PREM-Based Optimal Task Segmentation Under Fixed Priority Scheduling

Muhammad R. Soliman
University of Waterloo, Ontario, Canada
mrefaat@uwaterloo.ca

Rodolfo Pellizzoni
University of Waterloo, Ontario, Canada
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 are 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 approach.

2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases PREM, LLVM, scratchpad memory, scheduling, program segmentation

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.4

Funding This work was supported in part by NSERC and CMC Microsystems.

1 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 [17, 30, 16].

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 [22] 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 [33, 31, 32, 2, 1, 3, 27, 34, 19, 20, 12, 7, 21, 9, 10, 23, 4] 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. 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 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 2 summarizes the required background on PREM and related work. Section 3 introduces our new conditional PREM model and extends the existing schedulability analysis to cover such model. Section 5 then shows how to obtain an optimal segmentation for a given task set, while Section 4 describes our employed compilation framework, and our program segmentation algorithm based on such framework. Section 6 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 7.

2 Background and Related Work

In this section, we introduce existing research based on PREM and discuss required background and system assumptions. Note that while other predictable management approaches for local memories exist in the literature, we limit ourselves to PREM-based solutions due to space limitations. We consider a MPSoC platform comprising a set of possibly heterogeneous processors. Each processor has a fast private local memory in the form of a last level cache or ScrachPad Memory (SPM); all processors share the same main memory. As discussed in Section 1, the goal of PREM is to create a contentionless memory schedule. While the seminal work in [22] first proposed to split the execution of each application into a memory and a computation phase, the approach has been refined in successive works [32, 2] 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. 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. Second, it might be

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

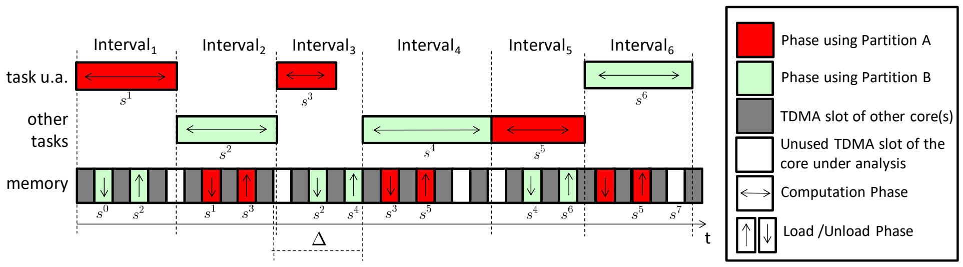


Figure 1 Example: TDMA memory schedule with $M = 2$ cores.

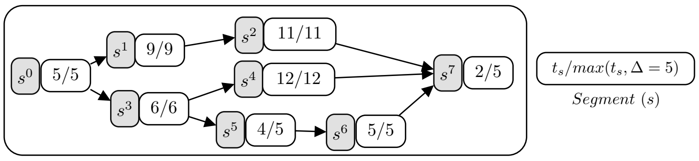


Figure 2 Example segment DAG (s^0 is s^{begin} and s^7 is s^{end}).

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 [22, 32, 9, 20] split a task into a sequence of PREM segments, where each segment has its own memory and computation phases and is executed non-preemptively.

2.1 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 [32, 13, 12, 27], at the cost of halving the available local memory space. As an example, we detail the approach in [32, 27], 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.) 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 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 $\Delta = \sigma \cdot (2M + 1)$: as again shown in Interval$_3$, the previous interval can finish right after the beginning of a TDMA slot assigned to the core under analysis, forcing that slot to be wasted. 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 as the computation phase of a segment and the memory phase of the next one cannot be executed in parallel: in general, the data required by a segment might not 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. 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 [33, 34, 21] 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.

2.2 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 [32, 7, 27, 4]. The seminal work in [22] 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 [13, 12] 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 [19] 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 [24], 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 4 to determine the data to load in each segment. Note that [24] only deals with a single-task, single processor case, and does not segment the program based on PREM.

The closest related work is [20], 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 3 and show in Section 6, 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 3 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.

3 Task 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. 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: $D_i \leq T_i$. 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 DAG could have many 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 $P' \succeq P$ iff: $P'.L \geq P.L$ and $P'.I \geq P.I$ and $P'.end \leq P.end$. If neither $P' \succeq P$ nor $P \succeq P'$ holds, we say that the two paths are incomparable.*

Since the $\succeq$ 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 [2] 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. While $P'.end = P.end$ for two paths belonging to the same DAG, we can also use Definition 1 to compare two DAGs for the same program.

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

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

[2] 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.

