Automated Compilation Framework for Scratchpad-based Real-Time Systems

by

Muhammad Refaat Sedky Soliman

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2019

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Isabelle Puaut Professor, University of Rennes I
Supervisor(s):	Rodolfo Pellizzoni Associate Professor, University of Waterloo
Internal Member:	Mark Aagaard Associate Professor, University of Waterloo
Internal Member:	Hiren Patel Associate Professor, University of Waterloo
Internal-External Member:	Ondřej Lhoták Associate Professor, University of Waterloo

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contribution

In what follows is a list of publications which I have co-authored and used their content in this dissertation. For each publication, I present a list of my contributions. The use of the content, from the listed publications, in this dissertation has been approved by all co-authors.

1. 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) - Best paper award [131].

    - Introduced the refined region-based program structure.
    - Proposed a WCET-driven automated allocation scheme for data scratchpad with prefetching.
    - Developed a scratchpad controller and a compiler tool for the allocation scheme.
    - Designed and executed the experiments, and analyzed the results.

2. M. R. Soliman and R. Pellizzoni, "PREM-based Optimal Task Segmentation under Fixed Priority Scheduling," in 31th Euromicro Conference on Real-Time Systems (ECRTS 2019) [132]

    - Introduced the conditional task model for PREM-based system and extended the schedulability analysis.
    - Developed a set of optimal segmentation algorithms for the fixed-size DMA model under fixed priority scheduling.
    - Designed and executed the experiments, and analyzed the results.

3. M. R. Soliman, G. Gracioli, R. Tabish, R. Pellizzoni and M. Caccamo, "Segment Streaming for the Three-Phase Execution Model: Design and Implementation", under review

    - Designed the segment streaming for the three-phase execution model.
    - Proposed a OS-level programming interface and its implementation.
    - Extended the schedulability analysis for the case of fixed-size DMA model.
    - Designed and executed the schedulability tests, and analyzed the results.

Abstract

ScratchPad Memory (SPM) is highly adopted in real-time systems as it exhibits a predictable behaviour. SPM is software-managed by explicitly inserting instructions to move code and data transfers between the SPM and the main memory. However, it is a tedious job to decide how to manage the SPM and to manually modify the code to insert memory transfers. Hence, an automated compilation tool is essential to efficiently utilize the SPM. Another key problem with SPM is the latency suffered by the system due to memory transfers. Hiding this latency is important for high-performance systems. In this thesis, we address the problems of managing SPM and reducing the impact of memory latency. To realize the automation of our work, we develop a compilation framework based on the LLVM compiler to analyze and transform the program code. We exploit our framework to improve the performance of the execution of single and multi-tasks in real-time systems. For the single task execution, Worst-Case Execution Time (WCET) is of great importance to assure correct and safe behaviour of the system. So, we propose a WCET-driven allocation technique for data SPM that employs software prefetching to efficiently manage the SPM and to overlap the memory transfer and the task execution in a predictable way. On the other hand, multi-tasking requires the system to be schedulable such that all the tasks can meet their timing requirements. However, executing multiple tasks on a multi-processor platform suffers from the contention of the accesses to the shared main memory. To avoid the contention, several scheduling techniques adopted the 3-phase execution model which executes the task as a sequence of memory and computation phases. This provides the means to avoid the contention as well as to hide the memory latency by using a Direct Memory Access (DMA) engine. Executing memory transfers using the DMA allows overlapping the memory transfers with the computations on the processor. Using the 3-phase model in systems with limited sizes of local SPM may necessitate a segmentation of the task. Automating the segmentation process is necessary especially for systems with large task sets. Hence, we propose a set of efficient segmentation algorithms that follow the 3-phase execution model. The application of these algorithms shows a significant improvement in the system schedulability. For our segmentation algorithms to be more applicable, we extend the 3-phase model to allow programs with multiple paths represented as conditional Directed Acyclic Graphs (DAGs), unlike the previous works that targeted sequential programs. We also introduce a multi-steaming model to exploit the benefits of prefetching by overlapping the memory and computation phases of the same task, which was not allowed in the previous approaches. By combining the automated compilation with the proposed algorithms, we are able to achieve our goal to efficiently manage data SPM in real-time systems.

Acknowledgements

First and foremost, I am grateful to Allah for empowering me to complete this thesis. Without his help, I would not have the ability to reach this stage in my life.

I would like to seize this opportunity to express my deepest gratitude to Prof. Rodolfo Pellizzoni (my supervisor), not only for his help and guidance through my Ph.D, but also for his sincere support at a personal level. He impacted my life and career in many ways and I was lucky to be his student.

I would like to thank my committee members: Professor Isabelle Puaut, Professor Mark Aagaard, Professor Hiren Patel, and Professor Ondřej Lhoták for taking the time and effort to participate in my examination committee and provide me with valuable feedback.

I would like thank my friends for the constructive discussions and their help and feedback: Mohamed Hassan, Saud Wasly, Ahmed Alhammad, and Michael Guo.

Foremost, I cannot put into words how grateful I am for my parents, Magda and Refaat, for their unconditional love, continuous support, endless sacrifices, and countless prayers. Thanks for my sister, Yasmeen, for believing in me and supporting me. Thank you, my wife Noura, for being patient and standing by my side through the good and the tough times.

Dedication

Indeed, my prayer, my rites of sacrifice, my living and my dying are for Allah, Lord of the worlds. [Quran 6:162]

To my parents, Magda and Refaat.

Table of Contents

List of Tables

List of Figures

List of Acronyms

SESE Single Entry Single Exit

PST Program Structure Tree

WCET Worst-Case Execution Time

WCEP Worst-Case Execution Path

ACET Average-Case Execution Time

SPM ScratchPad Memory

CFG Control Flow Graph

ICFG Inter-procedural Control Flow Graph

DMA Direct Memory Access

DAG Directed Acyclic Graph

IR Intermediate Representation

ILP Integer Linear Programming

LRU Least Recently Used

COTS Commercial-Off-The-Shelf

IPET Implicit Path Enumeration Technique

PREM PRedictable Execution Model

OS Operating System

API Application Programming Interface

Chapter 1

Introduction

Real-time systems are essential in many domains such as automotive, avionics, telecommunication infrastructures, medical devices, security systems, robotics, fabrication machines, and military applications [98]. A growing domain of real-time applications is Internet of Things (IoT) where smart devices are able to communicate, share information, and interact with their environment. With the rise of autonomous systems, the complexity of critical functionalities has been increasing, demanding high-performance real-time architectures and algorithms to cope with these needs.

In real-time systems, the correctness of the system depends on its logical functionality as well as the timing. Timing constraints are imposed to avoid unacceptable results or to maintain the quality of service. Hence, the execution time of a task running on a real-time platform must be bounded. The bound is derived using static or measurement-based analysis to estimate the Worst-Case Execution Time (WCET) [161] which accounts for the worst-case scenario to assure the predictability of the system. In a multi-tasking system, a schedulability analysis uses the WCET of each task to verify the timing constraints of a real-time system [24]. For a multi-tasking system, timing validation for the set of tasks running on the system is necessary. Each task in the system has a deadline such that the execution of a job of this task must finish before the deadline. A feasible task set means that all jobs of all tasks can meet their deadlines under all combinations of job arrivals of different tasks. Different schedulability algorithms are used to derive schedules for a task set execution. A schedulability analysis decides whether a task set is schedulable using a schedulability algorithm by ensuring that the WCET of any possible job in the system can meet its deadline. Architectural features like memory hierarchy, interconnect protocols and pipelining impact the ability to derive tight bounds on the WCET. The memory hierarchy is a key factor for both performance and predictability of the system. This is especially

true on architectures with multiple processors, because processors (or cores) share hardware resources, such as cache memory hierarchy, buses, DRAM, and I/O peripherals. Therefore, operations performed by one processing unit can result in unregulated contention at the level of any shared resource and thus unpredictably delay the execution of a task running on a different core.

Embedded systems usually comprise on-chip and off-chip memories. On-chip memories are small and fast compared to off-chip memories which are large and slow. Combining on-chip and off-chip memories in a multi-level memory hierarchy improves the performance [112] by bridging the speed gap between the processor and the off-chip main memory. Caches are the most common form of on-chip memory. They have been used in general purpose systems for a long time as they improve the average performance significantly. Caches employ a set of heuristics that exploit the temporal and spacial locality of memory accesses to keep the data that most likely will be accessed in the near-future. The execution time of a memory instruction in a cache-based system depends on whether the accessed data is a cache hit or a cache miss. The heuristic behavior of caches increases the variability in the execution time as the cache behavior depends on the history of the memory accesses. Assuming that every memory access is a cache miss to account for the worst case leads to a very pessimistic estimation of WCET. Static cache analysis tries to predict the cache behavior to be able to tighten the WCET bound [96]. The complexity of cache analysis significantly increases for multi-tasking systems and multi-core architectures as system resources are shared. Several works have been proposed to enforce a more deterministic behavior in real-time systems using cache partitioning and cache locking [56]. In the context of real-time systems, there has been significant attention to ScratchPad Memory (SPM) as an alternative to caches [155]. SPM is a small on-chip memory that is mapped to the address space of the processor. Unlike caches, SPM has to be explicitly managed by the software to move the data between the SPM and the main memory. Hence, SPM is highly predictable as its content is under software control. However, explicit management of SPM is challenging as it requires the programmer to be aware of the underlying hardware and manually embed the required managing instructions in the code. Several allocation algorithms have been proposed to automatically manage the SPM for both general purpose systems and real-time systems. An allocation mechanism determines the content of the SPM based on the SPM size and the platform configuration. The allocation of data in SPM requires explicit movement of the data to/from main memory. The time for these transfers is another challenge for SPM management. These transfers are usually performed using a Direct Memory Access (DMA) engine because of its efficiency.

In this thesis, our goal is two-fold: to automate the management of the SPM in real-time systems, and to efficiently hide the memory transfers by overlapping the DMA time and

the computation time. The first goal is achieved by introducing a compilation framework to analyze, optimize and transform a program based on the LLVM compiler. The second goal has two targets: single task execution and multi-tasking systems. For the execution of a single task, we use software prefetching to prefetch/write-back data in parallel with the computation of the task. For a multi-tasking system, this can be achieved using the 3-phase model [152] by overlapping the DMA time of one task with another task. The 3-phase model divides the execution of the task to memory and computation phases. Hiding memory time using the 3-phase model has been explored in many works. However, the previous works have two shortcomings: 1) the task is assumed to fit in the SPM or manually segmented by the programmer, 2) the DMA time of a segment of the task cannot be overlapped with the computation time of another segment of the same task. Hence, we tackle these two shortcomings in this thesis by: extending the execution model to allow streaming segments of the same task, i.e. execute them back-to-back, and proposing algorithms that consider both the task segmentation and the scheduling of the task set to obtain efficient segmentation and to improve the system schedulability.

1.1 Data SPM Management with Software Prefetching

Although using on-chip memory avoids frequent accesses to the main memory, the performance of embedded systems can be significantly affected by main memory latency due to the need to move the data between on-chip memory and main memory. While novel devices promise much increased memory bandwidth, in particular through DRAM stacking [1], the access latency for DRAM main memory has largely remained similar in recent years. In the context of general purpose systems, this problem is typically addressed through per-task prefetching techniques to bring content to on-chip memory before it is used and avoid stalling the processor. Cache prefetching has been extensively researched in the architecture and compilers communities [103]. Prefetching techniques incorporate hardware and/or software to hide cache miss latency by attempting to load cache lines from main memory before they are accessed by the program. The essence of these techniques is speculation of the data locality and the cache behavior, which makes them unsuitable to provide WCET guarantees for real-time programs.

Using prefetching techniques for SPM can provide similar benefit to hide memory latency. Current SPM management techniques for real-time systems do not solve the fundamental memory latency problem, because they generally assume that the core is stalled while the content of on-chip memory is reloaded.

We target the development of a compiler-directed prefetching scheme that optimizes

the allocation of program code and data in on-chip memory with the objective to minimize the WCET. For this phase, we focus on single program running on single core.

In Chapter 4, we present a novel prefetching scheme for program data. Our proposed method employs a Direct Memory Access (DMA) controller to move data between on-chip memory and main memory. Compared to related work, we do not stall the program while transferring data; instead, we rely on static program analysis to determine when data is used in the program, and we prefetch it into on-chip memory ahead of its use so that the time required for the DMA transfer can be *overlapped* with the program execution. The allocation and prefetching framework is automated in the compiler.

1.2 Task Segmentation and Scheduling for Multi-tasking Systems

Shared resources in Multi-Processor Systems-on-a-Chip (MPSoCs) represent a challenge for predictability in real-time systems. Main memory shared by all processing elements on the chip can cause significant performance degradation. For real-time systems, the contention for memory access among multiple processors may result in extremely high worst-case latency [66, 83, 141] which counter the benefit of using multiple processors. 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 3-phase model attempts to solve this issue by dividing the each task in one or multiple program segments and executing each segment in three phases: loading the data and code of the segment to the SPM, then executing the segment from the SPM, and finally writing back the modified data to the SPM. Since a segment does not need to access main memory during its computation phase, a DMA engine can be scheduled to perform memory transfers from/to the main memory in parallel. This enables scheduling the tasks as well as the DMA operations in a predictable way as contention on the main memory is mitigated.

Based on this core idea, successive works [6–8, 18, 22, 28, 45, 52, 97, 99, 101, 123, 136, 151, 152, 166, 167] have proposed a variety of contentionless approaches targeting different scheduling schemes and platforms. However, compiling a program to execute based on the 3-phase model is a key problem that has received significantly less attention. Due to the complexities inherent in each step, an automated tool is required to remove the burden from the programmer.

The 3-phase model only allows the overlap of the execution time of a task with the DMA time of another task, i.e. a multiple segments of the same task cannot execute back-to-back. In Chapter 5, we extend the model to allow multi-segment streaming. Streaming a segment into the next segment of the same task means that the code and data of the next segment are transfered to the SPM while the current segment is executing. This is important as the main structure in a program are usually the loops. So, techniques like loop tiling enable segment streaming in many cases. We also extend the 3-phase model to support a conditional Directed Acyclic Graph (DAG) representation for the tasks. Previous works adopted a sequential model of the program in which the segments of the program are executed in sequence. The conditional DAG representation allows multiple execution paths in the program and hence it is more general. Our evaluation has shown that the system schedulability improves significantly when multi-segment streaming is allowed.

In Chapter 6, we propose a set of program transformation constraints that allow us to convert a task into a conditional sequence of 3-phase segments. We use a region-based approach to simplify segment creation, in conjunction with loop splitting and tiling to split large loops into multiple segments. We address two models for the DMA: fixed-size DMA model, and variable-size DMA model. In both fixed and variable-size DMA models, the DMA is arbitrated between different cores using a TDMA memory schedule. The fixed-size model assigns TDMA slot that is sufficient to transfer the whole SPM space assigned to the task; while the variable-size model uses a fine granularity for the TDMA slots such that a transfer can span multiple slots. For the fixed-size model, we are able to derive a task segmentation algorithm that enumerates the best possible conditional segments for a given task on a platform with fixed-size memory phases. 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. Then, we propose a set of heuristics for the variable-size model as an optimal algorithm is too complex to consider. Our evaluation shows that our proposed algorithms improve the system schedulability significantly compared to other greedy and heuristic algorithms.

1.3 Thesis Outline

The thesis starts with a presentation of the compilation framework and the analysis and transformation passes used in Chapter 2. The rest of the thesis is structured in two parts. The first part is concerned with the case of the execution of a single task. Chapter 3 discusses the background and the related work of SPM management and prefetching. Then, we present our proposed technique for WCET-driven data SPM allocation and prefetching

in Chapter 4. The second part focuses on multi-tasking. In Chapter 5, we review the related work for the 3-phase model and then discuss the extension of the model with a formal schedulability analysis. After that, we present our developed algorithms for task segmentation and show the evaluation results in Chapter 6. Finally, we summarize the thesis in Chapter 7 and discuss the future extensions.

Chapter 2

Compilation Framework: Analysis and Transformation

In this chapter, we present the structure of our compilation framework. The framework is based on the LLVM compiler which is used to analyze and transform the program code. We start with an introduction to the LLVM compiler and the compilation flow in our framework in Section 2.1. Then, we focus on the set of analysis and transformation passes used to prepare the program and gather the required information about it: region analysis in Section 2.2, loop analysis and transformations in Section 2.3, memory access information in Section 2.4, and finally the back-end analysis in Section 2.5.

2.1 LLVM Compiler and Compilation Flow

The *Low Level Virtual Machine* (LLVM) is a compiler infrastructure introduced in [86]. The compiler is designed in a modular and reusable structure to support optimization of the program during its lifetime through compile time, link time and run time.

LLVM is based on the LLVM Intermediate Representation (LLVM-IR) which is a typed RISC-like instruction set. LLVM-IR is agnostic to the target machine and uses an infinite number of virtual registers. The register operations are in Static Single Assignment (SSA) form which means each register can be written only once. There are two file types for LLVM-IR: bytecode (.bc) and human readable assembly language (.ll). In this document, the human readable representation is used.

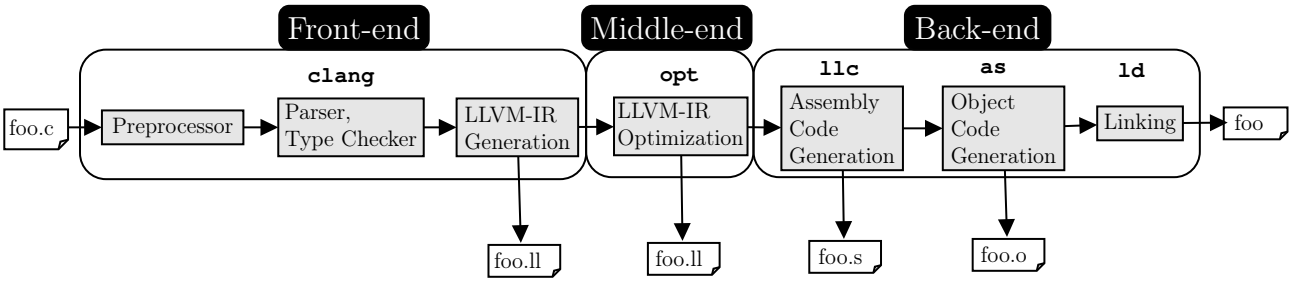


Figure 2.1: LLVM compiler tool-chain

The compilation tool-chain of LLVM is shown in Figure 2.1. The code is parsed using and converted to LLVM-IR on which most of the optimization passes are applied. Then, the back-end generates the assembly according to the specified target. The assembler generates the object file that is handled by the linker to emit the executable.

2.1.1 LLVM-IR Instructions

LLVM-IR set of instructions represent common operations to describe the program independent of the machine instructions. We focus on memory instructions and other instructions needed to understand its operation.

In LLVM, an object is represented by a pointer to its address in the memory. The pointer can refer to a global object, a stack-allocated object or a return from a function, *e.g.* malloc for heap allocation. All memory operations are pointer-based where the address is computed first -if needed- and then provided to the load/store instruction.

The following example shows `alloca, load, store, getelementptr` instructions:

```
1 @x = global [10x32] zerointializer
2 %ptr1 = alloca i32
3 store i32 5, i32* %ptr1
4 %ptr2 = getelementptr inbounds [10x32], [10x32]* @x, i32 0, i32 1
5 %x.1 = load i32, i32* %ptr2
```

`alloca` allocates an object in the stack and returns a pointer to it as in line 2 where a 32-bit integer is allocated and a pointer `ptr1` is returned to its address in the stack.

`getelementptr` is used to get the address of a subelement of an aggregate data structure. In line 1, an integer array `x` has 10 elements and line 4 gets the address of the second element in `ptr2`.

`store` is used to write to a memory address as in line 3 where a value of 5 is written to the integer pointed to by `ptr1`.

`load` is used to read from a memory address as in line 5 where the data in the address `ptr2` is loaded in register `x.1`.

Other instructions might be used for handling pointer types like `inttoptr` and `bitcast` or selection like `select` and `phi`.

2.1.2 LLVM Passes

LLVM provides a set of analysis and transform passes [2]. The analysis passes collect information about the program that can be used to apply transformations or for debugging.

The passes in LLVM works in a framework called LLVM Pass Manager. This framework is responsible for keeping the analysis information updated after the optimization of the program and maintaining the memory and execution dependency of different passes.

There are multiple types of passes depending on the scope of the pass. This helps the pass manager to schedule the passes in an efficient way. The pass can be a `ModulePass`, `CallGraphSCCPass`, `FunctionPass`, `LoopPass`, `RegionPass`, or `BasicBlockPass`. Each of these types imposes constraints on the information available to the pass and the scope of the transformation, *e.g.* `FunctionPass` can only work on the current function passed to it and has no information of the other functions.

2.1.3 Compilation Flow

Figure 2.2 depicts the compilation flow of the program analysis and transformations. The source code is compiled by the front-end of LLVM (`clang`) accompanied with the profiling information to Intermediate Representation (IR) code. Then, the middle-end generates the information about the region structure of the program, the loop bounds, the possible loop transformations, and the data footprint for each part of the program. The IR code is passed to the back-end of LLVM to create the assembly code and extract the timing and function stack information that can be mapped to the IR code. These information are fed to a set of real-time algorithms that are developed in this work. The output of these algorithms is a set of transformations to be applied on the IR code. The analysis-transformation cycle can run for multiple iterations in which the transformed IR code is analyzed and used by the algorithms. Finally, an executable is create for the optimized program.

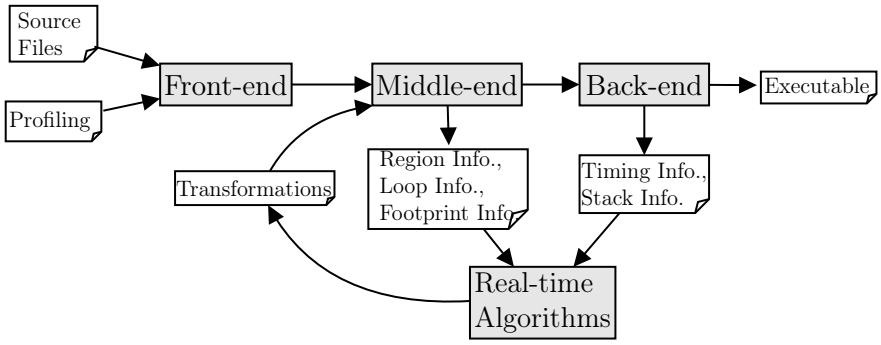


Figure 2.2: Compilation Flow

2.2 Region-Based Program Structure

In this section, we introduce the region-based program structure as the base of the program representation in our framework. Then, we discuss the region analysis in LLVM that generates the region tree representation for the program. After that, we show our proposed refined region-based structure that allows us to have a more detailed representation of the program.

The region tree is equivalent to the Program Structure Tree (PST) which is defined in [76] as a hierarchical representation of the program structure based on Single Entry Single Exit (SESE) regions of the Control Flow Graph (CFG). PST is used to speedup algorithms for compiler static analyses and optimizations. The benefit of PST is that each SESE region is a CFG on which analysis algorithms can be applied using divide-and-conquer approach. PST is then used to combine the results from each SESE region to the analysis result for the program.

The following definitions are the basic concepts used in the region representation:

Control Flow Graph (CFG) A control flow graph (CFG) $G = (N, E)$ of a program is a set of basic blocks represented by vertices N connected with a set of edges E. The graph starts with an *entry* basic block and ends with a *return* basic block. The basic block is a set of statements executed in a linear order and the basic block might end with a branch to compose a non-linear control flow.

Dominance and Post-dominance In the CFG, node a dominates node b if every path from the start of the CFG to b passes by a. Similarly, node b post-dominates node a if every path that from a to the end of the CFG passes by b.

SESE (Simple) Region A SESE (simple) region is a subgraph of the CFG that is connected to the other nodes in the CFG with only two edges, an incoming edge (entry edge) and an outcoming edge (exit edge). The SESE region is defined using the entry and exit edge such that the entry edge dominates the exit edge and the exit edge post-dominates the entry edge.

Canonical Region A region that cannot be constructed out of a set of regions is a canonical region. Two canonical regions are either disjoint or completely nested.

Trivial Region A trivial region is composed of one basic block.

Extended Region An extended region is a subgraph of the CFG that can be transformed to a simple region by adding empty basic blocks to combine multiple entry edges or exit edges.

Sequentially-Composed Regions Two regions are considered sequentially-composed if the exit of one region is the entry of another region.

Region Tree (Program Structure Tree) The region tree (PST) represents the relationship between canonical regions such that a region r_a is an ancestor of region r_b if r_b is completely contained in r_a. A r_a is the *parent* or region r_b if r_a is the closest containing region of r_b.

2.2.1 LLVM Region Analysis

The region analysis pass in LLVM constructs the region tree for canonical non-trivial regions. A region can be collapsed to a single node and modeled as a call to a function that contains the CFG of the region. This function can be analyzed and optimized; then it can replace the original region.

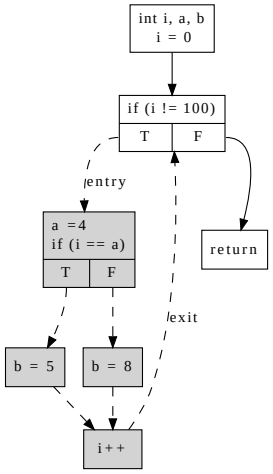
The examples in Figures 2.3, 2.4 and 2.5 are adopted from [58]. The CFG in Figure 2.3b is constructed from the program code in Figure 2.3a and it highlights the simple region that represents the if condition in the program code with single entry and single exit. The example in Figure 2.4 shows an extended region in Figure 2.4a and how it can be transformed to a simple region by inserting two empty basic blocks t_1 and t_2 in Figure 2.4b. The CFG in Figure 2.5 illustrates how the definition of an extended region generalizes the definition of a region. In the figure, the simple regions are fenced by solid borders while the extended regions are fenced by dashed borders. Note that in LLVM the top level region, which is the whole CFG that contains the node e, is also considered a region.

```
void foo() {
  int i, a, b;

  for (i = 0; i != 100; i++) {
    a = 3;

    if (i == a)
      b = 5;
    else
      b = 8;
  }
}
```

(a) Program code



(b) Program CFG and simple region

Figure 2.3: Simple and extended regions example

2.2.2 Refined Region Tree

In our framework, we use regions as the basic unit of the program to apply ScratchPad Memory (SPM) allocation or program segmentation. However, the region analysis in LLVM has two limitations:

- A basic block with multiple entries/exits is not considered a region.
- A basic block with a function call or multiple calls is considered one region.

We propose to construct a *refined region tree* that avoids these limitations and allows regions with finer granularity; hence provide more flexibility to our algorithms.

To obtain the refined regions, we first construct a modified graph $\bar{G} = (\bar{N}, \bar{E})$ from the CFG $G = (N, E)$, where $\bar{N}$ is the set of basic block nodes, call nodes and merge/split nodes and $\bar{E}$ is the set of edges such that:

- Each call to a function in G_f is split into a separate *call node*.
- A *merge/split node* is inserted before/after a basic block or a call node with multiple entry/exit edges.

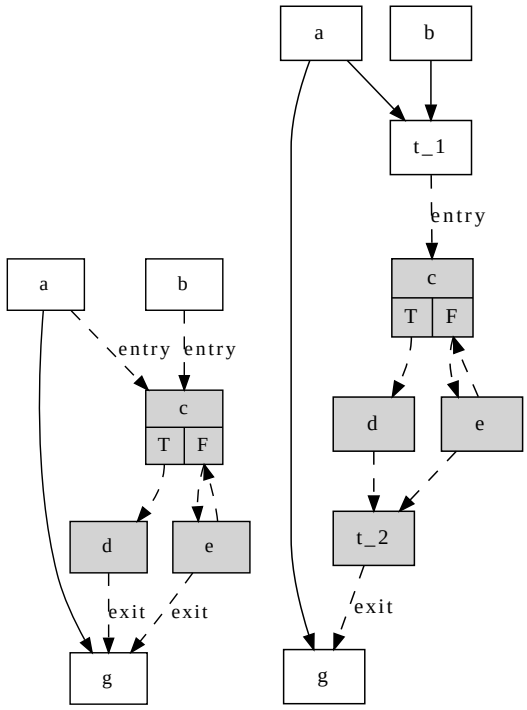


Figure 2.4: Extended region Example

Note that after the transformation, every node in $\bar{G}$ that is not a merge/split node has a single entry and a single exit; hence, it is a region. We use the term *trivial region* to denote any leaf of the refined region tree; note that by definition, each trivial region must comprise either a single basic block or a single call node, i.e., trivial regions represent code segments in the program. We denote a region that consists of a sequence of sequentially composed regions as a *sequential region*. A sequential region is not canonical as it is constructed by combining other regions. Finally, we construct the refined region tree by considering both canonical regions and maximal sequential regions, i.e., any sequential region that encompasses a maximal sequence of sequentially composed regions. It is proved in [142] that adding maximal sequential regions to the tree still results in a unique region tree.

The following example illustrates the process of constructing the refined region tree. Figure 2.6a shows an example CFG and its canonical regions. The corresponding region tree is shown in Figure 2.6b. In this example, region r_1 is the parent of regions r_2, r_3 and r_4.

