Optimal Task Segmentation under Fixed Priority Scheduling

Muhammad R. Soliman and Rodolfo Pellizzoni
University of Waterloo, Canada. {mrefaat, rpellizz}@uwaterloo.ca

Abstract— **Recently, a large number of works have discussed scheduling tasks consisting of a sequence of memory phases, where code and data is moved between main memory and local memory, and computation phases, where the task executes based on the content of local memory only; the key idea is to prevent main memory contention by scheduling the memory phase of one task in parallel with computation phases of tasks running on other cores. This paper provides two main contributions: (1) we present a compiler-level tool, based on the LLVM intermediate representation, that automatically converts a program into a conditional sequence of segments comprising memory and computation phases; (2) we propose an algorithm to find optimal segmentation decisions for a task set scheduled according to a fixed-priority partitioned scheme. Our evaluation shows that the proposed framework can be feasibly applied to realistic programs, and vastly overperforms a baseline greedy segmentation approach.**

I. Introduction

Multi-Processor Systems-on-a-Chip (MPSoCs) are becoming increasingly popular in the real-time and embedded system community. MPSoCs are characterized by the presence of shared memory resources. In particular, a single main memory shared by all processing elements on the chip can constitute a significant performance bottleneck. Even worse, hardware arbitration schemes used in Commercial-Off-The-Shelf (COTS) systems are optimized for average-case performance, resulting in extremely high worst-case latency in the presence of contention for memory access among multiple processors [16], [17], [29].

Hence, there is a significant interest in the real-time community in controlling the pattern of accesses in memory to avoid worst-case scenarios. This can be difficult in cache-based systems, where main memory accesses are generated by misses in last level cache, as the precise pattern of cache hits and misses is hard to predict. The PRedictable Execution Model (PREM) first proposed in [23] attempts to solve this issue by dividing the execution of each software task in two different parts: memory phases where the data and instructions required by the task are loaded from main memory into local memory (cache or scratchpad), and computation phases where a processor executes the task based on the content of local memory only. Since the task does not need to access main memory during its computation phase, other processors are free to do so without suffering contention.

Based on this core idea, successive works [2]–[5], [8]–[10], [12], [20]–[22], [24], [27], [31]–[34] have proposed a variety of contentionless approaches [1] targeting different scheduling schemes (preemptive vs non-preemptive, partitioned vs global) and platforms (general purpose processors vs GPU). However, the key problem of how to compile a program to execute based on PREM has received significantly less attention. In general, the following steps are required: (1) determine the data used by the program; (2) add instructions to create memory phases; (3) and possibly segment the program into multiple parts, so that the data and code of each part can fit in local memory. Due to the complexities inherent in each step, we strongly believe that an automated tool is required to remove the burden from the programmer.

The main contribution of this paper is a framework for automatically generating PREM-compatible code for sequential programs running on a general purpose processor; it is largely agnostic to the programming language being used since it operates on the intermediate representation of the LLVM compiler infrastructure [18]. In particular, we propose a set of program transformation constraints that allow us to convert a task into a conditional sequence of PREM segments. We use a region-based approach to simplify segment creation, in conjunction with loop splitting and tiling transformations [15] to split large loops into multiple segments. Based on the proposed framework, we then derive a task segmentation algorithm that is optimal for PREM platforms with fixed-length memory phases [27], in the sense that we prove that our algorithm produces the best possible conditional segments for a given task. Based on the proposed framework, we then derive a task segmentation algorithms that enumerates the best possible conditional segments for a given task on a platform with fixed-length memory phases [27]. Furthermore, for the case of fixed-priority partitioned scheduling, we show that applying the algorithm to each task in priority order leads to a solution that is optimal for the task set.

The rest of the paper is organized as follows. Section II summarizes existing research based on PREM. Section III introduces our new conditional PREM model and extends the existing schedulability analysis to cover such model. Section IV describes our employed compilation framework, and our program segmentation algorithm based on such framework, while Section V then shows how to obtain an optimal segmentation for a given task set. Section VI compares our optimal segmentation approach versus both a previous greedy approach, and a simple heuristic, using task set parameters extracted from real programs. Finally, we conclude in Section VII.

[1]Note that the model we are discussing is also referred to as three-phase model or acquisition-execution-replication model in related work.

II. Background and Related Work

We consider a MPSoC platform comprising a set of possibly heterogeneous processors. [2]. Each processor has a fast private local memory in the form of a last level cache [3] or ScratchPad Memory (SPM); all processors share the same main memory. As discussed in Section I, the goal of PREM is to create a contentionless memory schedule. While the seminal work in [23] first proposed to split the execution of each application into a memory and a computation phase, the approach has been refined in successive works [3], [32] into a three-phase model. Here, two memory phases are considered: an acquisition (or load) phase that copies data and instructions from main memory into local memory, and a replication (or unload) phase that copies modified data back to main memory. While the computation phase is always executed on a processor, memory phases can be either executed on the processor itself [2], [3], [5], [9], [10], [21]–[24], [33], [34], or on another hardware component [12], [13], such as a programmable DMA module [4], [8], [27], [32]. In all cases, memory phases are scheduled such that a single memory phase is executed at any one time in the entire system.

When the data used by a program is small and deterministic, the task can comprise a single sequence of load-computation-unload phases. However, the code and data of the program might be too large to fit in one partition of local memory. which we find common for applications such as image/video processing and deep neural networks. Second, it might be difficult to predict the data accessed by a job before it starts executing, as data accesses can be dependent on program inputs. To address such issue, the works in [9], [21], [23], [32] split a task into a sequence of PREM segments, where each segment has its own memory and computation phases and is executed non-preemptively.

A. Memory and Processor Schedule

The memory scheduling algorithm is different among related work, based on their specific goals and system assumption. Approaches targeted at multitasking systems optimize task execution by overlapping the computation of the current job with the memory phase for the next job to be scheduled on that processor. In essence, one can pipeline computation and memory phases using a double-buffering technique [12], [13], [27], [32], at the cost of halving the available local memory space. As an example, we detail the approach in [27], [32], which has been designed to schedule a set of fixed-priority, partitioned sporadic tasks, and fully implemented on an automotive COTS platform. The local memory of each processor is divided into two equal size partitions. Memory phases are executed by a dedicated DMA component using a TDMA memory schedule with fixed time slots; the size of each slot is sufficient to either load or unload the entirety of one partition. Figure 1 shows an example schedule on one processor; the task under analysis (u.a.)

[2]A processor can either be a general purpose core, or in the case of SIMD machine such as a GPU, a cluster of cores.

[3]A shared last level cache can still be used by partitioning it among the processors [19], [30].

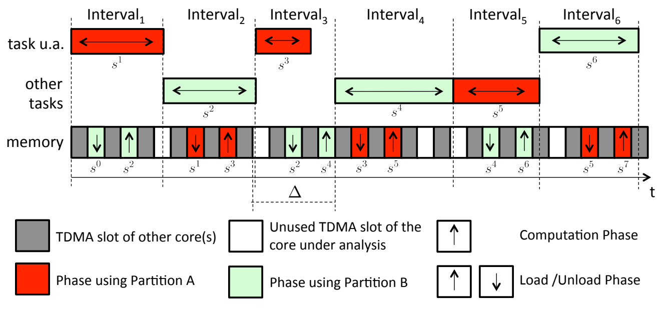


Fig. 1: Example: TDMA memory schedule with M = 2 cores.

consists 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. Segments are scheduled non-preemptively. During each interval, a segment of a job (ex: s^2 in Interval$_2$) computes using data and instruction 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 the other partition. Note that the length of each scheduling interval is the maximum of the computation time for the corresponding segment, and the time required for the load and unload operations. In the figure, $Interval_3$ is bounded by the memory time, while all other intervals are bounded by the computation time of the segment. Let M be the number of cores, and σ the size of each TDMA slot. Then as proven in [27], the worst-case memory time is equal to $\sigma \cdot (2M + 1)$: 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. To abstract from the details of the memory schedule, in the rest of the paper we assume a given bound Δ on the memory time for any interval. Hence, the length of an interval is the maximum of Δ and the computation time of the job in that interval. Finally, note that two segments of the same task cannot run back-to-back: in general, the data required by a segment cannot be determined until the previous segment completes; furthermore, to load a segment we might need to first evict some data and code of the previous one. For both reasons, the computation phase of a segment and the memory phase of the next one cannot be executed in parallel. To avoid idling the processor while a task loads its next segment, one or more segments of (possibly lower priority) other tasks are instead scheduled.

A downside of the described approach is that a high priority job can suffer blocking by a low priority job due to the non-preemptive interval schedule. The works in [22], [33], [34] adopt preemptive scheduling, but this requires a number of local memory partitions equal to the number of tasks: otherwise, a memory phase could be “wasted” by loading a job that is immediately preempted by a higher priority one. Given that local memory is typically a limited resource, we will not consider such fully-preemptive approaches.

B. Program Transformation

We next discuss how a program can be transformed to be PREM-compliant. Most single-segment works do not require program transformation; instead, the entire memory region allocated by the OS to the program is loaded in local memory [5], [8], [27], [32]. The seminal work in [23] introduces a set of macros, which the programmer could add to the program to both segment it, and mark data structures to be loaded / unloaded. Our experience with programs of even medium complexity is that this places an undue burden on the programmer, and it is likely to lead to a sub-optimal transformation. The authors of [12], [13] discuss a compiler-based approach to transform a GPU kernel. The approach focuses on generating code for the memory phase. On the other hand, our focus in this paper is how to automate data usage analysis and task segmentation for sequential programs running on a general purpose processor. Light-PREM [20] uses run-time profiling to detect memory areas used by a program to load during memory phases. We find the approach suitable for programs with highly dynamic data structures, but since it is based on profiling rather than static program analysis, it cannot guarantee worst-case bounds. Also, it does not discuss how to segment a task. In our previous work [25], we proposed a program analysis and transformation technique that uses static analysis to determine data accesses and predictably load/unload data from SPM while the program is executing. We reuse the same compiler framework in Section IV to determine the data to load in each segment. Note that [25] only deals with a single-task, single processor case, and does not segment the program based on PREM.

The closest related work is [21], where the authors introduce an automated task compilation and segmentation tool. The approach is similar to our work in that is relies on the LLVM compiler infrastructure, and employs loop splitting and tiling [15] to break loops that are too large to fit in local memory. However, the paper is focused on the case of a parallel, single-task system, and the tool employs a "greedy" segmenting approach that results in the longest possible segments. As we discuss in Section III and show in Section VI, such greedy approach is not suitable for multi-tasking systems where blocking time due to non-preemptive segments of lower priority tasks is a concern.

Finally, all related work assumes that a task comprises a single segment or a fixed sequence of segments. However, a program can have multiple execution paths whereas it accesses different data along each path, and must be PREM-compliant along all valid paths. Therefore, in Section III we introduce a new conditional PREM model in which the fixed segment sequence is replaced by a Directed Acyclic Graph (DAG) of segments, and we then show how to compile the program to execute segments conditionally. An important consequence of the conditional model is that we cannot extract a single worst-case path for the program: instead, the worst-case path depends on the maximum blocking time imposed by lower priority jobs. Hence, in Section III-A we also show how to extend the analysis in [27], [32], which considers a partitioned system with fixed per-task priorities, to the conditional model.

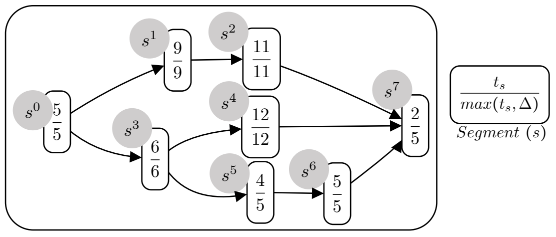


Fig. 2: Example segment DAG with $\Delta = 5$. In each node, the first number is the computation time of the segment, and the second its length. s^0 is s^{begin} and s^7 is s^{end}.

III. System Model and Schedulability Analysis

We consider scheduling a set of sequential, conditional PREM tasks on a multiprocessor. We assume non-preemptive segment execution, with a fixed memory time Δ to load/unload each segment. While in Section III-A we detail the analysis for the specific case of a partitioned system with fixed per-task priority, we point out that the task model presented in this section, and the compiler framework presented in Section IV, are independent of the specific scheduling scheme and can thus be used with any PREM-based approach. In details, we consider a set of sporadic tasks $\Gamma = \{\tau_1, \ldots, \tau_N\}$. 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: $D_i \le T_i$. τ_i is further characterized by a 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 edges representing precedence constraints between segments. We assume that the set S_i contains unique source and sink segments s^{begin}, s^{end}, as we consider programs with a single entry and exit point. We define the length $s.l$ of a segment $s \in S_i$ as the maximum length of any scheduling interval for the segment, that is, the maximum between the worst-case computation time t_s of s (including context-switch overheads) and the memory time Δ. In the remaining of the paper, we use p to denote a DAG path, that is, an ordered sequence of segments; $p.I$ is the number of segments in the path, $p.L$ the sum of their lengths, and $p.end$ the length of the last segment in the path. We say that a path is maximal if its first segment is s^{begin} and its last segment is s^{end}. To avoid confusion, in the rest of the paper we use uppercase letters (P) to denote maximal paths. Note that by definition $P.end = s^{end}.l$. Figure 2 shows an example DAG with three maximal paths: $P = \{s^0, s^1, s^2, s^7\}$, $P' = \{s^0, s^3, s^4, s^7\}$, and $P'' = \{s^0, s^3, s^5, s^6, s^7\}$. Note that we have $P.L = 30, P.I = 4, P'.L = 28, P'.I = 4, P''.L = 26, P''.I = 5$, and $P.end = P'.end = P''.end = 5$. Finally, we will use the notation $p = \{p_1, ..., p_n\}$ to indicate that path p can be obtained as a sequence of n (sub-)paths.

In general, a segment DAG could have a potentially large number of maximal paths, and a task could be segmented into many different DAGs. The following definitions will allow us to restrict the number of paths / DAGs to find a schedulable task system.

