performed compared to [16]. A detailed algorithm description is provided in [36].

The tool begins by creating a tree-based representation of the program based on its CFG. Nodes in the tree, called program regions, represent either basic blocks, conditionals, loops, or function calls. Program segmentation is equivalent to assigning regions to segments while maintaining the structure of the

[7]Given a partial order over a set of distinct elements, the Pareto frontier is the subset of elements that are not dominated by any other element.

[8]While this could be performed by logarithmic search over the value of $l_i^{\max}$, [16] employs a faster approach based on the concept of scheduling points introduced in [37].

program, and respecting constraints on the segment length $l^{\max}$ and its maximum footprint. Loop splitting and tiling are used to fit large data structures in the SPM, and break loops into shorter segment lengths. We modified our framework from [16] in two main ways. First, the tiling algorithm now returns a sequence of streaming segments. For a streaming loop with N tiles/segments, the first $N-1$ segments are streaming while the last segment is terminal. The data used by all segments are allocated before the loop stream as objects while data that are sub-arrays that is dependent on the tile are streamed using buffers and swapped during the loop execution. Second, we generate and account for the time of all API calls introduced in Section VI.

The segmentation algorithm can generate a large number of candidate solutions, especially when exploring tiling options. For this reason, simply pruning the set of resulting DAGs based on Definition 2 does not result in acceptable run-time. Instead, in [16] we introduced several pruning conditions that allow cutting dominating solutions while exploring the region tree. We further showed that the introduced conditions preserve the optimality of the algorithm, as the pruned solutions can only lead to dominating DAGs. To maintain a reasonable run-time for the algorithm, here we employ the same pruning conditions. However, the optimality proof in [16] relies on the assumption of a constant per-segment overhead. As we now account precisely for the overhead of different API calls in each segment, we cannot trivially extend the proof. Hence, the approach presented in this paper should be considered a heuristic, albeit in practice we expect it to miss a schedulable solution only in pathological corner cases.

VIII. Evaluation

In this section, we present the evaluation of the proposed OS-level API and the streaming execution model.

A. API Implementation

We have implemented the API and OS-related techniques described in Section VI (queues, tables, interrupt handler, and data structures) in Erika Enterprise RTOS version 3 release 55.

We then compiled and measured the execution times and memory footprint of the API functions using the Xilinx UltraScale+ ZCU102 MPSoC platform [39]. The platform features a four-core 1.2 Ghz ARM Cortex-A53 processor, each core having its own local instruction and data caches of 32 KB. There is a Last-Level Cache (LLC) of 1 MB shared by all the A53 cores. The MPSoC also provides a DDR4-2666 main memory controller of 64-bit data width connected to a 4 GB DDR4 memory module and a programmable logic. It is important to highlight that the ARM A53 processor is used by several embedded platforms, such as Raspberry Pi, NXP MINISASTOCSI, Hikey-96Boards, and Samsung S5P6818. We rely on the Jailhouse hypervisor [40] to provide hardware isolation and cache partitioning for individual Erika instances running on each A53 core [3], [41], and to perform code relocation (since Erika does not have virtual memory) [3].

We compiled the segmented code of the tasks using LLVM after adding the API calls. The generated assembly code of the tasks is then linked together with Erika using the gcc compiler version 7.2.1 for ARM64 architecture with the `-Os` flag. Table III shows the memory footprint (in bytes) for each API function. In addition to the API functions, we also wrote auxiliary functions to handle the queues (like remove the head of the queue and return a queue element). Those auxiliary functions consume 288 bytes of code and 8 bytes of data. The DMA interrupt handler is responsible for removing DMA requests from the appropriate queues when the DMA finishes a transfer request. In total, the API implementation has 2944 bytes. The actual memory overhead added by the function to each task depends on the calls made in each segment.

TABLE III: Code size (in bytes) and average (with standard deviation), worst-case, and best-case execution time (in ns) of the API implementation.