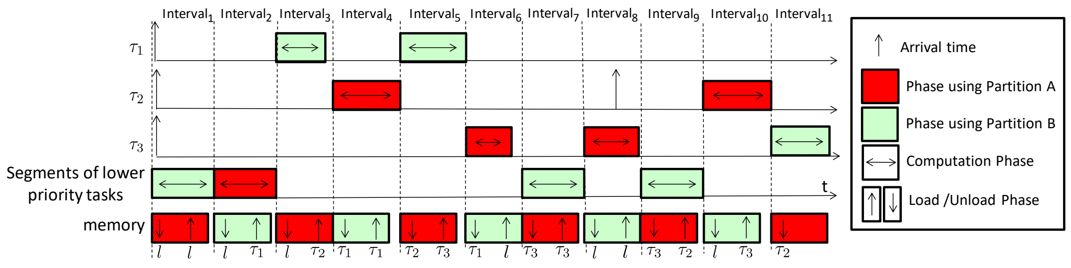


Figure 3 Example critical instant.

3.1 Schedulability Analysis and Preliminaries

We now consider a partitioned system with fixed per-task priority, and extend the analysis in [32, 27] 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 $\Gamma = \{\tau_1, \ldots, \tau_N\}$ represent 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 introduced in Section 2.1, where each SPM is divided in two partitions and the schedule is a sequence of scheduling intervals. 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.

The critical instant for a task under analysis τ_3 (the task arrival pattern that leads to the worst case response time for the task under analysis), as derived in [32], is depicted in Figure 3. Since scheduling decisions are only made when an interval starts, the worst case arrival pattern corresponds to the task under analysis and all higher priority tasks arriving 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: neither the task under analysis nor higher priority tasks can execute for the first 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. Albeit pessimistically, we then bound the blocking time as:

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

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

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 $P' \succeq P$. 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 $P.end$. 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 3.** *Task τ_i is schedulable if:*

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

▶ **Property 2.** *According to the analysis: (A) the schedulability of task τ_i depends on the maximum length $l_i^{l\,\max}$ of any segment of lower priority tasks $\tau_i + 1, \dots \tau_N$, but not on any other parameter of those tasks; (B) if τ_i is schedulable for a value l of $l_i^{l\,\max}$, 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 $\tau_1, \ldots, \tau_N$. However, we are interested in using the response time of tasks $\tau_1, \ldots, \tau_i$ in order to optimize the segmentation of task τ_{i+1}, hence $G_{i+1}, \ldots, G_N$ 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 used by our segmentation algorithm working on τ_{i+1}, as we detail in the next section: the algorithm considers a segmentation of τ_{i+1} to be valid only if its maximum segment length is no larger than $\overline{l_i^{l\,\max}}$. Note that in theory, one could determine $\overline{l_i^{l\,\max}}$ by performing a binary search over Equation 7. However, we show in technical report [26] that an alternative formulation based on the concept of scheduling points used in [5] 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 $G'_j \succeq G_j$. If τ_i is schedulable for G'_j according to the analysis, then it is also schedulable for G_j.*

Proof. Case $j = 1, \ldots i-1$: since $G'_j \succeq G_j$, 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 $G'_i \succeq G_i$, for each maximal path $P \in G_i.C$ there must exist a maximal path $P' \in G'_i.C$ such that $P' \succeq P$. 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. ◀

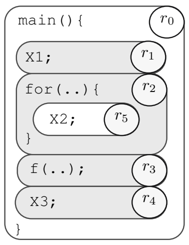
Property 3 shows that the dominance relation indeed corresponds to the notion of a DAG being better than another from a schedulability perspective. Hence, the objective of our segmentation algorithm is to find a set of "best" DAGs for a task based on Definition 2. Intuitively, the rest of the paper proceeds as follows. In Section 4 we present a segmentation algorithm that explores the set of all valid DAGs for a program, based on a set of constraints which include the maximum segment length, but quickly cuts dominating (i.e., worse) DAGs inspired by Property 3. Then, in Section 5 we show that, based on Property 2, we can invoke the segmentation algorithm on each task in priority order and obtain a set of DAGs (one for each task) that is optimal from a schedulability perspective.

4 Program Segmentation

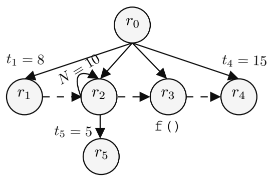
In this section, we show how a task is compiled into segments. We start by discussing the program structure based on regions. After that, we define valid segmentations according to our compiler framework, which is based on LLVM [18] and the work in [24]. Finally, we detail our algorithm, which segments the program and returns the set of all DAGs that could be optimal. Similarly to [24], 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.

4.1 Program Structure

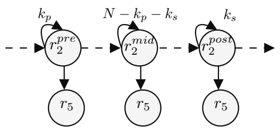
We adopt the region-based program structure introduced in [24] which represents each function in the program as a tree where each node is a *region*. A region encompasses a sub-graph of the program control flow graph (CFG) with a single entry and a single exit.