Definition 1: Given two maximal paths P, P', we say that P' dominates (is worse than or equal to) P and write $P' \ge P$

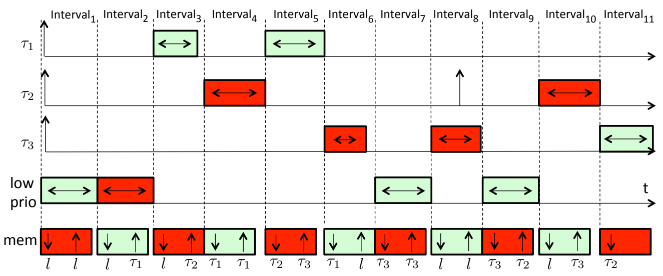


Fig. 3: Example critical instant. Up arrows represent arrival times. "low prio" denotes segments of lower priority tasks.

iff: $P'.L \geq P.L$ and $P'.I \geq P.I$ and $P'.end \leq P.end$. If neither $P' \geq P$ nor $P \geq P'$ holds, we say that the two paths are incomparable.

Since the $\geq$ relation defines a partial order between maximal paths, we can characterize a task based on its set of dominating paths. Formally, given segment DAG G, we use $G.C$ to denote the Pareto frontier [4] of all maximal paths in G. Intuitively, for a task τ_i, we will show that the set $G_i.C$ replaces the concept of worst-case execution time. For example, for Figure 2, $G.C$ is the set P, P''; P' is not included since P dominates it; but both P and P'' are included since they are incomparable.

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

Note that since $G.C$ is the Pareto frontier, $G' \geq G$ implies that for every path in G, there is a corresponding path in G' that dominates it.

A. Schedulability Analysis

We now consider a partitioned system with fixed per-task priority, and extend the analysis in [27], [32] to support conditional task execution. Since tasks are partitioned among cores and the effect of the memory schedule is captured by the memory time Δ, each core can be analyzed independently. Therefore, let $\Gamma = \{\tau_1, \ldots, \tau_N\}$ represents the set of tasks on the core under analysis, ordered by decreasing, distinct priorities, and assume that each task τ_i is associated with a given segment DAG G_i. The scheduling algorithm follows the scheme in Figure 1, in details: at the beginning of each scheduling interval, we execute on the processor the segment loaded during the previous interval (if any). In parallel, we unload and load the other local memory partition with the next segment of the highest priority ready task.

An example critical instant (modified from [32]) for task under analysis τ_3 is depicted in Figure 3. Here, the task under analysis and all higher priority tasks arrive just after the beginning of an interval for a lower priority task (Interval$_1$ in the figure). As a consequence, the task under analysis suffers an initial blocking time B_i equal to two intervals, as another lower priority segment loaded during Interval$_1$ executes during Interval$_2$. More in general, let τ_i be the task under analysis, and let $l_i^{l\max}$ denote the maximum length of any segment of a lower priority task. We have:

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

$$B_i = \begin{cases} 2 \cdot l_i^{l\max}, & \text{if } i \leq N-2. \\ l_{N-1}^{l\max} + \Delta, & \text{if } i = N-1. \\ \Delta, & \text{if } i = N. \end{cases} \tag{2}$$

For task τ_{N-1}, there is only one lower priority task; hence, the first blocking interval has only a memory phase and no task computation, while task τ_N computes in the second blocking interval. For τ_N, there is only one initial blocking interval consisting of a memory phase. Note that in the worst case, each successive segment of τ_i can suffer a blocking time equal to $l_i^{l\max}$ since two segments of τ_i cannot be executed back-to-back (Interval$_6$ and Interval$_8$ in the figure). For τ_N, we set $l_i^{l\max} = \Delta$ since there are no lower priority tasks, but a scheduling interval with memory only would be needed between successive segments of τ_N.

Since higher priority tasks arrive synchronously with the task under analysis, the interference suffered by τ_i in an interval of length t is equal to:

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

where L_j is the length of the path taken by τ_j. Since we cannot make any assumption on path execution, we maximize the interference by considering the path with maximum length:

$$L_j^{\max} = \max\{P.L \mid P \in G_j.C\}. \tag{4}$$

Note that it is sufficient to consider only the maximal paths in $G_j.C$ since each maximal path in G_j is dominated by a path in $G_j.C$, and by Definition 1 the dominating path has longer or equal L. Finally, since segments are executed non-preemptively, a task will complete by its deadline if its last segment starts execution $P.end$ time units before its deadline. Therefore, for a maximal path P, the response time $R_i(P)$ of τ_i up to its last segment can be computed as a standard iteration:

$$R_i(P) = B_i + (P.I - 1) \cdot l_i^{l\max} + P.L - P.end + \text{Inter}_i\big(R_i(P)\big), \tag{5}$$

and the task is schedulable along that path if:

$$R_i(P) \leq D_i - P.end. \tag{6}$$

Here, $P.L - P.end$ represents the length of intervals where τ_i computes (excluding the last segment), B_i is the blocking suffered by the first segment, $(P.I - 1) \cdot l_i^{l\max}$ is the blocking suffered by other segments, and $\text{Inter}_i\big(R_i(P)\big)$ is the interference of higher priority tasks. We next prove three key properties of the analysis.

Property 1: Consider two paths P, P' with $P' \geq P$. If Equation 6 holds for P', then it also holds for P.

Proof: Note that Equation 3 is increasing in t, and Equation 5 is increasing in $P.I$ and $P.L$ and decreasing in

[4]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.

$P.end$. Since it holds $P'.L \geq P.L$, $P'.I \geq P.I$, $P'.end \leq P.end$, at convergence it must hold: $R_i(P') \geq R_i(P)$.

Now by hypothesis it holds: $R_i(P') \leq D_i - P'.end$, which is equivalent to: $D_i \geq B_i + (P'.I - 1) \cdot l_i^{l\,\max} + P'.L + \text{Inter}_i\big(R_i(P')\big)$. But since we have: $B_i + (P'.I - 1) \cdot l_i^{l\,\max} + P'.L + \text{Inter}_i\big(R_i(P')\big) \geq B_i + (P.I - 1) \cdot l_i^{l\,\max} + P.L + \text{Inter}_i\big(R_i(P)\big)$, we obtain: $D_i - P.end \geq B_i + (P.I - 1) \cdot l_i^{l\,\max} + P.L - P.end + \text{Inter}_i\big(R_i(P)\big)$, completing the proof. ■

Based on Property 1, to check the schedulability of τ_i it is sufficient to test the set of dominating maximal paths. Hence, the following lemma immediately follows, where $\bigwedge$ denotes a logical and.

Lemma 1: Task τ_i is schedulable if:

$$\bigwedge_{P \in G_i.C} R_i(P) \leq D_i - P.end. \tag{7}$$

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

Proof: Part (A): by definition of Equations 5, 7. Part (B): since R_i is increasing in $l_i^{l\,\max}$, the response time for $l_i^{l\,\max} = l'$ cannot be larger than the one for l. ■

If the segment DAG G_i for each task $\tau_i \in \Gamma$ is known, then task set schedulability can be assessed by checking Equation 7 for all tasks in the order $\tau_1, \dots, \tau_N$. However, we are interested in using the response time of tasks $\tau_1, \dots, \tau_i$ in order to optimize the segmentation of task τ_{i+1}, hence $G_{i+1}, \dots, G_N$ are not known when analyzing τ_i. Based on Property 2, we instead use the analysis to determine the maximum value $\overline{l_i^{l\,\max}}$ of $l_i^{l\,\max}$ under which τ_i is still schedulable. Such value is then passed as an input to our segmentation algorithm working on τ_{i+1}, as we detail in the next section. Note that in theory, one could determine $\overline{l_i^{l\,\max}}$ by performing a binary search over Equation 7. However, we show Section III-B that an alternative formulation based on the concept of scheduling points used in [6] can be used to derive $\overline{l_i^{l\,\max}}$ directly.

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

Proof: Case $j = 1, \dots i-1$: since $G'_j \geq G_j$, the value of $L_j^{\max}$ for G_j is no larger than for G'_j. Since the interference $\text{Inter}_i(t)$ is increasing in $L_j^{\max}$, the resulting response time of τ_i for G_j cannot be larger than the one for G'_j.

Case $j = i$: since $G'_i \geq G_i$, for each maximal path $P \in G_i.C$ there must exist a maximal path $P' \in G'_i.C$ such that $P' \geq P$. Now since τ_i is schedulable for G'_i according to the analysis, by Equation 7 it must hold $R_i(P') \leq D_i - P'.end$; then by Property 1, it must also hold $R_i(P) \leq D_i - P.end$. This means that Equation 7 holds for G_i, concluding the proof. ■

Property 3 shows that the dominance relation indeed corresponds to the notion of a DAG being better than another from a schedulability perspective. Hence, the objective of our segmentation algorithm is to find a set of "best" DAGs for a task based on Definition 2.

B. Maximum Blocking Length Derivation

In this section, we show how to efficiently derive the maximum value of $\overline{l_i^{l\,\max}}$ for $l_i^{l\,\max}$, based on the strategy introduced in [6]. In details, define the set of points:

$$\mathcal{S}_i(P.end) = (D_i - P.end) \cup \{k \cdot T_j \mid j = 1 \dots i-1, k = 1 \dots \lfloor (D_i - P.end)/T_j \rfloor\}. \tag{8}$$

Note that $\mathcal{S}_i$ is the set of points where the interference value $\text{Inter}_i(t)$ changes, together with the deadline $D_i - P.end$ for the last segment to start computing. Then if for any time t, the total demand (including blocking time, interference of higher priority tasks and execution of τ_i) is less than t, it follows that τ_i is schedulable. We have thus obtained the alternative schedulability test:

$$\bigwedge_{P \in G_i.C} \bigvee_{t \in \mathcal{S}_i(P.end)} B_i + (P.I - 1) \cdot l_i^{l\,\max} + P.L - P.end + \text{Inter}_i(t) \leq t, \tag{9}$$

where $\bigvee$ represents the or of the conditions. In practice, the test can be efficiently evaluated because, as shown in [6], it is sufficient to check a subset $\bar{\mathcal{S}}_i(P.end)$ of the points in $\mathcal{S}_i(P.end)$.

We can now express Equation 9 as a condition on the maximum value of $l_i^{l\,\max}$ through simple algebraic manipulation. For a task τ_i with $i \leq N-2$ we obtain:

$$\begin{aligned}
&\bigwedge_{P \in G_i.C} \bigvee_{t \in \bar{\mathcal{S}}_i(P.end)} (P.I + 1) \cdot l_i^{l\,\max} + P.L - P.end + \text{Inter}_i(t) \leq t \\
&\Leftrightarrow \bigwedge_{P \in G_i.C} \bigvee_{t \in \bar{\mathcal{S}}_i(P.end)} l_i^{l\,\max} \leq \frac{t - P.L + P.end - \text{Inter}_i(t)}{P.I + 1} \\
&\Leftrightarrow \overline{l_i^{l\,\max}} = \min_{P \in G_i.C} \max_{t \in \bar{\mathcal{S}}_i(P.end)} \frac{t - P.L + P.end - \text{Inter}_i(t)}{P.I + 1}.
\end{aligned} \tag{10}$$

Similarly, for task τ_{N-1} we obtain:

$$\overline{l_i^{l\,\max}} = \min_{P \in G_i.C} \max_{t \in \bar{\mathcal{S}}_i(P.end)} \frac{t - P.L + P.end - \Delta - \text{Inter}_i(t)}{P.I}, \tag{11}$$

while for the lowest priority task we have $l_N^{l\,\max} = B_N = \Delta$, such that the schedulability test is equivalent to:

$$0 \leq \min_{P \in G_i.C} \max_{t \in \bar{\mathcal{S}}_i(P.end)} t - P.L + P.end - P.I \cdot \Delta - \text{Inter}_i(t). \tag{12}$$

IV. Program Segmentation

In this section, we show how a task is compiled into segments with the objective of optimizing the system schedulability. We start by discussing the program structure based on regions. After that, we define valid segmentations according to our compiler framework, which is based on LLVM [18] and the work in [25]. Finally, we detail our algorithm, which segments the program and returns the set of all DAGs that could be optimal. Similarly to [25], we assume that the program follows common real-time coding conventions. Therefore, the code should not use recursion or function pointers and all loops in the program are bounded. We also assume that the WCET and footprint of any part of the program are known either using static analysis or measurement.

A. Program Structure

We adopt the region-based program structure introduced in [25] which represents each function in the program as a tree where each node is a *region*. A region encompasses a sub-graph of the program control flow graph (CFG) with a single entry and a single exit. A leaf node in the region-tree is denoted as a *trivial region* and each trivial region comprises a single basic block or a single function call. Two regions r_1 and r_2 are *sequentially-composed* if the exit of r_1 is the entry of r_2. An internal node in the region-tree is a non-trivial region that can represent a loop, a condition, or a maximal set of sequentially-composed regions (i.e. a sequential region). A non-trivial region r_i is the *parent* of region r_j if r_i is the closest region containing r_j. Each loop region has one child that represents a single iteration of the loop. The top level region r_0^f of function f can either be a basic block or a sequential region. If r_0^f is sequential, then the last region in its children sequence must be a basic block that returns from f. Each region r_i in the region tree has WCET t_i and a data footprint $\mathcal{D}_i$.

Figure 4 shows an example of constructing the region trees of a program with two functions `main()` and `f()`. Figure 4b shows the region tree of the pseudo-code of `main()` in Figure 4a. Region r_0, which is the top level region of `main()`, is a sequential region with regions r_1 to r_4 as its children. Region r_2 is a loop with child r_5 that represents one iteration. All leaf regions r_1, r_3, r_4 and r_5 are trivial regions. Region r_3 is a call to `f()` in Figure 4d. Figure 4e is the region tree of `f()` where r_0^f is the top level region with r_1^f to r_3^f as its sequentially-composed children. Region r_2^f is an if-else conditional statement with region r_4^f as the true path and region r_5^f as the false path.

Loop transformations can be applied to loop regions that otherwise could not fit in a segment. A loop transformation must be legal, i.e. it preserves the temporal sequence of all dependencies and hence the result of the program. We are interested in two transformations: loop splitting and loop tiling. *Loop splitting* breaks the loop into multiple loops which have the same bodies but iterate over different contiguous portions of the index range. Figure 4c shows an example of splitting loop region r_2 in `main()` that has N iterations by expanding the loop region into three nodes: pre-loop node with k_p iterations, mid-loop node with $N - k_p - k_s$ iterations, and post-loop node with k_s iterations. *Loop tiling* combines strip-mining and loop permutation of a loop nest to create tiles of loop iterations which may be executed together. A tiled loop nest is divided into tiling loops that iterate over tiles and element loops that executes a tile. Note that the data footprint of a tile is derived in terms of the tile sizes. An example for tiling a 1-level loop is depicted in Figure 4f; however, note that multi-level (nested) loops can also be tiled. In the figure, r_4^f is a tiling loop region that has N_f iterations with tile size k_t. The number of tiles is $\lceil N_f/k_t \rceil$ with $M_f = \lceil N_f/k_t \rceil - 1$ complete tiles and a last tile $k_t^{last} \le k_t$ such that $k_t^{last} = N_f - M_f * k_t$. In Figure 4f, r_t^f is the tiling loop with M_f iterations over the element loop r_e^f. Note that, adding a tiling loop adds an overhead which is represented as r_o^f; we use t_{tile} to denote the WCET of the overhead region. The last tile is separated in r_{last}^f, where a tile of size k_t^{last} is executed after all complete tiles.