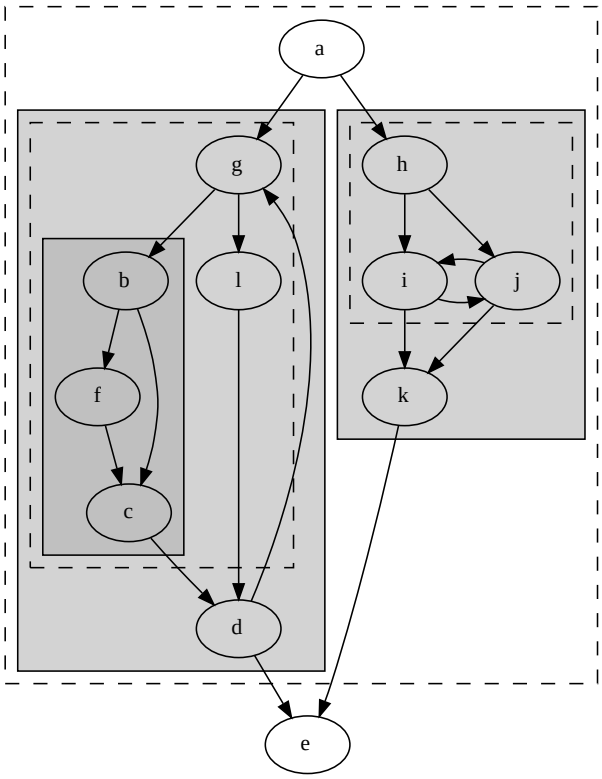
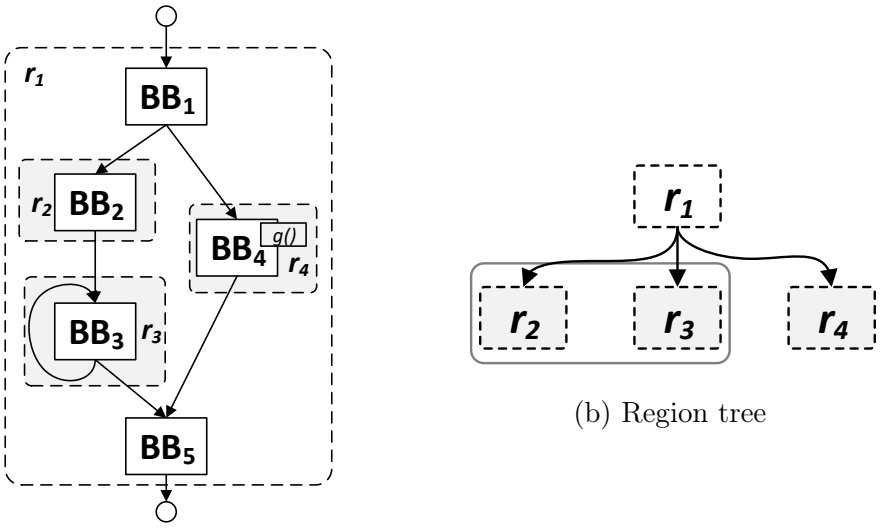


Figure 2.5: Simple (solid border) and extended (dashed border) regions in a CFG

Regions r_2 and r_3 are sequentially composed; this is represented by a solid-line box in the figure. Figure 2.7 shows the refined CFG and region tree for the example in Figure 2.6. We added merge points before BB_3 and BB_5, and split points after BB_1 and BB_3. Assuming that function $g()$ is called at the beginning of BB_4, we split BB_4 to a call node BB_{4a} that contains the function call and a basic block BB_{4b} for the rest of the instructions in BB_4. In the refined region tree in Figure 2.7b, regions r_1, r'_7 and r_4 are sequential regions. The regions r_1 to r_4 are the same as in the original region tree, while regions r'_5 to r'_{11} are added as a result of the refinement process. We refer to r'_3 as a *call region* as it contains the call node BB_{4a}. In this example, the leaf nodes $r'_5, r'_{11}, r_2, r'_8, r'_9$ and r'_{10} are trivial regions.



(a) Program CFG

(b) Region tree

Figure 2.6: Program CFG G and region tree

2.3 Loop Analysis and Transformation

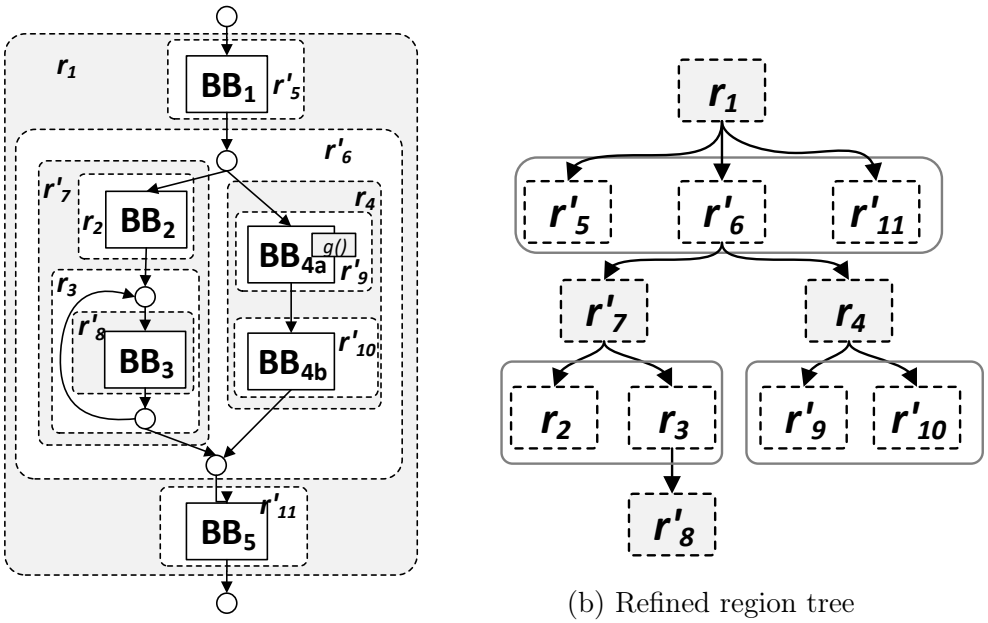
In this section, we discuss two aspects about loops that we employ in our framework: loop iteration bounds and loop transformations.

2.3.1 Loop Iteration Bound

As our framework targets real-time applications, each loop must have a bound on the number of iterations that can be used in timing analysis. We obtain a bound on a loop using one of three approaches:

- Using the loop trip-count analysis.
- Using programmer annotations.
- Profiling the program and use the profiling meta-data added to the IR of the program.

The first method that utilizes the loop trip-count analysis is accurate, but we can only



(a) Refined program CFG

(b) Refined region tree

Figure 2.7: Refined program CFG $\bar{G}$ and region tree

use it if the number of loop iterations is constant [1]. For the second method, an annotation can be inserted in the source code and attached to the loop so that it can be retrieved as a meta-data during the analysis. Currently, `clang`, the front-end of LLVM, does not support a `#pragma` attribute for the loop bound. Hence, we added a new loop attribute as following:

```
1 #pragma clang loop bound(x)
```

This allows the programmer to easily insert the required loop information in the source code.

The final method uses LLVM provides Branch Weight Metadata that is generated by profiling the program. Branch weights represent the likeliness of a branch instruction to be taken, hence, we can use weight of the loop back-edge branch to estimate the number of iterations. Note that this profiling method is dependent on the input data to the program. The branch weights appear in the IR in the following format:

[1]For some cases, if the number of iterations is an expression, e.g. depends on the outer loop counter, a max operation can be used to bound it.

```
!0 = metadata !{
  metadata !"branch_weights",
  i32 <TRUE_BRANCH_WEIGHT>,
  i32 <FALSE_BRANCH_WEIGHT>
}
```

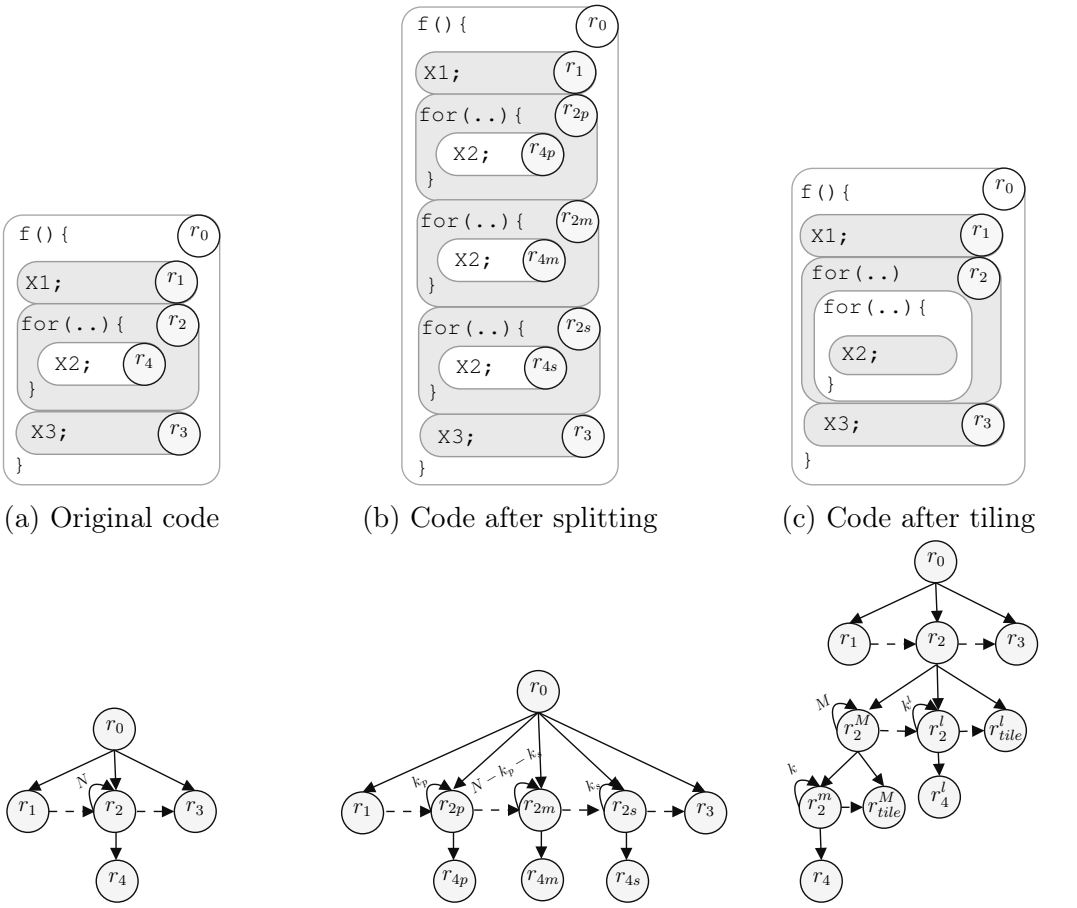
2.3.2 Loop Transformations

Loop transformations are an important tool to improve the execution time of the loops by effectively exploiting the features of the processor architecture. A loop transformation must be legal, i.e. it preserves the temporal sequence of all dependencies and hence the result of the program. There are many transformations that can be applied to loops, for example: loop fusion, loop fission, loop peeling, loop skewing, loop tiling, loop unrolling, ..etc. However, it is always a challenge to choose the best set of transformations that optimizes the required target. Although some transformations are supported in known compilers, like loop unrolling and loop vectorization; the space exploration of the possible loop transformations requires more expressive tools. There are multiple tools that support both source-level and IR-level transformations, like Pluto [21], PoCC [115], and Polly [58]. Many of these tools utilize the polyhedral model [116] to represent and manipulate the loops. As our framework is based on LLVM, we depend on Polly to perform the loop analysis and transformations on the IR-level.

The goal of the optimization is usually to improve data locality and minimize the communication in multi-core and distributed systems. In this work, we use loop transformation to manage the data in the local scratchpad memory and to allow program segmentation in multi-tasking systems. We are mainly interested in two transformations: loop splitting and loop tiling. We next discuss how to represent these transformations in the region tree of the program as well as the overhead incurred by them.

Loop splitting breaks the loop into multiple loops which have the same bodies but iterate over different contiguous portions of the index range. *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 execute a tile. An n-level tiled loop nest has n tiling loops and n element loops.

Figure 2.8 shows an example of loop splitting and loop tiling. The code of function `f()` is shown in Figure 2.8a and its region tree in Figure 2.8d in which region r_2 represents a



(a) Original code (b) Code after splitting (c) Code after tiling

(d) Original region tree (e) Region tree after splitting (f) Region tree after tiling

Figure 2.8: Region representation of loop transformations

single loop with N iterations. Region r_2 can be split by expanding the loop region into multiple regions. In the example, we split r_2 into three nodes as in Figure 2.8e: pre-loop node r_{2p} with k_p iterations, mid-loop node r_{2m} with $N - k_p - k_s$ iterations, and post-loop node r_{2s} with k_s iterations. This equivalent of having three loops in the code as shown in Figure 2.8b. Tiling region r_2 will result in a single tiling loop and a single element loop as in the equivalent code in Figure 2.8c which includes two nested loops. However, we represent the tiled loop in the region tree in more details. The loop in r_2 is tiled with tile size k. This results in $\lceil N/k \rceil$ tiles with first $M = \lceil N/k \rceil - 1$ tiles and a last tile with size $k^l \leq k$ such that $k^l = N - M * k$. The first M tiles are complete tiles while the last tile might not be a complete tile. This is illustrated in Figure 2.8f where r_2^M is the tiling loop that iterates over the first M tiles and r_2^l is the last tile. Note that r_2^M and r_2^l are considered sequential regions. Adding the tiling loop incurs an overhead, e.g. the loop counters. We account for such overhead by adding a region that represents the tiling overhead in sequence with the element loops, i.e. r_{tile}^M and r_{tile}^l in Figure 2.8f.

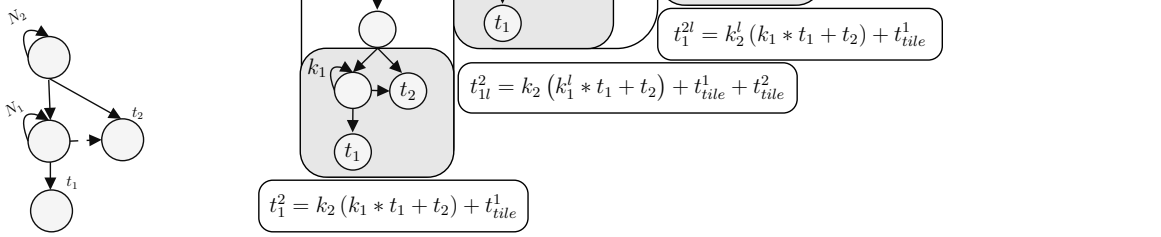
Tiling Overhead

As we discussed in the previous example, tiling a loop incurs an overhead due to the added tiling loops. When tiling n-level loops where $n > 1$, another overhead comes off due to the loop permutations. To illustrate this overhead, consider the region tree for a 2-level nested loop in Figure 2.9a. The inner loop has N_1 iterations and an execution time t_1 and an outer loop with N_2 iterations and an execution time of one iteration $N_1 * t_1 + t_2$. This implies that the total timing of the loop nest is:

$$t_{loop} = N_2 * (N_1 * t_1 + t_2)$$

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 $k_2^l = N_2 - M_2 * k_2$. The tiling overhead for each iteration of the tiling loop is t_{tile}^1 and t_{tile}^2 for the inner and outer loop, respectively. The resultant region tree in Figure 2.9b has 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. Adding the tile times will result in:

$$t'_{loop} = t_{loop} + N_2 * M_1 * t_2 + (M_2 + 1) * (M_1 + 1) * t_{tile}^1 + (M_2 + 1) * t_{tile}^2$$



(a) Region tree before tiling

(b) Region tree after tiling

Figure 2.9: Tiling 2-level loop nest

Hence, the total tiling overhead is:

$$t_{overhead} = t'_{loop} - t_{loop} = t^{permutation}_{overhead} + t^{tiling\,loops}_{overhead}$$

And the overhead terms are:

$$t^{permutation}_{overhead} = N_2 * M_1 * t_2$$

$$t^{tiling\,loops}_{overhead} = (M_2 + 1) * (M_1 + 1) * t^1_{tile} + (M_2 + 1) * t^2_{tile}$$

The permutation overhead is a consequence of the interchanging the tiling and the element loops. That is, as the tiling loop of the inner loop is moved on top of the element loop of the outer loop, the element loop of the outer loop is executed more times.

2.4 Memory Access Information

Optimizing memory accesses is the core of our work. So, our goal is to precisely identify the memory accesses for every part of the program. A memory instruction can target different sections of the memory. This includes the stack, the heap and the global data. Analyzing memory accesses is usually done in the compiler using pointer analysis.

Pointer analysis or points-to analysis tries to decide statically what are the objects that a pointer may refer to at run-time. It is an essential analysis for languages with pointers. There has been tens of papers that explore the trade-offs between efficiency and precision of the pointer analysis [69, 130].

Alias analysis is another term that is usually used interchangeably with pointer analysis. However, alias analysis focuses on the relations between pointers to determine if two pointers can point to the same object while pointer analysis tries to answer the question of what objects a pointer might point to.

There are a wide range of applications that benefit from pointer analysis. Pointer analysis is used in compilers for live range analysis to improve register allocation, constant propagation, static checking for run-time errors, multi-threading, cache analysis, etc. For real-time systems, pointer analysis can improve the accuracy of Worst-Case Execution Time (WCET) analysis by predicting the accesses to memory in systems with multiple memories.

The precision and the cost of pointer analysis depends on the implementation. There are multiple aspects to the analysis:

Flow-sensitivity Flow-sensitive analysis computes points-to sets for each program point while flow-insensitive analysis is concerned about points-to sets at any time in the program collectively. Flow-sensitive analysis is more expensive.

Context-sensitivity Context-sensitive analysis considers the calling context when analyzing the function while context-insensitive analysis analyzes the function independently. Context-sensitivity requires inter-procedural analysis and so adds complexity to the implementation.

There are other factors that affect the analysis like path-sensitivity, field-sensitivity and heap-modeling. There are many approximations that try to improve the precision of the analysis with a reasonable complexity. The application of pointer analysis is a main factor to determine if the added precision is worth the cost.

LLVM provides a set of alias analysis passes that are effective for most of the common access patterns. Using LLVM analyses, memory accesses that target distinct global objects, stack allocations and heap allocations can be easily identified. LLVM also provides a limited context-sensitive alias analysis for global objects, and an analysis based on Scalar Evolution for loops that reasons about the induction variables. Other pointer analysis techniques can be built upon the LLVM alias analysis infrastructure. For example, SVF [3] utilizes interprocedural dependence analysis to construct a more precise pointer analysis for LLVM. We use SVF to obtain flow-sensitive points-to information.

Note that pointer analysis is carried-out on the IR representation. So, it captures memory operations that are represented by load and store instructions. However, there are accesses to the program stack that only materialize when generating the assembly of the program; more specifically, stack accesses that are result of register spilling. Compilers assume an infinite number of registers in their IR representation. However, some of the registers have to be spilled to the stack when the back-end of the compiler schedules the assembly instructions due to the limited number of registers on the target processor. We account for such memory accesses in the back-end analysis. This is done by accounting for the load and store assembly instructions that target the stack, but do not access the allocated local objects.

For loops that are split or tiled, we extract the data footprint of each object used in the loop as a function of the transformation parameters. We make use of Polly to generate a rectangular bounding box over the data accessed inside the loop as a linear function of the number of iterations. If the accesses of an object are irregular or data dependent, the footprint of the object is the whole object.

In the developed algorithms in this work, we rely on software to mange the local SPM. So, we apply the following transformation pass to promote large local objects to be global objects. This enables more control on the content of the SPM.

2.4.1 Stack Object Promotion

The stack section in the memory has two components: a) temporary spilled registers and calling context b) allocated local objects. Allocated local objects can be large and hence it might not be possible to move the stack to the local memory. In order to allow a flexible allocation of the stack, a pass is implemented to promote large local objects to global objects [84]. This reduces the maximum stack size and provides the possibility to allocate local objects in the main memory or in the local memory without the need to manage multiple stacks. Applying the stack promotion on the IR level allows the compiler to optimize the code for the local objects before the promotion. Also, the promotion pass marks the promoted object as local to the function; hence the object does not need to have an initial value and does not have to be written back after the function scope, counter to static local objects in C for example.

The pass identifies `alloca` IR instructions and checks if the size of the allocated object is larger than a specified threshold, e.g. 32 bytes. For example:

```
1 %ptr = alloca [10x32]
```

This object is promoted to be a global object with the same size:

```
1 @ptr_global = global [10x32]
```

After the object is promoted, all accesses to the local object are modified to reference the new global object.

2.5 Back-end Analysis

The purpose of the back-end analysis is to analyze the machine instructions and create a map between the machine CFG and the IR CFG. This allows us to estimate the timing of different parts of the program at the IR level. Hence, we are able to perform timing-aware IR optimizations.

LLVM IR code is translated to a machine specific representation [2] with functions, basic blocks, and instructions. Mapping between IR CFG and machine CFG is possible as

LLVM keeps track of the relation between the basic blocks in the back-end. However, a one-to-one mapping between basic blocks does not always exist due to back-end optimizations that removes or adds basic blocks. We keep track of the removed and added basic blocks and use them to generate conservative timing estimates.

Analyzing the back-end also provides information about the memory requirement of the program stack at each point of the program. So, we extract the stack size for each function and use this information to assign the required space in the local memory.

2.6 Summary

In this chapter, we presented our compilation framework based on LLVM compiler and how it is integrated with the algorithms that we discuss in the next chapters. Analysis passes are utilized to collect information about the program structure, loops, memory accesses, and mappings to back-end assembly. Program transformations are passed by optimization algorithms to the middle-end of the compiler and applied on the program IR. We rely on this compilation flow as the basis for our proposed techniques.

Part I

The Case of Single Task Execution

Chapter 3

Scratchpad Management: Background and Related Work

In this chapter, we discuss the background and the related work for Chapter 4. Section 3.1.1 discusses the use of caches and ScratchPad Memory (SPM) in real-time systems, prefetching in general purpose processors, and the techniques for Worst-Case Execution Time (WCET) analysis. In Section 3.2, we focus on the previous research on SPM management and different allocation techniques for real-time systems as well as general-purpose computing.

3.1 Background

3.1.1 On-Chip Memory in Real-Time Systems

As estimating the WCET of a program is a critical aspect in real-time systems, various techniques have been proposed to provide a safe and tight bound when on-chip memory is used. In order to obtain a safe bound, the target of the memory accesses should be known to estimate the expected delay. The simplest solution is to assume the worst-case delay for all the accesses which highly overestimates the WCET. To be able to obtain a tighter bound, researchers have developed predictable techniques to analyze and control the memory accesses.

Caches and SPM are the two common forms of on-chip memories used in current processors. Caches are hardware-controlled, transparent to the software, and use heuristics to

exploit temporal and spacial locality. SPM is directly accessed by the processor, software-managed, and has a smaller footprint. Caches have proved a high efficiency in general-purpose computing that focuses on improving the Average-Case Execution Time (ACET) without the need to develop cache-aware software. SPM was introduced as a low energy alternative to caches [15]. It also provides better predictability for the WCET in real-time systems [155]. However, SPM has to be managed either by the programmer or using an automated compilation process. This limits the portability of the software as the SPM allocation is tied to the configuration of the platform.

The predictability of cache behavior is affected by several aspects: associtivity, replacement policy, write-back policy, and separation of data and instruction caches [67]. The cache replacement policy has the main impact on the cache predictability [122]. Static cache analysis is used to classify memory accesses as cache hits or misses [96]. The analysis of Least Recently Used (LRU) replacement policy in conventional caches based on abstract interpretation has been the foundation for cache analysis [51]. LRU policy offers high predictability of the cache behavior. However, the analysis of the common non-LRU replacement policies of set-associative instruction caches, like pseudo-round-robin in the ColdFire MCF 5307, and the PLRU (Pseudo-LRU) in the PowerPC MPC 750 and 755, produces pessimistic WCET bounds [67]. In [122], Reineke *et al.* analyzed different instruction cache replacement policies and showed that LRU policy is the most predictable while Pseudo-LRU and FIFO perform significantly worse than LRU. A quantitative approach is proposed in [61] to reduce the overestimation ratio of WCET for FIFO replacement policy. A k-miss classification is used in [60] to analyze MRU replacement policy used for instruction caches in processors like Intel Nehalem that showed a close WCET estimation to the LRU policy. A more precise analysis for PLRU policy is introduced in [59], but with a limited scalability. Although the cache analysis is applicable for both instruction and data caches, there are no available techniques to analyze non-LRU policies in data caches [67]. The main challenge of analyzing data caches is the precision of value analysis due to the usage of pointers, dynamically allocated data, and data dependent array indexes. In [50, 95], the authors try to derive the WCET by restricting the *reference string*, which is the sequence of addresses generated by memory operations, by skipping the cache when the address is unpredictable. This method overestimates the WCET as it eliminates the benefit of the cache for any unpredictable address. CAMA is a memory allocator proposed in [68] that employs shape analysis to ensure that data structures that exist simultaneously in the cache do not conflict.

To improve the predictability of the cache analysis, two important approaches have been proposed: cache partitioning and cache locking [56, 103]. Both approaches help to reduce the number of cache states that should be considered for the WCET analysis by disabling

the cache replacement policy for part of the cache. Cache partitioning considers a shared cache between tasks or cores [26, 32, 93, 105, 124, 150]. It divides the cache into partitions based on cache ways or aggregation of associative sets and assigns a partition for each task. This prevents the possible interference between tasks and isolates the cache analysis of each task. Cache locking allows to mark a cache line/way as locked using a hardware feature that is available in many embedded processors [11, 26, 41, 42, 124, 143–145]. Locking a cache line/way prevents the replacement of this line/way until it is unlocked which enforces a more predictable behavior for the cache accesses.

Method cache [125] is an alternative cache architecture for instruction cache that stores a complete method/function which means that cache misses can only happen on the call and return program points. The replacement of cache content depends on the call tree of the program rather than the addresses of the instructions which facilitate the WCET analysis. The method cache is used as part of real-time java processor in [126] and a WCET analysis tool is designed for the estimation of WCET at the byte-code level using Integer Linear Programming (ILP).

Using SPM in real-time systems is growing as it enhances the predictability of memory accesses. Unlike caches, the content of the SPM only depends on the program point and does not require the reference string to predict the target of the memory access [156]. Wehmeyer *et al.* studied the usage SPM on the WCET analysis in [153] and showed that it can significantly improve the predictability without the need to modify the timing analysis tool. They also presented a comparison between the impact of using caches and SPM on the WCET in [155]. The study showed that increasing the size of the cache increased the difference between the simulated ACET and the estimated WCET. On the other hand, increasing SPM size decreased the estimated WCET with a steady ratio between the ACET and the WCET. In a comparison between an instruction SPM and a method cache, Whitham *et al.* [158] showed that a method cache has a lower true WCET. However, an instruction SPM produces lower estimated WCET using WCET analysis. We review a wide range of techniques for SPM allocation for both general purpose and real-time systems in Section 3.2.

3.1.2 Cache Prefetching

Cache prefetching has been exploited in general purpose systems for a long time to effectively reduce the cache miss rate [25, 103]. The cache has an implicit prefetching capability when a cache miss happens as it fetches the whole line containing the required instruction/data to take advantage of spatial locality. Prefetching techniques try to hide the

transfer latency of a cache miss by loading the required line before its use using hardware or software. Hardware prefetchers use simple algorithms to speculate the next line to be referenced like detecting strided accesses to an array or prefetching next few lines. Software prefetching is performed by inserting prefetch instructions in the code. Data prefetching is more challenging than instruction prefetching as the data access patterns are more irregular. Cache prefetchers used in general purpose systems are based on speculation which is not suitable for real-time systems as it increases the intractability of the cache analysis.

Cache prefetching has been combined with instruction cache locking for real-time systems in [12, 35]. In [12], two address-tagged buffers are used for fetching and prefetching along with dynamic locking instruction cache. The memory lines to be loaded and locked in the instruction cache are selected using an ILP-based solution to minimize the WCET including the context switching time in a multitasking system. Their method shows that prefetching improves the WCET with small dynamic locking cache compared to locking techniques without prefetching. The approach in [35] focuses on program code converted to single-path form by transforming unpredictable branches to single execution trace. They exploit prefetching to improve spatial locality and cache locking to make use of temporal locality.

Prefetch distance is defined as the distance ahead in the execution of which a memory address should be requested [87]. Prefetching is useful if the prefetch request is sent early such that the prefetch distance hides the memory transfer latency. The prefetch distance varies during run-time due to different execution paths. In a cache system, the prefetch distance should not be too large or too small. Prefetching a cache line very early can result in evicting cache lines in use which incurs more cache misses. In summary, prefetch distance is a key factor that can affect the usefulness of the prefetching technique.

Both hardware and software prefetching approaches have their strengths and weaknesses [87, 108]. Unlike software prefetching, hardware prefetching techniques do not need help from the programmer or the compiler and can be applied to compiled programs without changes. Hence, they do not increase the program size as no prefetching instructions or hints are inserted. Also, hardware prefetchers can be used for both data and instruction caches while software prefetchers is mostly used for data. However, the dedicated hardware used for prefetching is large compared to software prefetching especially for complex data structures. The number of streams to be traced using hardware prefetchers are limited to the available hardware resources while software prefetching schemes are more flexible. Also, hardware prefetchers require training to be able to increase the accuracy of prefetching which may not be efficient for shore streams of data and irregular memory accesses. Software prefetching can be optimized for the application exploiting the available compile time information to produce more accurate prefetching results. However, the

software overhead added can significantly reduce the efficiency of the prefetching scheme. Other factors like the cache level in which the cache line to be prefetched are considered in software prefetching techniques. Many approaches are proposed to combine software and hardware prefetching techniques to leverage the merits of both approaches.