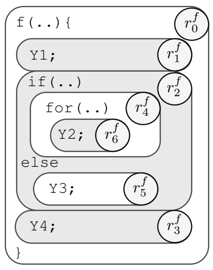
(a) `main` pseudo-code.



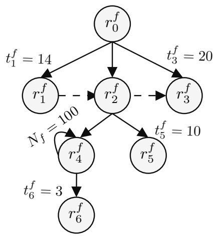
(b) `main` region tree.



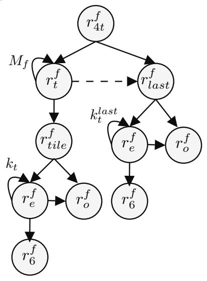
(c) Loop splitting of r_2.



(d) `f` pseudo-code.



(e) `f` region tree.



(f) Loop tiling of r_4^f.

Figure 4 Region representation ($\rightarrow$ ≡ parent-child / $\dashrightarrow$ ≡ sequential regions).

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 in the region tree has WCET t_r and a data footprint.

Figure 4 shows an example of a program with two functions: `main()` in Figure 4a and `f()` in Figure 4d. Figure 4b shows the region tree of `main()`. 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 representing 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()` . 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 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 n-level tiled loop nest, which means that the n outer loops are tiled, is divided into n tiling loops that iterate over tiles and n element loops that execute 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. 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 $k_t^{last} \leq k_t$ 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.

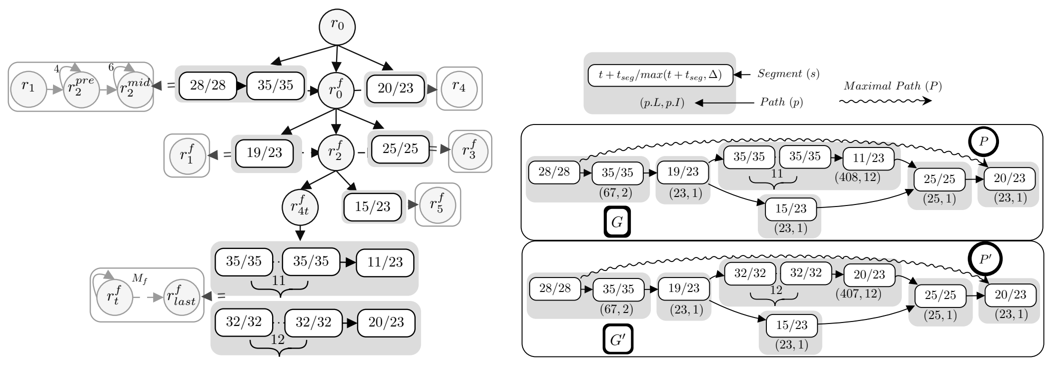
4.2 Valid Segmentation

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. The length constraint states that the length of each segment must be at most $l^{\max}$. As discussed in Section 3, 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 with WCET t_r is assigned to segment s, then $s.l = \max(t_r + t_{seg}, \Delta)$. If multiple regions in sequence are assigned to a segment s, then $s.l = \max\left(\left(\sum_r t_r\right) + t_{seg}, \Delta\right)$. 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 footprint and length 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.
- 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 loop iterations in a repeatable segment by inserting the segmentation instruction around the element loop.



(a) Segmented Tree. **(b)** DAGs.

Figure 5 Segmentation Example.

Based on the introduced constraints, we say that a set of regions in the tree constitute a *region sequence* if it comprises either: a single mergeable region, or a tiled loop, or a sequence of mergeable regions and/or splittable regions and tiles. Note that all regions in a sequence have the same parent. We say that a region sequence R is *maximal* if no children of its parent that is not in R can be merged with a region in R to form a segment. Our program segmentation produces a *segmented tree* $\mathcal{T}$, that is, a tree where every node is a set of segment paths $\mathcal{P}$. In particular, the segmented tree for a program is obtained by substituting region sequences in the region tree with sets of paths. A path $p \in \mathcal{P}$ for region sequence R is a sequence of segments, to which the regions and tiles in R are assigned. The segmented tree is derived inter-procedurally, i.e. for 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 $\mathcal{T}$ implicitly generates a set $\mathcal{G}$ of segment DAGs: each DAG in $\mathcal{G}$ is constructed by taking one path out of each path set and joining them according to the segmented tree hierarchy. A maximal path in the DAG thus comprises a sequence of paths $\{p_1, p_2, ..., p_n\}$ for some n, where p_1 encompasses s^{begin} and p_n encompasses s^{end} and hence the last region in the program r_{end}. Note that for a function that has multiple calls, a path that is chosen to construct a DAG from the path set of a region sequence in the function must be used for all the function calls as the region sequence represents the same code.