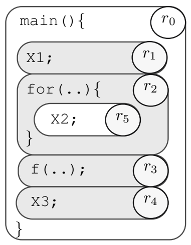
B. Valid Segmentation

Program segmentation is the process of assigning each part of the program code to a segment. In this paper, we restrict the parts of the program that can be assigned to a segment to be a region or a sequence of regions. A segmentation is valid if it satisfies the *footprint constraint*, the (optional) *length constraint* and the *compilation constraints*. The footprint constraint states that the footprint of each segment, i.e. the code and data of regions assigned to the segment must fit in the available SPM size $\mathcal{D}_{SPM}$. The length constraint states that the length of each segment must be at most $l^{\max}$. As discussed in Section III, this is done to limit the blocking time imposed on higher priority tasks; setting $l^{\max} = +\infty$ is equivalent to removing the constraint. Note that creating a segment incurs a segmentation overhead t_{seg} which contributes to the segment length. That is, if region r is assigned to segment s, then $s.l = t_r + t_{seg}$. If multiple regions in sequence are assigned to a segment s, then $s.l = (\sum_r t_r) + t_{seg}$. We further assume that the regions' WCETs satisfy the following property, which we argue is required for the WCET values to be sound:

Property 4: If r is a conditional region, then t_r is equal to the WCET of its longer children. If r is a sequential region or tiled loop, then its WCET is less than or equal to the sum of the WCETs of its children or tiles.

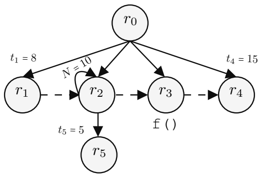
The compilation constraints are related to how the code is modelled and transformed. A necessary compilation constraint on a segment is that the data used by the segment is known before executing the segment. This implies that if a pointer is used to access a data object in a segment, the object(s) that the pointer may refer to must be known before the segment. In this paper, we add the following compilation constraints based on the region structure to develop a systematic segmentation process:

- A region cannot be assigned to more than one segment. If a region is assigned to a segment, all its children are assigned to the same segment.
- Each basic block region must be assigned to a segment.
- For all regions except function calls, we say that a region is *mergeable* if it satisfies the length and footprint constraints and all the children of the region are mergeable.
- A function is *mergeable* if the top level region of the function is mergeable. Accordingly, a function call region is mergeable if the called function is mergeable.
- A set of mergeable regions that are sequentially-composed can be combined in a *multi-region* segment that satisfies the length and footprint constraints.

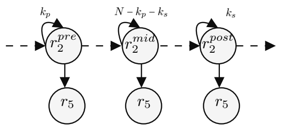


```
main(){
  X1;
  for(..){
    X2;
  }
  f(..);
  X3;
}
```

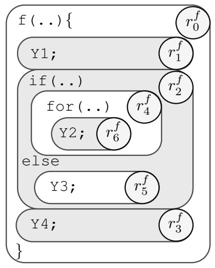
(a) `main` pseudo-code



(b) `main` region tree

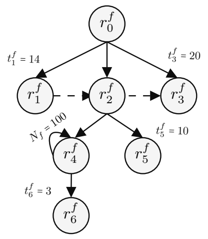


(c) Loop splitting of r_2

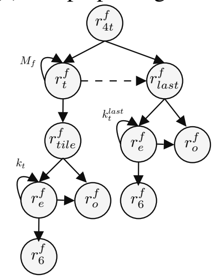


```
f(..){
  Y1;
  if(..)
    for(..)
      Y2;
  else
    Y3;
  Y4;
}
```

(d) `f` pseudo-code



(e) `f` region tree



(f) Loop tiling of r_4^f

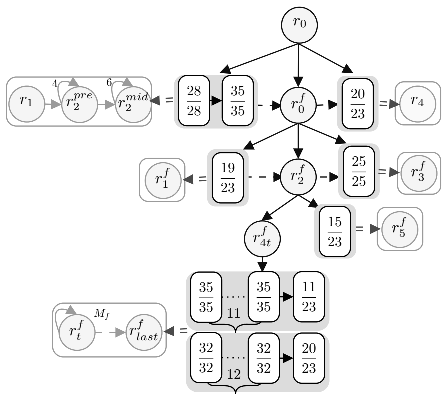
Fig. 4: Region representation of program code

- A loop can be divided into multiple segments using loop tiling and loop splitting. A loop region is *splittable* if its child that represents a single iteration of the loop is mergeable. A loop region that represents the outermost loop of a loop nest is *tileable* if it is legal to tile and a single iteration of the innermost loop of the tiling loops is mergeable. Note that a splittable loop is always tileable based on this definition. If a loop is tiled, then each tile must be assigned to a segment that comprises that tile only and the loop node represents a sequence of segments. Tiling allows combining multiple iteration of the loop in a repeatable segment by inserting the segmentation instruction around the element loop.

Segmenting the region tree of a function results in a *segmented tree* in which each *region sequence* R in the region tree is substituted with a set of paths $\mathcal{P}$. A region sequence can be one mergeable region, a tiled loop, or a sequence of mergeable regions and/or splittable regions and tiles. A path $p \in \mathcal{P}$ for region R is a sequence of segments, to which the regions and tiles in R are assigned. The segmented tree is constructed inter-procedurally, i.e. if a call to a function that is not mergeable, the segmented tree of that function is duplicated in place of the call region. If there are multiple calls to the function, the segmented tree for all the calls must be the same. The segmented tree of the program is accordingly the segmented tree of the $main$ function.

A segmented tree represents a set of DAGs of segments, such that a DAG is constructed from the segmented tree by taking one path out of each path set and joining them according to the tree hierarchy. This means that each maximal path is obtained by joining a sequence of path $\{p_1, p_2, ..., p_n\}$ for some n, where p_1 encompasses the source segment s^{begin} and p_n encompasses the sink segment s^{end} and hence the last region in the program. Note that if a path for a region sequence in a function is used to construct a DAG, the same path must be used for all the calls to the function since the region sequence represents the same code. We write $\mathcal{T}$ to represent the segmented tree, or equivalently the set of DAGs generated from it.

Figure 5 illustrates an example segmentation of the program introduced in Figure 4. Let the maximum segment length be $l_{max} = 35$, the memory time $\Delta = 23$, the segmentation overhead $t_{seg} = 5$, and the tiling overhead $t_{tiling} = 3$. We assume for this example that all the data of the program fits in the SPM, so the footprint constraint is always satisfied. Given the times for each basic block t in Figure 4b and Figure 4d, regions $\{r_1, r_4, r_1^f, r_3^f, r_5^f\}$ are mergeable regions. However, loop regions $\{r_2, r_4^f\}$ are not mergeable. Assume that we applied loop splitting on r_2 that has 10 iterations such that it is split to two loops: pre-loop with 4 iterations and mid-loop with 6 iterations. In Figure 5a, the region sequence $\{r_1, r_2^{pre}, r_2^{mid}\}$ is replaced by a path set with a single path that has 2 segments and a total length 67. The first segment combines r_1 and r_{pre}^2 while the second segment is r_{mid}^2. As region r_3 is a call to a non-mergeable function, it is replaced by a duplicate of the segmented tree of f. The segmented tree of f has two regions r_1^f and r_3^f each wrapped in a segment. Region r_2^f is a conditional that is not mergeable, so the false path r_5^f is wrapped in a segment while the true path r_4^f which is a loop with 100 iterations is tiled. There are many possible tiling options that would satisfy the max segment length. We choose two of them based on the tiling algorithm in the next section. The first path has length $p.l = 408$ and number of segments $p.I = 12$. The first 11 segments are complete tiles each with size $k_t = 9$ and length $\max(9 * 3 + t_{tiling} + t_{seg}, 23) = 35$, and the last segment is the last tile $k_t^{last} = 100 - 11 * 9 = 1$ with length $\max(1 * 3 + t_{tiling} + t_{seg}, 23) = 23$. Similarly, the other path has length $p.l = 497$ and number of segments $p.I = 13$. The


(a) Segmented Tree

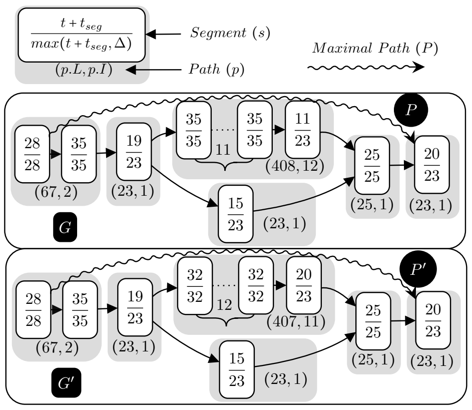

(b) DAGs

Fig. 5: Segmentation Example

first 12 segments are complete tiles each with size $k_t = 8$ and with length $\max(8*3+t_{tiling}+t_{seg},23) = 32$, and the last segment is the last tile $k_t^{last} = 100-12*8 = 4$ with length $\max(4*3+t_{tiling}+t_{seg},23) = 23$. The two DAGs generated from the segmented tree are shown in Figure 5b.

C. Segmentation Algorithm

The example in Section IV-B shows that different segmentation decisions can result in incomparable maximal paths according to Definition 1 as in Figure 5b: for the path P, we have $P.L = 547$, $P.I = 17$ and $P.end = 23$, while for the path P', we have $P'.L = 546$, $P'.I = 18$ and $P'.end = 23$. Since a DAG generated from the segmented tree $\mathcal{T}$ includes either P or P', the resulting two DAGs G and G' are also incomparable. This means that without considering the other tasks in the system, we cannot determine whether G or G' is better from a schedulability perspective. Hence, to guarantee that we can find an optimal segmentation for the task set, we need to consider both G and G'. On the other hand, if $G' \geq G$, we can safely ignore G' based on Property 3. This is formally captured by the following definition.

Definition 3: Let $\mathcal{G}$ be the set of all valid DAGs for a program according to the footprint, length and compilation constraints, and let $\mathcal{T}$ be the segmented tree returned by a segmentation algorithm for that program. We say that the algorithm preserves optimality iff for any program: $\mathcal{T}$ is valid according to the constraints, and $\forall G' \in \mathcal{G}, \exists G \in \mathcal{T} : G' \geq G$. Based on Definition 4, a naive optimality-preserving algorithm could proceeds as follows: first, enumerate all valid DAGs in $\mathcal{G}$. Then, cut dominating DAGs based on the dominance relation. However, due to possible variations of loop tiling/splitting and multi-region segments, this is practically unfeasible as the set $\mathcal{G}$ is too large. Therefore, we propose a much faster segmentation Algorithm 1 that preserves optimality according to Definition 4 but removes dominating DAGs without enumerating $\mathcal{G}$; instead, the algorithm explores the segmented tree recursively and removes unneeded paths from the path set $\mathcal{P}$ of each region sequence R.

Function SEGMENT(r) segments a subtree of the region tree and returns a segmented subtree with r as its root. The function traverses this subtree from its root r in depth-first order preserving the topological order between sequentially-composed children. If r is a sequential region, then a set of children in sequence that are mergeable or splittable loops may be combined in multi-region segments. This is achieved by adding these children to a region sequence R until a child that is not mergeable or splittable is found or until all children are traversed. Note that based on the compilation constraints, no children outside R can be combined with a region in R to form a segment; hence, we say that such R is a maximal region sequence. Then, the regions in R are replaced by a set of valid paths $\mathcal{P}$ that are generated using function SEGMENTSEQUENCE(R). If r is not sequential, a mergeable child r_c is directly replaced by a path of one segment, as r_c is a maximal region sequence by itself. If child r_c is not

Algorithm 1 Segmentation Algorithm

```
1:  function SEGMENT($r$)
2:      Initialize $R = \varnothing$        ▷ A set of sequential regions.
3:      Initialize $\mathcal{T}$ to be the r-subtree
4:      for all $r_c \in children(r)$ do
5:          if $r$ is sequential and $r_c$ is mergeable or splittable loop then
6:              Add $r_c$ to $R$
7:          else if $r_c$ is mergeable then ▷ $r$ is not sequential
8:              Replace $r_c$ with $\mathcal{P} = \{p\}$, where $p$ is single-segment path
9:          else
10:             Replace regions in $R$ with SEGMENTSEQUENCE($R$), empty $R$
11:             if $r_c$ is a tileable loop then
12:                 Replace $r_c$ with TILE($r_c$, $N_r$).
13:             else if $r_c$ is a call to $f$ then
14:                 Replace $r_c$ with SEGMENT($r_0^f$)
15:             else
16:                 Replace $r_c$ with SEGMENT($r_c$)
17:     If $R \neq \varnothing$, replace regions in $R$ with SEGMENTSEQUENCE($R$)
18:     Return $\mathcal{T}$
```

mergeable, then it has three cases: 1) r_c is a tileable loop, then a set of paths are generated by tiling the loop using function TILE(r_c); 2) r_c is a call to a function f, then the segmented tree of f is duplicated in place of r_c; 3) r_c is not a tileable loop or a function call, then it is segmented by recursively calling SEGMENT(r_c). Algorithm 1 consists of invoking SEGMENT(r_0) on the top level region of `main`, hence returning the segmented subtree for the whole program. There is an exception if r_0 satsfies all the constraints, then there is no need to construct the segmented tree as there is a single path with r_0 as a segment.

Since Algorithm 1 depends on SEGMENTSEQUENCE and TILE, we first state a key property of both functions, which will be detailed in Algorithms 2 and 3. Since the functions return a path set $\mathcal{P}$, we begin by defining a concept of domination among paths and path sets.

Definition 4: Given two paths p, p', we say that p' dominates p and write $p' \geq p$ iff: $p'.L \geq p.L$ and $p'.I \geq p.I$.

Note that Definition 4 is similar to Definition 1 for maximal paths, except that we do not consider the last segment, since its length is only relevant in the case of s^{end}. We can relate the two definitions through the following lemma.