3.1.3 WCET Analysis

WCET analysis is a necessary step in developing hard real-time systems. The analysis estimates an upper bound on the execution time of a program to ensure the satisfaction of the timing constraints of the system. A set of programming rules are adhered in real-time systems that helps in providing an execution bound such as: the program always terminate, recursion is not allowed and the loop iteration count is bounded. The WCET estimation difficulty depends on the complexity of the platform architecture. Also, the dependency of the execution time on the input makes it hard to derive a bound as the worst-case input is usually unknown.

The methods for WCET analysis can be classified to two main categories:

Measurements The program code is executed on the target platform or using a simulator. A range of execution times are measured for a set of inputs and combined to estimate the WCET for the program.

Static Analysis A static method does not rely on the execution or simulation of the program. The Control Flow Graph (CFG) of the program is analyzed in combination with an abstract model of the hardware to produce a safe upper bound on the execution time.

Processor architecture is modeled to be able to analyze its behavior for simulation or static analysis. The accuracy of the model is a key factor to the accuracy of the timing behavior. However, a concrete model of a processor is usually complex. So, a conservative abstract model is considered sufficient. Validation of the abstract model is necessary to consider the model trustful. Measurements, trace observation, and equivalence checking of abstraction levels are used to validate abstract models.

Measurements are more suitable for soft real-time systems as they produce estimations rather than bounds. An end-to-end measurement gives a distribution for the execution times of a subset of the possible executions based on some possible contexts. However, the bound on the WCET cannot be guaranteed unless the context of the worst case is known.

The bound can be calculated by measuring the execution time of parts of the program and combining them using path analysis. This can generate better approximations, but the same problem of the possible contexts apply for the execution of parts of the program. Exhaustive coverage of all the possible executions is usually infeasible.

Instrumentation using software and/or hardware is a common way to provide timing measurements. There are other methods that do not interfere with the program execution like using logic analyzers and hardware tracing. Simulators that incorporate a model for the processor can be also used to collect measurements for some set of inputs. Measurements can be used to validate the precision of the analytical methods by comparing the predicted execution times with the measured ones.

Static methods guarantee a safe bound that may be overestimated. A set of analyses is applied to the program code using an abstract model of the underlying platform to obtain upper bound on the execution time. Value analysis is used to compute ranges for the values in the processor registers and program variables. These ranges are used to obtain effective memory addresses and loop bounds and also to detect infeasible paths. The program can also be annotated to provide such information. Control-flow analysis determines a set or a super-set of the possible execution paths and ignores paths that do not contribute to the upper bound. The analysis can be applied to the source code, the intermediate representation or the machine code of the program. Several methods are used to map between the program structure of different code levels: Pattern-matching, data-flow analysis, symbolic execution of the source code, and abstract interpretation. The control-flow analysis generates a set of annotations or flow facts that can be used to constrain the program behavior. Another necessary information to derive a bound on the execution is the behavior of the processor when executing an instruction. Due to the architecture aspects like pipelining, caching, and branch prediction, the execution of an instruction is dependent on the history of the execution. An abstract model of the system is used to obtain possible states of the processor for a program point with conservative assumptions about unknown states. Data flow analysis based on abstract interpretation is usually used to analyze the processor behavior. A brief discussion about abstract interpretation is presented in Section 3.1.3

Bound calculation computes a bound on the execution time based on the information gathered by the static analysis or by combining the measurements of code parts to obtain an end-to-end execution time. There are three main methods for bound calculation: structure-based, path-based and implicit-path enumeration (IPET). Example in Figure 3.1 [47] shows the different bound calculation approaches. Figure 3.1a shows the CFG of the example program indicating the timing for each block and the loop bounds.

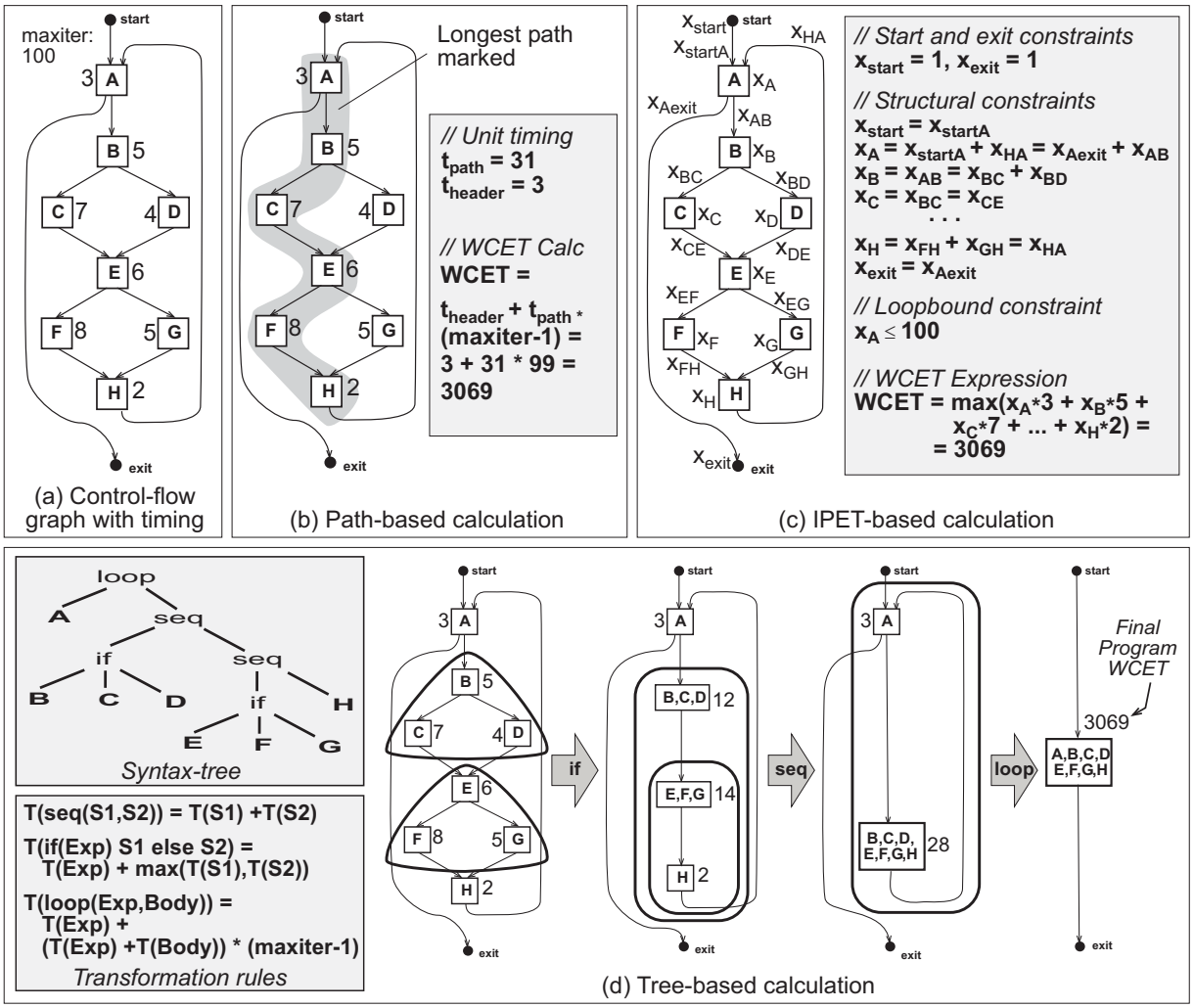


Figure 3.1: Methods of bound calculation [47]

Path-based bound calculation searches for the path with the longest execution time. This method represents the paths explicitly, however the number of paths is exponential in the number of branches which makes the search process prohibit-able in the case of nested loops without a heuristic. The path-based approach is applied in Figure 3.1b where the longest path is determined inside the loop and then the total WCET is calculated using the loop path, the loop bound and the exit path.

IPET method represents the structure of the program as a set of flow constraints and each basic block in the program is marked by an upper bound on its execution time and a number of execution times. The WCET bound is obtained by maximizing the sum of products of the execution counts and times. The size of the problem is proportional to the flow points converted to flow constraints which can grow exponentially with the program size. The constraint system is usually solved as an ILP problem. The flow constraints for the example in Figure 3.1c show a start and exit constraints that imply one entry and one exit. Then, the program flow is formulated as structural constraints to ensure that the number of times of entering and exiting a basic block are equal. Finally, the loop count constraint is added and the system is solved to maximize the WCET.

Structure-based methods apply a bottom-up traversal of the syntax tree of the program combining bounds of the constituting nodes according to the corresponding statement type. A set of nodes is replaced with a one node with a combined timing. To consider different flow contexts, transformation like loop unrolling can be applied to the syntax tree. Applying the structure-based approach to the syntax tree might not be straightforward due to code optimizations that alter the mapping between structures of the source code and the executable. The structure-based method is used in Figure 3.1d by collapsing set of nodes into combined nodes iteratively according the representing statement, *i.e.*, conditional, sequential, loop, to obtain a single node that represents the WCET.

Abstract Interpretation

Abstract interpretation [37] is a static program analysis based on abstract domains. That is, it executes an abstraction of the program on an abstract descriptions of values instead of executing the actual program on the concrete domain of values. An abstract state is associated with each program point which describes a set of concrete states at this point. An update function is used to update the abstract state based on the change that would happen to the concrete state when the program executes. The control flow is handled by merging the abstract states in a sound way whenever the flow merges.

Abstract domains are lattices [82], *i.e.*, partially ordered sets where all subsets have

least upper bounds. The relative information of two elements in the lattice is represented by the partial order such that a lower element in the lattice carry more information than a higher element in the lattice. The update functions are monotone [37] such that the information contained in the current state state is preserved when updated. The flow merging is handled with the least upper bound operation to join abstract values.

3.2 Related Work

SPM has been used to reduce energy consumption [15], improve the average-case performance, and improve the predictability of the system. In this section, we review allocation techniques in the literature for general-purpose systems and real-time systems.

There are two main categories of SPM allocation schemes: static and dynamic. Static allocation loads the content of the SPM at the beginning of the program and does not change it during run-time. On the contrary, dynamic allocation allows the contents of the SPM to change during run-time by inserting loading/unloading points in the program. Both static and dynamic allocation techniques are decided before run-time either by manual programming or automated compile-time algorithms. There have also been efforts to develop SPM allocators that decide where to allocate a memory object during run-time. We review the related work for static techniques in Section 3.2.1, dynamic techniques in Section 3.2.2 and run-time allocators in Section 3.2.3

3.2.1 Static Allocation Techniques

Static allocation is considered a partitioning problem in a system with multiple memories, *e.g.* on-chip SPM and main memory. Most of the proposed solutions for static allocation are based on a variation of the common knapsack problem [10].

Allocation of global objects has been modeled as a 0-1 knapsack problem in [128] to allocate the most frequently-accessed objects based on points-to analysis and profiling. Avissar *et al.* presented an optimal memory allocation for global and stack data in an SPM based system using 0/1 ILP solution in [14]. The approach distributes the program stack between multiple memories which offers allocation flexibility and improved performance. An extension to the approach for heap objects is discussed in [13]. However, the allocation of heap objects is not optimal and relies on profiling and modification of *malloc* function to target an allocation site to either the SPM or DRAM. A Tabu Search heuristic is proposed in [72] as an easy to implement alternative for solving the knapsack problem.

The work in [107] implements a static allocation scheme for compile-time-unknown SPM size. The scheme relies on the optimal solution developed in [14] to allocate global and stack objects and code in the SPM. The allocation problem is solved for a range of SPM sizes and stored in a compact format that is installed in the beginning of the program to modify the binary according to the available SPM size during run-time.

A WCET-oriented allocation algorithm for global and stack objects of non-recursive functions is introduced in [134]. The algorithm is based on a greedy heuristic to solve the ILP formulation of the allocation problem to minimize the WCET of the program. Another WCET-aware allocation of program code is discussed in [49] that incorporates an ILP-based allocator to minimize the WCET. In [162], a static allocation strategy for program code is described in a hybrid SPM-cache system to reduce the WCET. The approach is extended to both code and data in [170]. The allocation algorithm in [149] optimizes for the energy consumption while respecting an upper bound on the WCET. A static allocation scheme is introduced in [117] that targets the Precision Timed Architecture (PRET) [92] to ensure the temporal requirements are met. A greedy approach that targets both instruction and data for PRET is presented in [111]. The work in [80] combines static allocation of SPM and task scheduling for preemptive hard real-time systems.

Steinke *et al.* optimizes the static allocation of code and global objects for energy reduction in [133]. The approach is extended in [147] to partition arrays that do not fit in the SPM in the original approach. A similar approach in [154] presents an energy model and an optimal ILP formulation to solve the partitioning problem for energy reduction. The energy-oriented work is covered in [73].

A variety of approaches try to solve the partitioning problem between the SPM and DRAM to reduce cache pollution and minimize conflicts in cache-based systems [81, 109, 148, 171]. An allocation approach for global and local objects that improves the code size by optimizing the pointer type assignment using ILP formulation is presented in [129].

3.2.2 Dynamic Allocation Techniques

Verma *et al.* [146] has shown that the dynamic allocation is an extension of the global register allocation problem. They proposed an optimal formulation for the memory objects selection and spilling and a near optimal heuristic for the address assignment of the objects in the SPM.

Udayakumaran *et al.* [139] proposed a profile-dependent dynamic allocation strategy for global and stack objects and program code. The method partitions the program into regions and uses a Data-Program Relationship Graph (DPRG) to represent the timing

relationship between the regions. The memory transfers are inserted at the start of each region based on an allocation heuristic and a cost model to maximize the benefit of moving objects to the SPM. In order to be able to handle pointers, the authors proposed two alternatives, the first is based on run-time disambiguation of the pointer to determine if its reference is in the SPM using a compiler-inserted code and a height-balanced tree structure to translate the address from DRAM address to SPM address which incurs high run-time overhead. The second alternative is to fix the address of the object in the regions where it is accessed using pointers which limits the optimization of the allocation.

The greedy algorithm in [139] is adapted in [70] to optimize the allocation in a hybrid system consisting of SPM and non-volatile memory (NVM) for both memory access latency and the number of writes to the NVM. An Adaptive Genetic Algorithm for Data Allocation (AGADA) is proposed in [119] for a similar system that produces a better solution than the greedy algorithm.

Li *et al.* developed a dynamic allocation algorithm for data aggregates via graph coloring in [88] by partitioning the SPM into a pseudo-register file and solving the allocation problem as an extension to register allocation to obtain near-optimal solutions compared to ILP solutions. In [90], they proposed to use interval coloring which exhibited a better performance compared to [88].

A data allocation algorithm for global and stack objects is developed in [40] to optimize the allocation for WCET. The problem is formulated as an ILP and solved with a greedy heuristic that is applied iteratively to account for changes in the worst-case execution path after allocation.

Several works discussed array-based allocation of data SPM. In [77–79, 89], loop transformations are used to partition arrays into data tiles and fetch them to the SPM when used. On demand array-tiling and SPM allocation are combined in [163] using a comparability graph coloring allocator. An iteration-access-pattern-based technique in [165] overlaps SPM space between array blocks when the array elements are not used in later iterations which allows increasing the block size and decreasing the memory transfers. Absar *et al.* [4] handle tiling for irregular access pattern with indexing array referenced with an affine function. In [36, 74], a data reuse analysis is used to identify the reuse pattern of array elements within a loop to reduce the number of memory transfers. Run-time decisions are used to improve tiling approaches for irregular access pattern of array-based applications in [31, 34]. Chen *et al.* [31] handle irregular accesses using indexing array referenced with an affine function. In [34], a hardware component (DART), data access record table, records the memory access history to keep frequently accessed data in the SPM for indirectly accessed arrays with non-affine reference functions. Both approaches use compiler

generated address assignment for data layout in the SPM. Yemliha *et al.* [168] develop a model based on markov chain to predict irregular data accesses. Array-based approaches mainly target video/image processing applications where the kernel of the application is constructed from nested loops that access multi-dimensional arrays [31].

A scratchpad controller is integrated with the CPU in [75] to achieve a low overhead management of the instruction SPM. A dedicated instruction, Scratchpad Managing Instruction (SMI), is used to activate the SPM controller to stall the CPU and start copying instructions from DRAM using a Basic Block Table (BBT) that contains the required information to copy the basic blocks. The allocation algorithm uses a graph partitioning procedure to choose the points where SMI instructions should be inserted to reduce the energy consumption. A similar architectural extension is proposed in [23] where a cache-like tagging system is used for instruction SPM blocks and managed by explicit instructions inserted in the code to change the execution mode between the SPM and DRAM. A framework is introduced in [33] to dynamically allocate instructions and data in the SPM using an Address Translation Logic (ATL). Software interrupts are inserted in the binary code to invoke an SPM management routine to configure the ATL. A management technique for code placement for a memory system consisting of an SPM and a mini-cache with a memory management unit (MMU) is discussed in [46]. The technique profiles the application and classifies the code to cache-able and page-able and loads the page-able code regions on demand during run-time to the SPM. The memory management unit (MMU) and interrupts are also used in [71] to dynamically allocate pages of the program code by merging small basic blocks.

Whitham *et al.* [157,159,160] developed a scratchpad memory management unit (SMMU) to enhance the predictability of load/store operations. In their work, they depend on the separation of the logical address used in the program code and the physical address of an allocated object in the SPM to tackle the pointer aliasing and invalidation problems. The SMMU is programmed using OPEN/CLOSE operations to move data between SPM and main memory and a comparator network is used to translate logical addresses to physical addresses based on the location of the object. A comparison between the average-case performance of data caches and the worst-case performance of using SMMU with SPM in [156] showed that using SPM provided predictability while maintaining a performance similar to data caches.

The work in [44] is the only compile-time allocation for heap data. The approach in this paper is based on assigning a fixed size, which is called bin, to a dynamic structure with unknown size, like a linked list. If the size of the dynamic structure exceeds the size of the bin, the allocation is directed to the DRAM. A bin can be moved between the SPM and the main memory in the same way global and stack objects are handled in [139].

The approach does not guarantee a predictable behavior as the dynamic structure is split between the SPM and DRAM.

Dominguez *et al.* developed a compiler-directed scheme for recursive function data allocation in [43]. The approach is able to dynamically allocate a portion of recursive stack data in the SPM. The scheme incorporates a depth check for the function recursion and decides if the stack frame is profitable to allocate in the SPM using profiling.

In [16], the authors propose an allocation scheme for SPM for a higher level dataflow model of computation to make optimal use of the SPM based on memory access time and energy consumption.

The placement of the data with possible duplication in a multicore system is discussed in [62]. The paper targets executing a single task on a multicore system with virtual SPM in which local and remote accesses to distributed SPM are allowed. In their approach, the task is divided into parallel code regions that can be executed on multiple cores and dynamic movement of the data occurs between the execution of parallel regions. The optimization objective is the memory access cost considering different access times for local and remote SPM.

A prefetching technique is applied in [164] on the loop level. The authors use SPM data pipelining (SPDP) technique that utilizes Direct Memory Access (DMA) to achieve data parallelization for multiple iterations of a loop based on iteration access patterns of arrays. SPDP technique focuses on array-based applications where regular accesses can be statically analyzed. A similar technique is used in [38] to minimize the energy and maximize average performance. They propose a general prefetching scheme for on-chip memory using DMA priorities and pipelining to prefetch arrays with high reuse.

3.2.3 Run-time Allocation Techniques

Pyka *et al.* implemented a run-time allocation strategy in [118] that incorporates automated compiler transformation and operating system management of the SPM to reduce the energy consumption. The allocation scheme depends on information about the content of the SPM and the possible objects to be allocated during run-time to allocate program code, static global objects, and dynamically allocated objects. Code transformation adds locking and de-referencing layers to ensure correct addresses during run-time.

A run-time technique to manage the allocation of stack in the SPM is proposed in [110]. The technique is based on the use of Memory Management Unit (MMU) and splitting SPM into slots similar to virtual memory systems where DRAM is partitioned into pages. A

fault handler and a replacement policy are incorporated to handle the allocation during run-time. The implementation does not require additional hardware or compiler support. However, the technique lacks predictability. A similar approach is applied in [127] with a software implementation.

An OS-level run-time approach in [106] relies on annotations inserted by the programmer to provide the OS with hints to choose the most suitable memory using run-time allocator. In [63], a software management scheme of SPM that implements a fully associative cache is proposed. The scheme achieves results similar to a fully associative LRU organization which is practically infeasible using hardware. A similar approach is introduced in [104] where a compiler framework is used to substitute the tag-memory and cache controller hardware. This framework allows the selection of cache line size, replacement policy, and associations based on the program. However, such approaches add high overhead in the software. In [39], a memory reference sampling unit identifies the frequently accessed memory regions at run-time and processes the memory allocation using MMU unit in a hybrid cache/SPM system. The allocation does not require a compiler support and shows a similar performance to systems with cache only while reducing the energy consumption.

In [102], a management unit called dynamic instruction scratchpad (D-ISP) is added to the processor to allocate program code on function-based granularity with run-time address translation and FIFO replacement policy for allocating functions on demand in the SPM. The approach eliminates the interference of instruction and data memory accesses for more precision in the WCET analysis. CASA is a contention-aware allocation scheme proposed in [30] that adds a run-time cache miss tracker to allocate pages with significant misses to the SPM. The scheme uses a threshold to invoke an interrupt to move a page to the SPM and a translation lookaside buffer (TLB) to redirect the address to the SPM. SPM Allocator (SMA) is introduced in [100] as a light weight memory management for heap allocation in small on-chip memories.

In [9], a run-time SPM management approach is proposed for multicore architectures with hybrid on-chip memory comprising caches and SPM. The management scheme maps private inputs and outputs of the task to the SPM during run-time transparently to the programmer and overlaps the data transfers with task scheduler or previous task. The evaluation of the approach improved the ACET, the power consumption and the network traffic in a 32-core multicore system.

Prefetching using DMA is supported in [54]. Francesco *et al.* add a DMA engine to the processor to control the DMA transfers using a job queue such that multiple jobs are scheduled and processed one after the other without stalling the DMA. They also

provide high level functions to reserve the SPM space and manage the DMA transfers. A dynamic memory manager (DMM) to handle the allocation at run-time as the SPM space is allocated using malloc/free-like functions in the source code.

Although run-time techniques can enhance portability of the programs as they can adapt to the available SPM in the system, they lack predictable behavior.

3.3 Summary

In this chapter, we reviewed the background and the related work on SPM management. In the next chapter, we introduce our WCET-driven allocation and prefetching approach for data SPM in real-time systems.

Chapter 4

WCET-Driven Dynamic Data Scratchpad Management with Compiler-Directed Prefetching

In the last chapter, we reviewed the background and the related work on the ScratchPad Memory (SPM) management techniques. Our goal in this chapter is to develop an automated SPM management scheme that uses software prefetching to effectively manage the SPM and to tackle the memory latency problem in a predictable way. Our scheme incorporates SPM controller and a compilation flow based on the framework introduced in Chapter 2 to analyze, optimize the SPM allocation and transform the code.

The rest of the chapter is organized as follows. We introduce the framework in Section 4.1. We then show a motivating example in Section 4.2. We detail the region-based program representation as a basis for the allocation scheme in Section 4.3. Our proposed allocation mechanism is explained in Section 4.4, and the compilation flow in Section 4.5. Section 4.6 discusses the allocation algorithm, and Section 4.7 introduces the Worst-Case Execution Time (WCET) abstraction for our prefetch mechanism. Finally, we present insights into dynamic allocation and prefetching in Section 4.8 with experimental results in Section 4.9, and provide concluding remarks in Section 4.10.

4.1 Introduction

A lot of research effort has been invested in exploring different allocation schemes for code and data in SPM to exploit its benefits for performance, energy reduction and predictability. We focus on data allocation as it is more challenging than code allocation. The goal of this work is the design of an efficient and predictable prefetching technique for data SPM in real-time systems.

Static and dynamic allocation techniques discussed in Section 3.2 can provide predictability as long as it is known if the memory operation will access the SPM or the main memory. Some allocation algorithms target optimizing the WCET either statically or dynamically [40,134]. However, these techniques assume stalling the execution of the program to transfer the data between the SPM and the main memory. The performance of these techniques can be improved using prefetching to hide the transfer times by overlapping with CPU execution. Prefetching has been successful in cache-based systems, however applying prefetching to SPM-based systems needs different techniques that can be integrated with the software management schemes of the SPM.

SPM prefetching has been exploited for arrays in [164] on the loop level using software pipelining which provides significant improvements to the performance by eliminating or reducing the stalling time for memory transfers. Our work differs from their technique as they do not provide whole program management scheme and do not account for WCET.

A run-time SPM prefetching mechanism in [54] uses a Direct Memory Access (DMA) engine coupled with a job queue to schedule multiple data transfers and process them back to back without stalling the CPU or the DMA. The authors provide high level functions for the programmer to schedule the DMA operations and manage the SPM space during run-time in a heap-like style. In our work, we use a similar approach in a framework that supports automatic compile-time allocation to provide predictability and to optimize the WCET.

In real-time multitasking systems, SPM prefetching has been applied between tasks in the Predictable Execution Model (PREM) [5,101,113,151] such that DMA transfers are co-scheduled with CPU execution between different tasks. However, the proposed approaches suffer from three main limitations:

1. Statically loading all data and code before the beginning of the program severely limits the flexibility and precision of the allocation, especially if the data used by the program is dependent on the inputs and the path taken by the program through its control-flow graph.

2. DMA transfers cannot be overlapped with the execution of the same task, only other tasks. This makes the proposed approaches less suitable for many-core systems, where it might be preferable to execute a single task/thread on each core.

3. With the exception of [97], the proposed approaches assume manual code modification, which we find unrealistic in practice.

In this work, we tackle these limitations by providing a prefetching mechanism that can overlap the DMA transfers with the execution of the same task and a compiler-automated flow that can optimize the WCET.

Software prefetching has been known to add significant execution overhead in cache prefetching techniques [87]. In order to minimize the overhead, we use an SPM controller managed by the software to minimize the execution overhead.

An important issue with data allocation is the usage of pointers as they can cause incorrect execution if they are not handled properly. Most allocation techniques either assume that the program has no pointers or discard the pointer-referenced objects to avoid pointer related problems. Handling pointer references during run-time has been addressed in [139] and [157]. In [139], the authors proposed two alternatives, the first is based on run-time disambiguation of the pointer to determine if its reference is in the SPM using a compiler-inserted code and a height-balanced tree structure to translate DRAM address to SPM address which incurs high run-time overhead. We adopt a similar approach that is predictable with minimal run-time overhead using hardware SPM controller. The second alternative is to fix the location of the object in the regions where it is accessed using pointers which limits the optimization of the SPM allocation. The authors in [157] keep a unique logical address of the object that is translated to a physical address using a scratchpad management unit (SMMU) with a comparator array. The downside of this approach is the need for a per-access translation using the comparator network that is placed on the critical path of the processor. Also, there is a limitation on the size of the comparator array. We exploit the compiler analysis to avoid per-access translation by translating the address only at the pointer assignment points.

In summary, the contributions of this work are:

- We introduce an allocation mechanism for SPM that manages DMA transfers with minimum added overhead to the program. For simplicity and as a proof of concept, we implement our mechanism using a dedicated SPM controller, but we argue that a similar scheme could be supported by other platforms with the required DMA functionality.

- We implement an efficient run-time translation of pointers that does not require per-access translation and avoids pointer aliasing and invalidation issues.
- We develop an allocation algorithm for data in scratchpad memory that takes into account the overlap between DMA transfers and program execution.
- We show how to model the proposed mechanism in the context of static WCET analysis using a standard data-flow approach for processor analysis.
- We fully implement all required code analysis, optimization and transformation steps within the LLVM compiler framework [86], and test it on a collection of benchmarks. Outside of loop bound annotations, our prototype is able to automatically compile and optimize the program without any programmer intervention.

4.2 Motivating Example

In this section, we present an example that shows the benefit of data prefetching in SPM-based systems. Given a set of data *objects* used by a program, the general SPM allocation problem is to determine which subset of objects should be allocated in SPM to minimize the WCET of the program. Since the latency of accessing an object in the SPM is less than in main memory, we can compute the *benefit* in terms of WCET reduction for each object allocated in the SPM. We model the program's execution with a Control Flow Graph (CFG) where nodes represent basic blocks, i.e., straight-line pieces of code. In particular, Figure 4.1 shows the CFG of a program where object x is read/written in basic blocks BB_2 and BB_4 and object y is read in BB_4. Note that BB_2 and BB_4 are loops, since they include back-edges (i.e., the program execution can jump back to the beginning of the block); hence, x and y can be accessed many times. Assume that the SPM can only fit x or y. A static SPM allocation approach will choose to allocate either x or y for the whole program execution. A dynamic SPM allocation approach will try to maximize the benefit by possibly evicting one of the two objects to fit the other during the program execution.