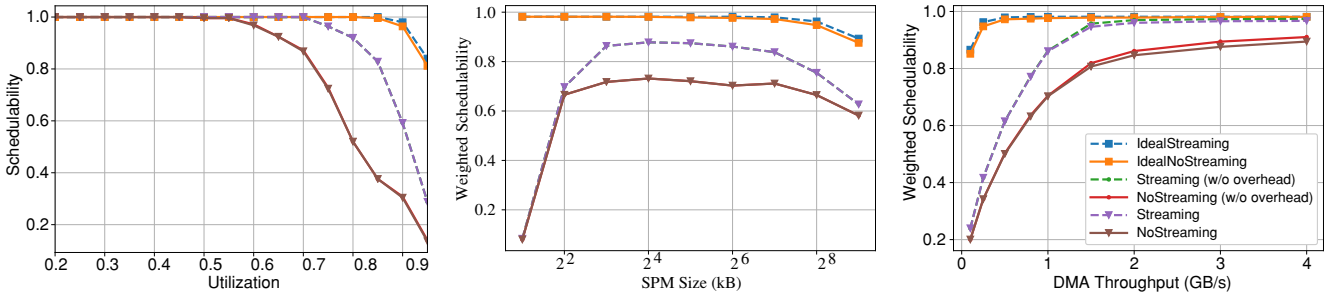
	Code (bytes)	AVG	STD	WCET	BCET	Var. Window
`allocate_buffer`	76	46.24	29.02	949	40	0.96
`dispatch`	112	197.32	18.14	717	191	0.73
`DMA_int_handler`	136	82.81	29.00	989	80	0.92
`allocate`	268	255.67	31.85	1252	252	0.80
`end_segment`	284	79.11	47.24	1565	70	0.96
`deallocate`	300	182.32	93.07	717	50	0.93
`allocate2d`	308	262.83	20.84	919	262	0.71
`deallocate_buffer`	320	256.18	140.52	646	60	0.91
`swap_buffer`	400	357.17	121.50	1595	262	0.84
`swap2d_buffer`	444	397.62	131.36	1040	282	0.73

Table III also shows the average (with standard deviation), worst-case (WCET), and best-case execution times (BCET) in nanoseconds of each API function and the variability window, which is a metric of predictability and is computed as $(WCET - BCET)/WCET$. These numbers were obtained by running each API function for 1000 times on the ZCU102 platform. In each repetition, we vary the memory attribute (read-only, read-write, and write-only) to ensure that all cases are analyzed. Time was measured using a timer from Erika (`osEE_aarch64_gtimer_get_ticks` function). Since the Xilinx ZCU102 platform does not provide an SPM, we implemented it as a Block RAM (BRAM) in the programmable logic, as done in [3]. As we will show in the schedulability analysis, the overhead of the API function is minimal for the system performance.

Table IV presents the memory footprint in bytes for the tables and queues, as described in Section VI. Since Erika does not support dynamic memory allocation, the number of DMA requests and list elements must be statically defined. Here we consider a maximum of 5 DMA requests and list elements per task (number of instances in Table VI). Similarly, we consider that each task can have at most 5 streaming buffers (one streaming table entry per each buffer). The total memory consumption for this configuration is 744 bytes. Note that the number of DMA requests and the number of entries in the tables are known for each task at compilation time based on the segmentation, hence they can be statically allocated as the maximum over all segments.

B. Schedulability

We evaluate the proposed streaming execution model in terms of schedulability and compare it against the three-phase

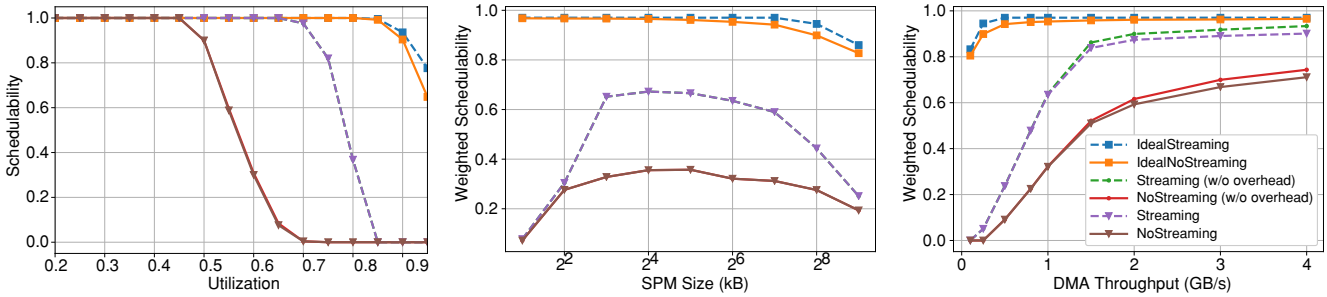


(a) Schedulability vs Utilization (SPM size = 64 kB / DMA throughput = 1.0 GB/s)

(b) Weighted Schedulability vs SPM size (DMA throughput = 1.0 GB/s)

(c) Weighted Schedulability vs DMA throughput (SPM size = 64 kB)

Fig. 7: Case 1: uniformly generated random task sets.