Lemma 2: Consider two maximal paths $P = \{p_1, ..., p_k, ..., p_n\}, P' = \{p'_1, ..., p'_k, ..., p'_n\}$ obtained by joining n paths. If $p'_n.end = p_n.end$ and $\forall k = 1...n : p'_k \geq p_k$, then $P' \geq P$.

Proof: Note by construction $P.L = \sum_{k=1...n} p_k.L, P'.L = \sum_{k=1...n} p'_k.L$. From $p'_k \geq p_k$ it follows $p'_k.L \geq p_k.L$, hence $P'.L \geq P.L$. In the same manner, we obtain $P'.I \geq P.I$. Finally, since p'_n and p_n contain the last segments in their corresponding maximal paths P' and P, $p'_n.end = p_n.end$ implies $P'.end = P.end$. Then by Definition 1 we have $P' \geq P$. ∎

Definition 5: Given the region tree of `main` function in the program, we define the *last segment region set* R_{last} to be the set of regions that may be included in the last segment s^{end} of the program. This set is composed by traversing the children of r_0, the top level region of `main`, in reverse order and adding them to R_{last} until a constraint is violated; except if r_0 is mergeable, in which case $R_{last} = \{r_0\}$.

Definition 6: Given two path sets $\mathcal{P}, \mathcal{P}'$ for the same region sequence R, we say that $\mathcal{P}'$ dominates $\mathcal{P}$ and write $\mathcal{P}' \geq \mathcal{P}$ iff: $\forall p' \in \mathcal{P}', \exists p \in \mathcal{P} : p' \geq p$, and if $R \supseteq R_{last}$, then $p'.end = p.end$.

Property 5: Let R be a region sequence and $\mathcal{P}'$ the set of all valid paths for R. Then SEGMENTSEQUENCE(R) returns a set of paths $\mathcal{P}$ such that $\mathcal{P} \subseteq \mathcal{P}'$ and $\mathcal{P}' \geq \mathcal{P}$.

Property 6: Let r_c be a tilable loop with N_r iterations and $\mathcal{P}'$ the set of all valid paths for r_c. Then TILE(r_c) returns a set of paths $\mathcal{P}$ such that $\mathcal{P} \subseteq \mathcal{P}'$ and $\mathcal{P}' \geq \mathcal{P}$.

Intuitively, this implies that TILE and SEGMENTSEQUENCE return a set of best path for the corresponding region sequence / loop. Based on Properties 5, 6, we next prove in Theorem 1 that Algorithm 1 preserves optimality. We start by showing that the algorithm can stop traversing the tree at mergeable regions, i.e. if a region is mergeable we do not need to segment its children.

Lemma 3: Consider a region r that is either mergeable (possibly after splitting) or a tile, and a valid DAG G' for the program where r is not assigned to a segment. Then there exists a valid DAG G where r is assigned to a segment and $G' \geq G$.

Proof: Consider any maximal path $\mathcal{P}'$ in G' of the form $P' = \{p_{begin}, p', p_{end}\}$, where p' is a path through the descendants of r (note that no path of the form $P' = \{p_{begin}, p'\}$ can exist, since the last region of `main`, and thus the program, is a basic block with no descendants). Note that in case of conditional regions, there could be multiple such p', and hence maximal paths $\mathcal{P}'$ with the same p_{begin} and p_{end}. **Example:** consider the conditional region r_2^f in Figure 5; a valid DAG G' has two maximal paths P' through the descendants of r_2^f: one for the true path, and one for the false path.

Now consider a valid DAG G obtained by replacing all such maximal paths P' with a path $P = \{p_{begin}, p, p_{end}\}$, where p comprises a single segment that includes r only; note the DAG is valid since r is mergeable or a tile. Since $p.I = 1$, it immediately follows $p'.I \geq p.I$. Based on Property 4, there must also exist one path p' with $p'.L \geq p.L$. By Lemma 2, we then proved that there must exist a maximal path P' such that $P' \geq P$. By definition, this implies $G' \geq G$, completing the proof. ∎

Lemma 4: Consider a segmented tree $\mathcal{T}$ where all region sequences are maximal, and the path set $\mathcal{P}'$ for each region sequence R includes all valid paths for R. Then the DAGs generated from $\mathcal{T}$ preserve optimality.

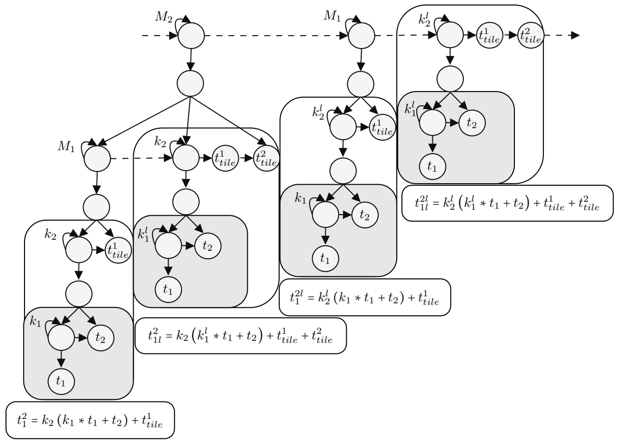
Proof: First note that by definition, each path $p \in \mathcal{P}'$ is a sequence of segments, to which the regions and tiles in R are assigned, i.e. $\mathcal{P}'$ does not include (still valid) paths that would segment the descendants of a region in R.

By the compilation constraints and definition of maximal region sequence R, it follows that any region that is in R cannot be merged in a segment with a region that is not in R. Hence, any valid maximal path for the program that includes segments of n region sequences can be constructed by joining n paths: $P = \{p_1, ..., p_k, ..., p_n\}$. By Lemma 3, we can restrict each p_k to be a path in $\mathcal{P}'$ (where each region $r \in R$ is assigned to a segment) and for each valid DAG G', generate a DAG G such that $G' \geq G$. By Definition 3, this means that generating DAGs from $\mathcal{T}$ preserves optimality. ∎

Lemma 4 shows that to preserve optimality, it is sufficient to return a single segmented tree with maximal region sequences, which is what Algorithm 1 builds by construction. Finally, we show that instead of generating the set $\mathcal{P}'$ of all valid paths for each region sequence R, we can use a dominated subset $\mathcal{P}$.

Lemma 5: Consider a segmented tree $\mathcal{T}$ as in Lemma 4. If for any maximal region sequence R, we substitute the set $\mathcal{P}'$ of all valid paths with a set $\mathcal{P}$ such that $\mathcal{P} \subseteq \mathcal{P}'$ and $\mathcal{P}' \geq \mathcal{P}$, the resulting segmented tree preserves optimality.

Proof: Let $\overline{\mathcal{T}}$ to denote the segmented tree obtained by replacing each path set $\mathcal{P}'$ with $\mathcal{P}$. Since for all regions $\mathcal{P} \subseteq \mathcal{P}'$, DAGs generated from $\overline{\mathcal{T}}$ are still valid. Consider any DAG G' generated from $\mathcal{T}$, and a maximal path P' of G'



M_2
M_1
k_2^l
k_1^l
k_2
k_1
t_1
t_2
t_{tile}^1
t_{tile}^2
$t_1^2 = k_2\,(k_1 * t_1 + t_2) + t_{tile}^1$
$t_{1l}^2 = k_2\,(k_1^l * t_1 + t_2) + t_{tile}^1 + t_{tile}^2$
$t_1^{2l} = k_2^l\,(k_1 * t_1 + t_2) + t_{tile}^1$
$t_{1l}^{2l} = k_2^l\,(k_1^l * t_1 + t_2) + t_{tile}^1 + t_{tile}^2$

Fig. 6: 2D Tiling Tree

through n region sequences: $P' = \{p'_1, ..., p'_k, ..., p'_n\}$. Since for all regions $\mathcal{P}' \geq \mathcal{P}$, then for every p'_k there exists another path p_k in $\overline{\mathcal{T}}$ such that $p'_k \geq p_k$, and furthermore $p'_n.end = p_n.end$ since the last region sequence in any maximal path must include R_{last}. By Lemma 2, this means that we can find a maximal path $P = \{p_1, ..., p_k, ..., p_n\}$ for $\overline{\mathcal{T}}$ such that $P' \geq P$. Since this is true for any maximal path through a given set of region sequences, and both $\mathcal{T}$ and $\overline{\mathcal{T}}$ have the same set of (maximal) region sequences, we have shown that $\overline{\mathcal{T}}$ can generate a DAG G such that for every maximal path $P \in G$, there is a maximal path $P' \in G'$ with $P' \geq P$. This implies $G' \geq G$, and since by Lemma 4 $\mathcal{T}$ preserves optimality, it thus follows that $\overline{\mathcal{T}}$ also preserves optimality according to Definition 3. ■

Theorem 1: If Properties 5, 6 hold, Algorithm 1 preserves optimality.

Proof: By construction, the algorithm returns a segmented tree $\mathcal{T}$ of maximal region sequences. Let $\mathcal{P}'$ to denote the set of all valid paths for each region R. The actual path set $\mathcal{P}$ used for R is generated at line 7, 11 or 13. At line 7, region r_c is not sequential. Hence, $R = \{r_c\}$ is a maximal region. The algorithm generates a path comprising a single segment for r_c, which is the only valid path for R; thus we have $\mathcal{P} = \mathcal{P}'$. At line 11 and 17, the path set $\mathcal{P}$ is generated by calling either SEGMENTSEQUENCE(R) or TILE(r_c); by Properties 5, 6 and Lemma 5, in both cases $\mathcal{P} \subseteq \mathcal{P}'$ and $\mathcal{P}' \geq \mathcal{P}$ hold. In summary, Lemma 5 applies to all maximal regions, hence the algorithm preserves optimality. ■

D. 2-level Tiling

In this section, we discuss our Algorithm 2 to find optimality-preserving tile sizes for a 2-level tileable loop. While our framework is restricted to 2-level loops (deeper levels of tiling are uncommon), in general the algorithm could be extended to tile more levels. Note that 1-level tiling is a special case of 2-level tiling in which the outer loop has a single iteration.

Figure 6 shows an expanded region tree that represents 2-level tiling. Consider two nested loops: an inner loop with N_1 iterations and an execution time of one iteration t_1 and an outer loop with N_2 iterations and an execution time of one iteration $N_1 * t_1 + t_2$. A 2-level tiling with tile sizes k_1 and k_2 of the inner and outer loops will create 2 tiling loops with $\lceil N_1/k_1 \rceil$ and $\lceil N_2/k_2 \rceil$ iterations, and 2 element loops with k_1 and k_2 iterations. Let $M_1 = \lceil N_1/k_1 \rceil - 1$, then the outer tiling loop has M_1 tiles with k_1 iterations of the outer element loop and a last tile $k_1^l = N_1 - M_1 * k_1$. Similarly, the inner tiling loop has $M_2 = \lceil N_2/k_2 \rceil - 1$ tiles with k_2 iterations of the inner element loop and a last tile $k_2^l = N_2 - M_2 * k_2$. Adding a tiling loop incurs a tiling overhead for each iteration of the tiling loop as discussed in Section IV-B; we indicate this as t_{tile}^1 and t_{tile}^2 for the inner and outer loop, respectively. As shown in Figure 6, there are 4 tile times: t_1^2 repeated $M_1 * M_2$ times, t_{1l}^2 repeated M_2 times, t_1^{2l} repeated M_1 times, and t_{1l}^{2l} executed one time. Based on these notation, a path $p(k_2, k_1)$ that is generated by the region sequence represented by the tiled loop nest has number of segments $p.I = (M_1 + 1)(M_2 + 1)$ and length:

$$\begin{aligned} p.L = M_2 * M_1 * \max(\Delta, t_1^2 + t_{seg}) \\ + M_2 * \max(\Delta, t_{1l}^2 + t_{seg}) \\ + M_1 * \max(\Delta, t_1^{2l} + t_{seg}) \\ + \max(\Delta, t_{1l}^{2l} + t_{seg}). \end{aligned} \tag{13}$$

We can rewrite the length as $p.L = t_{loop} + t_{overhead} + t_\Delta$ such that $t_{loop} = N_2.(N_1.t_1 + t_2)$ is the original loop time and does not depend on the tile size, $t_{overhead} = (M_2 + 1)[(M_1 + 1) * (t_{tile}^1 + t_{seg}) + t_{tile}^2]$ is the tiling and segmentation overhead, and t_Δ is the total segment under-utilization:

$$\begin{aligned} t_\Delta = M_1 * M_2 * \max(\Delta - (t_1^2 + t_{seg}), 0) + \\ + M_2 * \max(\Delta - (t_{1l}^2 + t_{seg}), 0) \\ + M_1 * \max(\Delta - (t_1^{2l} + t_{seg}), 0) \\ + \max(\Delta - (t_{1l}^{2l} + t_{seg}), 0). \end{aligned} \tag{14}$$

Note that the $p.I$ and $p.L$ are non-linear functions in k_1 and k_2, as the expressions for M_1 and M_2 include ceiling functions.

Algorithm 2 takes as input a region r and a number of iterations N_2 for the outer loop[5] and returns a set of valid paths $\mathcal{P}$ for r. The algorithm starts by computing the upper limit of the outer loop tile k_2^{max} as the maximum k_2 such that any tile segment s in $p(k_2, k_1 = 1)$ has length $s.l \leq l_{max}$ and footprint $s.\mathcal{D} \leq s.\mathcal{D}_{SPM}$. For each k_2, $k_1^{max}(k_2)$ is similarly computed as the maximum k_1 such that any tile segment s in $p(k_2, k_1)$ has length $s.l \leq l_{max}$ and footprint $s.\mathcal{D} \leq \mathcal{D}_{SPM}$. A threshold $k_1^\Delta(k_2)$ is then computed; in Lemma 6, we show that all segments generated from tile sizes (k_2, k_1) with $k_1 \leq k_1^\Delta(k_2)$ are underutilized, meaning that the length of the segment is less than or equal to Δ. Two variables $\hat{t}_\Delta, \hat{t}_{overhead}$ are used to track the valid solution with total minimum under-utilization so far in k_1 loop such that $\hat{t}_\Delta$ is the minimum under-utilization and $\hat{t}_{overhead}$ is the overhead due to tiling and segmentation for that solution. Note that the solution with minimum under-utilization is not necessarily the

[5]Note that N_2 is the number of iteration in the mid-loop node when Algorithm 2 is called by Algorithm 3.