Let the benefit of accessing x from the SPM instead of the main memory be 100 cycles for BB_2 and 10 cycles for BB_4. Similarly, the benefit for accessing y from the SPM in BB_4 is 70 cycles. Let the cost to transfer x from the main memory to the SPM or vice-versa be 20 cycles, and the cost for y 40 cycles. Note that individual accesses to the main memory is more costly than transferring a block from the main memory to the SPM. Hence, the benefit of accessing x/y from the SPM prevails the transfer cost. Then, for static allocation,

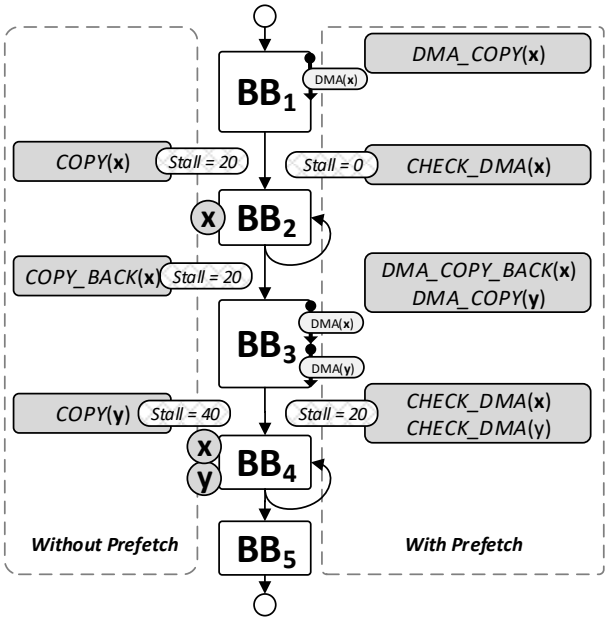


Figure 4.1: Motivating Example

the total benefit of allocating x is $100 + 10 = 110$ cycles and the cost is 2*20 cycles (fetch x from memory to SPM at the beginning of the program and write it back from SPM to main memory at the end). Similarly, the benefit for allocating y is 70 cycles and the cost is 20 cycles (fetch only as y is not modified, so there is not need to write it back to main memory). The optimal allocation would choose x as it has a net benefit of 70 cycles versus 50 cycles for y.

In previous approaches that adopt dynamic allocation, the program execution has to be interrupted to transfer objects either using a software loop or a DMA unit. We represent this case in the *without prefetch* box in Figure 4.1. In the example, x is fetched before BB_2 and written back after BB_2 to empty the SPM for y. Then, y is fetched before BB_4. Since x is allocated in the SPM for BB_2 and y is allocated for BB_4, this results in a total benefit of $100 + 70 = 170$. The program will stall before BB_2 to fetch x, after BB_2 to write-back x, and before BB_4 to fetch y. The total cost is $20 + 20 + 40 = 80$ cycles as the execution has to be stalled for each fetch/write-back transfer. The net benefit is $170 - 80 = 90$ cycles, which is 20 cycles better than the static allocation.

However, if memory transfers can be parallelized with the execution time of the program, we next show that we can exploit the SPM more efficiently. We illustrate the prefetching sequence in the *with prefetch* box in Figure 4.1. Let us assume that the amount

of execution time that can be overlapped with DMA transfers is 30 and 40 cycles for BB_1 and BB_3, respectively. We start prefetching x before BB_1 by configuring the DMA to copy x from main memory to SPM. Then, we poll the DMA before BB_2 where x is first used to ensure that the transfer has finished. Since transferring x requires less cycles than the maximum overlap for BB_1 (20 versus 30), the prefetch operation for x finishes in parallel with the execution of BB_1; hence, there is no need to stall the program before x can be accessed from the SPM in BB_2. Before BB_3, we first write-back x so that we have enough space in the SPM to then prefetch y. We propose to schedule both transfers back-to-back, *e.g.* using a scatter-gather DMA, in parallel with the execution of BB_3. Since the amount of overlap for BB_3 is 40, the write-back for x completes after 20 cycles, leaving 20 additional cycles of overlap for the prefetch of y. Hence, by the time BB_4 is reached, the CPU stalls for $40 - 20 = 20$ cycles to complete prefetching y before using it in BB_4. For the described prefetching approach, the benefit is the same as the dynamic allocation. However, the cost is lower as the CPU only stalls for 20 cycles. The net benefit is $170 - 20 = 150$ cycles, compared to 90 cycles without prefetching.

4.3 Region-Based Program Representation

The motivating example shows that the cost of copying objects between main memory and SPM can be reduced by overlapping DMA transfers with program execution. However, to achieve a positive benefit, we also need to predict whether any given memory access targets the SPM rather than main memory. In general, programs contain branches and function calls, making such determination possibly dependent on the execution path. To produce tight WCET bounds, a fundamental goal of our approach is to *statically determine which memory accesses are in the SPM regardless of the flow through the program.* To achieve this objective, we employ the refined region structure that we introduced in Section 2.2.

For this reason and to simplify figures, in the following sections we omit drawing the node inside each trivial region when representing the CFG. Since allocations are based on regions, for simplicity we will omit individual nodes when representing CFGs and instead draw regions.

4.4 Allocation Mechanism

As discussed in the motivational example, to efficiently manage the dynamic allocation of multiple objects we require a DMA unit capable of queuing multiple operations. In gen-

eral, many commercial DMA controllers with scatter-gather functionality support such a requirement, albeit the complexity of managing the DMA controller in software and checking whether individual operations have been completed could increase with the number of transfers. As a proof of concept, in the following section we describe the implementation based on a dedicated SPM controller, reserving implementation on a COTS platform as future work [1].

4.4.1 Assumptions

In the rest of this chapter, we assume the following:

- We adopt the refined region structure that we introduced in Section 2.2 for the allocation. We showed in the motivating example that the cost of copying objects between main memory and SPM can be reduced by overlapping DMA transfers with program execution. However, to achieve a positive benefit, we also need to predict whether any given memory access targets the SPM rather than main memory. In general, programs contain branches and function calls, making the prediction of the target of a memory access possibly dependent on the execution path. To produce tight WCET bounds, a fundamental goal of our approach is to *statically determine which memory accesses are in the SPM regardless of the flow through the program.* The region structure fits our purpose as any region has a single entry and a single exit. This means that allocating an object in the SPM in a region, implies that any access to that object during the execution of any path in this region is guaranteed to access the SPM.

- We focus solely on the allocation of data SPM, as it is generally more challenging. We assume a separate instruction SPM that is large enough to fit the code.

- The allocation is object-based, meaning that we do not allow allocation of parts of an object. Transformations like tiling and software pipelining could further improve the allocation, especially for small sizes of SPM. We keep this possible expansion to future work.

- We assume that the target program does not use recursion or function pointers and that local objects have fixed or bounded sizes. We argue that these assumptions

[1] For example, the Freescale MPC5777M SoC used in previous work [136] includes both SPM memory and a dedicated I/O processor that could be used to implement the described management functionalities.

conform with standard convention for real-time applications. Also, we do not analyze system calls as we do not include OS support in our prototype.

- We assume that all loops in the program are bounded. The bounds can be derived using compiler analysis, annotations or profiling.
- We employ pointer analysis to determine the references of the load/store instructions. A points-to set is composed for each pointer reference. The size of the points-to set depends on the precision of the pointer analysis. We utilize the pointer analysis from [3] that is based on inter-procedural static value-flow analysis [135]. Allocation-site abstraction is used for dynamically allocated objects to represent objects, i.e., to consider a single abstract object to stand in for each run-time object allocated by the same instruction [130]. To be able to allocate a dynamically allocated object, an upper bound on the size of the object should be provided at compile-time.
- For simplicity, we focus on a system comprising a single core running one program. However, the proposed method could be extended to a multicore system supporting a predictable arbitration for main memory as long as each core is provided with private or partitioned on-chip memory.

4.4.2 SPM controller

We define a set of allocation commands that are inserted in the code and executed by the *SPM controller*. The SPM controller is a memory mapped unit connected to the processor to manage the on-chip fast data memory. Cache memory is transparent to the software and the cache controller implicitly accesses the caches by mapping the memory address to the cache. In contrast, the SPM has a distinct address range that is used to access its content.

Figure 4.2 shows the proposed SPM controller connections to an SPM-based system. There is a separate instruction SPM (I-SPM) that is assumed to fit the code of the program. The data SPM is managed by the SPM controller through the DMA. The system incorporates a DMA unit for memory transfers. The D-SPM is assumed to have dual-ports, which means that the CPU can access the SPM while the DMA transfers data between SPM and main memory. The allocation method and WCET analysis can be applied for single-port SPM, but this will offer less opportunity to overlap the memory transfers. The main memory is connected to a shared bus the arbitrates the memory access between the CPU or the DMA. To efficiently support the parallization of memory transfers with the execution time, the DMA is designed to work in transparent mode. That is, the DMA

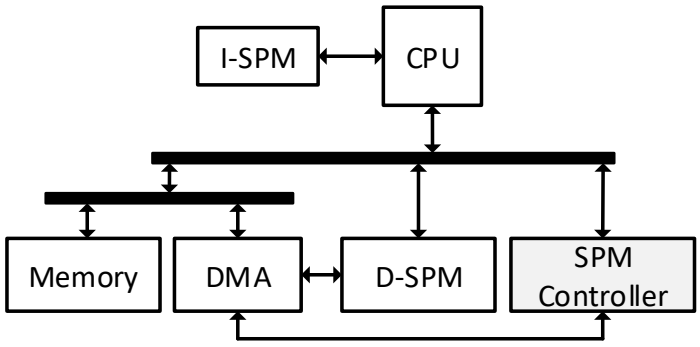


Figure 4.2: SPM-based System

transfers an object only when the CPU is not using the main memory. Whenever the CPU requests the main memory bus, the DMA yields to the request and stalls any ongoing transfer until the memory bus is released.

The general process of allocation starts with reserving the space in the SPM and copying the object from the main memory if necessary. Then, the references of the object are directed to the new address in the SPM. Finally, the object is evicted from the SPM and written-back to the main memory if necessary. We use allocation commands to achieve this process: **ALLOC***XX*, **GETADDR**, **DEALLOC**, **SETPTR**, **SETMM**, **SETSIZE**.

The purpose of using this controller is to manage the DMA transfers between the main memory and the SPM, keep track of the allocated objects, and resolve pointers during run-time. Figure 4.3 shows the components of the SPM controller. The following is a description of these components:

CMD Decoder As the SPM controller is a memory mapped device, we use load/store operations to implement the allocation commands as we will discuss later in this section. The CMD decoder decodes the embedded commands in the data and address to produce the corresponding command (CMD).

CMD Queue The execution of the allocation commands may require multiple cycles depending on the implementation of the controller. So, the allocation queue is used to store the commands until they are executed in FIFO order. Using this queue hides the latency by releasing the bus after one cycle if the command is not blocking. However, the allocation queue is an optional component that can be eliminated if the allocation commands execute in one cycle or if it is acceptable to stall until the command is executed in case of multiple cycles.

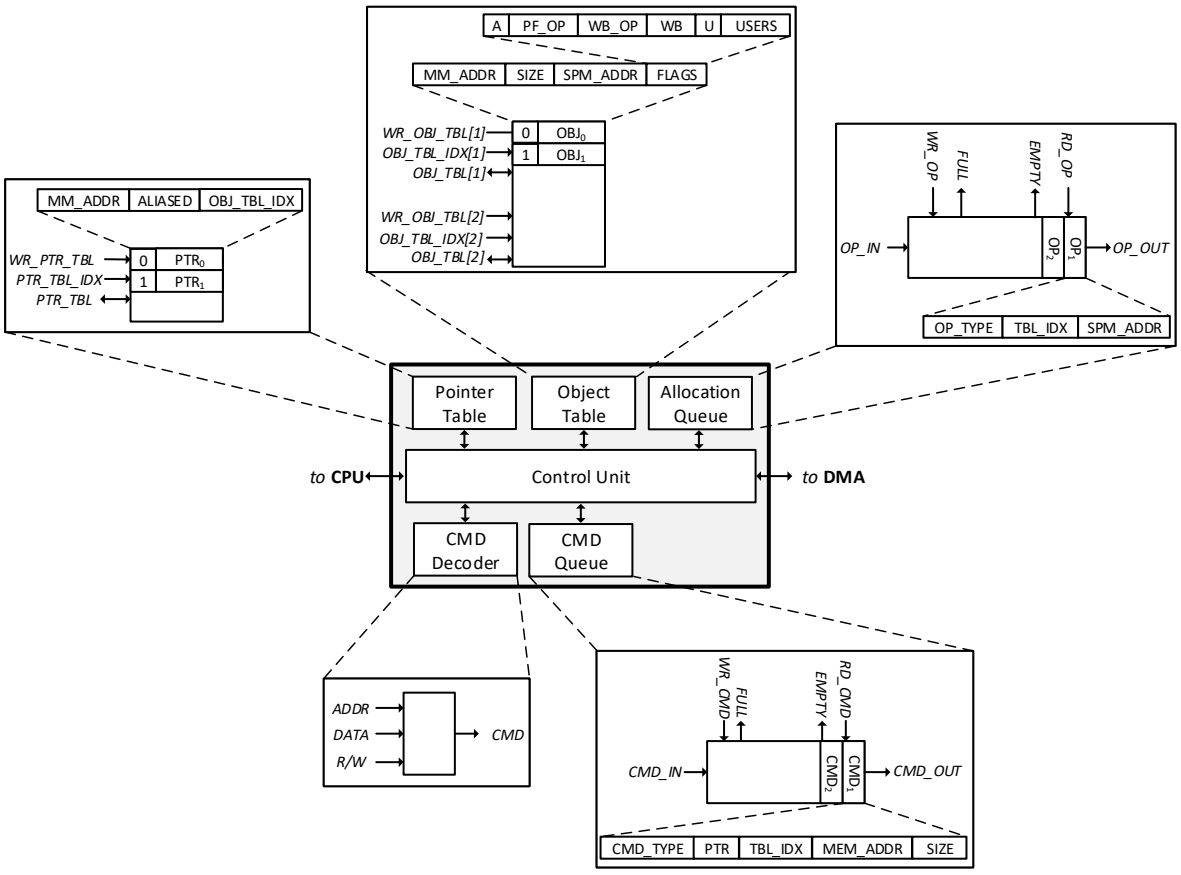


Figure 4.3: Data SPM Controller

Object Table The object table stores the status of the allocated objects and the required information to manage the DMA and resolve pointers. As shown in Figure 4.3, each entry in the table identifies an object using its main memory address (MM_ADDR) and its size ($SIZE$), and its SPM address (SPM_ADDR) if the object is allocated in the SPM. The allocation status of the object is tracked using a set of flags ($FLAGS$):

V	the entry is (V)alid and associated with an object
A	(A)llocated in the SPM
PF_OP	(P)re(F)etching (OP)eration has been scheduled
WB_OP	(W)rite-(B)ack (OP)eration has been scheduled
WB	(W)rite-(B)ack when de-allocated if used
U	(U)sed in the SPM
$USERS$	number of current users (allocations) of the object

These flags are updated based on the allocation commands (**ALLOC**XX, **DEALLOC** and **GETADDR**) and the DMA transfers.

Pointer Table The pointer table is used to store pointers during run-time to determine if the pointee of a pointer is allocated in the SPM. An entry in the pointer table contains the pointer value (MM_ADDR), a flag to indicate if the pointee exists in the object table ($ALIASED$) and the index of the pointee entry in the object table (OBJ_TBL_IDX). The pointer table is managed using the command **SETPTR** as we will discuss later.

Allocation Queue The allocation queue allows scheduling multiple DMA transfers and executing them in FIFO order. An entry in the queue comprises the operation type (OP_TYPE) -prefetch or write-back-, the index of the object to be transferred in the object table (TBL_IDX), and the SPM address (SPM_ADDR). When the operation is at the front of the queue, the object entry is checked in the object table for the main memory address (MM_ADDR) and the size ($SIZE$) to pass to the DMA along with the SPM address (SPM_ADDR). Note that the object table is also checked for the allocation flags to determine if the scheduled operation is not canceled.

Control Unit the control unit is responsible for communicating with the CPU and the DMA, executing the allocation commands, updating the tables, and controlling the queues. Appendix A explains how the control unit works in detail.

Note that the controller does not perform per-access translation from main memory addresses to SPM addresses as the SPM is addressed directly in the code. Hence, the controller does not add overhead to the critical path of the processor unlike the proposed unit in [157].

4.4.3 Allocation Commands

We propose a set of load/store operations called *allocation commands* to manage the SPM controller. The allocation commands are used to achieve the following purposes during run-time:

- Reserve/release a space for an object in the SPM and trigger DMA transfers if required using **ALLOC***XX*/**DEALLOC** commands.
- Translate main memory address to SPM address when required using **GETADDR** command.
- Disambiguate pointers using **SETPTR** command.
- Update the object table using **SETMM** and **SETSIZE** commands.

We will explain the functionality of these commands in this section and illustrate how to use them for different allocation cases in Section 4.4.4.

- **SETPTR** command targets the pointer table in the SPM controller.

  SETPTR *TBL_IDX, MEM_ADDR*

  The entry at *TBL_IDX* in the pointer table is configured with a main memory address *MEM_ADDR* which is compared with the main memory address range of the objects with valid entries in the object table to find the entry of the pointee object. If the pointee is found, *ALIASED* flag is set and the table index of the object is stored in *OBJ_TBL_IDX* of the pointer table entry. All the allocation commands on pointers checks the pointer entry for the aliasing object to use in allocation. The alias checking process can be implemented in one cycle using a one-shot comparator or over multiple cycles comparing one entry at a time. If **SETPTR** command is executed over multiple cycles, the command queue is used to keep the order of commands as the command is non-blocking and its execution is overlapped with the

CPU execution. If the number of entries in the object table is large, an alias set that specifies which objects that can alias with the pointer can be used to reduce the number of comparisons. For the sake of simplicity of analysis, we assume in this work a one-cycle implementation.

- **ALLOC***XX* command reserves the space in the SPM and schedules a DMA transfer if necessary. The command has the following syntax:

  ALLOC*XX* *TBL_IDX, PTR, MEM_ADDR*

  It requires a table index for an object/pointer (*TBL_IDX*) and an SPM address (*MEM_ADDR*). The *PTR* flag distinguishs between an allocation of an object or a pointer. If the allocation is for a pointer, the index for the pointee in the object table is *OBJ_TBL_ADDR* field in the entry at *TBL_IDX* of the pointer table. **ALLOC***XX* command for a pointer must be preceded with **SETPTR** command to set the corresponding pointer entry. This mechanism to translate pointer table index to object table index is used also for **DEALLOC** and **GETADDR** commands.

  There are four versions of **ALLOC***XX* command according to the flags (*XX*): **ALLOC**, **ALLOC***P*, **ALLOC***W*, **ALLOC***PW*. The *P* flag directs the controller to prefetch the object from the main memory. The SPM controller will schedule a prefetch transfer for the object and set *PF_OP* flag in the object entry. The *P* flag is used if the object has an initial value or has been previously written. If the *P* flag is not used, the object is allocated directly and *A* flag is set. Otherwise, *A* flag is set once the prefetch transfer completes. The *W* flag informs the controller that this object should be copied back to the main memory when de-allocated if it has been used as it might have been modified. The *W* flag sets the *WB* flag in the object entry.

- **DEALLOC** command de-allocates the object/pointer in table index (*TBL_IDX*).

  DEALLOC *TBL_IDX, PTR*

  If the *WB* and *U* flags are set in the object entry in the object table, the controller will schedule a write-back transfer, set *WB_OP* flag and reset *A* flag. Otherwise, the object will be de-allocated by simply resetting *A* flag.

  If a prefetch transfer has been scheduled and a **DEALLOC** command is issued for the object to be prefetched, the transfer is canceled as the object is not needed anymore. Also, if a write-back transfer has been scheduled for an object and it was followed by **ALLOC***XX* for the same object, the transfer is canceled if the object is

allocated to the same SPM address, otherwise the transfer is not canceled. This is particularly important for allocations within loops, when the object can be allocated to the same address over multiple iterations.

- **GETADDR** command returns the current address of the object.

  GETADDR *TBL_IDX, PTR*

  If PF_OP or WB_OP flag is set in the the object entry in the object table, the controller stalls until the DMA completes transferring the object. If no transfer is scheduled or after the transfer finishes, the controller returns SPM_ADDR if A flag is set and MM_ADDR otherwise.

  GETADDR command is inserted only before the first use of the object/pointer after the allocation/de-allocation in the SPM. The address returned by the command is then applied for all the next uses until another allocation/de-allocation occurs. This process is compiler-automated and it eliminates the per access address translation used in previous approaches [157]. For pointers, **SETPTR** and **GETADDR** commands are required if the pointer can alias with the content of the SPM even if the pointer itself is not allocated.

- **SETMM** and **SETSIZE** commands are used to configure the entry at *TBL_IDX* in the object table with the information of an object.

  SETMM *TBL_IDX, MEM_ADDR*

  SETSIZE *TBL_IDX, SIZE*

  These commands are used to initialize the object table. Also, it can be used to add the information dynamically-allocated objects or change the set of objects tracked by the table during run-time.

We explain in Appendix A the encoding and IR representation of the allocation commands and the abstraction of the SPM controller and its initialization process.

4.4.4 Example

Figures 4.4, 4.5 depict an example for the allocation process. There are two objects x and y corresponding to entries 3 and 5 in the object table. Also, a pointer p is an argument to function f and uses entry 0 in the pointer table. Figure 4.4 shows the CFG of two

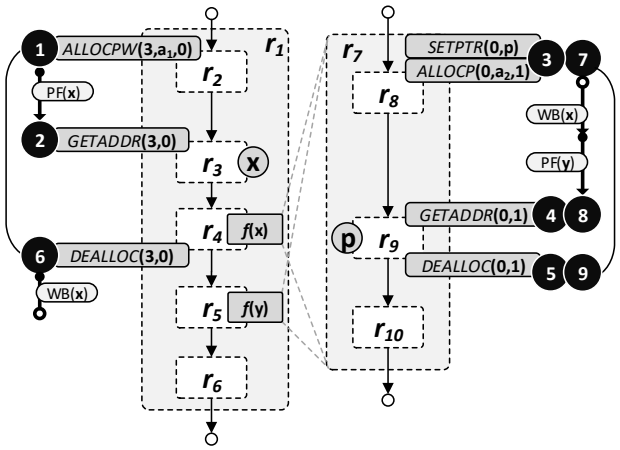


Figure 4.4: Allocation Example

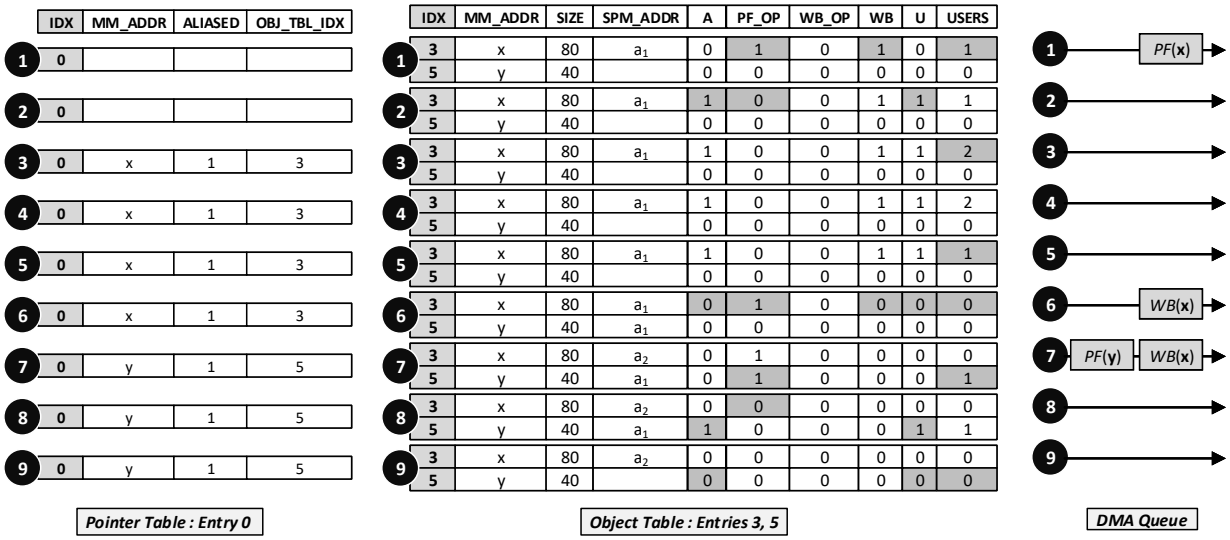


Figure 4.5: SPM Controller State for Allocation Example in Figure 4.4

functions where r_1-r_{10} represent regions. x is read/written in r_3, and the pointee of p is read in r_9. Note that function f, comprising regions r_7 to r_{10}, is called from two different call regions, r_4 with $p = \&x$ and r_5 with $p = \&y$. In the example, we assume that x is allocated at address a_1 in the SPM in sequentially composed regions r_2, r_3 and r_4. The argument of function f is allocated at a different address a_2 inside the function.

We use program points ❶ to ❾ to follow the allocation process. Entries 3 and 5 of the

object table, Entry 0 of the pointer table and the DMA queue are traced in Figure 4.5 for these program points. At ❶, x is allocated to address a_1 with P and W flags. In the object table, PF_OP is set to indicate x is being prefetched, WB is set to indicate a write-back when de-allocated, and $USERS$ is incremented. A prefetch transfer $PF(x)$ is scheduled in the DMA queue. At ❷, **GETADDR** checks entry 3 for the address; as $PF(x)$ did not finish at this point, the CPU is stalled. When the prefetch finishes, x is allocated, PF_OP is reset, A is set; and the CPU continues execution. Also, U is set to mark x as used in the SPM. In r_4, function f is called. At ❸, **SETPTR**$(0, p)$ sets the address for p in entry 0 of the pointer table and apply alias checking with the object table. As p aliases with x at this point,flag $ALIASED$ is set and OBJ_TBL_IDX refers to entry 3 in the object table. An allocation of p to a_2 is issued; however, its pointee x is already in the SPM at address a_1. So, no new allocation at a_2 is performed, and $USERS$ is incremented in entry 3 to indicate that two **ALLOC** commands (users) have been executed for x. **GETADDR** at ❹ returns a_1. When p is deallocated at ❺, $USERS$ is decremented in entry 3 of the object table. However, x is not evicted as there is another user for it. When x is deallocated at ❻, x is evicted as this is the last user of x in the SPM. As WB and U are set, a write-back is scheduled for x. f is called again in r_5. The same process to set entry 0 in the pointer table at point ❼ is done with the address of y and OBJ_TBL_IDX refers to entry 5 in the object table. Then, p is also allocated to a_2 with P flag. So, a prefetch is scheduled for y. Before p is used in r_9, **GETADDR** is executed. At this point the write-back transfer of x is done and the prefetch for y is partially completed. The CPU stalls till y is completely prefetched, then U flag is set in entry 5, address a_2 is returned, and the execution continues at ❽. Finally, p is deallocated at ❾ which evicts y. No write-back is needed as WB flag is not set.

An essential observation is that the state of the SPM and the sequence of DMA operations in function f depend on which region calls f: if f is called from r_4, then x is already available in SPM at address a_1, and the allocation of p to a_2 is not used. If instead f is called from r_5, p is allocated to a_2 and the object y must be prefetched from main memory. Therefore, let σ be the *context* under which a region executes, i.e., the sequence of call regions starting from the main function; note that since the main function of the program is not called by any other function, the only valid context for regions in the main is $\sigma = \emptyset$. We denote the execution of a region r_n in a context σ as r_n^σ, which we call a *region-context* pair. Then, allocation decisions, which involve adding allocation commands in the code, must be based on regions, but the state of the SPM and DMA operations, which are needed for WCET estimation, depend on region-context pairs. Intuitively, this is equivalent to considering multiple copies of each region r_n, one for each context in which r_n can execute.

4.5 Compilation Flow

The proposed SPM allocation algorithm in this chapter optimizes which objects to allocate, where in the SPM, and at which program point using a set of heuristics detailed in Section 4.6. These heuristics are integrated with our compilation framework introduced in Chapter 2. We utilize the analysis information about regions, loops, memory accesses, and the backend of the program along with WCET analysis for the optimization. The WCET analysis is estimated based on an abstract interpretation approach as detailed in Section 4.7. The allocation results are used for transforming the Intermediate Representation (IR) of the program, then the assembly code for the transformed IR is generated. The final executable is created using a linker script that specifies the memory sections representing the connected units; *i.e.*, instruction SPM, data SPM, SPM controller, and main memory. We focus in the rest of this section on the IR transformation pass.

4.5.1 IR Transformation

The transformation of the program IR to apply the SPM allocations includes the following:

- Insertion of **ALLOC***XX* and **DEALLOC** commands at load/unload program points.
- Insertion of **SETPTR** commands before allocation of pointers and before checking for pointer aliasing.
- Insertion of **GETADDR** commands before first uses of objects/pointers after allocations and de-allocations.
- Modifying the uses of the allocated object/pointer in the scope of allocation to use the address returned by **GETADDR** commands.

We collect the information required for IR code transformation traversing the program CFG accounting for SPM allocations.

ALLOC*XX* (**DEALLOC**) command is inserted before the entry (exit) of an region. For trivial regions, **ALLOC***XX* (**DEALLOC**) is inserted before the first IR instruction in the region. Otherwise, a basic block is inserted before the entry (exit) of the region with **ALLOC***XX* (**DEALLOC**) command. For a top-level region of a function where there is no exit node, **DEALLOC** command is inserted before the return instruction of the function.

SETPTR command is used with pointers to setup the pointer table either for a pointer allocation or to check for aliases in the SPM. If the pointer will be allocated in the SPM, **SETPTR** command is inserted before the **ALLOC** command. If the pointer can alias with the content of the SPM, **SETPTR** is inserted before the usage of the pointer to access the memory.

GETADDR command is inserted before the first use of an allocated object/pointer. That is, we traverse the different paths till a load/store instruction to the object/pointer is found after allocation/de-allocation points. This also applies for a pointer that has a possible allocated object in its points-to set.

The load/store instruction is referenced to the IR instruction that calculates the memory address of the instruction and **GETADDR** command is inserted before it. The reference to the object/pointer is then modified to the return address of the **GETADDR** command. The same modification is applied for other load/store references after the first use without the insertion of another **GETADDR** command.

For de-allocated objects/pointers, we need to ensure that the write-back transfer -if scheduled- is done before the next use of the object/pointer in the main memory. So, **GETADDR** command is inserted before the first load/store instruction without changing references as the purpose is to check the DMA transfer.