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 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.

4.3 Segmentation Algorithm

The example in Section 4.2 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 $G' \succeq G$, we can safely ignore G' based on Property 3. This is formally captured by the following definition.

▶ **Definition 4.** *Let $\mathcal{G}$ be the set of all valid DAGs for a program according to a set of constraints, and let $\mathcal{G}'$ be the set of DAGs returned by a segmentation algorithm for that program. We say that the algorithm preserves optimality iff for any program: $\mathcal{G}'$ is valid according to the constraints, and $\forall G \in \mathcal{G}, \exists G' \in \mathcal{G}' : G \succeq G'$.*

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. Note that the length, footprint and compilation constraints are implied in all the following algorithms whenever a region is checked to be mergeable, splittable, or tileable and whenever a segment is checked to be valid.

Algorithm 1 starts with a call to SegmentTask function. Then Segment(r_0) is called on r_0, the top level region of `main`, hence returning the segmented subtree for the whole program. Finally, a DAG set $\mathcal{G}$ is generated from the segmented tree and returned as a result of SegmentTask. Note that if r_0 is mergeable, then the segmented tree is composed of a single, maximal region sequence R that comprises r_0 only; hence, in this case we simply return a DAG with r_0 as its single segment.

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

Algorithm 1 Segmentation Algorithm.

```
1: function SegmentTask($\tau$)
2:     if $r_0$ is mergeable then
3:         Create DAG $G$ with a single segment comprising $r_0$, return $\mathcal{G} = \{G\}$
4:     Generate DAG set $\mathcal{G}$ from $\mathcal{T}$ = Segment($r_0$), return $\mathcal{G}$
5: function Segment($r$)
6:     Initialize $R = \emptyset$                                  ▷ A set of sequential regions.
7:     Initialize $\mathcal{T}$ to be the subtree whose root is $r$
8:     for all $r_c \in children(r)$ do
9:         if $r$ is sequential and $r_c$ is mergeable or splittable loop then
10:            Add $r_c$ to $R$
11:        else if $r_c$ is mergeable then                         ▷ $r$ is not sequential
12:            Replace $r_c$ with $\mathcal{P} = \{p\}$, where $p$ is single-segment path
13:        else
14:            Replace regions in $R$ with SegmentSequence($R$), empty $R$
15:            if $r_c$ is a tileable loop then
16:                Replace $r_c$ with Tile($r_c$).
17:            else if $r_c$ is a call to $f$ then
18:                Replace $r_c$ with Segment($r_0^f$)
19:            else
20:                Replace $r_c$ with Segment($r_c$)
21:    If $R \neq \emptyset$, replace regions in $R$ with SegmentSequence($R$)
22:    return $\mathcal{T}$
```

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, the obtained R is maximal. 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 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).

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

Note that Definition 5 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 6.** *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 $\forall k = 1...n : p'_k \succeq p_k$, then $P' \succeq P$.*

Proof. Note by construction $P.L = \sum_{k=1...n} p_k.L, P'.L = \sum_{k=1...n} p'_k.L$. From $p'_k \succeq p_k$ 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 $P' \succeq P$. ◀

▶ **Definition 7.** *Given two path sets $\mathcal{P}, \mathcal{P}'$ for the same region sequence R, we say that $\mathcal{P}'$ dominates $\mathcal{P}$ and write $\mathcal{P}' \succeq \mathcal{P}$ iff: $\forall p' \in \mathcal{P}', \exists p \in \mathcal{P} : p' \succeq p$, and if $r_{end} \in R$, 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 $\mathcal{P}' \succeq \mathcal{P}$.*

▶ **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 $\mathcal{P}' \succeq \mathcal{P}$.*

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 11 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 8.** *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 $G' \succeq G$.*

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` r_{end}, 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 6, we then proved that there must exist a maximal path P' such that $P' \succeq P$. By definition, this implies $G' \succeq G$, completing the proof. ◀

▶ **Lemma 9.** *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 DAG set generated from $\mathcal{T}$ preserves 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 8, 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 $G' \succeq G$. By Definition 4, this means that generating DAGs from $\mathcal{T}$ preserves optimality. ◀

Lemma 9 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 $\mathcal{P}$.