Algorithm 2 2-Level Tiling

```
1:  function TILE($r$, $N_2$)
2:      $\mathcal{P} = \varnothing$
3:      Compute $k_2^{max}$
4:      for all $k_2 \le k_2^{max}$ do
5:          Compute $k_1^{max}(k_2)$
6:          $k_1^{\Delta}(k_2) = \max\{k_1 \mid t_1^2 + t_{tile}^1 + t_{seg} \le \Delta\}$
7:          $k_1 = k_1^{max}$, $\hat{t}_{\Delta} = \infty$, $\hat{t}_{overhead} = 0$
8:          repeat
9:              Generate $p(k_2, k_1)$
10:             if $p(k_2, k_1)$ is valid then
11:                 Add $p(k_2, k_1)$ to $\mathcal{P}$
12:                 Compute $t_{overhead}$, $t_{\Delta}$
13:                 if $t_{\Delta} > \hat{t}_{\Delta}$ then
14:                     $\hat{t}_{\Delta} = t_{\Delta}$, $\hat{t}_{overhead} = t_{overhead}$
15:             $k_1 = k_1 - 1$
16:         until $k_1 = k_1^{\Delta}$ or $\hat{t}_{\Delta} = 0$ or $t_{overhead} \ge \hat{t}_{overhead} + \hat{t}_{\Delta}$
17:     Filter $\mathcal{P}$ by removing dominating paths based on Definition 4
18:     return $\mathcal{P}$
```

solution with the minimum total length for all the tiles. That is due to the non-linear relation between the tile size and the last tile size. Then, we iterate over k_1 starting from k_1^{max}. In each iteration, if path $p(k_2, k_1)$ is valid we add it to $\mathcal{P}$, then we compute the tiling and segmentation overhead $t_{overhead}$ and under-utilization t_{Δ}, and update $\hat{t}_{overhead}$ and $\hat{t}_{\Delta}$ accordingly. The loop exits if k_1^{Δ} is reached or if the overhead of the current solution $t_{overhead}$ exceeds $\hat{t}_{overhead} + \hat{t}_{\Delta}$, or if t_{Δ} of the current solution is 0. Finally, the path set $\mathcal{P}$ is filtered and returned, in the same way as in Algorithm 2. We prove in Lemma 8 that the algorithm preserves Property 6.

Lemma 6: All segments in a path $p(k_2, k_1)$ with $k_1 \le k_1^{\Delta}(k_2)$ have length Δ.

Proof: Note that t_1^2 is increasing in k_1. Hence, by definition of $k_1^{\Delta}(k_2)$, it must hold for k_1: $t_1^2 + t_{tile}^1 + t_{seg} \le \Delta$. By definition, we also have $k_1^l \le k_1$ and $k_2^l \le k_2$. This implies that $t_1^2 + t_{seg}$, $t_{1l}^2 + t_{seg}$, $t_1^{2l} + t_{seg}$ and $t_{1l}^{2l} + t_{seg}$ are all smaller than or equal to $t_1^2 + t_{tile}^1 + t_{seg}$, and thus Δ. Since the length of a segment is the maximum of its computation (including t_{seg}) or Δ, it follows that all segments have length Δ. ■

Lemma 7: Consider two valid solutions (k_2, k_1) with overhead $t_{overhead}$ and (k_2, k_1') with overhead $t'_{overhead}$. If $k_1' \le k_1$ then $p(k_2, k_1').I \ge p(k_2, k_1).I$ and $t'_{overhead} \ge t_{overhead}$.

Proof: Follows directly by noticing that both $p(k_2, k_1).I$ and $t_{overhead}$ depends on M_1, which is non-increasing in k_1. ■

Lemma 8: Property 6 holds for Algorithm 2.

Proof: Note that r cannot be part of R_{last}, since the last region in a program must be a basic block and tiles cannot be merged with other regions. By the compilation constraints, every generated tile must be assigned to a segment that comprises the tile only. Then by the footprint and length constraints, the set of all valid paths $\mathcal{P}'$ comprises all valid paths $p(k_2, k_1)$ such that $k_2 \le k_2^{max}$ and $k_1 \le k_1^{max}(k_2)$.

For a given value of k_2, define $\bar{k}_1$ as the value of k_1 for which the algorithm break at line 16. Furthermore, let $\hat{k}_1$ be the value of k_1 corresponding to $\hat{t}_{\Delta}$, $\hat{t}_{overhead}$. We prove that for every $k_1' < \bar{k}_1$, there exists a valid k_1 in $\bar{k}_1, ..., k_1^{max}$ such that $p(k_2, k_1') \ge p(k_2, k_1)$. Since furthermore the filtering on line 17 respects Definition 6 (given that r is not R_{last}), this implies that Property 6 holds.

We have to consider three cases, based on which breaking condition at line 16 evaluates to true. Note that we have $k_1 < \bar{k}_1 \le \hat{k}_1$. Furthermore, $p(k_2, \hat{k}_1)$ must be valid (otherwise we would not have set the values of $\hat{t}_{\Delta}$, $\hat{t}_{overhead}$ at line 14), and so must be $p(k_2, k_1^{\Delta})$ (unless $l_{max} < \Delta$ and then there are no valid paths and the lemma trivially holds).

- Assume $\bar{k}_1 = k_1^{\Delta}$. By Lemma 6, we have $p(k_2, k_1^{\Delta}).L = p(k_2, k_1^{\Delta}).I * \Delta$ and $p(k_2, k_1').L = p(k_2, k_1').I * \Delta$. Given $k_1' < \bar{k}_1$, we also have $p(k_2, k_1').I \ge p(k_2, k_1^{\Delta}).I$ by Lemma 7 and thus $p(k_2, k_1') \ge p(k_2, k_1^{\Delta})$.
- If $\hat{t}_{\Delta} = 0$, then it must hold $t'_{\Delta} \ge \hat{t}_{\Delta}$. By Lemma 7, we also have $t'_{overhead} \ge \hat{t}_{overhead}$; therefore, $p(k_2, k_1').L \ge p(k_2, \hat{k}_1).L$. Also by Lemma 7 we have $p(k_2, k_1').I \ge p(k_2, \hat{k}_1).I$. Therefore $p(k_2, k_1') \ge p(k_2, \hat{k}_1)$.
- If $t_{overhead} \ge \hat{t}_{overhead} + \hat{t}_{\Delta}$, then $t'_{overhead} + t'_{\Delta} \ge t_{overhead} \ge \hat{t}_{overhead} + \hat{t}_{\Delta}$; this implies $p(k_2, k_1').L \ge p(k_2, \hat{k}_1).L$. Since again by Lemma 7 we have $p(k_2, k_1').I \ge p(k_2, \hat{k}_1).I$, it holds $p(k_2, k_1') \ge p(k_2, \hat{k}_1)$.

■

1) Region Sequence Segmentation: Next, we consider Algorithm 3 that generates a path set $\mathcal{P}$ from a set of sequential regions. If we do not apply loop splitting, then there are 2^{m-1} possible paths for m mergeable regions in sequence. An enumeration of these ways is possible as m is usually small. However, adding loop splitting greatly increases the number of paths. To tackle this complexity, the algorithm works by incrementally constructing a set of *partial paths*. We denote a path with segments that encompasses all the regions in a region sequence R as a complete path. Consequently, we define a partial path $\bar{p}$ as a path that encompasses a sub-sequence $\bar{R} \subseteq R$ that includes all regions from the beginning of R up to region r. Since a partial path is still a valid program path (just on a smaller region sequence), we use $\bar{p}.I$, $\bar{p}.L$ and $\bar{p}.end$ with the usual meaning. However, we also use $\bar{p}.t_{end}$ to denote the WCET of the regions included in the last segment of $\bar{p}$, such that $\bar{p}.end = \max(\Delta, \bar{p}.t_{end} + t_{seg})$. The algorithm iterates over the regions in R, maintaining a set of partial paths $\bar{\mathcal{P}}$. For each region r with computation time t_r, a new set of partial paths is constructed by taking each partial path $\bar{p}$ in $\bar{\mathcal{P}}$ and adding r to it. Note that when doing so, two new partial paths might be generated in the following way:

1) Add r to a new segment and add it to $\bar{p}$. This results in a new partial path $\bar{p}_n$ such that $\bar{p}_n.I = \bar{p}.I + 1$, $\bar{p}_n.t_{end} = t_r$, and $\bar{p}_n.L = \bar{p}.L + \bar{p}_n.end$. Note that $\bar{p}_n$ is always valid, since r is mergeable (or a tile).
2) Add r to the last segment of $\bar{p}$. This results in a new partial path $\bar{p}_m$ such that $\bar{p}_m.I = \bar{p}.I$, $\bar{p}_m.t_{end} = \bar{p}.t_{end} +$

t_r, and $\bar{p}_m.L = \bar{p}.L - \bar{p}.end + \bar{p}_m.end$. Note that $\bar{p}_m$ might not be a valid path according to the constraints; hence, it is only added to the new set of partial paths if valid.

The process continues until after we reach the last region r in R; at that point, the path in $\bar{\mathcal{P}}$ are complete, so we return a path set $\mathcal{P} = \bar{\mathcal{P}}$. We next prove a set of conditions that allow us to remove some partial paths from $\bar{\mathcal{P}}$ at each step. Given a partial path $\bar{p}$ for $\bar{R}$, we say that $\bar{p}$ generates a complete path p if there are valid segmentation decisions for the remaining regions in $R \setminus \bar{R}$ that result in p.

Lemma 9: Given a sub-sequence $\bar{R} \subseteq R$ and two partial paths $\bar{p}'$ and $\bar{p}$ over $\bar{R}$, then for any complete path p' for R generated from $\bar{p}'$, there exists a complete path p for R generated from $\bar{p}$ such that $p' \geq p$ if any of the following conditions is satisfied:

1) $\bar{p}'.I \geq \bar{p}.I$ **and** $\bar{p}'.L - \bar{p}'.end \geq \bar{p}.L - \bar{p}.end$ **and** $\bar{p}'.t_{end} \geq \bar{p}.t_{end}$.
2) $\bar{p}'.I = \bar{p}.I$ **and** $\bar{p}'.L \geq \bar{p}.L$ **and** $\bar{p}'.t_{end} \geq \bar{p}.t_{end} > \Delta - t_{seg}$
3) $\bar{p}'.I > \bar{p}.I$ **and** $\bar{p}'.L - \bar{p}'.end \geq \bar{p}.L$ **and** $\bar{p}'.t_{end} \leq \bar{p}.t_{end}$ **and** $\bar{p}'.t_{end} < \Delta - t_{seg}$.
4) $\bar{p}'.I > \bar{p}.I$ **and** $\bar{p}'.L \geq \bar{p}.L + \Delta$ **and** $\bar{p}'.t_{end} \leq \bar{p}.t_{end}$ **and** $\bar{p}'.t_{end} > \Delta - t_{seg}$.

Proof:

By induction on the number of remaining regions in $R \setminus \bar{R}$. The base case is that $R \setminus \bar{R}$ is empty (no remaining regions); the induction case is that there is at least one remaining region r that can be added to $\bar{p}'$ and $\bar{p}$.

Base case: Since $R \setminus \bar{R} = \emptyset$, both $\bar{p}'$ and $\bar{p}$ are already complete paths. Hence, it suffices to prove that $\bar{p}'.I \geq \bar{p}'.I$ and $\bar{p}'.L \geq \bar{p}.L$, from which $\bar{p}' \geq \bar{p}$. By cases based on which of Conditions 1-4 apply between $\bar{p}'$ and $\bar{p}$.

1) We have $\bar{p}'.I \geq \bar{p}.I$. Furthermore, from $\bar{p}'.L - \bar{p}'.end \geq \bar{p}.L - \bar{p}.end$ and $\bar{p}'.t_{end} \geq \bar{p}.t_{end}$ we obtain $\bar{p}'.L \geq \bar{p}.L$.
2) We have $\bar{p}'.I = \bar{p}.I$ and $\bar{p}'.L \geq \bar{p}.L$.
3) We have $\bar{p}'.I > \bar{p}.I$ and from $\bar{p}'.L - \bar{p}'.end \geq \bar{p}.L$ we obtain $\bar{p}'.L \geq \bar{p}.L$.
4) We have $\bar{p}'.I > \bar{p}.I$ and from $\bar{p}'.L \geq \bar{p}.L + \Delta$ we obtain $\bar{p}'.L > \bar{p}.L$.

Induction case: let $(\bar{p}_m, \bar{p}_n)$ / $(\bar{p}'_m, \bar{p}'_n)$ to denote the partial paths generated by adding r to $\bar{p}/\bar{p}'$; note that $\bar{p}_m/\bar{p}'_m$ could be an invalid partial path, while $\bar{p}_n$ / $\bar{p}'_n$ is always valid. Assuming that one of Conditions 1-4 apply between $\bar{p}'$ and $\bar{p}$, we then prove that after adding r, one of Conditions 1-4 apply between $\bar{p}'_n$ and $\bar{p}_n$, and if $\bar{p}'_m$ is valid, then one of Conditions 1-4 also apply either between $\bar{p}'_m$ and $\bar{p}_m$ or between $\bar{p}'_m$ and $\bar{p}_n$. By cases, based which of Conditions 1-4 apply between $\bar{p}'$ and $\bar{p}$. Note that if $\bar{p}'.t_{end} \geq \bar{p}.t_{end}$, this implies that $\mathcal{D}'_{end} \geq \mathcal{D}_{end}$. This is always true as $\bar{p}'$ and $\bar{p}$ are constructed from regions in sequence, and since the execution time of last segment in $\bar{p}'$ is larger than the execution time of last segment in $\bar{p}$, then the set of regions and/or split in the last segment of $\bar{p}$ are part of the last segment of $\bar{p}'$. Hence, $\mathcal{D}'_{end} \geq \mathcal{D}_{end}$.