(a) Schedulability vs Utilization (SPM size = 64 kB / DMA throughput = 1.0 GB/s)

(b) Weighted Schedulability vs SPM size (DMA throughput = 1.0 GB/s)

(c) Weighted Schedulability vs DMA throughput (SPM size = 64 kB)

Fig. 8: Case 2: heavy load task sets with 'disparity' using 50% of the system utilization.

TABLE IV: Memory footprint per task (in bytes) of queues and tables.

	Individual Size	Number of Instances	Total (bytes)
Three Phase Queue (3PQ)	16	1	16
Stream Waiting Queue(SWQ)	16	1	16
Stream Dispatch Queue (SDQ)	16	1	16
Auxiliary variables	4	2	8
DMA request	56	5	280
Free list elements	24	5	120
Three Phase Table (3PT)	48	1	48
Streaming Table (ST)	48	5	240

model without streaming in [16]. In both models, we use an ideal program segmentation as a reference for the best possible performance. That is, the ideal case assumes that there is no restrictions on the SPM size and that the program code can be segmented at any arbitrary point without any overhead. To assess the impact of the API implementation, we also report results assuming that the API introduces no overhead. We do not provide results using the greedy segmentation approach proposed in [28], since in [16] we already found that most tested task sets are unschedulable. Unfortunately, we are not able to provide an analytical comparison against other scheduling approaches for the three-phase model, since they do not fit the key assumptions of our model, namely, a multitasking system where sequential tasks are segmented. In particular, the distributed analysis employed in [30] cannot be easily extended to handle multitasking systems[9]. The static scheduling approaches in [12], [28] have been designed for a single parallel task; when applied to a single sequential task per core, they reduce to our scheme. The remaining CPU-oriented approaches do not support segmentation.

TABLE V: Evaluation Benchmarks.

Benchmark	Suite	LOC	Data (bytes)	WCET (ns)
adpcm_dec	TACLeBench	476	404	176947
cjpeg_transupp	TACLeBench	474	3459	12083696011
fft	TACLeBench	173	24572	89540809
compress	UTDSP	131	136448	168984645
lpc	UTDSP	249	8744	233390
spectral	UTDSP	340	4584	109074793
disparity	CortexSuite	87	2704641	339361377

A set of real benchmarks are used in the evaluation from multiple benchmark suites, UTDSP [42], TACLeBench [43], and CortexSuite [44]. Table V shows the data footprint and the WCET (in ns) for each of the employed benchmarks. Since program segmentation requires WCET for each of the program regions, we used static analysis as in [16] using a simple model and compared the obtained numbers with the execution times on the platform to estimate the WCET for each region in the program. We use the benchmarks to generate synthetic task sets for two cases:

[9]This would require modeling the initial memory phase as a task that is simultaneously executed on two different resources, main memory and the processor; to the best of our knowledge, no existing analysis can support this.

- For case 1, we generate a random number of tasks between 5 and 15. The utilization of each task is uniformly generated [45] given a system utilization and the number of tasks. Then the period is assigned to each task with a minimum period of 10 ms[10].
- For case 2, we are interested in a system that can benefit the most from the streaming model. Therefore, we force the benchmark 'disparity' to be a heavy load task with 50% of the system utilization, while generating 3 to 5 tasks with the other 50% of the system utilization. That is, if the system utilization is 0.5, then 'disparity' has 0.25 utilization while the other tasks have 0.25 utilization. We chose 'disparity' since it is an image processing application that processes 256x256 image with a large data footprint (2.6 MB). Hence, it is forced to be segmented using loop tiling due to the limited SPM size.

For both cases, we conduct the schedulability test with system utilization between 0.2 and 0.95 using 250 task sets per utilization level. To depict the effect of memory size on the system performance, we vary the SPM size between 2 kB and 512 kB. Similarly, we vary the DMA throughput available to the processor under analysis between 0.1 GB/s to 4.0 GB/s; note that given our TDMA arbitration assumption, the required memory bandwidth is equal to the DMA transfer speed multiplied by the number of processors. For a four-core system, this puts the maximum considered bandwidth at 16 GB/s, which is still lower than the theoretical memory throughput of 21 GB/s offered by the ZCU102 MPSoC and similar platforms.