Algorithm 2 1-Level Tiling.

```
1: function TILE($r$)
2:     Compute $k_t^{max}$, $\mathcal{P} = \emptyset$
3:     for all $k_t \leq k_t^{max}$ do
4:         Generate $p(k_t)$ and add it to $\mathcal{P}$ if it is valid
5:     Filter $\mathcal{P}$ by removing dominating paths based on Definition 5
6:     return $\mathcal{P}$
```

▶ **Lemma 10.** *Consider a segmented tree $\mathcal{T}$ as in Lemma 9. Let $\overline{\mathcal{T}}$ denote the segmented tree obtained by replacing, for each maximal region sequence R in $\mathcal{T}$, the set $\mathcal{P}'$ of all valid paths with a set $\mathcal{P}$ such that $\mathcal{P} \subseteq \mathcal{P}'$ and $\mathcal{P}' \succeq \mathcal{P}$. Then the DAG set generated from $\overline{\mathcal{T}}$ preserves optimality.*

Proof. 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' through n region sequences: $P' = \{p'_1, ..., p'_k, ..., p'_n\}$. Since for all regions $\mathcal{P}' \succeq \mathcal{P}$, then for every p'_k there exists another path p_k in $\overline{\mathcal{T}}$ such that $p'_k \succeq p_k$, and furthermore $p'_n.end = p_n.end$ since the last region sequence in any maximal path must include the last region in the program r_{end}. By Lemma 6, this means that we can find a maximal path $P = \{p_1, ..., p_k, ..., p_n\}$ for $\overline{\mathcal{T}}$ such that $P' \succeq P$. 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 $P' \succeq P$. This implies $G' \succeq G$, and since by Lemma 9 $\mathcal{T}$ preserves optimality, it thus follows that the DAG set generated from $\overline{\mathcal{T}}$ also preserves optimality according to Definition 4. ◀

▶ **Theorem 11.** *If Properties 5, 6 hold, Algorithm 1 preserves optimality based on the footprint, length and compilation constraints.*

Proof. By construction, the algorithm creates 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 12, 16 or 21. At line 12, 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 16 and 21, the path set $\mathcal{P}$ is generated by calling either SEGMENTSEQUENCE(R) or TILE(r_c); by Properties 5, 6 and Lemma 10, in both cases $\mathcal{P} \subseteq \mathcal{P}'$ and $\mathcal{P}' \succeq \mathcal{P}$ hold. In summary, Lemma 10 applies to all maximal regions, hence the algorithm preserves optimality. ◀

4.3.1 Tiling Algorithm

We now discuss how to optimize the tile size for a 1-level tiled loop r, similarly to the example in Section 4.1. Note that while we present the case of 1-level tiling for simplicity, in practice 2-level tileable loops are common in embedded programs. Hence, our framework also implements a more general algorithm that can find tile sizes for 2-level tiling; due to space limitations, we detail it in the provided technical report [26].

Given an execution time for one iteration of t_1, a number of iterations N_r and a tile size k_t with $M = \lceil N_r/k_t \rceil - 1$ and $k_t^{last} = N_r - M * k_t$, tiling results in a path $p(k_t)$ comprising M segments of length $\max(\Delta, k_t * t_1 + t_{tile} + t_{seg})$, and one segment of length $\max(\Delta, k_t^{last} * t_1 + t_{tile} + t_{seg})$. Algorithm 2 simply iterates over k_t starting with $k_t^{\max}$, the maximum value of k_t such that the length of any segments in $p(k_t)$ is less than or equal to

$l_{\max}$ and its footprint is less than or equal to the SPM size. It then generates each path $p(k_t)$ and adds it to $\mathcal{P}$ if it is valid. Finally, based on Definition 5, it removes any path $p(k'_t)$ from $\mathcal{P}$ if there exists another path $p(k_t) \in \mathcal{P}$ with $p(k'_t) \succeq p(k_t)$. The following lemma then easily follows.

▶ **Lemma 12.** *Algorithm 2 satisfies Property 6.*

Proof. First note that r cannot be r_{end}, since the last region in a program must be a basic block and not a loop. 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, any path $p(k_t)$ with $k_t > k_t^{max}$ cannot be valid. Also since the algorithm adds all valid paths $p(k_t)$ with $k_t \leq k_t^{max}$ to $\mathcal{P}$, before executing line 5, $\mathcal{P}$ contains all valid paths for r. Since furthermore the filtering on line 5 respects Definition 7, Property 6 holds. ◀

4.3.2 Region Sequence Segmentation

Next, we consider Algorithm 3 that generates a path set $\mathcal{P}$ from a region sequence R. The algorithm iterates over each region r in R, incrementally constructing a set of paths $\bar{\mathcal{P}}$ for the sub-sequence that includes all regions from the beginning of R up to r. For simplicity of notation, for a path $\bar{p} \in \bar{\mathcal{P}}$, we use $\bar{p}.t_{end}$ to denote the WCET of the regions included in the last segment of $\bar{p}$, such that $\bar{p}.end = \max(\Delta, \bar{p}.t_{end} + t_{seg})$. At each step of the algorithm, for a region r with computation time t_r, a new set of paths is constructed by taking each path $\bar{p}$ in $\bar{\mathcal{P}}$ and adding r to it. Note that when doing so, two new paths might be generated in the following way:

1. Add a new segment comprising r to $\bar{p}$ to construct a new 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.
2. Add r to the last segment [3] of $\bar{p}$ to construct a new 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 valid according to the constraints; so, it is only added to the new set of paths if valid.