1) Since $\bar{p}'.t_{end} \geq \bar{p}.t_{end}$, then $\bar{p}_m$ is valid if $\bar{p}'_m$ is valid. We prove that Condition 1 applies between $\bar{p}'_n$ and $\bar{p}_n$, and if $\bar{p}'_m$ is valid, Condition 1 also applies between $\bar{p}'_m$ and $\bar{p}_m$.
   - Condition 1 applies between $\bar{p}'_m$ and $\bar{p}_m$:
     Since $\bar{p}'.L - \bar{p}'.end \geq \bar{p}.L - \bar{p}.end$ and $\bar{p}'.t_{end} \geq \bar{p}.t_{end}$, then Equation 15 holds:
     $$\bar{p}'_m.L - \bar{p}'_m.end \geq \bar{p}_m.L - \bar{p}_m.end \tag{15}$$
     And since $\bar{p}'.I \geq \bar{p}.I$, then Equation 16 holds:
     $$\bar{p}'_m.I \geq \bar{p}_m.I \tag{16}$$
     And since $\bar{p}'.t_{end} \geq \bar{p}.t_{end}$, then by adding t_r to both sides Equation 17 holds:
     $$\bar{p}'_m.t_{end} \geq \bar{p}_m.t_{end} \tag{17}$$
     Since Equations 15, 16 and 17 hold, then Condition 1 applies bewteen $\bar{p}'_m$ and $\bar{p}_m$.
   - Condition 1 applies between $\bar{p}'_n$ and $\bar{p}_n$:
     Since $\bar{p}'.L - \bar{p}'.end \geq \bar{p}.L - \bar{p}.end$ and $\bar{p}'.t_{end} \geq \bar{p}.t_{end}$, then Equation 18 holds:
     $$\bar{p}'_n.L - \bar{p}'_n.end \geq \bar{p}_n.L - \bar{p}_n.end \tag{18}$$
     And since $\bar{p}'.I \geq \bar{p}.I$, then Equation 19 holds:
     $$\bar{p}'_n.I \geq \bar{p}_n.I \tag{19}$$
     And since $\bar{p}'_n.t_{end} = \bar{p}_n.t_{end}$ and Equations 18 and 19 hold, then Condition 1 applies bewteen $\bar{p}'_n$ and $\bar{p}$.
2) Since $\bar{p}'.t_{end} \geq \bar{p}.t_{end}$, then $\bar{p}_m$ is valid if $\bar{p}'_m$ is valid. We prove that Condition 1 applies between $\bar{p}'_n$ and $\bar{p}_n$, and if $\bar{p}'_m$ is valid, Condition 2 also applies between $\bar{p}'_m$ and $\bar{p}_m$.
   - Condition 2 applies between $\bar{p}'_m$ and $\bar{p}_m$: Since $\bar{p}'.I = \bar{p}.I$, then Equation 16 holds.
     Since $\bar{p}'.t_{end} \geq \bar{p}.t_{end} > \Delta - t_{seg}$, Equation 17 holds.
     Since $\bar{p}'.t_{end} \geq \bar{p}.t_{end} > \Delta - t_{seg}$, then Equations 20 and 21 hold:
     $$\bar{p}'_m.end - \bar{p}'.end = t_r \tag{20}$$
     $$\bar{p}_m.end - \bar{p}.end = t_r \tag{21}$$
     From Equations 20 and 21 and since $\bar{p}'.L \geq \bar{p}.L$, hence $\bar{p}'.L - \bar{p}'.end + \bar{p}'_m.end \geq \bar{p}.L - \bar{p}.end + \bar{p}_m.end$ and Equation 22 holds:
     $$\bar{p}'_m.L \geq \bar{p}_m.L \tag{22}$$
     Since Equations 16, 17 and 22 hold, then Condition 2 applies bewteen $\bar{p}'_m$ and $\bar{p}_m$.
   - Condition 1 applies between $\bar{p}'_n$ and $\bar{p}_n$:
     Since $\bar{p}'.I = \bar{p}.I$, then Equation 19 holds. And since $\bar{p}'.L \geq \bar{p}.L$, then Equation 18 holds. Since $\bar{p}'_n.t_{end} = \bar{p}_n.t_{end}$ and Equations 18 and 19 hold, then Condition 1 applies bewteen $\bar{p}'_n$ and $\bar{p}_n$.
3) Since $\bar{p}'.t_{end} \leq \bar{p}.t_{end}$, then $\bar{p}_m$ may not be valid. We prove that Condition 1 applies between $\bar{p}'_n$ and $\bar{p}_n$, and

if $\bar{p}'_m$ is valid, Condition 1 also applies between $\bar{p}'_m$ and $\bar{p}_n$.

- Condition 1 applies between $\bar{p}'_m$ and $\bar{p}_m$:
  Since $\bar{p}'_m.end \geq \bar{p}_n.end$, then $\bar{p}'.L - \bar{p}'.end + \bar{p}'_m.end \geq \bar{p}.L + \bar{p}_n.end$ and Equation 23
  $$\bar{p}'_m.L \geq \bar{p}_n.L \tag{23}$$
  Since $\bar{p}'.I > \bar{p}.I$, then Equation 24 holds:
  $$\bar{p}'_m.I \geq \bar{p}_n.I \tag{24}$$
  And since, $\bar{p}'.t_{end}+t_r > t_r$, then Equation 25 holds:
  $$\bar{p}'_m.t_{end} \leq \bar{p}_n.t_{end} \tag{25}$$
  Since Equations 23, 24 and 25 hold, then Condition 1 applies bewteen $\bar{p}'_m$ and $\bar{p}_n$.
- Condition 1 applies between $\bar{p}'_n$ and $\bar{p}_n$:
  Since $\bar{p}_n.end = \bar{p}'_n.end$ and $\bar{p}'.L - \bar{p}'.end \geq \bar{p}.L$, then Equation 18 holds. And since $\bar{p}'.I > \bar{p}.I$, then Equation 19 holds. From Equations 18 and 19 and since $\bar{p}_n.end = \bar{p}'_n.end$, then Condition 1 applies bewteen $\bar{p}'_n$ and $\bar{p}_n$.

4) Since $\bar{p}'.t_{end} \leq \bar{p}.t_{end}$, then $\bar{p}_m$ may not be valid. We prove that Condition 1 applies between $\bar{p}'_n$ and $\bar{p}_n$, and if $\bar{p}'_m$ is valid, Condition 1 also applies between $\bar{p}'_m$ and $\bar{p}_n$.
   - Condition 1 applies between $\bar{p}'_m$ and $\bar{p}_n$:
     Since $\bar{p}'.I > \bar{p}.I$ **and** $\bar{p}'.L \geq \bar{p}.L + \Delta$ **and** $\bar{p}'.t_{end} \leq \bar{p}.t_{end}$ **and** $\bar{p}'.t_{end} > \Delta - t_{seg}$:
     Since $\bar{p}'_m.end \geq \bar{p}_n.end$, then Equation 26 holds:
     $$\bar{p}'_m.L - \bar{p}'_m.end + \bar{p}'.end \geq \bar{p}_n.L - \bar{p}_n.end + \Delta \tag{26}$$
     And since $\bar{p}'.t_{end} \geq \Delta - t_{seg}$, then Equation 27 holds:
     $$\Delta - \bar{p}'.t_{end} \leq 0 \tag{27}$$
     From Equations 26 and 27, then Equation 28 holds:
     $$\bar{p}'_m.L - \bar{p}'_m.end \geq \bar{p}_n.L - \bar{p}_n.end \tag{28}$$
     And since, $\bar{p}'.I > \bar{p}.I$, then Equation 23 holds.
     From Equations 28 and 23 and since $\bar{p}'_m.t_{end} \geq \bar{p}'_n.t_{end}$, then Condition 1 applies between $\bar{p}'_m$ and $\bar{p}_n$.
   - Condition 1 applies between $\bar{p}'_n$ and $\bar{p}_n$:
     Since $\bar{p}'_n.end = \bar{p}_n.end$ and $\bar{p}'.L \geq \bar{p}.L + \Delta$ then Equation 18 holds. And since $\bar{p}'.I > \bar{p}.I$, then Equation 19 holds. From Equations 18 and 19 and since $\bar{p}'_n.t_{end} \geq \bar{p}'_n.t_{end}$, then Condition 1 applies between $\bar{p}'_n$ and $\bar{p}_n$.

■

Algorithm 3 traverses the regions in R in topological order generating partial paths using the current region r. If r is not a splittable loop, then new partial paths $\bar{p}_m$ and $\bar{p}_n$ are generated by adding r to each previous partial path in function CREATEPARTIALPATHS. The new partial paths are placed in $\bar{\mathcal{P}}_{next}$, which is then filtered based on Lemma 9 before becoming the set of partial paths $\bar{\mathcal{P}}$ at the next iteration. If r is a splittable loop, then before generating new partial path, the loop must be split to pre-loop region r_p, mid-loop region r_t and post-loop region r_s. Note that all combinations of pre-loop k_p and post-loop k_s splits are visited. For each (k_p, k_s), partial paths $\bar{\mathcal{P}}_{loop}$ for r_p are generated using CREATEPARTIALPATHS, then r_t is tiled and each tile path is sequenced with the paths in $\bar{\mathcal{P}}_{loop}$. Then, partial paths are created using k_s for all paths in $\bar{\mathcal{P}}_{loop}$. All paths $\bar{\mathcal{P}}_{loop}$ are finally accumulated in $\bar{\mathcal{P}}_{next}$.

Algorithm 3 Segment a Sequence of Regions

```
Require: A set of sequential regions $R$ and the set of last segment regions $R_{last}$
 1: function SEGMENTSEQUENENCE($R$)
 2:     $\bar{\mathcal{P}} = [\bar{p} = \varnothing]$, $\mathcal{P}_{last} = \varnothing$, $\bar{\mathcal{P}}_{next} = \varnothing$, $\bar{R}_{last} = R_{last}$
 3:     for all $r \in R$ do        ▷ Traverse the sequence in topological order.
 4:         if $r$ is a splittable loop then
 5:             for all $k_p$, $k_s$ do:
 6:                 Split $r$ to $r_p, r_t$ and $r_s$
 7:                 $\bar{\mathcal{P}}_{loop}$ = CREATEPARTIALPATHS($r_p, \bar{\mathcal{P}}$)
 8:                 Filter $\bar{\mathcal{P}}_{loop}$ using Lemma 9
 9:                 $\bar{\mathcal{P}}_{loop}$ = all path by joining $\bar{\mathcal{P}}_{loop}$ with TILE($r_t$, $N_r - k_p - k_s$)
10:                 $\bar{\mathcal{P}}_{loop}$ = CREATEPARTIALPATHS($r_s, \bar{\mathcal{P}}_{loop}$)
11:                 if $r_s \in \bar{R}_{last}$ then
12:                     Create $s^{end}$ from all regions in $\bar{R}_{last}$
13:                     For each $\bar{p} \in \mathcal{P}_{loop}$, create $\bar{p}_{last}$ by adding $s^{end}$ to $\bar{p}$, add $\bar{p}_{last}$ to $\mathcal{P}_{last}$
14:                 $\bar{\mathcal{P}}_{next} = \bar{\mathcal{P}}_{next} \cup \mathcal{P}_{loop}$
15:         else        ▷ $r$ is a mergeable region that is not a splittable loop
16:             $\bar{\mathcal{P}}_{next}$ = CREATEPARTIALPATHS($r, \bar{\mathcal{P}}$)
17:             if $r \in \bar{R}_{last}$ then
18:                 Create $s^{end}$ from all regions in $\bar{R}_{last}$
19:                 For each $\bar{p} \in \bar{\mathcal{P}}$, create $\bar{p}_{last}$ by adding $s^{end}$ to $\bar{p}$, add $\bar{p}_{last}$ to $\mathcal{P}_{last}$
20:         Filter $\bar{\mathcal{P}}_{next}$ using Lemma 9, $\bar{\mathcal{P}} = \bar{\mathcal{P}}_{next}$, $\mathcal{P}_{next} = \varnothing$, $\bar{R}_{last} = \bar{R}_{last} \setminus r$
21:     Filter $\bar{\mathcal{P}}$ by removing dominating paths based on Definition 4
22:     return $\mathcal{P} = (\mathcal{P}_{last}$ if $R \supseteq R_{last}$ else $\bar{\mathcal{P}})$
23: function CREATEPARTIALPATHS($r, \bar{\mathcal{P}}$)
24:     $\bar{\mathcal{P}}_{tmp} = \varnothing$
25:     for all $\bar{p}$ in $\bar{\mathcal{P}}$ do
26:         Create $\bar{p}_m$ by adding $r$ to the last segment in $\bar{p}$, add $\bar{p}_m$ to $\bar{\mathcal{P}}_{tmp}$ if valid
27:         Create $\bar{p}_n$ by adding new segment using $r$ to $\bar{p}$, add $\bar{p}_n$ to $\bar{\mathcal{P}}_{tmp}$
28:     return $\bar{\mathcal{P}}_{tmp}$
```

The final complexity regards the case where $R \supseteq R_{last}$. In this case, Definition 6 requires us to consider all possible combinations of the last segment s^{end}. If the current region $r \in R_{last}$ and r is mergeable, there is a last segment s^{end} composed of all regions in $\bar{R}_{last}$ such that $\bar{R}_{last}$ is the set of all regions starting from r to the end of R_{last}. Then s^{end} is combined with partial paths $\bar{\mathcal{P}}$ to form complete paths in $\bar{\mathcal{P}}_{last}$. If r is a splittable loop, then the part that contribute to s^{end} is the post-loop split (tiles cannot be merged with other regions). Hence for each (k_p, k_s), we generate the partial paths using r_p and add tile paths from r_t, then a last segment s^{end} is composed from the post-loop split r_s and all the regions after r until the end of R_{last}. Complete paths are generated by adding s^{end} to each partial path in $\bar{\mathcal{P}}_{loop}$ to produce a complete path in $\bar{\mathcal{P}}_{last}$. Finally, the path set $\mathcal{P}$ for R is $\bar{\mathcal{P}}_{last}$ if $R \supseteq R_{last}$, otherwise it is $\bar{\mathcal{P}}$.

Lemma 10: Algorithm 3 satisfies Property 5.

Proof: By construction, the algorithm explores all possible combinations for the parameters of a splittable loop, all possible valid assignments of sequential regions in R to segments, and tiling decisions based on Algorithm 2 (note that on Lines 12,18, adding the regions in $\bar{R}_{last} \subseteq R_{last}$ to a single segment s^{end} must be valid based on the definition of R_{last}). Therefore, it must hold $\mathcal{P} \subseteq \mathcal{P}'$. It remains to show that if a path p' is discarded (i.e., the path is in $\mathcal{P}'$ but not in $\mathcal{P}$), then there exists a path p such that $p' \geq p$, and if $R \supseteq R_{last}$, then $p'.end = p.end$. A path can be discarded for three reasons: (1) Algorithm 2 removes a tiling solution; (2) a partial path is discarded based on the conditions in Lemma 9; (3) a complete path is filtered based on Definition 4.