We present results in terms of the system schedulability and weighted schedulability metric [46]. The *system schedulability* is the proportion of the schedulable task sets out of the total tested task sets. We define the *weighted schedulability metric* μ of a system as [46]:

$$\mu = \frac{\sum_u sched(u) \cdot u}{\sum_u u}$$

In this definition, $sched(u)$ is the system schedulability for system utilization u. This metric allows us to depict the performance of the schedulability algorithm on a two-dimensional plot while varying a system parameter (SPM size or DMA throughput), without restricting the evaluation to a fixed utilization value. For all cases, the running time of the program segmentation algorithms varies from a few seconds to a maximum of a minute. Running the combined segmentation and schedulability algorithm for a task set of 15 tasks takes an average of a minute to a maximum of a few minutes.

For case 1, Figure 7a shows the system schedulability (y-axis) vs utilization (x-axis) for SPM size of 64 kB and DMA throughput of 1.0 GB/s. The streaming execution model outperforms the non-streaming three-phase model as the system utilization increases. This is consistent with Figure 7b that shows the weighted schedulability (y-axis) vs the SPM size (x-axis) with DMA throughput of 1.0 GB/s, and Figure 7c that shows the weighted schedulability (y-axis) vs the DMA throughput (x-axis) for SPM size 64 kB. The streaming model results in an improvement up to 16% for the weighted schedulability compared the non streaming model in Figure 7b and Figure 7c.

In case 2, Figure 8a depicts the system schedulability vs utilization for SPM size 64 kB and DMA throughput 1.0 GB/s. We see an even more significant improvement in the system schedulability using the streaming model. Figure 8b and Figure 8c show the weighted schedulability vs SPM size with DMA throughput of 1.0 GB/s, and weighted schedulability vs DMA throughput for SPM size of 64 kB. An improvement up to 33% is achieved for the weighted schedulability compared the non streaming model in Figure 8b and Figure 8c.

In Figure 7b and Figure 8b, the schedulability of the system improves at the beginning as the SPM size increases since larger SPM sizes cause less segmentation and hence less overhead and less blocking from lower priority tasks. Then, the performance peaks and starts to decline as large SPM sizes imply large DMA times due to the fixed-time assumption used in the analysis; this causes under-utilization of the segments with smaller execution time. On the other hand, increasing the DMA throughput as in Figure 7c and Figure 8c improves the schedulability as the impact of the DMA time on the response time of the tasks is reduced. We can see that the performance of the streaming model saturates close to DMA throughput of 4.0 GB/s as it approaches the ideal performance. Finally, comparing the results with and without the API overhead shows that the effect of the overhead on the schedulability is minimal and becomes only relevant at high DMA throughput when segment length is shorter.

IX. Conclusion and Future Work

Managing hardware resource contention is key to unleash the performance of modern MPSoCs in real-time systems. The three-phase execution model has proven effective in increasing predictability of memory accesses while hiding access latency through the use of a DMA engine. We feel that the biggest challenge going forward is in making the model more applicable by relaxing program constraints and providing automatic program transformation algorithms. To this end, in this paper we have proposed a new streaming model for three-phase tasks that allows consecutive segments of the same task to execute sequentially, improving schedulability by reducing blocking time. Furthermore, we have described an automatic compilation framework that adds all required OS calls to the programs, and evaluated the approach on a commercial RTOS and realistic benchmarks. On the theoretical side, schedulability could be further improved by making more complex assumptions on DMA time (variable-time model). On the implementation side, we plan to further extend the API to handle different I/O device types (synchronous and asynchronous) and hardware accelerators in a way compatible with the three-phase model.

[10]We argue that tasks with periods lower than 10 ms are typically control tasks with very small footprint and execution time; hence, we simply propose to pin them to SPM and execute them as preempting highest priority tasks, rather than three-phase tasks.

ACKNOWLEDGMENT

This work has been supported in part by ONR N00014-17-1-2783, by NSF grant number CNS 18-15891, by the NSERC and by CMC Microsystems. Marco Caccamo was supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsors.

REFERENCES

[1] A. Burns and R. I. Davis, "A survey of research into mixed criticality systems," *ACM Comput. Surv.*, vol. 50, no. 6, pp. 82:1–82:37, Nov. 2017.