The process continues until after we reach the last region in R; at that point, we return the final path set $\bar{\mathcal{P}}$.

Note that 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 and tiling greatly increases the number of paths. We tackle this complexity in the extended technical report [26] by introducing a set of conditions that allows us to prune the generated paths from $\bar{\mathcal{P}}$ at each step while preserving optimality and hence improve the segmentation time.

In details, Algorithm 3 traverses the regions in R in topological order. If the current region r is not a splittable loop, then new paths $\bar{p}_m$ and $\bar{p}_n$ are generated by adding r to each previous path in function CreatePaths. The new paths are placed in $\bar{\mathcal{P}}_{next}$, before becoming the set of paths $\bar{\mathcal{P}}$ at the next iteration. If r is a splittable loop, then before generating a new 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), paths $\bar{\mathcal{P}}_{loop}$ for r_p are generated using CreatePaths, then r_t is tiled and each tile path is sequenced with the paths in $\bar{\mathcal{P}}_{loop}$. Then, paths are created using k_s for all paths in $\bar{\mathcal{P}}_{loop}$. All paths $\bar{\mathcal{P}}_{loop}$ are finally accumulated in $\bar{\mathcal{P}}_{next}$. After traversing all regions in R, the paths in $\bar{\mathcal{P}}$ are filtered using Definition 5 if $r_{end} \notin R$. Otherwise, all the generated paths are kept. Finally, the path set $\bar{\mathcal{P}}$ for R is returned.

[3] Note that adding a region r to a segment s implies that the footprint of the resulting segment is the union of the footprints of r and of the regions in s.

Algorithm 3 Segment a Region Sequence.

```
Require: A region sequence $R$ and the last basic block region $r_{end}$
 1: function SEGMENTSEQUENCE($R$)
 2:     $\bar{\mathcal{P}} = \{\bar{p} = \emptyset\}$, $\bar{\mathcal{P}}_{next} = \emptyset$
 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:                 $\bar{\mathcal{P}}_{loop}$ = CREATEPATHS($r_p, \bar{\mathcal{P}}$)
 8:                 $\bar{\mathcal{P}}_{loop}$ = generate all paths by joining $\bar{\mathcal{P}}_{loop}$ with TILE($r_t$)
 9:                 $\bar{\mathcal{P}}_{loop}$ = CREATEPATHS($r_s, \bar{\mathcal{P}}_{loop}$)
10:                 $\bar{\mathcal{P}}_{next} = \bar{\mathcal{P}}_{next} \bigcup \bar{\mathcal{P}}_{loop}$
11:         else                 ▷ $r$ is a mergeable region that is not a splittable loop
12:             $\bar{\mathcal{P}}_{next}$ = CREATEPATHS($r, \bar{\mathcal{P}}$)
13:         $\bar{\mathcal{P}} = \bar{\mathcal{P}}_{next}$, $\bar{\mathcal{P}}_{next} = \emptyset$
14:     If $r_{end} \notin R$, Filter $\bar{\mathcal{P}}$ by removing dominating paths based on Definition 5
15:     return $\bar{\mathcal{P}}$
16: function CREATEPATHS($r, \bar{\mathcal{P}}$)
17:     $\bar{\mathcal{P}}_{tmp} = \emptyset$
18:     for all $\bar{p}$ in $\bar{\mathcal{P}}$ do
19:         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
20:         Create $\bar{p}_n$ by adding new segment using $r$ to $\bar{p}$, add $\bar{p}_n$ to $\bar{\mathcal{P}}_{tmp}$
21:     return $\bar{\mathcal{P}}_{tmp}$
```

▶ **Lemma 13.** *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. Therefore, it must hold $\mathcal{P} \subseteq \mathcal{P}'$. It remains to show that if a path p' for R is discarded (i.e., the path is in $\mathcal{P}'$ but not in $\mathcal{P}$), then there exists a path p such that $p' \succeq p$, and if $r_{end} \in R$, then $p'.end = p.end$. A path in $\mathcal{P}'$ can be discarded if: (1) Algorithm 2 removes a tiling solution; (2) the path is filtered based on Definition 5.