As an example, consider an access to element m of the global array $x[10]$ inside a loop while m is changing every iteration.

```
1 %ptr = getelementptr inbounds [10x32], [10x32]* @x, i32 0, i32 %m
2 %t = load i32, i32* %ptr
```

Assume that the object x is at entry 0 of the object table and it has been allocated. The transformation of this access will be:

```
1 %getaddr_ptr = getelementptr [32 x [8 x i32]], [32 x [8 x i32]]* ↩
    @SPM_CONT, i32 0, i32 0, i32 5
2 %addr = load i32, i32* %getaddr_ptr
3 ....
4 %ptr = getelementptr inbounds [10x32], [10x32]* @addr, i32 0, i32 %m
5 %t = load i32, i32* %ptr
```

The first two lines are the implementation of **GETADDR** command to entry 0. This command is inserted before the loop, then any access to x inside the loop is directed to use *addr* which is the current address of x in the SPM.

Note that the scope of the transformation is local as the addresses of the objects/pointers are not changed. The transformation is only applied to specific accesses to memory

when a possible reference to an allocated object/pointer is detected. SPM controller guarantees the correctness of the address during run-time.

After the transformation, the pointer analysis cannot determine that the access to *addr* is actually an access to x as the reference chain will refer it to the SPM controller. However, we need to differentiate between allocation commands and accesses to objects to be able to conduct the WCET analysis after the transformation. So, we add metadata to the added/modified instructions to reflect their usage after the transformation.

4.6 Allocation Algorithm

4.6.1 Problem Description

The dynamic allocation is the problem of *choosing the content of the data SPM at each program point to minimize the program WCET*. We divide the problem into the following sub-problems in the context of prefetching support:

- Decide which objects should be moved to the SPM and which objects should be evicted.
- Determine the program points to prefetch and write-back objects so that the prefetch distance is large enough to hide the memory transfer time.
- Assign spaces in the SPM for the allocated objects to assure that objects that exist in the SPM simultaneously do not have overlapped address ranges.
- Provide predictable memory accesses to be able to derive a bound on the WCET of the program. An object that is accessed at a program point can be classified to: must be in the SPM, must be in the main memory, may be in the SPM.

The design space for the allocation problem comprises every program point, *i.e.*, every instruction in the program. This space is very large while most of these program points do not provide different solutions and can complicate the WCET analysis. For this reason, we reduce the allocation problem design space by adopting the region structure described in Section 4.3.

We now discuss how to determine a set of allocations for the entire program with the objective to minimize the program's WCET. For the remainder of the section, we let S_{SPM}

denote the size of the SPM. $V = \{v_1, \ldots, v_j, \ldots\}$ is the set of allocatable objects, where $S(v_j)$ denotes the size of object v_j. We let $R = \{r_1, \ldots, r_n, \ldots\}$ be the set of program regions across all functions. Without loss of generality, we assume that region indexes are topologically ordered, so that each parent region has smaller index than its children, each call region has smaller index than the regions in the called function, and sequentially composed regions have sequential indexes; this is also the order used in Figure 4.4. Note that such topological order must exist since the refined region tree for each function is unique, and furthermore the call graph has no loops due to the absence of recursion. To define the relation between region-context pairs we introduce a parent function $\wp(r_n^\sigma)$ for a region-context r_n^σ in function f as follows: if r_n is the root region of the refined region tree for f, then $\wp(r_n^\sigma) = r_m^{\sigma'}$, where $r_m^{\sigma'}$ is the region-context that calls f in context σ. Otherwise, $\wp(r_n^\sigma) = r_m^\sigma$, where r_m is the parent region of r_n. As an example based on Figure 4.4, assume that r_4 executes in context σ. Then when r_7 is called from r_4, r_7 executes in context $\sigma \cup r_4$. We further have $\wp(r_7^{\sigma \cup r_4}) = r_4^\sigma$, while for example $\wp(r_8^{\sigma \cup r_4}) = r_7^{\sigma \cup r_4}$. Finally, to generalize the problem for the usage of pointers, let $P = \{p_1, \ldots, p_k, \ldots\}$ be the set of pointers in the program. As the pointee of the pointer can change based on the program flow, we define $\chi_{r_n^\sigma}(p_k)$ as the points-to set for pointer p_k in region-context r_n^σ. For simplicity, we refer to the allocation of the pointee of pointer as the allocation of the pointer. We define $S_{r_n^\sigma}(p_k)$ as the size of p_k in region-context r_n^σ which is computed as:

$$S_{r_n^\sigma}(p_k) = \max_{v \in \chi_{r_n^\sigma}(p_k)} S(v)$$

Note that the pointee of p_k can be a global, local objects from the set of objects in V or a dynamically allocated object. In case of dynamic allocation, v refers to a dynamic allocation site in the program.

We begin by formalizing the conditions under which a set of allocations are feasible as a satisfiability problem. This is similar to a multiple knapsack problem where regions are knapsacks (available space in SPM), except that we add additional constraints to model the relation between regions. Remember that to allocate an object v_j (pointer p_k) in a region r_n, we have to assign an address in the SPM to the object (pointer). Hence, an allocation solution is represented by an assignment to the following decision variables over all regions $r_n \in R$ and all objects $v_j \in V$ (pointers $p_k \in P$):

$$alloc_{r_n}^{v_j} = \begin{cases} 1, & \text{if } v_j \text{ is allocated in } r_n \\ 0, & \text{otherwise} \end{cases}$$

$assign_{r_n}^{v_j}$ = address assigned to v_j in r_n

$$alloc_{r_n}^{p_k} = \begin{cases} 1, & \text{if } p_k \text{ is allocated in } r_n \\ 0, & \text{otherwise} \end{cases}$$

$assign_{r_n}^{p_k}$ = address assigned to p_k in r_n

An allocation solution is feasible if the allocated objects fit in the SPM at any possible program point. As discussed in Section 4.4.4, the state of the SPM depends on the context under which a region is executed. Hence, we introduce new helper variables to define the availability of an object v_j (pointer p_k) in a region-context r_n^σ:

$$avail_{r_n^\sigma}^{v_j} = \begin{cases} 1, & \text{if } v_j \text{ is available in SPM for execution of } r_n^\sigma \\ 0, & \text{otherwise} \end{cases}$$

$address_{r_n^\sigma}^{v_j}$ = address of v_j in the SPM during execution of r_n^σ

$$avail_{r_n^\sigma}^{p_k} = \begin{cases} 1, & \text{if } p_k \text{ is available in SPM for execution of } r_n^\sigma \\ 0, & \text{otherwise} \end{cases}$$

$address_{r_n^\sigma}^{p_k}$ = address of p_k in the SPM during execution of r_n^σ

We can determine the value of the helper variables based on the allocation. We first discuss the basic constraints assuming that the points-to information is not available. In this case, allocation of a pointer is handled as an allocation of an object. After that, we discuss how the constraints can be modified to consider aliasing between objects and pointers.

Basic Constraints

We present a set of necessary and sufficient constraints for an allocation problem in which points-to information is not available.

$$\forall v_j, r_n^\sigma : alloc_{r_n}^{v_j} \vee avail_{\wp(r_n^\sigma)}^{v_j} \Leftrightarrow avail_{r_n^\sigma}^{v_j}. \tag{4.1}$$

$$\forall p_k, r_n^\sigma : alloc_{r_n}^{p_k} \vee avail_{\wp(r_n^\sigma)}^{p_k} \Leftrightarrow avail_{r_n^\sigma}^{p_k}. \tag{4.2}$$

Equation 4.1 (4.2) simply states that v_j (p_k) is available in the SPM during the execution of r_n^σ if either v_j (p_k) is allocated in r_n, or if v_j (p_k) was already available in the SPM during the execution of the parent region-context pair.

$$\forall v_j, r_n^\sigma : avail_{\wp(r_n^\sigma)}^{v_j} \Rightarrow address_{r_n^\sigma}^{v_j} = address_{\wp(r_n^\sigma)}^{v_j}. \tag{4.3}$$

$$\forall p_k, r_n^\sigma : avail^{p_k}_{\wp(r_n^\sigma)} \Rightarrow address^{p_k}_{r_n^\sigma} = address^{p_k}_{\wp(r_n^\sigma)}. \tag{4.4}$$

$$\forall v_j, r_n^\sigma : \neg avail^{v_j}_{\wp(r_n^\sigma)} \wedge alloc^{v_j}_{r_n} \Rightarrow address^{v_j}_{r_n^\sigma} = assign^{v_j}_{r_n}. \tag{4.5}$$

$$\forall p_k, r_n^\sigma : \neg avail^{p_k}_{\wp(r_n^\sigma)} \wedge alloc^{p_k}_{r_n} \Rightarrow address^{p_k}_{r_n^\sigma} = assign^{p_k}_{r_n}. \tag{4.6}$$

Equations 4.3, 4.5 (4.4, 4.6) specify the address in the SPM for objects (pointers). If the object (pointer) was already available in the parent region-context, then the address is the same. Otherwise, if the object (pointer) is allocated in r_n, then the address is the one assigned by the allocation.

Finally, given the object (pointer) availability and address for each region-context pair, we can express the feasibility conditions for the allocation problem.

$$\forall v_j, r_n^\sigma : avail^{v_j}_{r_n^\sigma} \Rightarrow address^{v_j}_{r_n^\sigma} + S(v_j) \leq S_{SPM}. \tag{4.7}$$

$$\forall p_k, r_n^\sigma : avail^{p_k}_{r_n^\sigma} \Rightarrow address^{p_k}_{r_n^\sigma} + S_{r_n^\sigma}(p_k) \leq S_{SPM}. \tag{4.8}$$

$$\begin{aligned} \forall v_j, v_k, r_n^\sigma, j \neq k : (avail^{v_j}_{r_n^\sigma} \wedge avail^{v_k}_{r_n^\sigma}) \Rightarrow \\ (address^{v_j}_{r_n^\sigma} + S(v_j) \leq address^{v_k}_{r_n^\sigma}) \vee \\ (address^{v_k}_{r_n^\sigma} + S(v_k) \leq address^{v_j}_{r_n^\sigma}) \end{aligned} \tag{4.9}$$

$$\begin{aligned} \forall p_j, p_k, r_n^\sigma, j \neq k : (avail^{p_j}_{r_n^\sigma} \wedge avail^{p_k}_{r_n^\sigma}) \Rightarrow \\ (address^{p_j}_{r_n^\sigma} + S_{r_n^\sigma}(p_j) \leq address^{p_k}_{r_n^\sigma}) \vee \\ (address^{p_k}_{r_n^\sigma} + S_{r_n^\sigma}(p_k) \leq address^{p_j}_{r_n^\sigma}) \end{aligned} \tag{4.10}$$

$$\begin{aligned} \forall v_j, p_k, r_n^\sigma : (avail^{v_j}_{r_n^\sigma} \wedge avail^{p_k}_{r_n^\sigma}) \Rightarrow \\ (address^{v_j}_{r_n^\sigma} + S(v_j) \leq address^{p_k}_{r_n^\sigma}) \vee \\ (address^{p_k}_{r_n^\sigma} + S_{r_n^\sigma}(v_k) \leq address^{v_j}_{r_n^\sigma}) \end{aligned} \tag{4.11}$$

Equation 4.7 (4.8) states that if v_j (p_k) is in the SPM during the execution of r_n^σ, then it must fit within the SPM size. Equations 4.9 to 4.11 state that if two objects/pointers are in the SPM during the execution of r_n^σ, then their addresses must not overlap. Note that the size for a pointer is dependent on the region-context pair. Giving a points-to set $\chi_{r_n^\sigma}(p_k)$, the size required for allocating p_k in r_n^σ is the maximum size of all objects in the points-to set.

$$\forall p_k, r_n^\sigma : def^{p_k}_{r_n^\sigma} \Rightarrow \neg alloc^{p_k}_{r_n} \tag{4.12}$$

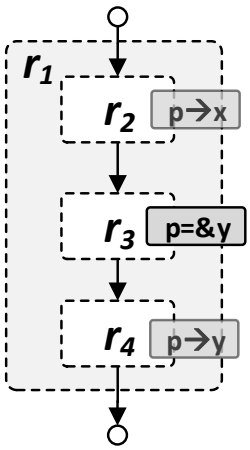


Figure 4.6: Example of Pointer Definition

The constraint in Equation 4.12 states that the allocation of pointer p_k is not allowed in r_n ($alloc_{r_n}^{p_k} = 0$) if $def_{r_n^\sigma}^{p_k} = 1$ where $def_{r_n^\sigma}^{p_k}$ is defined as following:

$$def_{r_n^\sigma}^{p_k} = \begin{cases} 1, & \text{if } p_k \text{ is defined in } r_n^\sigma, \textit{ i.e.}, \ p_k \text{ can change its reference in } r_n^\sigma \\ 0, & \text{otherwise} \end{cases}$$

This constraint is required for correctness of execution and analysis. This case is depicted in Figure 4.6 where pointer p is defined in region r_3 to point to y rather than x. Hence, p can be allocated in r_2 and r_4 while $alloc_{r_3}^{p} = 0, alloc_{r_1}^{p} = 0$. The allocation of p in r_3 or r_1 will result in pointer invalidation as any reference to p after its definition in r_3 should point to y not x.

As long as Equations 4.7 to 4.11 are satisfied for a given solution in all region-context pairs, all objects fit in the SPM; hence, the allocation problem can be feasibly implemented. To do so, we next discuss how to determine the list of commands (**ALLOC**/**DEALLOC**/**GETADDR**) that must be added to each region. For a region r_n that is not sequentially composed, an **ALLOC** is inserted at the beginning of the region and a **DEALLOC** at the end of the region.

In the case of sequential regions, to reduce the number of DMA operations, we note the following: if the same object v_j is allocated in two sequentially composed regions r_p and r_q with the same assigned address, then there is no need to **DEALLOC** v_j at the end of r_p and **ALLOC** it again at the beginning of r_q. Hence, we consider the maximal sequence of sequentially composed regions $r_p, \ldots, r_q$ such that for every region r_n in the sequence:

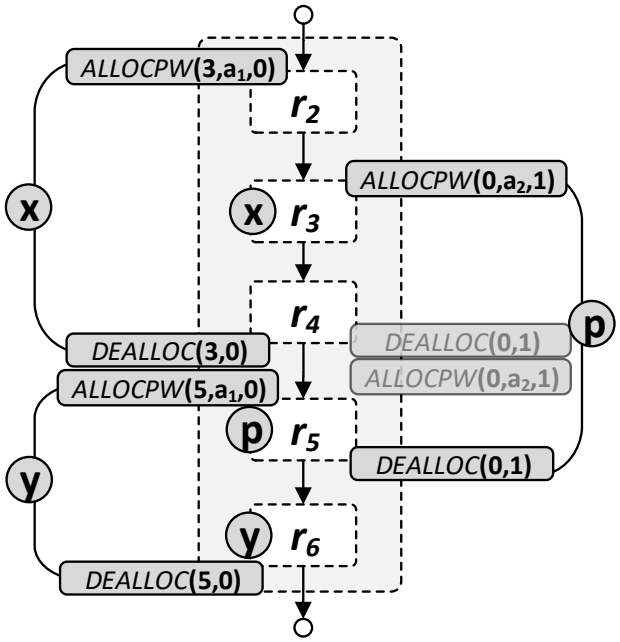


Figure 4.7: Allocation Overlap Example

$alloc_{r_n}^{v_j} = 1$ and the address $assign_{r_n}^{v_j}$ assigned to v_j is the same. We then add the **ALLOC** command at the beginning of r_p and the **DEALLOC** command at the end of r_q. The P and W flags of the **ALLOC** command are set as discussed in Section 4.4.3 based on the usage throughout the whole sequence. The same procedure applies for a pointer p_k allocated in a sequence of sequentially composed regions.

Also, we note the following for object v_j and pointer p_k such that $v_j \in \chi_{r_n^\sigma}$: if object v_j and pointer p_k are allocated in an overlapped sequence of sequentially composed regions, DMA operations on the pointer are inserted to re-locate the pointee to avoid pointer invalidation. The example shown in Figure 4.7 shows the case where object x is allocated to address a_1 in r_2, r_3 and r_4, pointer p is allocated to address a_2 in r_3, r_4 and r_5, and object y is allocated to a_1 in r_5 and r_6. If p can point to x, it will be already in address a_1 in the SPM when p is allocated in r_3 and x will not be copied to address a_2. However, the copy of x at a_1 should be written-back after r_4 to allocate y to a_1. Using p in r_5 with the assumption that x is in the SPM will result in a conflict. So, a re-location must be guaranteed after r_4, so that the copy of x is moved to a_2 before y is fetched to a_1. Note that the relocation commands will cancel each other if p is not pointing to x.

Example: refer to the example in Figure 4.4, where p is allocated in two regions in

sequence (r_8 and r_9). **ALLOC** is inserted before r_8 and **DEALLOC** is inserted after r_9. P flag is set in **ALLOC** even though x is not used in r_8, but it is read in r_9. Similarly, W is not set as x is not modified in neither r_8 nor r_9.

Finally, to compute the WCET for the program, we need to determine whether an **ALLOC**/**DEALLOC** command triggers a DMA operation; this again depends on the context σ in which a given region r_n is executed, as demonstrated by the example in Section 4.4.4. As in Equation 4.3, we know that the **ALLOC** will be canceled if v_j was already available in the parent region-context; hence, for a region r_n that performs an **ALLOC** on v_j and a context σ, the **ALLOC** generates a DMA prefetch on v_j only if both the P flag in the **ALLOC** is set and $avail^{v_j}_{\wp(r^\sigma_n)} = 0$ (similarly for DEALLOC, a DMA operation is generated if the W flag is set and $avail^{v_j}_{\wp(r^\sigma_n)} = 0$).

Aliasing Constraints

The feasibility problem can be relaxed using the points-to information of each pointer. Points-to information are derived from a must-may alias analysis. We consider the must-alias points-to sets with object v_j and pointer p_k such that $\chi_{r^\sigma_n}(p_k) = \{v_j\}$; which is a common case with passing by reference in functions. In this case, the allocation of either v_j or p_k in region-context r^σ_n means that both v_j and p_k are available in the SPM in this region-context. The constraints can be extended if there are multiple pointers that only point to v_j in a region-context.

$$alloc^{v_j}_{r_n} \vee alloc^{p_k}_{r_n} \vee avail^{v_j}_{\wp(r^\sigma_n)} \vee avail^{v_j}_{\wp(r^\sigma_n)} \Leftrightarrow avail^{v_j}_{r^\sigma_n}. \tag{4.13}$$

$$alloc^{v_j}_{r_n} \vee alloc^{p_k}_{r_n} \vee avail^{v_j}_{\wp(r^\sigma_n)} \vee avail^{p_k}_{\wp(r^\sigma_n)} \Leftrightarrow avail^{p_k}_{r^\sigma_n}. \tag{4.14}$$

Equations 4.13, 4.14 replace Equation 4.1 , 4.2 and state that v_j and p_k are available in the SPM during the execution of r^σ_n if v_j or p_k is allocated in r_n, or if v_j or p_k was already available in the SPM during the execution of the parent region-context pair. Note that in Equation 4.14, v_j can be available in the parent region-context $\wp(r^\sigma_n)$ while p_k is not available in it if p_k changes its reference in the children of $\wp(r^\sigma_n)$, *i.e.*, $\chi_{r^\sigma_n}(p_k) \neq \{v_j\}$. In that case, Equation 4.14 is not applicable to $\wp(r^\sigma_n)$ and $alloc^{p_k}_{\wp(r^\sigma_n)} = 0$ according to Equation 4.12.

$$avail^{v_j}_{\wp(r^\sigma_n)} \Rightarrow address^{p_k}_{r^\sigma_n} = address^{v_j}_{\wp(r^\sigma_n)}. \tag{4.15}$$

$$\neg avail^{v_j}_{\wp(r^\sigma_n)} \wedge alloc^{v_j}_{r_n} \wedge \neg alloc^{p_k}_{r_n} \Rightarrow address^{v_j}_{r^\sigma_n} = address^{p_k}_{r^\sigma_n} = assign^{v_j}_{r_n}. \tag{4.16}$$

$$\neg avail^{v_j}_{\wp(r^\sigma_n)} \wedge alloc^{p_k}_{r_n} \wedge \neg alloc^{v_j}_{r_n} \Rightarrow address^{p_k}_{r^\sigma_n} = address^{v_j}_{r^\sigma_n} = assign^{p_k}_{r_n}. \tag{4.17}$$

$$\neg avail^{v_j}_{\wp(r^\sigma_n)} \wedge alloc^{v_j}_{r_n} \wedge alloc^{p_k}_{r_n} \Rightarrow (address^{v_j}_{r^\sigma_n}, address^{v_j}_{r^\sigma_n}) = \gamma(assign^{v_j}_{r_n}, assign^{p_k}_{r_n}). \tag{4.18}$$

Equation 4.3 still applies for v_j as the availability in the parent dominates any allocation in the region. However, the address p_k inherits the address of the v_j if it is available in its parent as in Equation 4.15. If v_j is not available in the parent, there are two cases:

Equations 4.16,4.17 state that if only v_j or p_k is allocated, the address of v_j and p_k is the assigned address of the allocated one.

Equation 4.18 state that if both v_j and p_k are allocated, the assigned address for each of them to be determined with an arbitrary function γ that depends on how the allocation is implemented. In this work, we use $\gamma(assign^{v_j}_{r_n}, assign^{p_k}_{r_n}) = assign^{v_j}_{r_n}$ as we consider relocation of the pointer as we illustrated before in the example shown in Figure 4.7.

Example: refer to the example in Figure 4.4, where p is allocated with assigned address a_2 in r_8. For context $\sigma \cup r_5$, we have $avail^{p}_{r_7^{\sigma\cup r_5}} = avail^{y}_{r_7^{\sigma\cup r_5}} = 0$, since $\chi_{r_7^\sigma \cup r_5}(p) = \{y\}$ and y is not available in r_5^σ, the parent of $r_7^{\sigma\cup r_5}$. Hence, we also have $address^{p}_{r_8^{\sigma\cup r_5}} = address^{y}_{r_8^{\sigma\cup r_5}} = a_2$. However, for context $\sigma \cup r_4$ we obtain $avail^{p}_{r_7^{\sigma\cup r_4}} = avail^{x}_{r_7^{\sigma\cup r_4}} = 1$, since $\chi_{r_7^\sigma \cup r_4}(p) = \{x\}$ and x is available in r_4^σ. So, we get $address^{p}_{r_7^{\sigma\cup r_4}} = address^{x}_{r_7^{\sigma\cup r_4}} = a_1$.

The constraints for SPM size are the same as in Equations 4.7, 4.8. Equations 4.10, 4.11 for address overlap are not applied for must alias cases. That is, if $\chi_{r^\sigma_n}(p_{k1}) = \chi_{r^\sigma_n}(p_{k2}) = \{v_j\}$, then their address ranges match $address^{p_{k1}}_{r^\sigma_n} = address^{p_{k2}}_{r^\sigma_n} = address^{v_j}_{r^\sigma_n}$. So, Equations 4.11, 4.10 are not applied between p_{k1}, p_{k2}, v_j for region-context r^σ_n.

We illustrated the possible aliasing constraints for one pointer and one object. Another set of constraints can be derived for aliasing pointers or pointers with multiple pointees in their points-to set. We do not detail these constraints as they exploit a may-alias which means the constraints do not represent necessity.

4.6.2 WCET Optimization

For a given allocation solution $\{alloc_{r_n}^{v_j}, assign_{r_n}^{v_j}, alloc_{r_n}^{p_k}, assign_{r_n}^{p_k} | \forall v_j, p_k, r_n\}$, the described procedure determines the set of objects available in the SPM and the set of DMA operations for each region-context r_n^σ. Assuming that bounds on the time required for SPM and main memory accesses are known, this allows us to determine the benefit (WCET reduction) for every trivial region in context σ, as well as the length of DMA operations. For a dynamic allocation approach without prefetch, the length of DMA operations could simply be summed to the execution time of the corresponding region, since DMA operations stall the core.

However, for our proposed approach with prefetching, the cost of DMA operations depend on the overlap: since DMA works in transparent mode, for a trivial region the maximum amount of overlap is equal to the execution time of its code minus the time that the CPU accesses main memory directly. Furthermore, since the length of DMA operations is generally longer than the execution of a trivial region, the total overlap depends on the program flow. Therefore, we compute the amount of overlap as part of an integrated WCET analysis, which we present in Section 4.7. We solve the allocation problem by adopting a heuristic approach that first searches for feasible allocation solutions, and then runs the WCET analysis on feasible solutions to determine their fitness; we discuss it next in Section 4.6.3.

Finally, we note that the proposed region-based allocation scheme is a generalization of the approaches used in related work on dynamic allocation. In [134], the authors applied a structured analysis to choose a set of variables for static allocation. They analyzed innermost loop as Directed Acyclic Graph (DAG) for worst case path and then collapsed the loop into a basic block to analyze the outer loop. The region tree representation captures this structure as loops, conditional statements and functions as regions. The dynamic allocation in [139] is based on program points around loops, if statements and functions which can be matched with an entry/exit of a region. In [40], Deverge *et al.* proposed a general graph representation that allows different granularities of allocation. The authors formulated the dynamic allocation problem based on the flow constraints which can also be applied to the region representation. All such approaches use heuristics to determine the overall program allocation. Hence, to allow a fair evaluation focused on the benefits of data prefetching, in Section 4.9 we compare our proposed scheme against a standard dynamic allocation approach with no overlap using the same region-based program representation and search heuristic.

The problem formulation is presented as a variation of the general knapsack problem which is an NP-hard optimization problem [27]. This means that the allocation problem is

complex even if we assumed that the WCET is a linear function of the allocations. We opt to using heuristics in a divide-and-conquer approach to produce a reasonable solution. We developed a set of algorithms for the allocation, the address assignment and the WCET analysis in the next sections.

4.6.3 Allocation Heuristic

The allocation heuristic divides the allocation to three sub-problems: profit estimation, knapsack allocation, and address assignment. The profit estimation provides the expected profit of a single allocation. The knapsack allocation adopts a genetic algorithm that uses the estimated profit and a feedback mechanism using the WCET analysis to find a near-optimal solution. The address assignment is used as a part of the genetic algorithm evaluation to check the feasibility of fitting the allocations in the SPM with distinct addresses.

Genetic Algorithm

The allocation heuristic adopts a genetic algorithm to search for near-optimal solutions to the allocation problem.

- **Chromosome Model:** The chromosome is a binary string where each bit represents one of the $alloc_{r_n}^{v_j}$ decision variables. Note that we do not represent the $assign_{r_n}^{v_j}$ decision variables in the chromosome; instead, we use a fast address assignment algorithm as part of the fitness function to find a feasible address assignment for a chromosome.
- **Fitness Function:** The fitness fit of a chromosome represents the improvement in the WCET of the program with this allocation if it is feasible. The fitness function first applies the address assignment algorithm to the chromosome. If the allocation is not feasible, the chromosome has $fit = 0$. Otherwise, we execute the WCET analysis after the program is transformed to insert the allocation commands; the fitness of the allocation is then assigned as $fit = WCET_{MM} - WCET_{alloc}$ where $WCET_{MM}$ is the WCET with all the objects in main memory and $WCET_{alloc}$ is the WCET for the analyzed solution.
- **Initialization:** The initial population $P(0)$ is generated randomly with feasible solutions, *i.e.*, $fit > 0$.

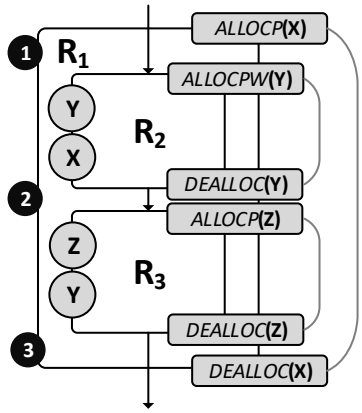


Figure 4.8: Example of allocation order

- **Evolution Operations:** The evolution process incorporates random random selection, one-point crossover and random bit mutation to generate $P'(t+1)$. The elite chromosomes with highest fitness from $P(t)$ and $P'(t+1)$ are chosen to form the next population $P(t+1)$.
- **Termination:** The algorithm is terminated after k generations or if the best chromosome does not change for n generations.

Allocation Order

The order of allocation commands can significantly impact the profit of a solution. The solution generated by the genetic algorithm consists of a set of allocations, *i.e.*, objects allocated in allocation regions. The order in which the allocation commands corresponding to these allocations are executed is known if they are inserted in different program points, *e.g.* two objects allocated in two allocation regions different functions. This does not apply if the allocation regions share program points. The example in Figure 4.8 illustrates this case.

In the example, three objects X, Y and Z are allocated in allocation regions R_1, R_2 and R_3 respectively. Program point ❶ is the entry for both R_1 and R_2. So, at this point both X and Y are allocated and prefetched. The order between them can be $X \rightarrow Y$ or $Y \rightarrow X$. In R_2, Y is used before X. If we chose the order $X \rightarrow Y$, then at the first use of Y, the program will stall until both X and Y are prefetched which might eliminate the profit of allocation. However if we chose $Y \rightarrow X$, the program will stall for Y only at the first use of Y and stall for X only at the first use of X.

A similar case occurs at point ❷ as Y is deallocated and Z is allocated while Z is used first in R_3. However, the order might be forced in this case if the address range assigned to Y overlaps with the address range assigned to Z in the SPM. In this situation, Y has to be written-back before Z is prefetched to avoid overwriting Y. If they do not overlap, it might be more profitable to prefetch Z first, then write-back Y.