[2] T. Mück, A. A. Fröhlich, G. Gracioli, A. M. Rahmani, J. G. Reis, and N. Dutt, "Chips-ahoy: A predictable holistic cyber-physical hypervisor for mpsocs," in *Proc. of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, ser. SAMOS '18. NY, USA: ACM, 2018, pp. 73–80.

[3] G. Gracioli, R. Tabish, R. Mancuso, R. Pellizzoni, and M. Caccamo, "Designing mixed criticality applications on modern heterogeneous mpsoc platforms," in *2019 31th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2019, pp. 1–23.

[4] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding memory interference delay in cots-based multi-core systems," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2014, pp. 145–154.

[5] M. Hassan and R. Pellizzoni, "Bounding dram interference in cots heterogeneous mpsocs for mixed criticality systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2323–2336, Nov 2018.

[6] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for COTS-based embedded systems," in *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, ser. RTAS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 269–279.

[7] S. Wasly and R. Pellizzoni, "Hiding memory latency using fixed priority scheduling," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2014, pp. 75–86.

[8] P. Burgio, A. Marongiu, P. Valente, and M. Bertogna, "A memory-centric approach to enable timing-predictability within embedded many-core accelerators," in *2015 CSI Symposium on Real-Time and Embedded Systems and Technologies (RTEST)*. IEEE, 2015.

[9] N. Capodieci, R. Cavicchioli, P. Valente, and M. Bertogna, "SiGAMMA: Server based integrated GPU Arbitration Mechanism for Memory Accesses," in *Proceedings of the 25th International Conference on Real-Time Networks and Systems - RTNS '17*. New York, New York, USA: ACM Press, 2017.

[10] B. Forsberg, L. Benini, and A. Marongiu, "HePREM: Enabling predictable GPU execution on heterogeneous SoC," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018.

[11] G. Durrieu, M. Faugère, S. Girbal, D. G. Pérez, C. Pagetti, and W. Puffitsch, "Predictable Flight Management System Implementation on a Multicore Processor," *Embedded Real Time Software (ERTS'14)*, 2 2014.

[12] B. Rouxel, S. Skalistis, S. Derrien, and I. Puaut, "Hiding communication delays in contention-free execution for spm-based multi-core architectures," in *2019 31th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2019, pp. 1–23.

[13] M. Becker, D. Dasari, B. Nicolic, B. Akesson, V. Nelis, and T. Nolte, "Contention-Free Execution of Automotive Applications on a Clustered Many-Core Platform," in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2016.

[14] B. Forsberg, A. Marongiu, and L. Benini, "GPUguard: Towards supporting a predictable execution model for heterogeneous SoC," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017.

[15] R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo, "A Real-Time Scratchpad-Centric OS for Multi-Core Embedded Systems," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2016.

[16] M. R. Soliman and R. Pellizzoni, "PREM-based Optimal Task Segmentation under Fixed Priority Scheduling," in *2019 31th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2019, pp. 1–23.

[17] A. Alhammad and R. Pellizzoni, "Schedulability analysis of global memory-predictable scheduling," in *Proceedings of the 14th International Conference on Embedded Software - EMSOFT '14*. New York, New York, USA: ACM Press, 2014.

[18] A. Alhammad, S. Wasly, and R. Pellizzoni, "Memory efficient global scheduling of real-time tasks," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2015, pp. 285–296.

[19] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. Buttazzo, "Memory-processor co-scheduling in fixed priority systems," in *Proceedings of the 23rd International Conference on Real Time and Networks Systems - RTNS '15*. New York, New York, USA: ACM Press, 2015.

[20] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo, "Memory-centric scheduling for multicore hard real-time systems," *Real-Time Systems*, vol. 48, no. 6, 2012.

[21] G. Yao, R. Pellizzoni, S. Bak, H. Yun, and M. Caccamo, "Global Real-Time Memory-Centric Scheduling for Multicore Systems," *IEEE Transactions on Computers*, vol. 65, no. 9, 2016.

[22] P. Gai, E. Bini, G. Lipari, M. D. Natale, and L. Abeni, "Architecture for a portable open source real time kernel environment," in *In Proceedings of the Second Real-Time Linux Workshop and Hand's on Real-Time Linux Tutorial*, 2000.