Case (1): Assume that Algorithm 2 removes a path p'_t from the returned path set for a tiled loop; by Property 6, it must return another path p_t such that $p'_t \succeq p_t$. Then if we consider any path p' for R of the form $p' = \{p_1, ..., p'_t, ..., p_n\}$, there must exist another path $p = \{p_1, ..., p_t, ..., p_n\}$, and by Lemma 6, it must hold $p' \succeq p$. Next consider the case $r_{end} \in R$: 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): Note this applies only if r_{end} is not contained in R. It thus suffices to notice that by Definition 5 $p' \succeq p$ must hold. ◀

5 Optimal Task Set Segmentation

Based on the analysis Properties 2, 3 introduced in Section 3.1 and segmentation Algorithm 1, we now show that we can obtain an optimal task set segmentation using Algorithm 4. The algorithm recursively calls function SEGMENTTASKSET for task index i from 1 to N

Algorithm 4 Task Set Segmentation.

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

by keeping track of the DAGs $G_1, \dots G_{i-1}$ 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 a DAG set $\mathcal{G}_i$ for τ_i. If $i < N$, the function iterates over all possible $G_i \in \mathcal{G}_i$; 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} ($\mathcal{G}_i$ 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 $G_N \in \mathcal{G}_N$, in which case we terminate by finding a solution $\{G_1, \dots, G_N\}$. 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 compilation constraints in Section 4.2. We start with a corollary.

▶ **Corollary 14.** *Consider two DAGs G_j, G'_j for task τ_j where $1 \le j \le i$ and $G'_j \succeq G_j$. 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 $\overline{l_i^{l\max}} \ge \overline{l_i^{l\max}}'$.*

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 $l_i^{l\max} \le \overline{l_i^{l\max}}'$ for G'_j, based on Property 3 it is also schedulable with $l_i^{l\max} \le \overline{l_i^{l\max}}'$ for G_j; this implies $\overline{l_i^{l\max}} \ge \overline{l_i^{l\max}}'$. ◀

▶ **Theorem 15.** *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 $G'_1, \dots, G'_N$ 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 $G_1, \dots, G_N$ 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 SegmentTaskSet that results in a DAG set $G_1, \dots G_i$ such that $G'_j \succeq G_j$ for every $j = 1 \dots i$; 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 $j = 1 \dots i$ 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 $G'_1, \dots, G'_j$, and $\overline{l_j^{l\max}}$ is the same value for DAGs $G_1, \dots, G_j$.

Base Case ($i = 1$): note $l^{\max} = +\infty$, meaning that only the footprint and compilation constraints apply when invoking Algorithm 1. Hence, by Definition 4 the algorithm must find a DAG $G_1 \in \mathcal{T}_1$ such that $G'_1 \succeq G_1$. By Corollary 14, this also implies $\overline{l_1^{l\max}}' \leq \overline{l_1^{l\max}}$.

Induction Step ($i = 2 \dots N$): consider the recursive call sequence that results in $G'_j \succeq G_j$ and $\overline{l_j^{l\max}}' \leq \overline{l_j^{l\max}}$ for each $j = 1 \dots i-1$ (such sequence exists by induction hypothesis); we have to show that we can find a DAG $G_i \in \mathcal{G}_i$ such that $G'_i \succeq G_i$ 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 $G'_1, \dots, G'_N$, 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 4, this implies that Algorithm 1 finds a valid DAG G_i with maximum segment length $l^{\max}$ such that $G'_i \succeq G_i$. $\overline{l_i^{l\max}}' \leq \overline{l_i^{l\max}}$ then again follows by Corollary 14. ◀

Complexity. Since it iterates over all $G_i \in \mathcal{G}_i$, Algorithm 4 is exponential. Intuitively, it might seem sufficient to only use the DAG in $\mathcal{G}_i$ 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 6, in practice this results in an acceptable runtime considering the algorithm is an offline optimization.

Composability and Generality. As we (re-)compile all tasks, our approach requires the source code of all applications in the system. Since Algorithm 4 segments tasks in priority order, any code change in a program will not affect higher priority tasks; however, 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 6 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.

6 Evaluation

We implemented our segmentation framework using LLVM to analyze and generate the region trees for the program as in [24], 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 [25]. For the data SPM, we vary its size from 4 kB to 512 kB. For