Case (1): Assume that Algorithm 2 removes a path p'_t from the returned path set; by Property 6, it must return another path p_t such that $p'_t \geq p_t$. Then if we consider any complete path $p' = \{p_1, ..., p'_t, ..., p_n\}$ for R, there must exist another path $p = \{p_1, ..., p_t, ..., p_n\}$, and by Lemma 2, it must hold $p' \geq p$. Next consider the case $R \supseteq R_{last}$: by the compilation constraints, a tiled loop cannot generate the last segment in the program (the last region is a basic block, and tiles cannot be merged with another region). Therefore p_n is not empty and it must hold $p'.end = p.end = p_n.end$.

Case (2): We first consider the sub-case when R_{last} is not contained in R. If a partial path is discarded, then a path p' in $\mathcal{P}'$ might be removed from $\mathcal{P}$; however, by Lemma 9, there must be a path $p \in \mathcal{P}$ such that $p' \geq p$. Next, consider the sub-case where $R \supseteq R_{last}$. Each complete path p' can then be written as $p' = \{p'_1, \{s^{end}\}\}$, where s^{end} is a segment made of the regions in some $\bar{R}_{last} \subseteq R_{last}$, and p'_1 is a partial path for $R \setminus \bar{R}_{last}$. Then by applying Lemma 9 to $R \setminus \bar{R}_{last}$, if a partial path is discarded causing p'_1 to be removed, then there must still be a path p_1 for $R \setminus \bar{R}_{last}$ such that $p'_1 \geq p_1$. This implies that we can find a complete path $p = \{p_1, \{s^{end}\}\}$ in $\mathcal{P}$, where by Lemma 2 it holds $p' \geq p$, and $p'.end = p.end = s^{end}.l$.

Case (3): Note this applies only if R_{last} is not contained in R. It thus suffices to notice that by Definition 4 $p' \geq p$ must hold. ■

V. Optimal Task Set Segmentation

Based on the analysis Properties 2, 3 introduced in Section III-A and segmentation Algorithm 1, we now show that we can obtain an optimal task set segmentation using

Algorithm 4 Task Set Segmentation

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

Algorithm 4. The algorithm recursively calls function SegmentTaskSet for task index i from 1 to N by keeping track of the DAGs $G_1, \ldots G_{i-1}$ selected for the previous tasks. The function maintains a maximum segment length $l^{\max}$, which is provided as a constraint to Algorithm 1 to generate the segmented tree $\mathcal{T}_i$ for τ_i. If $i < N$, the function iterates over all possible $G_i \in \mathcal{T}_i$; the schedulability analysis is used to determine $\overline{l_i^{l\max}}$, the maximum schedulable value of $l_i^{l\max}$, and the function is then invoked recursively for task $i+1$ after updating $l^{\max}$ based on the computed value. Note that if G_i is not schedulable, then we obtain $l^{\max} < 0$; hence, there will be no valid DAG for τ_{i+1} ($\mathcal{T}_i$ is empty), and the recursive call will immediately return. Once we reach task τ_N, the function checks if τ_N is schedulable for any DAG $G_N \in \mathcal{T}_N$, in which case we terminate by finding a solution $\{G_1, \ldots, G_N\}$. If no solution can be found, the algorithm eventually terminates on Line 2.

We now prove the optimality of Algorithm 4 for a program segmentation obeying the footprint and program constraints in Section IV-B. We start with a corollary of the properties.

Corollary 1: Consider two DAGs G_j, G'_j for task τ_j where $1 \leq j \leq i$ and $G'_j \geq G_j$. Let $\overline{l_i^{l\max}}, \overline{l_i^{l\max}}'$ be the maximum value of $l_i^{l\max}$ under which τ_i is schedulable for G_j and G'_j, respectively, according to an analysis satisfying Properties 2, 3. Then $\overline{l_i^{l\max}} \geq \overline{l_i^{l\max}}'$.

Proof: By Property 2, $\overline{l_i^{l\max}}$ and $\overline{l_i^{l\max}}'$ are well defined (i.e., there must exist such maximum values). Since τ_i is schedulable with $l_i^{l\max} \leq \overline{l_i^{l\max}}'$ for G'_j, based on Property 3 it is also schedulable with $l_i^{l\max} \leq \overline{l_i^{l\max}}'$ for G_j; this implies $\overline{l_i^{l\max}} \geq \overline{l_i^{l\max}}'$. ■

Theorem 2: Algorithm 4 is an optimal segmentation algorithm for a conditional PREM task set Γ according to any (sufficient) schedulability analysis satisfying Properties 2, 3 and based on the footprint and compilation constraints.

Proof: We have to show that if there exists a set of segment DAGs $G'_1, \ldots, G'_N$ for Γ that is valid according to the footprint and compilation constraints and is schedulable according to the analysis, then Algorithm 4 finds a (same or different) DAG set $G_1, \ldots, G_N$ that is also valid and schedulable.

By induction on the index i. We show that for every i, there exists a recursive call sequence of function SEGMENT-TASKSET that results in a DAG set $G_1, \ldots G_i$ such that $G'_j \geq G_j$ for every $j = 1 \ldots i$; by Property 3 with $i = N$, this proves the theorem (note that τ_N is schedulable by Property 3, while all other tasks are schedulable because the recursion reaches G_N). We also show that for every $j = 1 \ldots i$ it holds $\overline{l_j^{l\max}}' \leq \overline{l_j^{l\max}}$, where $\overline{l_j^{l\max}}'$ is the maximum schedulable value of $l_j^{l\max}$ computed by the analysis with DAGs $G'_1, \ldots, G'_j$, and $\overline{l_j^{l\max}}$ is the same value for DAGs $G_1, \ldots, G_j$.

Base Case ($i = 1$): note $l^{\max} = +\infty$, meaning that only the footprint and compilation constraints apply when invoking Algorithm 1. Hence, by Definition 3 the algorithm must find a DAG $G_1 \in \mathcal{T}_1$ such that $G'_1 \geq G_1$. By Corollary 1, this also implies $\overline{l_1^{l\max}}' \leq \overline{l_1^{l\max}}$.

Induction Step ($i = 2 \ldots N$): consider the recursive call sequence that results in $G'_j \geq G_j$ and $\overline{l_j^{l\max}}' \leq \overline{l_j^{l\max}}$ for each $j = 1 \ldots i-1$ (such sequence exists by induction hypothesis); we have to show that we can find a DAG $G_i \in \mathcal{T}_i$ such that $G'_i \geq G_i$ and $\overline{l_i^{l\max}}' \leq \overline{l_i^{l\max}}$.

Based on the recursive call at line 7 of the algorithm, it must hold: $l^{\max} = \min_{j=1}^{i-1} \overline{l_j^{l\max}}$. Define $l^{\max\prime} = \min_{j=1}^{i-1} \overline{l_j^{l\max}}'$; since the task set is schedulable for $G'_1, \ldots, G'_N$, the maximum length of any segment in G'_i is at most $l^{\max\prime}$. By induction hypothesis, it must be $l^{\max\prime} \leq l^{\max}$, which means that the maximum segment length in G'_i is also no larger than $l^{\max}$. Hence, if we define $\mathcal{G}_i$ to be the set of all valid DAGs for a program according to the constraints with maximum segment length $l^{\max}$, we have $G'_i \in \mathcal{G}_i$. By Definition 3, this implies that Algorithm 1 finds a valid DAG G_i with maximum segment length $l^{\max}$ such that $G'_i \geq G_i$. $\overline{l_i^{l\max}}' \leq \overline{l_i^{l\max}}$ then again follows by Corollary 1. ■

A. Discussion

Complexity: since it iterates over all $G_i \in \mathcal{T}_i$, Algorithm 4 is exponential. Intuitively, it might seem sufficient to only use the DAG in $\mathcal{T}_i$ that results in the highest value of $\overline{l_i^{l\max}}$; however, given two DAGs G_i and G'_i with $\overline{l_i^{l\max}} \geq \overline{l_i^{l\max}}'$, it might be that $L_i^{\max} \geq L_i^{'\max}$, that is, G_i results in larger slack for τ_i, but it increases the interference caused by τ_i on lower priority tasks based on Equations 4. In this case, we have to test both G_i and G'_i. However, if $L_i^{\max} \leq L_i^{'\max}$, then we can safely ignore G'_i. As we show in Section VI, in practice this results in an acceptable runtime considering the algorithm is an offline optimization.

Composability and Generality: since Algorithm 4 requires to segment tasks in priority order, any code change in a program will not affect higher priority tasks, but it might force a recompilation of all lower priority tasks. This might be undesirable, especially if the priority ordering does not match criticality levels. Therefore, in Section VI we also explore a simpler and faster (but non-optimal) heuristic that uses the same value of $l^{\max}$ for all tasks, thus ensuring that each program can be compiled independently. In this sense, we would like to stress that even if the optimality of Algorithm 4 depends on analysis Properties 2, 3, our compiler framework in conjunction with Algorithm 1 can still be used to produce a set of valid program segmentations for any PREM-based system.

VI. EVALUATION

We implemented our segmentation framework using LLVM to analyze and generate the region trees for the program as in [25], and estimate the data footprint for each part of the program. Poly [14] is used to handle loop transformations. For code generation, we target a simple MIPS processor model with 5-stages pipeline and no branch prediction. We assume that there are data SPM, and code SPM and that the task code fits in the code SPM. Note that the WCET of each region in a program is statically estimated using the simple MIPS processor model similar to [26]. For the data SPM, we vary its size from 4 kB to 512 kB. For the memory transfer, we assume that the DMA needs 1 cycle per word (4 bytes). The segmentation overhead t_{seg} includes the DMA intialization and the context switching, and it is assumed to be 100 cycles.

We evaluate the segmentation and scheduling algorithms using a set of synthetic and real benchmarks. We used applications from UTDSP [1], TACLeBench [11] and Cortex-Suite [28] benchmark suites. The application are chosen to represent a variety of sizes, complexities and data footprints (see Table I). The applications are used to generate sets of random tasks. Each task set is composed of a random number of tasks between 4 and 12 tasks. Given a system utilization and the number of tasks, the utilization of each task is generated with uniform distribution [7], and then a period is assigned to each task. The period of τ_i is computed as $u_i * c_i$ where u_i is the generated utilization and c_i is the WCET of the application if executed without premption from the SPM. We assume deadlines equal to periods. Schedulability tests are conducted for 250 task sets.

We report the results in terms of the system schedulability and the weighted utilization. The *system scheduability* is the proportion of the schedulable task sets out of the total tested task sets. We define the *weighted utilization* μ of a system as: $\mu = \frac{\sum_u sched(u) * u}{\sum_u u}$ where $sched(u)$ is the system schedulability for system utilization u. We compare our optimal algorithm with ideal, greedy and heuristic algorithms. The *ideal* algorithm assumes no restriction on SPM size and that the program code can be segmented at any arbitrary point without any increased overhead. Hence, the only constraint is l_{max} which is produced from Algorithm 4 [6]. The *greedy* and *heuristic* algorithms do not depend on Algorithm 4 to drive the segmentation of each task based on the schedulability analysis. The greedy algorithm resembles the algorithm used

[6]Note that the ideal algorithm is still compliant with the PREM model, i.e. the next segment has to be decided and loaded while the current segment is executing.

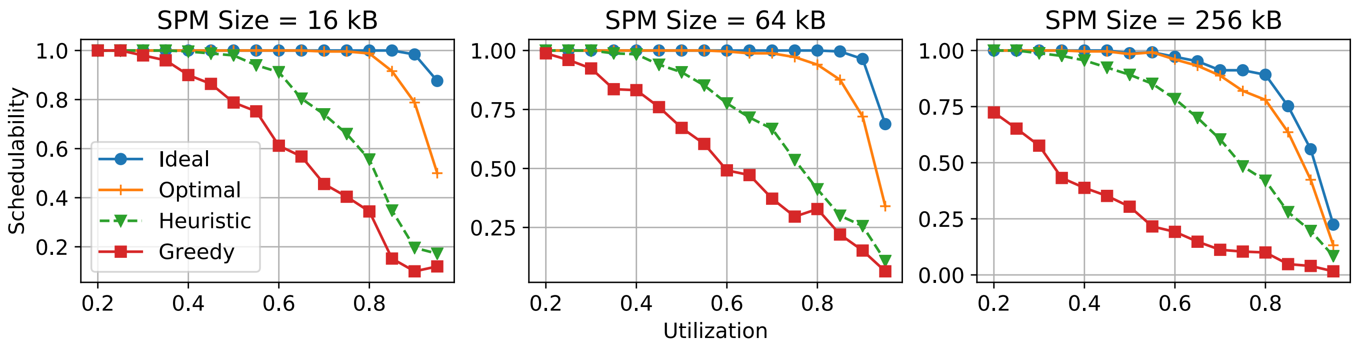
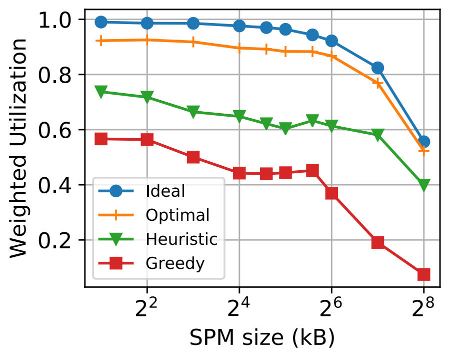
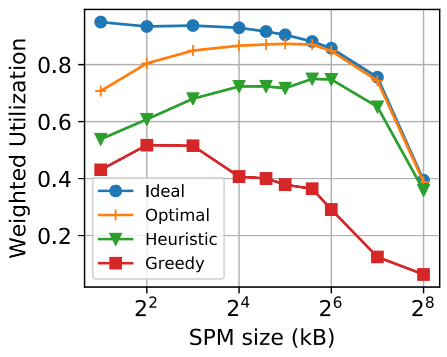


Fig. 7: Schedulability vs Utilization



(a) t_{seg} = 100



(b) t_{seg} = 1000, footprint > 24 kB

Fig. 8: Weighted Utilization VS SPM Size

Benchmark	**Suite**	**LOC**	**Data(B)**
adpcm_dec	TACLeBench	476	404
cjpeg_transupp	TACLeBench	474	3459
fft	TACLeBench	173	24572
compress	UTDSP	131	136448
lpc	UTDSP	249	8744
spectral	UTDSP	340	4584
disparity	CortexSuite	87	2704641

TABLE I: Benchmarks
(LOC: lines of code)

in [21] and assumes $l_{max} = \infty$ for all tasks. The heuristic algorithm uses the same l_{max} for all tasks by varying l_{max} between Δ and $10 * \Delta$ with step $0.5 * \Delta$, and picking the value of l_{max} that achieves the highest weighted utilization.