We account for the allocation order of allocations that happen at the same program points in the feedback WCET analysis and the IR transformation. That is, when we record the usage order of objects in the allocation regions during the profit estimation phase, then we order the allocation commands such that the object that is used first is scheduled first whenever possible.

Address Assignment Algorithm

The goal of the algorithm is to assign an address for each of the object allocations in the SPM. The assignment must conform to the set of constraints on the addresses in Section 4.6.1.

The address constraints can be linearized using auxiliary decision variables. Then, an ILP solver can be used to obtain an assignment that satisfies the constraints. This problem is a generalization of the register allocation problem which has a correspondence to the graph coloring problem [29, 146]. The general graph coloring is known to be NP-hard [55].

The address assignment algorithm is depicted in Algorithm 1. Given a chromosome, the region tree is traversed in topological order assigning addresses to the allocated objects and pointers in each region. The topological order visits all the nodes with the same parent before visiting the children. For the root of a function, all the parents (call regions) of the function are visited before the root of the function. Also, for a sequence of sequentially composed regions, the order of the sequence is maintained. After the objects in a region are assigned to SPM addresses, an end address to the last allocated address is maintained. For each region r_n, the previous end address is the maximum of all parent regions (note that if r_n is not the root of its function, it has a single parent region). For a region that is not sequentially composed or the first region in a sequence of regions, addresses are iteratively assigned to the allocated objects starting from the previous end address. For a region in a sequence, an allocated object maintains the same address as the previous region if the object is allocated in both. Otherwise, a best fit algorithm is used to assign the remaining addresses. The end address for each region is then computed as the maximum end address for any allocated object. Note that the algorithm trivially ensures that objects/pointers allocated in a region cannot overlap with any object or pointer that is available in a parent;

Algorithm 1 Address Assignment

Input: region information, $\{alloc_{r_n}^{v_j}, alloc_{r_n}^{p_k} | \forall v_j, p_k, r_n\}$

1: **for all** region r_n by increasing index starting with r_1 **do**
2: $\quad end\_addr_{r_n} \leftarrow \text{ASSIGN\_ADDRESSES}(r_n)$

3: **function** ASSIGN_ADDRESSES(r_n)
4: $\quad end\_addr_{r_n} = \max_\sigma\{end\_addr_{\wp(r_n^\sigma)}\}$
5: $\quad$ **if** r_{n-1} is not sequentially composed with r_n **then**
6: $\quad\quad$ **for all** v_j such that $alloc_{r_n}^{v_j}$ **do**
7: $\quad\quad\quad assign_{r_n}^{v_j} \leftarrow end\_addr_{r_n}$
8: $\quad\quad\quad end\_addr_{r_n} \leftarrow end\_addr_{r_n} + S(v_j)$
9: $\quad\quad$ **for all** p_k such that $alloc_{r_n}^{p_k}$ **do**
10: $\quad\quad\quad assign_{r_n}^{p_k} \leftarrow end\_addr_{r_n}$
11: $\quad\quad\quad end\_addr_{r_n} \leftarrow end\_addr_{r_n} + \max_\sigma(S_\sigma(p_k))$
12: $\quad$ **else**
13: $\quad\quad$ **for all** v_j such that $alloc_{r_n}^{v_j} \wedge alloc_{r_{n-1}}^{v_j}$ **do**
14: $\quad\quad\quad assign_{r_n}^{v_j} \leftarrow assign_{r_{n-1}}^{v_j}$
15: $\quad\quad$ **for all** p_k such that $alloc_{r_n}^{p_k} \wedge alloc_{r_{n-1}}^{p_k}$ **do**
16: $\quad\quad\quad assign_{r_n}^{p_k} \leftarrow assign_{r_{n-1}}^{p_k}$
17: $\quad\quad$ **for all** v_j such that $alloc_{r_n}^{v_j} \wedge \neg alloc_{r_{n-1}}^{v_j}$ **do**
18: $\quad\quad\quad$ Compute $assign_{r_n}^{v_j}$ using best fit based on already assigned addresses
19: $\quad\quad$ **for all** p_k such that $alloc_{r_n}^{p_k} \wedge \neg alloc_{r_{n-1}}^{p_k}$ **do**
20: $\quad\quad\quad$ Compute $assign_{r_n}^{p_k}$ using best fit based on already assigned addresses
21: $\quad\quad max_{r_n}^{v} \leftarrow \max_{v_j \text{ s.t. } alloc_{r_n}^{v_j}}\{assign_{r_n}^{v_j} + S(v_j)\}$
22: $\quad\quad max_{r_n}^{p} \leftarrow \max_{p_k \text{ s.t. } alloc_{r_n}^{p_k}}\{assign_{r_n}^{p_k} + \max_\sigma(S_\sigma(p_k))\}$
23: $\quad\quad end\_addr_{r_n} \leftarrow \max\{max_{r_n}^{v}, max_{r_n}^{p}\}$

hence, Equations 4.9, 4.11 are always satisfied. However, the algorithm is not optimal, since it does not consider that an allocation might not be required in any context where the object is already available in the SPM or the aliasing between objects and pointers. Finally, the allocation is considered feasible only if the end address never exceeds the SPM size; this guarantees that Equations 4.7, 4.8 are also satisfied.

4.7 WCET Analysis

We discuss how to model the behavior of our prefetch mechanism in the context of static timing analysis so that a safe bound to the WCET of the program running uninterrupted can be computed. We assume a given allocation solution computed based on Section 4.6. We rely on the standard approach of Data Flow Analysis (DFA) [161], where the detailed state of the hardware is generalized into an *abstract state* based on the theory of abstract interpretation [37, 137]. To avoid maintaining a different state for each path through the program, the analysis relies on computing fixed points by "merging" states when paths join (i.e., branch join and loops entry/exit). In detail, given two abstract states d and d', we need to compute a *join operator* $\vee$ such that the resulting state $d'' = d \vee d'$ is more general than either d or d'. We model time as natural numbers, i.e., processor clock cycles.

We begin by providing an intuitive discussion of the challenges of handling our prefetching mechanism, followed by our intended solutions. In what follows, we use function $(x)^+$ as a shorthand or $\max(0, x)$ and $\mathcal{P}(A)$ to denote the powerset of set A. As discussed in Section 4.3, let $\bar{G}_f = (\bar{N}, \bar{E})$ denote the refined CFG for function f. To keep track of the program execution, it is useful to formally define the concept of program state:

Definition 1 (State)**.** *The program execution is defined as the transformation of a* program state*. We let Σ be the set of all possible program states; we use $\mathbf{s} \in \Sigma$ to denote an individual state and $S \subseteq \Sigma$ to denote a set of states. The state at any given point in the execution of the program represents the amount of* elapsed time $\mathbf{s}.t$ *since the beginning of the program, and the content of all hardware registers and memories.*

Definition 2 (Transfer Function)**.** *For every edge $e : BB_i \to BB_j$ in $\bar{G}_f$ and context σ for f, we define a transfer function $\mathcal{T}_{e,\sigma} : \Sigma \to \Sigma$ such that: if $\mathbf{s}$ is the program state at the beginning of the execution of BB_i and the program execution flows from BB_i to BB_j, then $s' = \mathcal{T}_{e,\sigma}(\mathbf{s})$ is the program state at the beginning of the execution of BB_j. Function $\mathcal{T}'_{e,\sigma} : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ denotes the obvious set-extension of function $\mathcal{T}_{e,\sigma}$, i.e. $\mathcal{T}'_{e,\sigma}(S) = \cup_{\mathbf{s} \in S} \mathcal{T}_{e,\sigma}(\mathbf{s})$.*

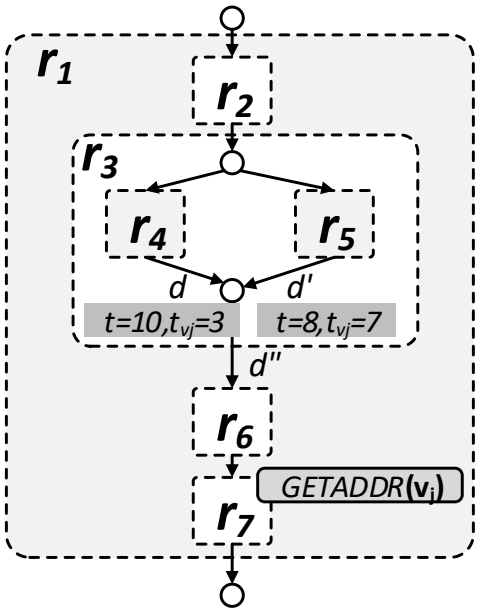


Figure 4.9: WCET Example: Merging states from different paths

Note that based on Definition 2 [2], if the execution cannot flow from node BB_i to BB_j for any state in S, then $\mathcal{T}'_{e,\sigma}(S) = \emptyset$. Given a set of initial program states S_{entry} with $t = 0$ [3], a WCET analysis could then simply proceed as follows: enumerate all paths through every function in the program; for each path through the program, iteratively apply function $\mathcal{T}'_{e,\sigma}$ starting from state set S_{entry} for all edges comprising the path, obtaining a set of final states S_{exit}. The WCET can then be obtained as the maximum time elapsed for any state in any final state set S_{exit}. Since based on our assumptions the number of paths through the whole program is finite, this approach is computable, but it is generally computationally intractable for all but the simplest programs, as the number of paths is exponential in the number of branches/loops in the program.

To obtain a tractable analysis, WCET techniques typically attempt to prune paths that cannot lead to the WCET by making *local* decisions: ideally, we could examine each branch point in the CFG one at a time, determine which branch leads to the WCET, and

[2] Also note that $\mathcal{T}_{e,\sigma}$ defines a *deterministic* machine: assuming we know the state $\mathbf{s}$ at the beginning of the basic block, we can compute the exact state $\mathbf{s}'$ along e. If the machine is non-deterministic, the definition can be modified to return a set of states rather than a single state while maintaining the same theoretical framework, see [137].

[3] Note that in general, a set of program states must be considered, rather than a single state, because the initial state of the hardware, including the program inputs, is not known.

exclude from the analysis all other branches, thus implicitly identifying the Worst Case Execution Path (WCEP) through the program. In practice, this might not be possible, because the worst case path through a branch might depend on the path taken through another branch preceding or following the one under analysis, either due to the program semantic (i.e., some paths might be invalid) or due to architectural considerations (i.e., an hardware operation started during a basic block might influence the timing of a successive basic block).

Since our objective is to show how to integrate our proposed prefetching scheme in existing WCET frameworks, in the rest of the section we focus on architectural analysis. To explain how we model the behavior of DMA operations, consider as an example the execution of the CFG and associated region tree in Figure 4.9 in a context σ. Assume that the analysis for the path through r_4^σ has computed a program state $\mathbf{s}$ with an upper bound to the execution time of the program up to this point equal to $t = 10$, and an upper bound to the remaining time to complete a DMA fetch operation for an object v_j equal to $t_{v_j} = 3$. For the path through r_5^σ, we instead have a state $\mathbf{s}'$ with $t = 8, t_{v_j} = 7$, i.e., the execution takes longer along the path through r_4^σ than through r_5^σ, but results in a shorter remaining DMA time. Assume now that a **GETADDR** command on object v_j is executed at the beginning of region/context r_7^σ. The amount of time that the command will block is then equal to t_{v_j} minus the amount of overlap that the DMA operation has with r_6^σ, or zero if the operation completes during r_6^σ. Assume a simple case where the execution through r_6^σ requires Δ units of time and performs no access to main memory, so that the DMA operation can overlap up to Δ. The program can then resume from **GETADDR** at time $t + \Delta + \max(t_{v_j} - \Delta, 0)$. Hence, note that for $\Delta = 7$, the worst case path is through r_4^σ, resulting in a time of 17 units against 15 for the path through r_5^σ. However, for $\Delta = 3$, the worst case path is through r_5^σ, with a time of 15 time units against 13 for the path through r_4^σ. In summary, we cannot determine which path through a branch leads to the worst case unless we analyze the regions following the branch in the CFG (r_6^σ and r_7^σ in the example). This shows that the WCEP determination is a *global* decision.

A typical solution to the global decision problem is to employ a *Meet Over Path* (MOP) solution: if we do not know which state to use for b_4, we can *abstract* the execution of the program by considering a new join state $\mathbf{s}''$ that is worse than either $\mathbf{s}$ or $\mathbf{s}'$. Such state does not need to represent any real execution of the system (i.e., it is *abstract*), as long as we can prove that the WCET obtained based on $\mathbf{s}''$ is no smaller than the ones determined based on $\mathbf{s}$ and $\mathbf{s}'$. In this case, a trivial solution would be to computing a join state $\mathbf{s}''$ with $t = \max(10, 8) = 10$ and $t_{v_j} = \max(3, 7) = 7$. However, this would lead us to over-approximate the time for the **GETADDR**, resulting in 17 time units for $\Delta = 3$, rather than the computed bound of 15 time units. Therefore, we seek to derive a tighter

abstraction.

Intuitively, this can be achieved by abstracting the states $\mathbf{s}$ and $\mathbf{s}'$ for the execution through r_4^σ and r_5^σ into abstract states d and d'. An abstract state d is composed of two information: the elapsed program execution time $d.t$, and a set of *timers* $\{t_{v_j}\}$. For an object v_j, $d.t_{v_j}$ represents the worst case time required to complete either a prefetch or write-back operation in the allocation queue; since the allocation queue is served in FIFO order, this represents the time to transfer that specific object, plus the time required for all operations ahead of it in the queue. For the example in Figure 4.9, let d be the state through r_4^σ and d' be the state through r_5^σ. Since there is only one DMA operation in the queue, we have $d.t = 10, d.t_{v_j} = 3$ and $d'.t = 8, d'.t_{v_j} = 7$, i.e., the abstract states are equivalent to the corresponding program states. The join state $d'' = d \vee d'$ is then computed as follows:

$$d''.t = t_{\max} = \max(d.t, d'.t), \tag{4.19}$$

and for every timer t_{v_j}:

$$d''.t_{v_j} = \max\left(d.t_{v_j} - (t_{\max} - d.t), d'.t_{v_j} - (t_{\max} - d'.t)\right). \tag{4.20}$$

Based on Equations 4.19, 4.20, we compute a join state for the example $d''.t = \max(10, 8) = 10, d''.t_{v_j} = (3-(10-10), 7-(10-8)) = 5$. Note that this abstraction is tighter compared to the values $t = 10, t_{v_j} = 7$ obtained by the trivial over-approximation; in particular, it is easy to see that for the provided example, the time for the **GETADDR** command computed based on d'' is *exactly* equal to the worst case between d and d' for any value of Δ, albeit for more complex cases involving multiple DMA operations it is still a (tighter) over-approximation. However, the abstraction does not correspond to any "real" program state, since the values of t and t_{v_j} are different than the program state at r_7^σ for either execution path. The key intuition is that adding Δ units of time to the execution time of the program is always worse than adding Δ units of time to the length of timers, since a **GETADDR** might block the program for a time at most equal to the length of the corresponding timer. Hence, if the execution time along two paths differs by a value Δ, we are guaranteed to obtain an upper bound if we consider the longest execution time but subtract Δ units of time from the timers along the shortest path, as performed in Equation 4.20.

Note that in general, a single DMA operation could overlap with many regions, and the amount of overlap can be further modified by the path through each region and allocation commands for both the same and other objects. Due to the presence of the max term in Equation 4.20, modeling the WCET problem as an ILP (a technique also known as implicit path enumeration [161]) would require adding a large number of auxiliary variables.

Therefore, we propose to instead compute the WCET by performing the MOP procedure using a structure-based approach [161] that relies on the region tree, as summarized in Algorithm 2.

Algorithm 2 WCET Analysis

Input: initial program state d with $d.t = 0$, region information, allocation solution

1: $d \leftarrow$ ANALYZE_REGION$(r_1, \emptyset, d)$
2: **return** $d.t + \max_{v_j}\{d.t_{v_j}\}$

3: **function** ANALYZE_REGION(r, σ, d)
4: **if** r is trivial region **then**
5: $d \leftarrow$ STATE_TRANSFER(r, σ, d)
6: **if** r calls a region r_n **then**
7: $d \leftarrow$ ANALYZE_REGION$(r_n, \sigma \cup r, d)$
8: **else**
9: **for all** paths p_i in r **do**
10: $d_i \leftarrow d$
11: **for all** subregions r_n along p_i **do**
12: $d_i \leftarrow$ ANALYZE_REGION(r_n, σ, d_i)
13: $d \leftarrow$ JOIN$(r, \sigma, \{d_i\})$
14: **return** d

Starting from an initial abstract program state d and region r_1, the root of the main function, the algorithm recursively calls function $ANALYZE\_REGION$ to update state d based on the execution of region r in context σ. If r is a trivial region, then function $STATE\_TRANSFER$ is used to update d based on the region's code, including any allocation command. Note that we need to pass the context σ to the function, since as explained in Section 4.6, the availability and address of objects in the SPM depends on the context for the region. If the region is a call region, we also need to recursively invoke $ANALYZE\_REGION$ on the called region after updating the context. If region r is not trivial, then we need to recursively analyze all sub-regions along every path in r; this results in an updated state d_i for each path p_i. The states are then joined by function $JOIN$. If region r has no backedge (i.e., it is not a loop), then the function simply applies the join operator over all states d_i. If the region is a loop, then function $JOIN$ performs a fixed-point iteration over the abstract state (since such fixed point iteration is a well-understood technique in DFA [37, 137], we do not discuss it further). At the end of the analysis, we return the total elapsed time plus the maximum timer length, to indicate the

need to complete any remaining write back operation.

In the next section, we first provide required preliminaries on the underlying mathematical principles of DFA using the MOP approach. We then formally introduce our abstraction and prove it correct in Section B.0.2. Note that while Algorithm 2 enumerates regions, in practice the only regions that contain code and must thus be analyzed are trivial regions, containing one basic block each. Hence, for simplicity and to be consistent with previous analyses, we discuss the MOP procedure over basic blocks using the refined CFG $\bar{G}_f$. Finally, note that while we focused on modeling the behavior of DMA operations, the abstract state can also model both architectural states, such as the state of the processor pipeline [137], as well as the value of program variables, which can be used to exclude invalid paths (flow analysis) and compute loop bounds [94].

We provide a detailed discussion of the WCET analysis in Appendix B.

4.8 Insights into Dynamic Allocation and Prefetching

As discussed in [139], the dynamic allocation without prefetching is more beneficial than the static allocation only for intermediate SPM sizes which can fit some but not all of the objects in the SPM. That is, for small sizes of the SPM where none of the objects can fit in the SPM and for large sizes where most of the objects can fit in the SPM, the benefit of dynamic and static allocation is similar without prefetching. For object-based approaches like our method, the range of the SPM sizes that shows benefit for the dynamic allocation is dependent on the number, sizes and live ranges of the objects in the program. The significance of dynamic allocation appears when there are multiple objects with distinct live ranges and the size of the SPM can fit some but not all of them.

Prefetching allows the allocation of objects for which the cost of memory transfers is larger than the profit of allocation as it can hide all or part of the transfer cost. When the size of the SPM is large enough to fit most of the objects, prefetching outperforms the static allocation in choosing the memory transfer points to minimize the transfer cost. For intermediate SPM sizes where dynamic allocation is useful, prefetching can still offer additional benefit by hiding the transfer cost when there are opportunities to overlap the memory transfers.

In this section, we present insights through examples into dynamic allocation and prefetching.

Let the profit of an allocation be P and the cost to do memory transfers between the SPM and the main memory be C. Hence, the net profit: $P_{net} = P - C$. We analyze the

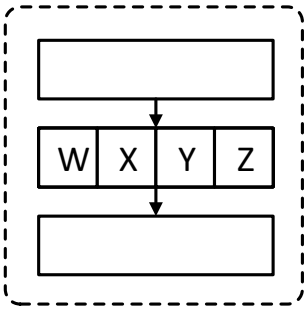


Figure 4.10: Structure of program unit U

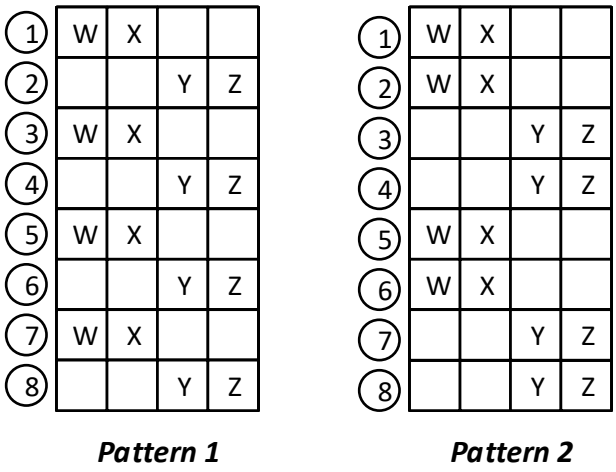


Figure 4.11: Two usage patterns

factors that affect the profit and the cost of dynamic allocation, mainly the size of the SPM and the characteristics of the program.

To study these factors, we use a regular program unit U that consists of 3 regions in sequence such that the top and bottom regions have computations only and the middle region has memory accesses only as shown in Figure 4.10. The computational region has an execution time e and the memory region has a usage of all or some of four objects W, X, Y and Z. All the objects have size s and profit p when allocated in U where the object is used. A memory transfer has cost c to prefetch/write-back to/from the SPM.

Assume a program that consists of eight units $U_1, \ldots, U_8$ in sequence. In each unit, either W, X or Y, Z are used. Figure 4.11 shows two usage patterns of objects W, X, Y and Z: Pattern 1 and Pattern 2. Each object is used in four program units in the two

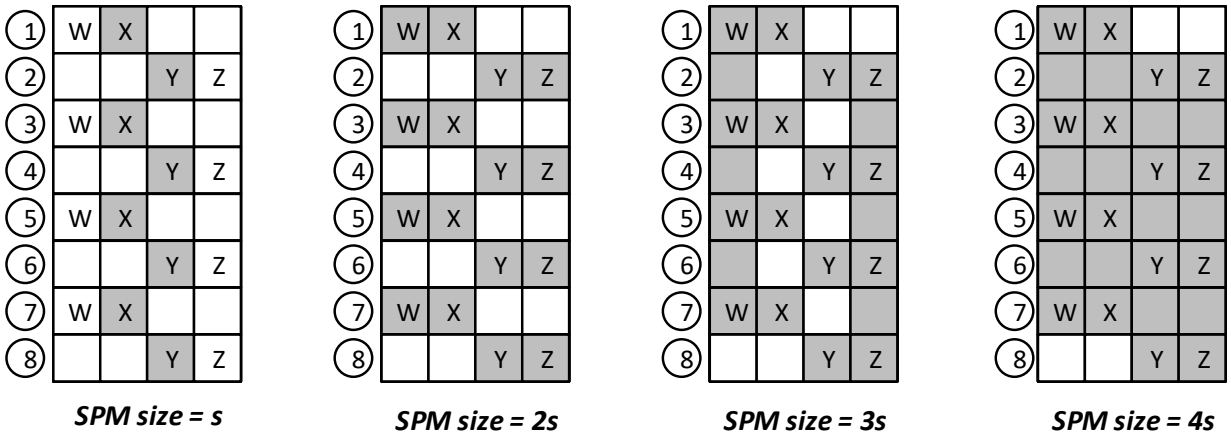


Figure 4.12: Allocations for Pattern 1 (shaded object is used in the unit)

patterns. We assume that the object is read/written when used. We next analyze the static allocation, dynamic allocation and prefetching for these patterns.

4.8.1 Static Allocation

We use static allocation as a reference to assess the efficiency of dynamic allocation. For static allocation, the allocated object resides in the SPM for the whole program. So, the profit for the pattern in Figure 4.11 depends only on how many objects can fit in the SPM. The profit of allocation one object in the program is $4 * p$ for each object. A cost $2 * c$ is needed to transfer the object to/from the SPM at the beginning and end of the program. The net profit for allocating n objects is $P_{net} = n * (4 * p - 2 * c)$.

4.8.2 Dynamic Allocation

For dynamic allocation, an algorithm is used to determine the program points at which the content of the SPM changes to maximize the profit.

Figures 4.12, 4.13 depict the allocated objects in each unit to maximize the profit for different SPM sizes. We show the profit for static and dynamic allocation when varying the SPM size in Figure 4.14 in terms of p and the cost to achieve this profit in terms of c. We next analyze the profit, the cost and the optimization space for dynamic allocation.

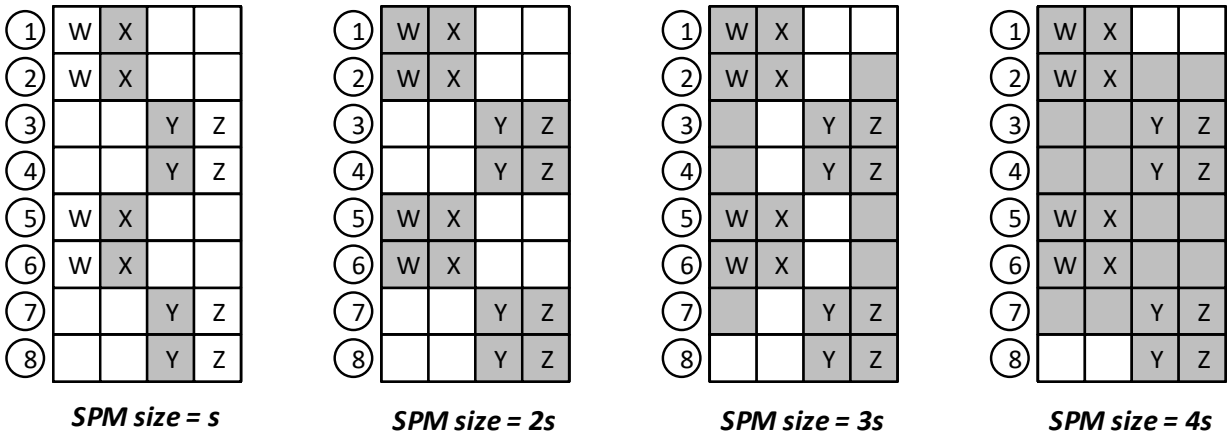


Figure 4.13: Allocations for Pattern 2 (shaded object is allocated in the unit)

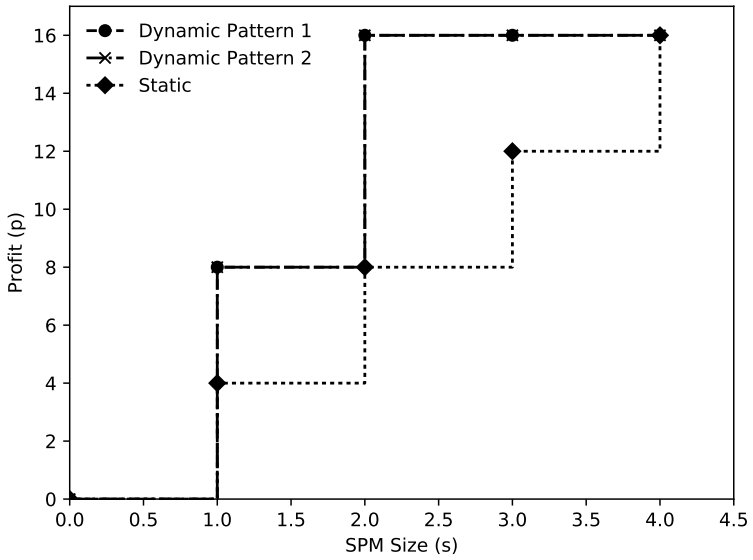
Allocation Profit

In both usage patterns, the profit for SPM size of s is $8p$ while static allocation has a profit of $4p$. For sizes of $2s$ and $3s$ or larger, dynamic allocation is able to allocate all the objects in their usage regions achieving profit $16p$ versus $8p$ and $12p$ for static allocation. When all objects can fit in the SPM at size $4s$, dynamic allocation has no preference over the static allocation.

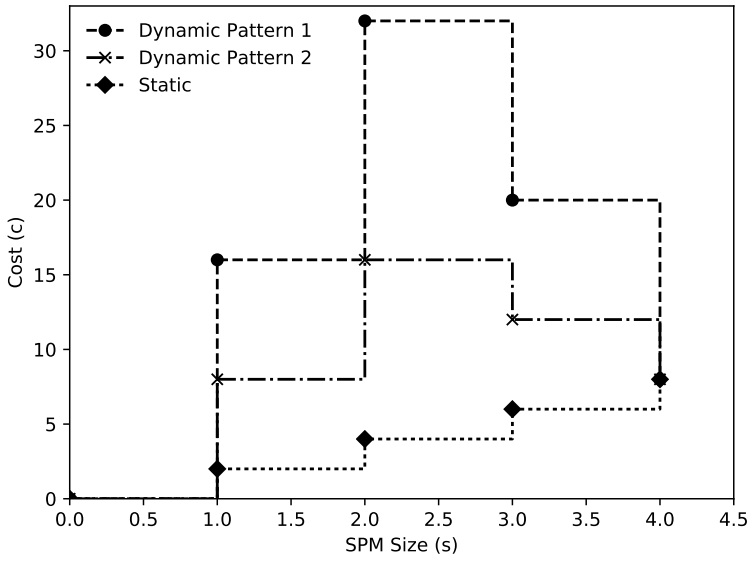
The efficiency of dynamic allocation stems from the ability of changing the content of the SPM to evict an object that is not used and fetch another object that will be used, *i.e.*, when objects have distinct live ranges. When multiple objects are used simultaneously, the dynamicity of the program is limited. In the two reference patterns, two objects are used simultaneously. So, the profit of dynamic allocation is limited when SPM size is s as only fit one object can be allocated. The profit of dynamic allocation is maximum when the SPM can fit two objects at least, *i.e.*, all the objects used simultaneously. When the SPM can fit all the objects with size $4s$, there is no preference between dynamic and static allocation.