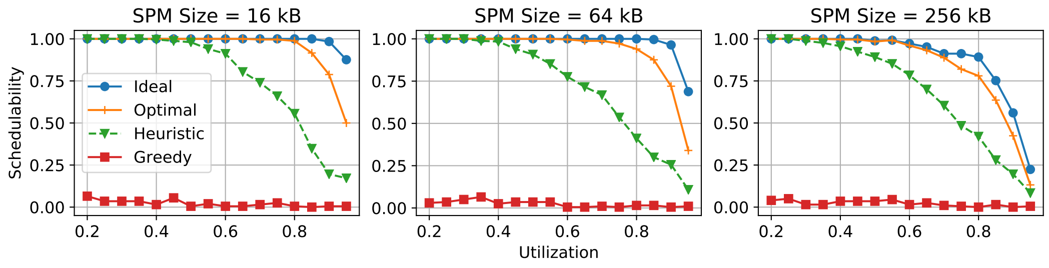
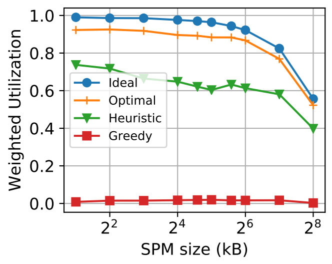
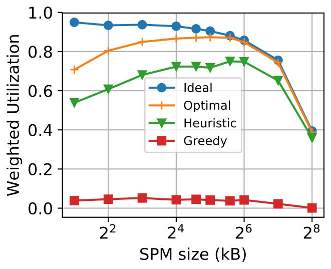


Figure 6 Schedulability vs Utilization.



(a) $t_{seg} = 100$.



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

Figure 7 Weighted Utilization VS SPM Size.

Table 1 Benchmarks (LOC: lines of code).

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

the memory transfer, we assume that the DMA speed is 1 cycle per word (4 bytes) [4]. 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 [29], TACLeBench [11] and CortexSuite [28] benchmark suites. The application are chosen to represent a variety of sizes, complexities and data footprints (see Table 1). 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 [6], 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 [5]. The *greedy* and *heuristic* algorithms do not depend on Algorithm 4 to drive

[4] In the extended technical report [26], we discuss the effect of the DMA speed on the system schedulability
[5] Note that the ideal algorithm is still compliant with PREM, i.e. the next segment has to be decided and loaded while the current segment is executing. Hence, to our understanding we cannot employ existing scheduling analyses for limited-preemptive task sets [8].

the segmentation of each task based on the schedulability analysis. The greedy algorithm resembles the algorithm used in [20] 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 6 shows the system schedulability for the four algorithms for SPM sizes of 16, 64 and 256 kB. 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 7a 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 7a 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 7b 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.

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. In the extended technical report [26], we discuss how the algorithm time scales with the number of tasks in a task set in more details.

7 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 Ahmed Alhammad and Rodolfo Pellizzoni. Schedulability analysis of global memory-predictable scheduling. In *Proceedings of the 14th International Conference on Embedded Software - EMSOFT '14*, New York, New York, USA, 2014. ACM Press.

2 Ahmed Alhammad and Rodolfo Pellizzoni. Time-predictable execution of multithreaded applications on multicore systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014*, New Jersey, 2014. IEEE Conference Publications.

3 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*. IEEE, 2015.

4 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)*. IEEE, 2016.

5 E. Bini and G.C. Buttazzo. Schedulability analysis of periodic fixed priority systems. *IEEE Transactions on Computers*, 53(11), 2004.

6 Enrico Bini and Giorgio C. Buttazzo. Measuring the Performance of Schedulability Tests. *Real-Time Systems*, 30(1-2), 2005.

7 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)*. IEEE, 2015.

8 Giorgio C. Buttazzo, Marko Bertogna, and Gang Yao. Limited Preemptive Scheduling for Real-Time Systems. A Survey. *IEEE Transactions on Industrial Informatics*, 2013.

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*, New York, New York, USA, 2017. ACM Press.

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)}*, February 2014.

11 Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn 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.

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)*. IEEE, 2018.

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*. IEEE, 2017.

14 TOBIAS GROSSER, ARMIN GROESSLINGER, and CHRISTIAN LENGAUER. Polly — Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters*, 22(04), 2012.

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.

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), 2018.

17 Hyoseung Kim, Dionisio De Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding memory interference delay in COTS-based multi-core systems. In *Real-Time Technology and Applications - Proceedings*, 2014.

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.

19 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*. IEEE, 2014.

20 Joel Matějka, Björn Forsberg, Michal Sojka, Zdeněk 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*, New York, New York, USA, 2018. ACM Press.

21 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*, New York, New York, USA, 2015. ACM Press.

22 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.

23 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), 2017.

24 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.

25 Muhammad R. Soliman and Rodolfo Pellizzoni. Data Scratchpad Prefetching for Real-time Systems. Technical report, UWSpace, 2017.

26 Muhammad R Soliman and Rodolfo Pellizzoni. Optimal Task Segmentation for PREM-based Systems Under Fixed Priority Scheduling. Technical report, University of Waterloo, Canada, 2019. URL: `http://ece.uwaterloo.ca/~rpellizz/techreps/optimal_seg_tech_report.pdf`.

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)*. IEEE, 2016.

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)*. IEEE, 2014.

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

30 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)*. IEEE, 2016.

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*. IEEE, 2013.

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)*. IEEE, 2014.

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), 2012.

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), 2016.