[23] E. Srl, "Erika enterprise RTOS v3," May 2019, online; accessed 27 May 2019. [Online]. Available: http://www.erika-enterprise.com/

[24] S. Wasly and R. Pellizzoni, "A Dynamic Scratchpad Memory Unit for Predictable Real-Time Embedded Systems," in *2013 25th Euromicro Conference on Real-Time Systems*. IEEE, 2013.

[25] A. Alhammad and R. Pellizzoni, "Time-predictable execution of multithreaded applications on multicore systems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014*. New Jersey: IEEE Conference Publications, 2014.

[26] R. Tabish, R. Mancuso, S. Wasly, S. S. Phatak, R. Pellizzoni, and M. Caccamo, "A Reliable and Predictable Scratchpad-centric OS for Multi-core Embedded Systems," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2017.

[27] R. Mancuso, R. Dudko, and M. Caccamo, "Light-PREM: Automated software refactoring for predictable execution on COTS embedded systems," in *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE, 2014.

[28] J. Matějka, B. Forsberg, M. Sojka, Z. Hanzálek, L. Benini, and A. Marongiu, "Combining PREM compilation and ILP scheduling for high-performance and predictable MPSoC execution," in *Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores - PMAM'18*. New York, New York, USA: ACM Press, 2018.

[29] B. Rouxel, S. Derrien, and I. Puaut, "Tightening Contention Delays While Scheduling Parallel Applications on Multi-core Architectures," *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 5s, 2017.

[30] J. M. Rivas, J. Goossens, X. Poczekajlo, and A. Paolillo, "Implementation of Memory Centric Scheduling for COTS Multi-Core Real-Time Systems," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, vol. 133, 2019.

[31] M. Soliman and R. Pellizzoni, "WCET-driven dynamic data scratchpad management with compiler-directed prefetching," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, 2017.

[32] Y. Yang, M. Wang, Z. Shao, and M. Guo, "Dynamic scratch-pad memory management with data pipelining for embedded systems," in *Computational Science and Engineering, 2009. CSE '09. International Conference on*, vol. 2, Aug 2009, pp. 358–365.

[33] B. Flachs, S. Asano, S. Dhong, H. P. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H. J. Oh, S. M. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, N. Yano, D. A. Brokenshire, M. Peyravian, V. To, and E. Iwata, "The microarchitecture of the synergistic processor for a cell processor," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 1, 2006.

[34] S. Saidi, P. Tendulkar, T. Lepley, and O. Maler, "Optimizing explicit data transfers for data parallel applications on the cell architecture," *ACM Transactions on Architecture and Code Optimization*, vol. 8, no. 4, 2012.

[35] E. Hammami and Y. Slama, "An overview on loop tiling techniques for code generation," in *Proceedings of IEEE/ACS International Conference on Computer Systems and Applications, AICCSA*, vol. 2017-October, 2018.

[36] M. R. Soliman, "Automated compilation framework for scratchpad-based real-time systems," Ph.D. dissertation, University of Waterloo, 2019. [Online]. Available: http://hdl.handle.net/10012/14835
[37] E. Bini and G. Buttazzo, "Schedulability analysis of periodic fixed priority systems," *IEEE Transactions on Computers*, vol. 53, no. 11, 2004.
[38] G. Yao, G. Buttazzo, and M. Bertogna, "Bounding the maximum length of non-preemptive regions under fixed priority scheduling," in *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2009.
[39] Xilinx, "Zynq UltraScale+ Device - technical reference manual." [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf
[40] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer, "Look mum, no VM exits! (almost)," in *Proc. of the 13th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2017)*, 2017, pp. 13–18.
[41] 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 Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Montreal, Canada, April 2019.
[42] "UTDSP Benchmark Suite." [Online]. Available: http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html
[43] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener, "TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research," *DROPS-IDN/6895*, vol. 55, 2016.
[44] S. Thomas, C. Gohkale, E. Tanuwidjaja, T. Chong, D. Lau, S. Garcia, and M. Bedford Taylor, "CortexSuite: A synthetic brain benchmark suite," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2014.
[45] E. Bini and G. C. Buttazzo, "Measuring the Performance of Schedulability Tests," *Real-Time Systems*, vol. 30, no. 1-2, 2005.
[46] A. Bastoni, B. B. Brandenburg, and J. H. Anderson, "Cache-Related Preemption and Migration Delays: Empirical Approximation and Impact on Schedulability," in *Proc. of the OSPERT '10*, Brussels, Belgium, Jul 2010.