Allocation Cost

Dynamic allocation can increase the memory transfer cost compared to static allocation as it changes the content of the SPM more frequently. The cost depends on the scattering of the object usage. The usage of an object in Pattern 1 is more scattered than its usage in Pattern 2. That is, in order to achieve the maximum profit at a certain size, more memory



(a) Profit



(b) Cost

Figure 4.14: Profit and cost of the dynamic and static allocations

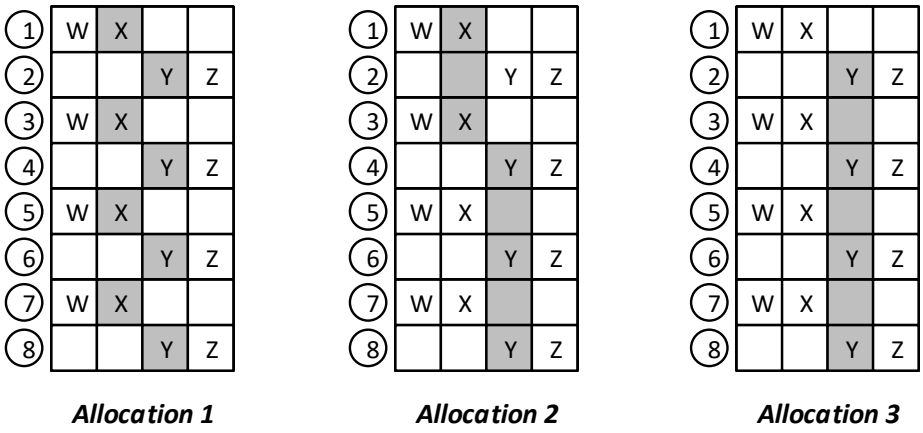


Figure 4.15: Possible allocations for Pattern 1 with SPM size = s

transfers are needed for Pattern 1 than Pattern 2 even though they have the same profit. Consider size $2s$ for instance, the cost of allocation for Pattern 1 is $32c$ compared to $16c$ for Pattern 2. Note that as the size of the SPM increases to fit more objects, the cost of dynamic allocation decreases. For example, the cost of dynamic allocation for Pattern 1 is $32p$ for SPM size s and $20c$ for size $2s$ for the same profit $16p$. Note that in the analyzed program, we assumed that the object is read/written which induce two memory transfers to prefetch and write-back the object. If the object is only read, one memory transfer is required reducing dynamic allocation cost. So, the cost directly depends on the usage of the object.

Optimization

Due to the added cost for dynamic allocation, the allocation algorithm can ignore some allocations to reduce the cost and optimize the net profit.

To analyze the impact of the memory transfer cost on the net profit, we focus on pattern 1 and fix the SPM size to s. In Figure 4.15, we show three possible allocations. Allocation 1 achieves profit $8p$ with cost $16c$ resulting in net profit $P^1_{net} = 8p - 16c = p * [8 - 16(c/p)]$. Allocation 2 achieves profit $5p$ with cost $4c$ resulting in net profit $P^2_{net} = 5p - 4c = p * [5 - 4 * (c/p)]$. Allocation 3 is similar to static allocation and achieves profit $4p$ with cost $2c$ resulting in net profit $P^3_{net} = 4p - 2c = p * [4 - 2 * (c/p)]$.

A dynamic allocation algorithm will explore these possible allocations based on the ratio c/p. Figure 4.16 shows how the algorithm would choose the most profitable allocation based on the net profit. In this figure, we plot P^1_{net}, P^2_{net} and P^3_{net} in terms of p on the y-axis

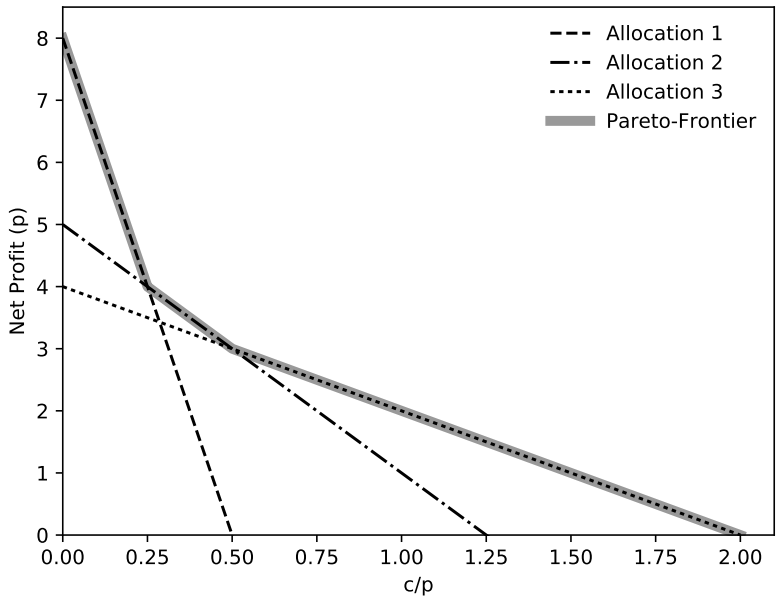


Figure 4.16: Net profit and pareto-frontier for different allocations

while varying the ratio c/p on the x-axis. We highlight the pareto-frontier of the optimal allocation at each point. For $c/p \leq 0.25$, it is more profitable to use the allocation 1. For $0.25 < c/p \leq 0.5$, the algorithm would choose allocation 2. Finally, static allocation is more profitable than dynamic allocation for $0.5 < c/p < 2$. Neither dynamic allocation nor static allocation is profitable for $c/p \geq 2$.

To summarize, the cost of allocation is an important factor that affects the efficiency of dynamic allocation. So, we next discuss how prefetching can enhance the allocation by hiding the cost of allocation.

4.8.3 Prefetching

We showed that dynamic allocation might not be able to perform better than static allocation due to limited profit relative to the cost to change the content of the SPM. Prefetching can improve the dynamic allocation by overlapping the memory transfer with the program execution, hence reducing the cost.

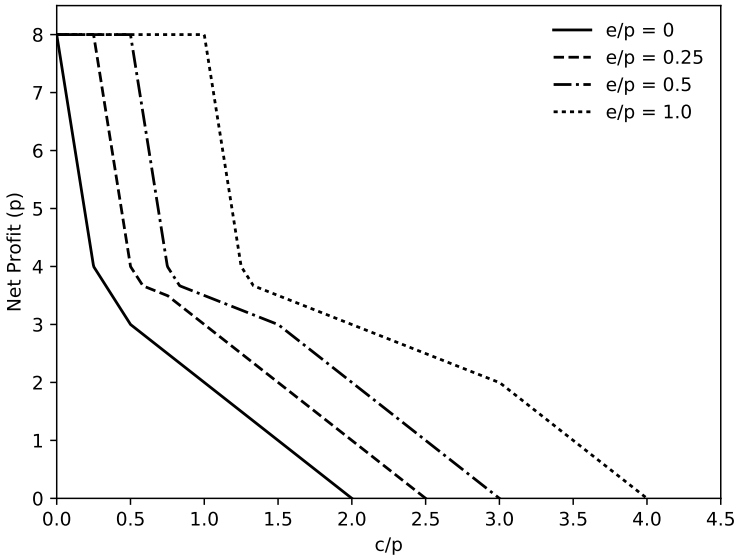


Figure 4.17: Pareto-frontier for $e/p = [0, 0.25, 0.5, 1]$

The net profit of the allocations presented in Figure 4.15 are modified to add the overlap as following: Allocation 1: $P^1_{net} = p * [8 - 16 * (c/p - e/p)^+]$, Allocation 2: $P^2_{net} = p * [5 - 4 * (c/p - e/p)^+]$, Allocation 3: $P^3_{net} = p * [4 - (c/p - 3 * e/p)^+ - (c/p - e/p)^+]$. We use the notation x^+ to denote $max(x, 0)$ as the overlap can reduce the cost till it is completely hidden, but cannot add profit after that.

For allocation 1 and 2, the object is prefetched at the beginning of the program unit U and is written-back after the memory region. That is, the cost c to prefetch/write-back is overlapped with computation e. For allocation 3, prefetching y has more overlap opportunity as it can start in the beginning of U_1 while y is first used in U_2.

We show the pareto-frontier to maximize the net profit as a variable of c/p in Figure 4.17 for e/p values 0, 0.25, 0.5 and 1. For low c/p ratio, increasing the overlap percentage makes it more profitable till it saturates at the maximum possible profit. Prefetching also increases the range of c/p values for which allocation is profitable. That is, the allocation is profitable when $c/p \leq 2$ for no overlap $e/p = 0$ while the allocation is profitable for $c/p \leq 4$ for $e/p = 1$.

We showed how the possible overlap in terms of the computational part of the program affects the efficiency of prefetching to hide the allocation cost. For more complex programs,

other factors have impact on the possible overlap. We summarize them as following:

- The distance between two uses in sequence of an object. In the example, if x is allocated in U_1 and is written back after the memory region of U_1, the maximum overlap for the write-back is limited by the next usage in U_3 as the DMA has to finish before the next usage.
- The address assignment of the allocated objects can limit the possible overlap. In this example, if X is allocated in U_1 and Y is allocated in U_2, writing back X must finish before allocating Y to avoid overwriting as X and Y are allocated to the same space in the SPM.
- The order of DMA operations also impacts the utilization of the possible overlap as explained in Section 4.6.3.
- Dual-port SPM allows for more overlap as the memory accesses to the SPM can be overlapped with the DMA transfers. This means that the possible overlap for the allocation of an object is dependent on the allocation of other objects.

4.9 Evaluation

The evaluation of the prefetching approach for the data SPM allocation is performed using a simple model of MIPS processor with a 5-stage pipeline and no branch predictor. For memory instructions, we consider a latency for a word access to main memory of 10 cycles, 1 cycle to SPM and 1 cycle to the SPM controller. For the DMA, we use a similar model as in [163] such that the latency to initialize the transfer to/from main memory is 10 cycles and the latency per word is 2 cycles.

We consider three cases: 1) dynamic allocation without prefetching; 2) dynamic allocation with prefetching; 3) and dynamic allocation with no cost (ideal). Note that the stack always resides in the SPM as its size becomes small after reducing its depth by the converting stack variables to globals as discussed in Section 4.5 and its access rate is usually high.

We tested the allocation algorithm for multiple benchmarks from UTDSP [140], and CHStone [169] suites. We evaluate 8 benchmarks from these suites as described in Table 4.1. We avoided benchmarks that have the following criteria: 1) benchmarks with system calls, as we cannot analyze their WCET without the OS code; 2) benchmarks that access only the stack or have very small sizes for static and local objects.

Benchmark	Description	Suite	No. of Objects
histogram	Enhances a 256-gray-level, 128x128 pixel image by applying global histogram equalization	UTDSP	3
lpc	Linear predictive coding (LPC) encoder	UTDSP	14
g722	Implementation of the CCITT G.722 ADPCM coding algorithm	UTDSP	18
edge detect	Detects the edges in a 256 gray-level 128 x 128 pixel image	UTDSP	6
compress	Compresses a 128 x 128 pixel image	UTDSP	8
spectral	Calculates the power spectral estimate	UTDSP	10
gsm	Linear predictive coding analysis of global system for mobile communications	CHStone	10
aes	Advanced encryption standard	CHStone	6

Table 4.1: Evaluation Benchmarks

As the benchmarks available for real-time systems are usually small kernels, we focus on the performance of the prefetching algorithm compared to dynamic allocation rather than the total profit of the allocation. We were not able to apply the algorithm to other suites with more realistic applications, *e.g.* SPEC2000, as they have system calls, recursion, unknown loop bounds, and calls to standard libraries which makes it unsuitable to derive WCET estimation as part of the framework. We plan to explore other benchmarks in the future.

We define the ideal case as the dynamic allocation with no cost for memory transfer, *i.e.*, the best case for prefetching where all memory transfers are overlapped with CPU execution. Figures 4.18 to 4.25 show the *ideality factor* as a function of the size of the SPM. The ideality factor is computed as $\big(WCET(\text{dynamic w/o prefetching}) - WCET(\text{dynamic w/ prefetching})\big)/\big(WCET(\text{dynamic w/o prefetching}) - WCET(\text{ideal})\big)$. The denominator represents the best hypothetical improvement in WCET that prefetching can achieve relative to the ideal dynamic allocation and the numerator is the improvement for the prefetching case. The ideality factor is an indication for the performance of the prefetching approach, with a value of 1 indicating a performance equivalent to the ideal case. For each benchmark, we vary the range of the SPM sizes starting from the size in which at least one object can fit in the SPM.

The solving time for the allocation algorithm depends on the number of possible allocations, the size of the CFG of the program and the genetic algorithm parameters. In the experiments, we used a population of 100 chromosomes and termination parameters $k = 500, n = 10$. The solving time varied between a few seconds to around 15 minutes. Inserting the allocation commands increases the executable code size by at most 1.2% for the tested programs.

Results Analysis

Benchmark 'histogram' has two main arrays with size 1024 bytes each. When the size of the SPM is 1024 bytes, it can fit only one of them and dynamic allocation is able to arbitrate between the two arrays. Prefetching can overlap part of the cost needed for dynamic allocation as shown in Figure 4.20. When the SPM size is 2048, both arrays can fit in the SPM and also prefetching technique can hide the whole memory time required to transfer the arrays as it can overlap the transfer of one array with the use of the other array in the SPM.

For benchmark 'g722' in Figure 4.21, prefetching technique can only overlap part of the memory transfer as the live range of the used objects are overlapped, *i.e.*, the chance to transfer one object while using the others is low.

Benchmark 'edge_detect' has three arrays with size 64 Kbyte and a small array with size 36 bytes. For small SPM size, only the small array can fit and prefetching can overlap its memory transfer time as shown in Figure 4.25. When the SPM can fit one of the large arrays, prefetching can overlap around 25% of its memory transfer time. Similarly, prefetching can overlap around 33% of the transfer time when the SPM can fit two large arrays. When the SPM can fit all the large arrays, dynamic allocation and prefetching choose not to allocate all three arrays as the memory transfer cost is larger than the profit. Hence, the ideal case is much better than the prefetching. This is a typical program where techniques to allocate portions of the array, *e.g.* tiling, are important to be able to overcome the transfer cost issue.

The other benchmarks have more objects and the live ranges are more nested. The ideality factor changes as the SPM space increases as more objects can fit in the SPM and hence more memory transfers are introduced. If the space is used to arbitrate for objects, prefetching does not have enough time to overlap the memory transfers. If the space allows objects to exist in the SPM simultaneously, prefetching performs better as it has more opportunity to overlap the memory transfers.

4.10 Summary

In this chapter, we introduced a framework for predictable data SPM prefetching. Our approach is automated within a compilation flow that is integrated with the LLVM compiler. We provided a hardware/software design that includes an SPM controller, an allocation algorithm and a WCET analysis. The experiments have shown the potential of

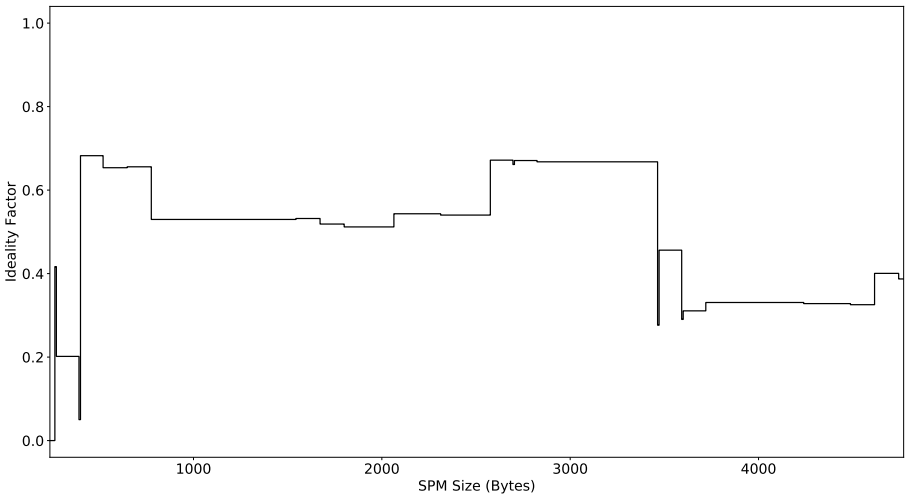


Figure 4.18: Ideality factor (aes)

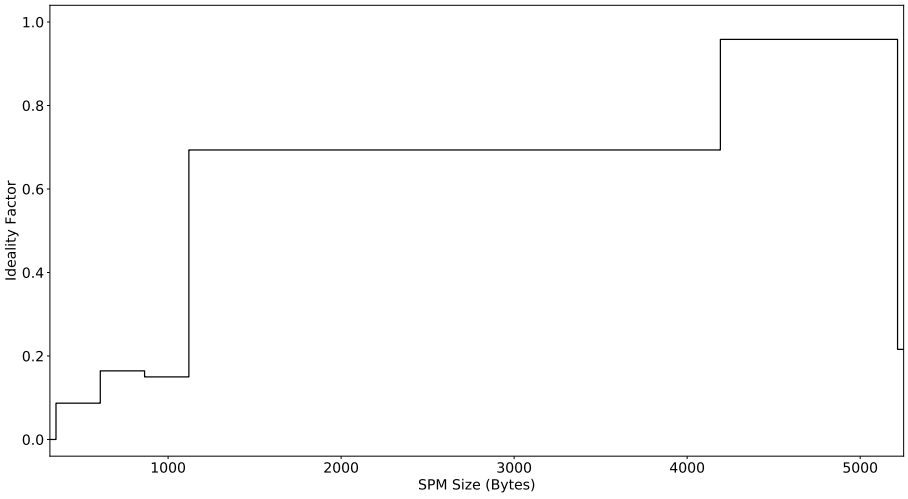


Figure 4.19: Ideality factor (compress)

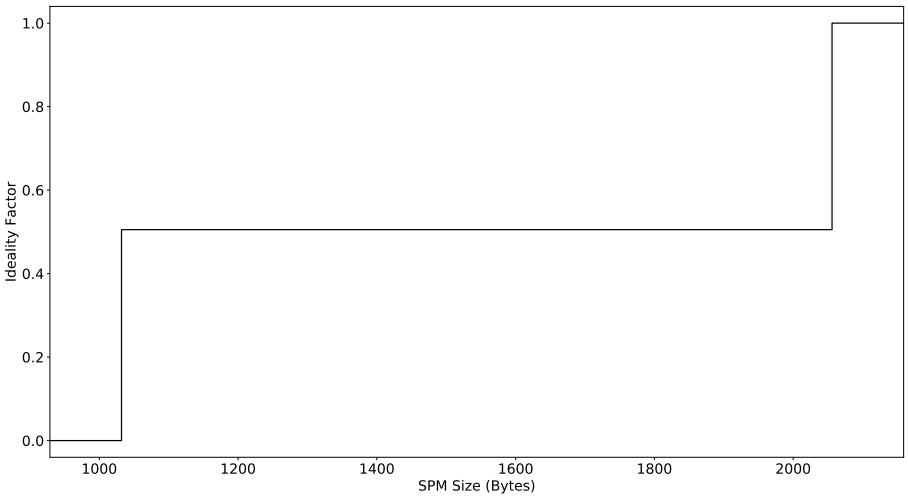


Figure 4.20: Ideality factor (histogram)

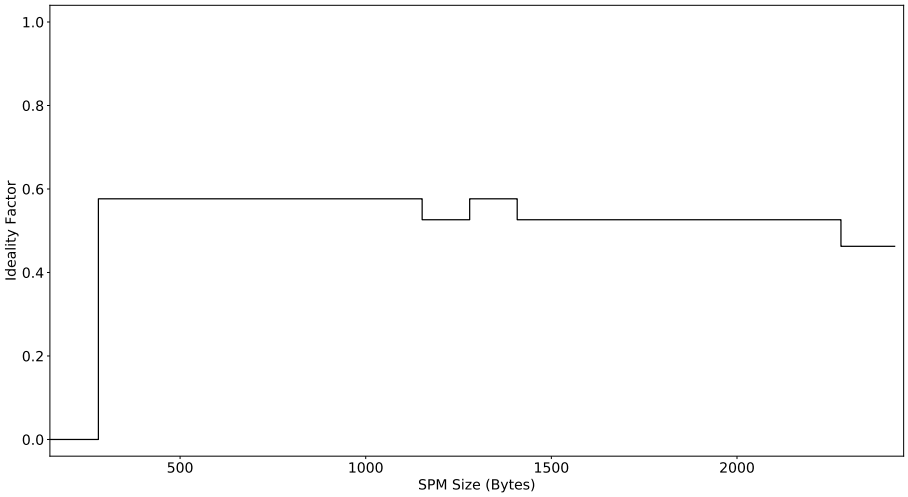


Figure 4.21: Ideality factor (g722)

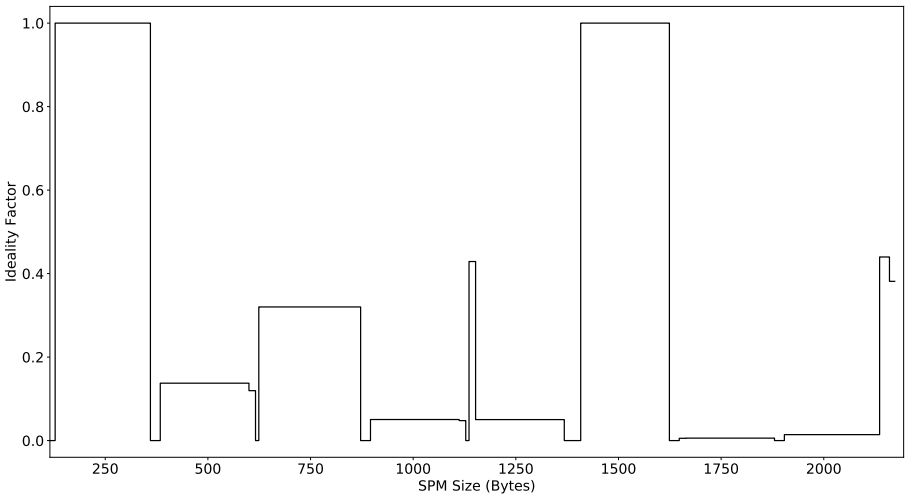


Figure 4.22: Ideality factor (spectral)

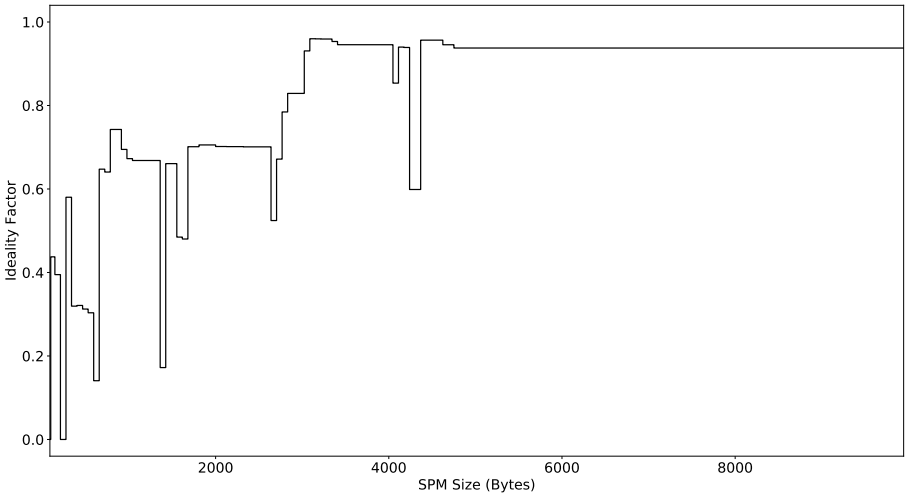


Figure 4.23: Ideality factor (lpc)

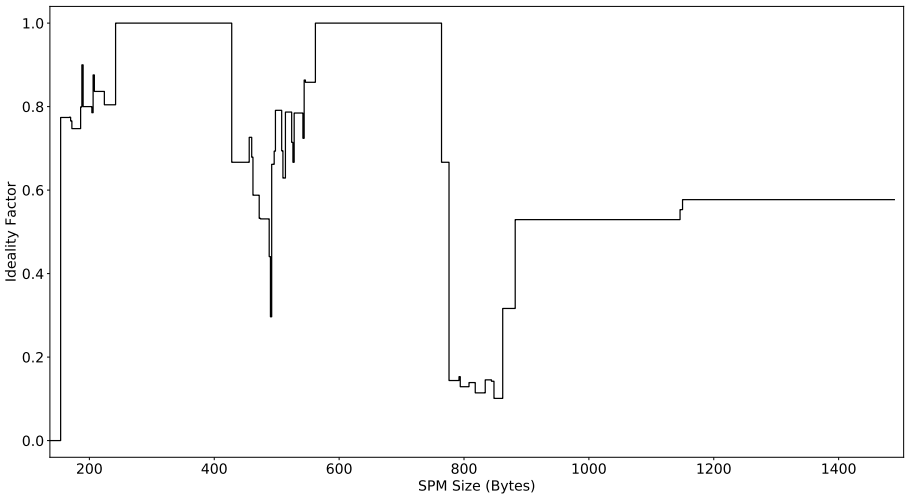


Figure 4.24: Ideality factor (gsm)

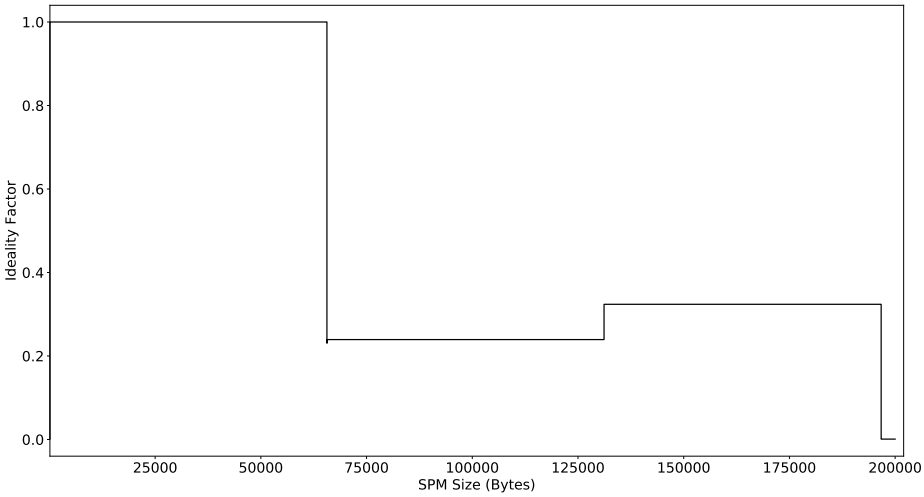


Figure 4.25: Ideality factor (edge detect)

our prefetching technique to provide a predictable mechanism to hide the latency of main memory transfers and efficiently manage the data SPM with low overhead. Our framework can handle pointer-based memory accesses for static, stack and dynamically allocated objects.

The performance of the allocation algorithm can be enhanced to tackle large objects and loops using transformations like tiling and data pipelining. We plan to integrate these mechanisms in our framework in future work.

Part II

The Case of Multi-Tasking Scheduling

Chapter 5

Multi-Segment Streaming using the 3-Phase Execution Model

In Chapter 1, we discussed the challenges of managing shared resources in Multi-Processor Systems-on-a-Chip (MPSoCs). The shared memory in such systems can incur a very high latency due to the contention for memory access among multiple processors [66, 83, 141]. Hence, there is a significant interest in the real-time community in controlling the pattern of accesses in memory to avoid worst-case scenarios. PRedictable Execution Model (PREM) first proposed in [114] 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 the main memory into the local memory (cache or ScratchPad Memory (SPM)), and computation phases where a processor executes the task based on the content of local memory only. This approach avoids the memory contention as the task does not need to access the main memory during its computation phase; and hence other processors are free to access the memory without contention. The first work on PREM focused on avoiding the memory contention, but did not hide the latency of the memory transfers. This is achieved by scheduling the memory and computation phases of the tasks such that the memory transfers are performed by the processor. To improve the efficiency of the system, several works proposed using a dedicated Direct Memory Access (DMA) unit to perform the memory transfers efficiently and to hide the memory phase latency. The 3-phase model [152] divided the task execution into three phases, acquisition-execution-replication, by loading the SPM during the acquisition phase, executing the task, then unloading the modified data to the main memory in the replication phase. The model is able to hide the memory latency by overlapping the DMA transfer of a task with the execution of another task, which significantly improves the system schedulability [152].

To apply the 3-phase model, the program code and data have to fit in the available SPM. However, many applications require memory larger than the SPM size. In this case, the program has to be divided into a sequence of multiple segments that execute according to the 3-phase model. Compiling a program to execute based on the 3-phase model is a key problem that has received significantly less attention despite the numerous contention-less approaches based on the 3-phase model that have been proposed in the literature. In general, the following steps are required to compile a program according to the 3-phase model: (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, an automated tool is required to remove the burden from the programmer. Our goal is to utilize the framework introduced in Chapter 2 to analyze a set of tasks and generate program transformations based on a set of constraints to convert each task into a conditional sequence of 3-phase segments. However, before we delve into the segmentation process in Chapter 6, we extend the 3-phase model to our new multi-segment conditional streaming model. The new model addresses two limitations of the 3-phase model: 1) the previous works considered a program with a single execution path comprising a set of segments executing in sequence; while many applications have multiple execution paths, 2) the model did not allow streaming multiple segments of the same task, i.e. two segments of the same task cannot execute back-to-back; which limits the overlap of the memory phases and the execution phases only between different tasks. We address the first limitation by extending the model to consider a conditional Directed Acyclic Graph (DAG) representation, which can represent a program with multiple execution paths. For the second limitation, we introduce the multi-segment streaming model which allows executing two segments of the same task back-to-back by loading the code and data required by the next segment while executing the current segment.