Figure 7 shows the system schedulability for the four algorithms for SPM sizes of 16, 64 and 256 kB. For the heuristic algorithm, the value of l_{max} with the best weighted utilization is showed in the figure for each SPM size. The graphs show that the optimal algorithm performs much better than the greedy and the heuristic algorithms and close to the ideal algorithm for different SPM sizes. This is confirmed in Figure 8a that shows the weighted utilization for the compared algorithms for SPM sizes between 4 kB and 512 kB. Note that the ideal algorithm may suffer from segmentation overhead, the interference and blocking overhead from other tasks in the system, and also segment under-utilization. This leads to lower schedulability at high system utilization.

We can notice in Figure 8a that the weighted utilization does not increase as SPM size increases. This might be counter-intuitive as increasing the SPM size allows more data to be loaded for each segment which leads to decreased segmentation overhead. However, the tasks suffer from a higher under-utilization penalty as Δ increases. The second effect is dominant since the segmentation overhead is relatively small and 4 benchmarks have data footprints of less than 8kB. For this reason, we show in Figure 8b the weighted utilization using only applications with data footprint greater than 24 kB and t_{seg} = 1000. The figure shows that the system schedulability ascents at first and then declines around SPM size of 48 kB.

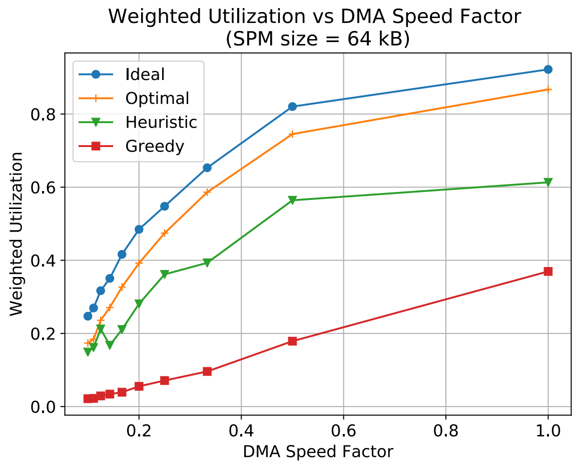


Fig. 9: Weighted Utilization VS DMA Speed Factor

The DMA speed is an main factor in the schedulability of the system. In order to illustrate its effect, we show in Figure 9 the change of the weighted utilization vs the DMA speed factor when SPM size is 64 kB. The DMA speed factor is relative to the base speed of 1 cycle per word, i.e. a factor of 0.5 means the DMA speed is 2 cycles per word. We can see that the weighted utilization improves as the DMA speed factor increases which is intuitive. The figure also shows that the optimal algorithms is superior to the greedy and heuristic algorithms for all the tested speed factors.

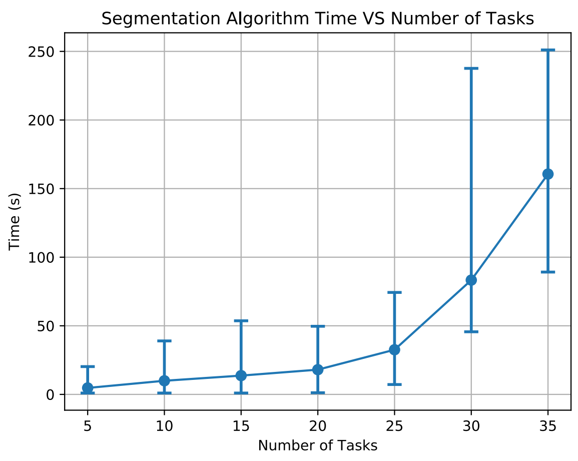


Fig. 10: Segmentation Time VS Number of Tasks

The segmentation algorithm takes a few seconds to finish with a maximum of a minute compared to few hours for the naive segmentation algorithm with exhaustive search. Running the scheduling algorithm for one of the tested task sets takes an average of a minute to segment the tasks and apply the schedulability test with a maximum of few minutes. We show in Figure 10 the min/mean/max time in seconds to segment a task set with number of tasks per set varying between 5 and 35. The numbers were obtained by collecting the time of segmentation 100 task sets for each number of tasks.

VII. Conclusions and Future Work

PREM-based scheduling schemes have recently attracted significant attention in the literature, but to make the approach applicable to industrial practice, there is a stringent need for automated tools. To this end, we have proposed a compiler-level framework that automatize the process of analyzing a program and transforming it into a conditional sequence of PREM segments. Furthermore, for the case of fixed-priority partitioned scheduling with fixed-length memory phases, which has been fully implemented and tested in [27], we have shown that it is possible to find optimal segmentation decisions within reasonable time for realistic programs.

This work could be extended in two main directions: first, by applying it to other PREM-based scheduling schemes. Note that since searching for an optimal segmentation solution might become too expensive, we might have to resort to a heuristic instead. Second, by extending it to other task and platform models. In particular, we are highly interested in looking at parallel tasks executed on heterogeneous multicore devices.

References

[1] UTDSP Benchmark Suite. URL: `http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html`.

[2] Ahmed Alhammad and Rodolfo Pellizzoni. Schedulability analysis of global memory-predictable scheduling. In *Proceedings of the 14th International Conference on Embedded Software - EMSOFT '14*, pages 1–10, New York, New York, USA, 2014. ACM Press. URL: `http://dl.acm.org/citation.cfm?doid=2656045.2656070`, `doi:10.1145/2656045.2656070`.

[3] Ahmed Alhammad and Rodolfo Pellizzoni. Time-predictable execution of multithreaded applications on multicore systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014*, pages 1–6, New Jersey, 2014. IEEE Conference Publications. URL: `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6800243`, `doi:10.7873/DATE.2014.042`.

[4] Ahmed Alhammad, Saud Wasly, and Rodolfo Pellizzoni. Memory efficient global scheduling of real-time tasks. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 285–296. IEEE, 4 2015. URL: `http://ieeexplore.ieee.org/document/7108452/`, `doi:10.1109/RTAS.2015.7108452`.

[5] Matthias Becker, Dakshina Dasari, Borislav Nicolic, Benny Akesson, Vincent Nelis, and Thomas Nolte. Contention-Free Execution of Automotive Applications on a Clustered Many-Core Platform. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 14–24. IEEE, 7 2016. URL: `http://ieeexplore.ieee.org/document/7557865/`, `doi:10.1109/ECRTS.2016.14`.

[6] E. Bini and G.C. Buttazzo. Schedulability analysis of periodic fixed priority systems. *IEEE Transactions on Computers*, 53(11):1462–1473, 11 2004. URL: `http://ieeexplore.ieee.org/document/1336766/`, `doi:10.1109/TC.2004.103`.

[7] Enrico Bini and Giorgio C. Buttazzo. Measuring the Performance of Schedulability Tests. *Real-Time Systems*, 30(1-2):129–154, 5 2005. URL: `http://link.springer.com/10.1007/s11241-005-0507-9`, `doi:10.1007/s11241-005-0507-9`.

[8] Paolo Burgio, Andrea Marongiu, Paolo Valente, and Marko 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)*, pages 1–8. IEEE, 10 2015. URL: `http://ieeexplore.ieee.org/document/7369851/`, `doi:10.1109/RTEST.2015.7369851`.

[9] Nicola Capodieci, Roberto Cavicchioli, Paolo Valente, and Marko 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*, pages 48–57, New York, New York, USA, 2017. ACM Press. URL: `http://dl.acm.org/citation.cfm?doid=3139258.3139270`, `doi:10.1145/3139258.3139270`.

[10] Guy Durrieu, Madeleine Faugère, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti, and W. Puffitsch. Predictable Flight Management System Implementation on a Multicore Processor. {*Embedded Real Time Software (ERTS'14)*}, 2 2014. URL: `https://hal.archives-ouvertes.fr/hal-01121700`.

[11] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Bjrn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. *DROPS-IDN/6895*, 55, 2016. URL: `http://drops.dagstuhl.de/opus/volltexte/2016/6895/https://github.com/tacle/tacle-bench`, `doi:10.4230/OASICS.WCET.2016.2`.

[12] Bjorn Forsberg, Luca Benini, and Andrea Marongiu. HePREM: Enabling predictable GPU execution on heterogeneous SoC. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 539–544. IEEE, 3 2018. URL: `http://ieeexplore.ieee.org/document/8342066/`, `doi:10.23919/DATE.2018.8342066`.

[13] Bjorn Forsberg, Andrea Marongiu, and Luca Benini. GPUguard: Towards supporting a predictable execution model for heterogeneous SoC. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 318–321. IEEE, 3 2017. URL: `http://ieeexplore.ieee.org/document/7927008/`, `doi:10.23919/DATE.2017.7927008`.

[14] TOBIAS GROSSER, ARMIN GROESSLINGER, and CHRISTIAN LENGAUER. Polly Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters*, 22(04):1250010, 2012. URL: `https://www.infosun.fim.uni-passau.de/publications/docs/GGL2012ppl.pdfhttp://www.worldscientific.com/doi/abs/10.1142/S0129626412500107`, `doi:10.1142/S0129626412500107`.

[15] Emna Hammami and Yosr Slama. An overview on loop tiling techniques for code generation. In *Proceedings of IEEE/ACS International*

Conference on Computer Systems and Applications, AICCSA, volume 2017-October, 2018. doi:10.1109/AICCSA.2017.168.

[16] Mohamed Hassan and Rodolfo Pellizzoni. Bounding DRAM Interference in COTS Heterogeneous MPSoCs for Mixed Criticality Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2323–2336, 11 2018. URL: https://ieeexplore.ieee.org/document/8493604/, doi:10.1109/TCAD.2018.2857379.

[17] Hyoseung Kim, Dionisio De Niz, Bjrn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding memory interference delay in COTS-based multi-core systems. In *Real-Time Technology and Applications - Proceedings*, 2014. doi:10.1109/RTAS.2014.6925998.

[18] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, CGO*, 2004. doi:10.1109/CGO.2004.1281665.

[19] Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, Rodolfo Pellizzoni, Emiliano Betti Marco Cesati Marco Caccamo Renato Mancuso, Roman Dudko, and Rodolfo Pellizzoni. Real-Time Cache Management Framework for Multi-core Architectures. *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013. doi:10.1109/RTAS.2013.6531078.

[20] Renato Mancuso, Roman Dudko, and Marco 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*, pages 1–10. IEEE, 8 2014. URL: http://ieeexplore.ieee.org/document/6910515/, doi:10.1109/RTCSA.2014.6910515.

[21] Joel Matějka, Bjrn Forsberg, Michal Sojka, Zdenk Hanzálek, Luca Benini, and Andrea 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*, pages 11–20, New York, New York, USA, 2018. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=3178442.3178444, doi:10.1145/3178442.3178444.

[22] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio Buttazzo. Memory-processor co-scheduling in fixed priority systems. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems - RTNS '15*, pages 87–96, New York, New York, USA, 2015. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=2834848.2834854, doi:10.1145/2834848.2834854.

[23] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for COTS-based embedded systems. In *Real-Time Technology and Applications - Proceedings*, 2011. doi:10.1109/RTAS.2011.33.

[24] Benjamin Rouxel, Steven Derrien, and Isabelle Puaut. Tightening Contention Delays While Scheduling Parallel Applications on Multi-core Architectures. *ACM Transactions on Embedded Computing Systems*, 16(5s):1–20, 9 2017. URL: http://dl.acm.org/citation.cfm?doid=3145508.3126496, doi:10.1145/3126496.

[25] M.R. 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. URL: http://drops.dagstuhl.de/opus/volltexte/2017/7175/, doi:10.4230/LIPIcs.ECRTS.2017.24.

[26] Muhammad R. Soliman and Rodolfo Pellizzoni. Data Scratchpad Prefetching for Real-time Systems. Technical report, UWSpace, 5 2017. URL: https://uwspace.uwaterloo.ca/handle/10012/11837.

[27] Rohan Tabish, Renato Mancuso, Saud Wasly, Ahmed Alhammad, Sujit S. Phatak, Rodolfo Pellizzoni, and Marco Caccamo. A Real-Time Scratchpad-Centric OS for Multi-Core Embedded Systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11. IEEE, 4 2016. URL: http://ieeexplore.ieee.org/document/7461321/, doi:10.1109/RTAS.2016.7461321.

[28] Shelby Thomas, Chetan Gohkale, Enrico Tanuwidjaja, Tony Chong, David Lau, Saturnino Garcia, and Michael Bedford Taylor. CortexSuite: A synthetic brain benchmark suite. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 76–79. IEEE, 10 2014. URL: http://ieeexplore.ieee.org/document/6983043/http://cseweb.ucsd.edu/groups/bsg/https://bitbucket.org/taylor-bsg/cortexsuite, doi:10.1109/IISWC.2014.6983043.

[29] Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming Non-Blocking Caches to Improve Isolation in Multicore Real-Time Systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12. IEEE, 4 2016. URL: http://ieeexplore.ieee.org/document/7461361/, doi:10.1109/RTAS.2016.7461361.

[30] Bryan C. Ward, Jonathan L. Herman, Christopher J. Kenna, and James H. Anderson. Outstanding Paper Award: Making Shared Caches More Predictable on Multicore Platforms. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 157–167. IEEE, 7 2013. URL: http://ieeexplore.ieee.org/document/6602097/, doi:10.1109/ECRTS.2013.26.

[31] Saud Wasly and Rodolfo Pellizzoni. A Dynamic Scratchpad Memory Unit for Predictable Real-Time Embedded Systems. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 183–192. IEEE, 7 2013. URL: http://ieeexplore.ieee.org/document/6602099/, doi:10.1109/ECRTS.2013.28.

[32] Saud Wasly and Rodolfo Pellizzoni. Hiding memory latency using fixed priority scheduling. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 75–86. IEEE, 4 2014. URL: http://ieeexplore.ieee.org/document/6925992/, doi:10.1109/RTAS.2014.6925992.

[33] Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Emiliano Betti, and Marco Caccamo. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Systems*, 48(6):681–715, 11 2012. URL: http://link.springer.com/10.1007/s11241-012-9158-9, doi:10.1007/s11241-012-9158-9.

[34] Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Heechul Yun, and Marco Caccamo. Global Real-Time Memory-Centric Scheduling for Multicore Systems. *IEEE Transactions on Computers*, 65(9):2739–2751, 9 2016. URL: http://ieeexplore.ieee.org/document/7328709/, doi:10.1109/TC.2015.2500572.