In this chapter, we start with a review of the background and related work of the 3-phase model in Section 5.1. Then, we introduce the new multi-segment conditional streaming model in Section 5.2. Section 5.3 presents an Operating System (OS)-level programming interface for the management of the SPM along with a software implementation. After that, we derive a sufficient schedulability analysis according to the proposed model for two DMA models, fixed-size in Section 5.4 and variable-size in Section 5.5. Finally, we summarize the work in this chapter in Section 5.6.

5.1 Background and Related Work

In this section, we first present required background on the 3-phase model and discuss related work on scheduling of 3-phase tasks on multiprocessor. We then discuss existing limitations of the model.

We consider a MPSoC platform comprising a set of possibly heterogeneous processors [1]. Each processor has a fast private local memory in the form of a last level cache or ScratchPad Memory (SPM); all processors share the same main memory. As discussed in the chapter introduction, the 3-phase model allows the creation of contentionless memory schedule. While the seminal work in [114] first proposed to split the execution of each application into a memory and a computation phase, the approach has been refined in successive works [7, 152] into three phases. 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 [6, 7, 18, 28, 45, 99, 101, 114, 123, 166, 167], or on another hardware component [52, 53], such as a programmable DMA module [5, 22, 136, 152]. 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. 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 [28, 99, 114, 152] split a task into a sequence of 3-phase segments, where each segment has its own memory and computation phases and is executed non-preemptively.

5.1.1 Memory and Processor Schedule

The authors of [114] were initially concerned with protecting task execution from I/O DMA transfers, such that memory phases of a general purpose processor were assigned higher priority than I/O transfers. The approach in [28] assigns higher priority to memory phases executed by a GPU. Other algorithms employ a round-robin [52] or TDMA [136, 152, 166]

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

schedule among processors, or a static [7, 18, 45, 99, 123] or priority-based [5, 6, 101, 167] schedule among tasks.

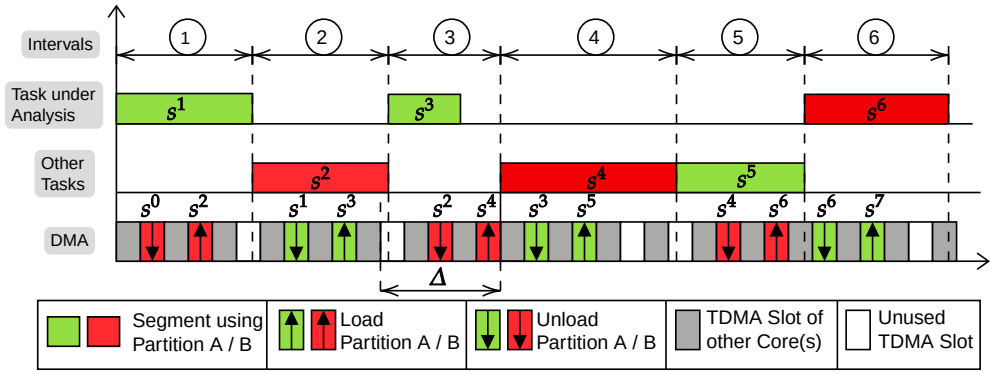


Figure 5.1: Example: TDMA memory schedule with $M = 2$ cores.

The memory scheduling algorithm is different among related work, based on their specific goals and system assumptions. 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 [52, 53, 136, 152], at the cost of halving the available local memory space. As an example, we detail the approach in [136, 152], 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 5.1 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 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 phases. In the figure, interval ③ is bounded by the memory time, while all other intervals are bounded by the computation time of the segment.

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. First of all, since scheduling decisions are only made at the beginning of scheduling intervals, the first segment of a task can be blocked by up to two segments of lower priority tasks, as we will formally illustrate in Sections 5.4 and 5.5. To avoid blocking on the first segment, the works in [101, 166, 167] 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. Second, 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, existing approaches do not allow the computation phase of a segment and the memory phase of the next segment of the same task to be executed in parallel. To avoid idling the processor while a task loads its next segment, at least one segment of another task is instead scheduled, but this segment could belong to a lower priority task if no higher priority jobs are active at that moment. If a task comprises a large number of segments, such inter-segment blocking could greatly affect the response time of the task. Therefore, in Section 5.2 we introduce a new, streaming scheduling model where segments of the same task are allowed to run back-to-back whenever the program code allows it - that is, the data used by a segment can be determined before the previous segment starts and both segments fit in SPM.

Finally, we discuss the length of DMA operations. In the rest of this dissertation, we will consider two DMA models: fixed-size and variable-size. Consider first the *fixed-size* DMA approach detailed in Figure 5.1: let M be the number of cores, and σ the size of each TDMA slot. Then as proven in [136], 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, when considering the fixed-size model, we will simply assume a given value of Δ as the fixed memory time for any interval. Hence, under such a model the length of an interval is the maximum of Δ and the computation time of the job in that interval. The variable-time model is discussed in [152]. The scheduling rules are the same as the ones detailed above for the fixed-size case, except that the memory time for each interval is proportional to the amount of time required to unload / load required data; in essence, if the data used by a task is smaller than the size of a partition, then the memory time can be reduced compared to the fixed-size case. We formalize the computation of the memory time based on the size of DMA operations in Section 5.2.2. While the variable-size model can result in more efficient usage

of the available DMA bandwidth, it also leads to a more complex schedulability analysis, as we discuss in Section 5.5.

5.1.2 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 [18,22,136,152]. The seminal work in [114] 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 [52,53] 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 [97] 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.

The closest related work is [99], 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 [64] 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 will show in Section 6.4, such a 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 5.2.3 we introduce a new conditional PREM model in which the fixed segment sequence is replaced by a Directed Acyclic Graph (DAG) of segments. In Chapter 6 we will then show how to compile the program to a set of conditional segments.

5.2 Multi-Segment Conditional Streaming Model

We now introduce our new multi-segment conditional streaming model. In particular, we first detail the scheduling rules in Section 5.2.1; then we clarify our assumptions on the hardware platform in Section 5.2.2; and finally we formalize the task model in Section 5.2.3. Compared to previous approaches discussed in Section 5.1, we extend existing work on the 3-phase model in two directions: 1) instead of assuming a fixed sequence of segments for each task, we consider conditional execution where at run-time, the sequence of segments for a job depends on the execution path through the program; 2) we allow streaming the data of the next segment of a task while its previous segment is executing, such that two segments of the same task can be executed back-to-back.

5.2.1 Streaming Execution Model

We consider a 3-phase task model in which a task is executed as a set of segments. The code and data of each segment are loaded into the SPM before the segment starts execution and the segment is executed only from the local memory without any access to the main memory. A segment can be either terminal or streaming as we define next:

- *Streaming Segment*: a segment is streaming if data swapping can be done during the current segment such that the data used by the previous segment of the same task is swapped-out to the main memory -if needed- and the data required by the next segment is swapped-in the SPM -if needed-. Note that data shared between the current segment and the next segment, or between the current segment and the previous segment, is not swapped. For example, if array a and array b are loaded in the SPM and used by the current segment, and the next segment requires array a and array c, then only array c is swapped-in during the execution of the current segment. In details, the swapped-out data comprises all data used by the previous segment but not the current one, while the swapped-in data comprises all data used by the next segment but not the current one.

- *Terminal Segment*: a segment is terminal if it cannot be streamed into the next segment of the same task. That is, the code and data required by the next segment cannot be loaded during the current segment either due to data dependency or limited space in the SPM or a constraint imposed by the compiler, e.g. a control dependency where the next segment is only known at the end of the current segment. Note that the first and the last segments of a task are always terminal segments.

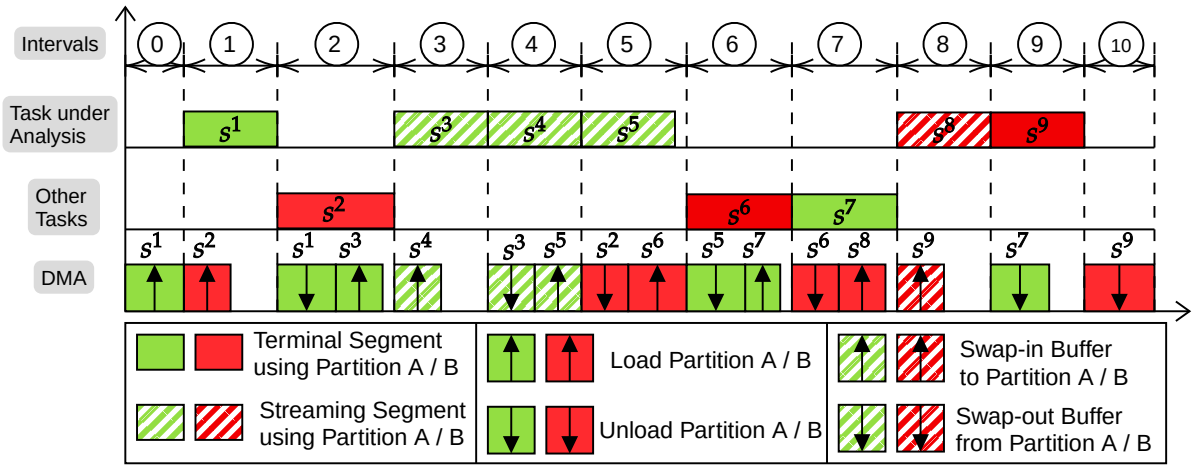


Figure 5.2: Streaming Execution Model

Figure 5.2 illustrates an example schedule for a streaming task under analysis. In the figure, we have a schedule for several tasks running on one processor and a local SPM with two partitions: A and B. The task under analysis has six segments: s^1 and s^9 are terminal segments while s^3, s^4, s^5, and s^8 are streaming segments. Segments s^2, s^6 and s^7 are terminal and belong to other tasks. Like in the example in Figure 5.1, the schedule consists of a sequence of scheduling intervals; in each scheduling interval, a segment can be executed in parallel with DMA operations.

The schedule in Intervals ⓪, ① and ② follows the same scheme as in previous work on the 3-phase model, as discussed in Section 5.1.1, since it involves terminal segments. Assume that initially no task is ready, both SPM partitions are empty, and that the task under analysis (u.a) arrives at the beginning of Interval ⓪. Then the scheduler first loads s^1 in partition A, so that it can be executed in Interval ① after the load operation finishes. Since s^1 is terminal, it cannot be followed by a segment of the task u.a.; instead, assuming that a segment s^2 of another task is ready, s^2 is loaded in partition B during Interval ① to be executed in Interval ②. As s^3 is the first streaming segment after a terminal segment, it is completely loaded in partition A during Interval ② after s^1 is unloaded. Intervals ③, ④, and ⑤ depict the case of multiple streaming segments executed in sequence. All such intervals use the same partition A: first, s^4 is swapped-in partition A during Interval ③ and executed in Interval ④. Then, s^5 is swapped-in during Interval ④ in exchange for swapping-out s^3, then executed in Interval ⑤. Although s^5 can be streamed into s^8, we assume that the stream is preempted by segments s^6 and s^7 from other tasks with

higher priority than the task u.a. To execute s^6 during Interval ⑥, s^2 must be unload and s^6 loaded in partition B during Interval ⑤, while the preempted streaming segment s^5 executes. Since s^7 must then be executed in partition A, s^5 is also completely unloaded from partition A during Interval ⑥. As the task u.a. can resume execution in Interval ⑧, s^8 is loaded to partition B in Interval ⑦. Due to the preemption of the stream, all the code and data for s^8 has to loaded. Also note that due to preemption, in this case the task u.a. resumes its execution from the other partition. While s^8 executes in Interval ⑧, we swap-in the data of s^9, which terminates both the stream and the task. Note that we do not need to swap-out the previous segment s^5 of the task u.a., since it has been unloaded in Interval ⑥. Finally, assuming that no other task is ready, s^7 is unloaded and partition A becomes empty in Interval ⑨, while s^9 is unloaded and partition B becomes empty in Interval ⑩.

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

1. The schedule comprises a sequence of scheduling intervals. During each Interval$_i$, at most one segment s^i is executed in parallel with at most one memory operation (unload and load, or only load, or only unload, or swap-out and swap-in, or only swap-in). The interval ends when both the segment execution (if any) and the memory operation (if any) have completed.

2. If a segment s^i executes during Interval$_i$ -i.e. there is a loaded partition-, or if there is a ready task at the end of Interval$_i$, then Interval$_{i+1}$ starts immediately after Interval$_i$ ends. Otherwise, Interval$_{i+1}$ starts when a task becomes ready.

3. Given two segments s^i and s^{i+1} executed in successive scheduling intervals, if $\tau(s^i) = \tau(s^{i+1})$ then s^i must be a streaming segment.

4. Scheduling decisions are only taken at the beginning of a scheduling interval; namely, at the beginning of Interval$_i$, the scheduler decides which segment s^{i+1} (if any is ready and it does not violate Rule 3) to execute in the following Interval$_{i+1}$.

5. Consider memory operations performed during Interval$_i$, there are two cases:

   - If there is a segment s^i executing in Interval$_i$, a segment s^{i+1} to be executed in Interval$_{i+1}$, and $\tau(s^i) = \tau(s^{i+1})$; then: if Interval$_{i-1}$ executed segment s^{i-1} and $\tau(s^{i-1}) = \tau(s^i)$, s^{i-1} is swapped-out and segment s^{i+1} is swapped-in; otherwise only segment s^{i+1} is swapped-in.

- If no segment is executed in Interval$_i$, or no segment is scheduled to be executed in Interval$_{i+1}$, or there is a segment s^i executing in Interval$_i$, a segment s^{i+1} to be executed in Interval$_{i+1}$, and $\tau(s^i) \neq \tau(s^{i+1})$; then: if there exists a segment s^p that was executed but not unloaded, then s^p is unloaded; and if Interval$_{i+1}$ will execute segment s^{i+1}, then s^{i+1} is loaded.

We now make the following observations based on the rules. First, Rule 4 does not specify how to choose the next scheduled segment; to construct a schedulability analysis, we will assume a fixed per-task priority assignment. Second, Rule 5 specifies that swap-in/swap-out operations are performed for the current task in the same partition if the next segment belongs to the same task, otherwise the previous segment that has not been unloaded (if any) is unloaded and the next segment (if any) is loaded. Note that a previous segment could have been executed multiple intervals before the current interval, but has not been unloaded. This is the streaming case where the swap-in/swap-out operations are scheduled for the current executing task until the stream is preempted. An example of this case is segment s^2 in Figure 5.2 which is unloaded in Interval ⑤.

5.2.2 Platform Assumptions

We assume that the model is executed on a MPSoC platform comprising multiple processors. Tasks are partitioned to processors. A DMA component is used to execute memory phases and shared among all processors. To ensure that task execution on each processor is independent of the other processors, we further assume a TDMA arbitration for the DMA component. As discussed in Section 5.1.1, we consider two timing models for the DMA. Under the fixed-size DMA model, the time required to complete all memory phases in each scheduling interval is constant and equal to Δ. Such model has been implemented, as an example, on a Freescale MPC5777M SoC platform in [136].

Under the variable-size DMA model, the time depends on the actual length of memory phases (load / unload / swap-in and out) performed during the interval. As an example, the implemented tri-core Ultrascale+ platform in [57] employs a fine-grained TDMA arbitration among the cores. Consider M cores, and assume each core is assigned a TDMA slot of size σ_j, with $\Sigma = \sum_{j=1}^{M} \sigma_j$ the length of the TDMA round. Further assume an overhead *over* for switching between slots. Let t denote the time required to execute the memory phases in a given interval on core j, assuming that the DMA services core j only. Then the total memory time for that interval is bounded by:

$$\left\lceil \frac{t}{\sigma_j - over} \right\rceil \cdot \Sigma + \sigma_j; \tag{5.1}$$

a number of slots equal to $\lceil t/(\sigma_j - over) \rceil$ is required to complete the DMA phases, the core receives one slot every Σ, and the first slot can be wasted if the memory phase arrives just after its beginning.

To abstract from the complexities of the underlying TDMA implementation, in the rest of the dissertation we will assume that the variable-time model is characterized by a rate parameter ρ and a latency parameter δ, such that the memory time in an interval can be bounded as: $\rho \cdot t + \delta$. Here, the rate parameter represents the slowdown due to the need to arbitrate between M cores, while the latency parameter represents the overhead introduced by the granularity of the TDMA schedule. Note that from Equation 5.2 we obtain:

$$\left\lceil \frac{t}{\sigma_j - over} \right\rceil \cdot \Sigma + \sigma_j \leq t \cdot \frac{\Sigma}{\sigma_j - over} + \Sigma + \sigma_j, \tag{5.2}$$

which means that for the scheme in [57], setting $\rho = \Sigma/(\sigma_j - over)$ and $\delta = \Sigma + \sigma_j$ results in a valid upper bound on the memory time.

5.2.3 Task Model

We consider a set of sporadic tasks $\Gamma = \{\tau_1, \ldots, \tau_N\}$ executed on a given processor. We use T_i to denote the period (or minimum inter-arrival time) of task τ_i, and D_i for its relative deadline. We assume constrained deadline: $D_i \leq T_i$. 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. A job of τ_i that arrives at time t is feasible if s^{end} completes execution no later than $t + D_i$ [2].

A segment can be either terminal or streaming. We say that segment s^j streams into segment s^k if the content of the memory required by s^k can be loaded in the SPM during the execution of s^j. s^{begin}, s^{end} are terminal segments. Any other segment s^j is a *streaming segment* if s^j has a unique immediate successor segment s^k that it streams into; otherwise,

[2] Note that some previous related work [136] required the unload phase of s^{end}, rather than its execution phase, to complete by $t + D_i$. The use of either definition depends on platforms-related assumptions, i.e., whether output operations are performed during the execution of the task, or during the unload phase. We use the definition based on the execution phase of the last segment as it leads to a simpler analysis, and the main objective of this dissertation is to show how to compile and segment the tasks. However, we point out that the analysis could be modified to incorporate the assumption in [136]. In particular, if no streaming is employed, the unload phase of s^{end} must be performed at the beginning of the following segment.

s^j is a *terminal segment*. We further say that a segment s^j is *initial* if its immediate predecessor(s) is terminal, or if the segment is s^{begin} (note that by definition, if s^j has multiple immediate predecessors, they must all be terminal). Note that an initial segment can either be terminal or streaming.

We use $s.c$ to denote the worst-case computation time of a segment s (including context-switch overheads). For the variable-size DMA model, we further use $s.ld$ and $s.ul$ to denote the time of the load and unload phases for s; furthermore if s is a streaming segment, we use $s.st$ to denote the time for the swap-in/out phase(s) executed in parallel with s. Note that based on the DMA model in Section 5.2.2, this means that if a streaming phase of length $s.st$ is performed in a scheduling interval, the memory time for that interval is $\delta + \rho \cdot s.st$; while if an unload phase for segment s^j is performed together with a load phase for another segment (possibly of a different task) s^k, the memory time is $\delta + \rho \cdot (s^j.ld + s^k.ul)$.

For the fixed-size model, the memory time for every scheduling interval is equal to Δ. Hence, we define the length $s.l$ of segment s as the maximum length of any scheduling interval for the segment, that is, $\max(s.c, \Delta)$. For the variable-size model, we similarly define the length $s.l$ of a streaming segment to be equal to $\max(s.c, \delta + \rho \cdot s.st)$; while we define the length $s.l$ of a terminal segment to be simply equal to $s.c$, as the memory time depends on the executed memory phases. We use p to denote a DAG path, that is, an ordered sequence of segments; $p.S$ is the number of segments in the path, $p.I$ is the number of terminal segments, $p.L$ is the sum of the lengths of all segments, and $p.end$ the length of the last segment. We write $s \in p$ to mean that segment s belongs to path p, and $p \in G_i$ to mean that p is a path of G_i. We say that a path is maximal if its first segment is s^{begin} and its last segment is s^{end}. To avoid confusion, we use uppercase letters (P) to denote maximal paths. Note that by definition $P.end = s^{end}.l$. 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.

5.3 OS Programming Interface

We propose an OS-level Application Programming Interface (API) to be inserted in the code of a task to partition it into segments and to communicate the changes in the SPM content to the OS. The API can be inserted manually or automatically during the program compilation as we propose in this work. The API is used for allocation and de-allocation of objects and buffers. A buffer is used for segment streaming such that the content of the buffer can be swapped with other data during execution. We refer to other SPM allocations that are not buffers as objects.

An object/buffer can be a 1D or a 2D memory block. A 1D object/buffer represents a 1D (sub-)array or any linear structure such that all its content is contiguous in the main memory. Hence, a 1D object/buffer has a single length `size`. A 2D object/buffer represents a sub-array of a 2D array in the main memory. We refer to the width of the source 2D array in the main memory as `src_pitch` and the width of the destination 2D array in the SPM as `dst_pitch`. For the transfer of a 2D object/buffer between main memory and the SPM, we use parameters `height` and `width`, which represent the number of rows and the number of bytes per row to be copied. Each object/buffer has a usage attribute `attr` that indicates if the data in the object/buffer is write-only, read-only, or read-write.

Objects are allocated/deallocated in terminal segments only using `allocate`/`allocate2d`/`deallocate` functions. Buffers are allocated before the first streaming segment using `allocate_buffer` and deallocated using `deallocate_buffer` at the end of the segment stream. The content of a buffer can be modified during the execution of streaming segments using `swap_buffer` and `swap2d_buffer` functions. A swap implies that the current data will not be needed by the next segment, and that new data should be loaded in the buffer before the next-to-next segment of the task; note that the swap-out and swap-in operations defined in Section 5.2 correspond to reading from / writing to main memory the content of buffers.

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

The following presents a detailed description of each function in the API:

- `int allocate(uint64_t *src, uint64_t *dst, int size, int attr)`:
  - **Description:** create SPM block at `dst` and copy `size` bytes from `src` to `dst` if `attr` is read-only or read-write.
  - **Parameters:** `src`→ address in main memory, `dst`→ address in SPM, `size`→ size, `attr`→ object usage ([0] read-only, [1] write-only, [2] read-write).
  - **Return:** the ID of the SPM object.
- `int allocate2d(uint64_t *src, uint64_t *dst, int width, int height, int src_pitch, int dst_pitch, int attr)`:

  - **Description:** create 2D SPM block with size `dst_pitch*height` at `dst` and copy a 2D-block with width `width` bytes and height `height` bytes `src` to `dst` if `attr` is read-only or read-write.
  - **Parameters:** `src`→ address in main memory, `dst`→ address in SPM, `width`→ width of the transfer, `height`→ of the transfer, `src_pitch`→ pitch of the source, `dst_pitch`→ pitch of the destination, `attr`→ object usage ([0] read-only, [1] write-only, [2] read-write).
  - **Return:** the ID of the SPM object.

- `void deallocate(int obj_id)`:
  - **Description:** release SPM object with ID `obj_id`.
  - **Parameters:** `obj_id`→ object ID.

- `int allocate_buffer(uint64_t *dst, int size, int attr)`:
  - **Description:** allocate SPM buffer at `dst` with size `size` bytes.
  - **Parameters:** `dst`→ address in SPM, `size`→ size,
  - **Return:** the ID of the SPM buffer.

- `void swap_buffer(int buf_id, uint64_t *src, int size)`:
  - **Description:** swap the data in SPM buffer with ID `buf_id` by writing the current data if required and fetching the new data from `src` if required.
  - **Parameters:** `buf_id`→ buffer ID, `src`→ address in main memory, `size`→ size,

- `void swap2d_buffer(int buf_id, uint64_t *src, int width, int height, int src_pitch, int dst_pitch)`:
  - **Description:** swap the data in SPM buffer with ID `buf_id` by writing the current data if required and fetching the new data from `src` if required (both source and destination are 2D arrays).
  - **Parameters:** `buf_id`→ buffer ID, `src`→ address in main memory, `width`→ width of the transfer, `height`→ of the transfer, `src_pitch`→ pitch of the source, `dst_pitch`→ pitch of the destination.

- `void deallocate_buffer(int buf_id)`:
  - **Description:** deallocate buffer with ID `buf_id`.

  - **Parameters:** `buf_id`→ buffer ID.

- `void wait()`:
  - **Description:** wait for pending memory transfers.

- `void dispatch(void)`:
  - **Description:** force all buffer DMA requsts to move from waiting queue to dispatch queue.

5.3.1 API Implementation

Table 5.1: Data fields of an entry in the Three-Phase Table (3PT) and Streaming Table (ST).

Data Structure Field	Description
usage	Usage of the object/buffer (read-only, write-only, or read-write).
size	Size for 1D object/buffer.
width, height, src_pitch, dst_pitch	Information for a 2D object/buffer.
src_ptr	Pointer to the address in main memory.
dst_ptr	Pointer to the SPM address.

We now discuss how one can implement the API in a Real-time Operation System (RTOS) and show how it works with an example. The OS tracks the objects and buffers used by each task using two tables, *Three-Phase Table (3PT)* and *Streaming Table (ST)*. An entry in the 3PT or ST includes the fields in Table 5.1. An entry is created in the 3PT when either `allocate` or `allocate2d` is called and the entry ID is returned. When `deallocate` is called with the entry ID, the entry is removed from the table. Similarly, an entry is created in the ST using `allocate_buffer` and is removed using `deallocate_buffer`.

The memory transfers of a task are managed by the OS using three different queues: *Three-Phase Queue (3PQ)* for objects and *Streaming Wait Queue (SWQ)*/*Streaming Dispatch Queue (SDQ)* for buffers. Each queue contains a list of DMA transfer requests: either reading an object/buffer from main memory to the SPM, or writing back an object/buffer from the SPM to main memory. Allocating a read-only/read-write object adds a read request to the 3PQ, and deallocating a write-only/read-write object adds a write request to the 3PQ. Swapping a buffer adds a write request for the current data to the

SWQ if the buffer is write-only/read-write, and a read request for the new data if the buffer is read-only/read-write. Note that the first swap of a buffer does not add a write request as the buffer is empty. Deallocating a buffer adds a write request for the current data to the SWQ. Using two queues for the streaming buffers is necessary: if a single queue is used, the current segment may call `swap_buffer`/`swap2d_buffer`/`deallocate_buffer` and hence add new read/write requests to the queue while the requests for the next segment are processed. Using two queues avoids this issue by distinguishing between the requests for the next segment, which are processed during the current segment from the SDQ, and requests added to the SWQ by the buffer swap/deallocation. Moving requests from the SWQ to the SDQ is done by the scheduler, with the exception of the explicit usage of `dispatch` function before the segment stream starts.

Note that the OS keeps separate tables and queues for each task as attributes of the Task Control Block (TCB). Since the data required for the first segment in the task has to be allocated in the SPM before executing the segment, the TCB also contains the allocation state for the first segment, which the OS copies in the 3PT when a new job of the task starts.

Besides allocation/swapping/deallocation, DMA requests are managed based on the scheduling decision at the beginning of each interval. Algorithm 3 shows the steps taken by the scheduler for a given processor at the beginning of Interval_i. We use the notation τ_i/τ_{i+1} for tasks scheduled in Interval_i/ Interval_{i+1}. Task $\tau(s^i)/\tau(s^{i+1})$ is assigned to τ_i/τ_{i+1} if s^i/s^{i+1} is scheduled in the interval, otherwise τ_i/τ_{i+1} is empty. We also use τ_p for the task of an executed segment s^p that has not been unloaded. If s^p does not exist, then τ_p is empty. The scheduler starts by determining the segment s^{i+1} to be executed in Interval_{i+1} (Rule 4). Then, the scheduler dispatches buffer DMA requests (if any) from SWQ to SDQ of task τ_i if not empty. After that, it determines the memory operations to be carried out in Interval_i based on Rule 5. If τ_i is not empty and $\tau_i = \tau_{i+1}$ -i.e. s_i is streaming into s_{i+1}-, then a swap-out/swap-in operation is scheduled. In this case, the write/read requests for the previous/next segment are processed by sending the write requests from the SDQ to the DMA, then sending the read requests from the SDQ to the DMA. If τ_i is empty or $\tau_i \neq \tau_{i+1}$ -i.e. s_i is not streaming into s_{i+1}-, then load and unload operations are scheduled. For an unload operation, write requests for modified (write-only or read-write) objects in the 3PT of $\tau(s^p)$ are added to its 3PQ. Note that if an object is in 3PT, this means that it has not been deallocated yet and thus needs to be reloaded when the task resumes execution. Therefore, read requests are also added to 3PQ for all objects in 3PT [3]. Then, write requests in the 3PT are sent to the DMA. For buffers, all write requests in SDQ

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

and SWQ are sent to the DMA. Note that it is necessary to process the write requests in SWQ even though they have not dispatched to SDQ as all the modified data has to been written back to main memory. For a load operation, if τ_{i+1} has not started yet, the initial state of the 3PT is copied from the TCB of τ_{i+1} and read requests are added to the 3PQ for read-only/read-write objects. Then, all read requests in 3PQ and SDQ are sent to the DMA. Finally, the task τ_p is updated to τ_i if no segment is scheduled in Interval_i or if $\tau_i \neq \tau_{i+1}$, i.e. s^i is not streaming into s^{i+1}. That is, a segment that has executed, but has not been unloaded (if any) is unloaded during Interval_i in that case according to Rule 5. Then, s^i will be the last executed segment that has not been unloaded and hence $\tau_p = \tau_i$ if s^i exists, or $\tau_p = \tau_i = \emptyset$ if no segment is executed in Interval_i.

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

Example

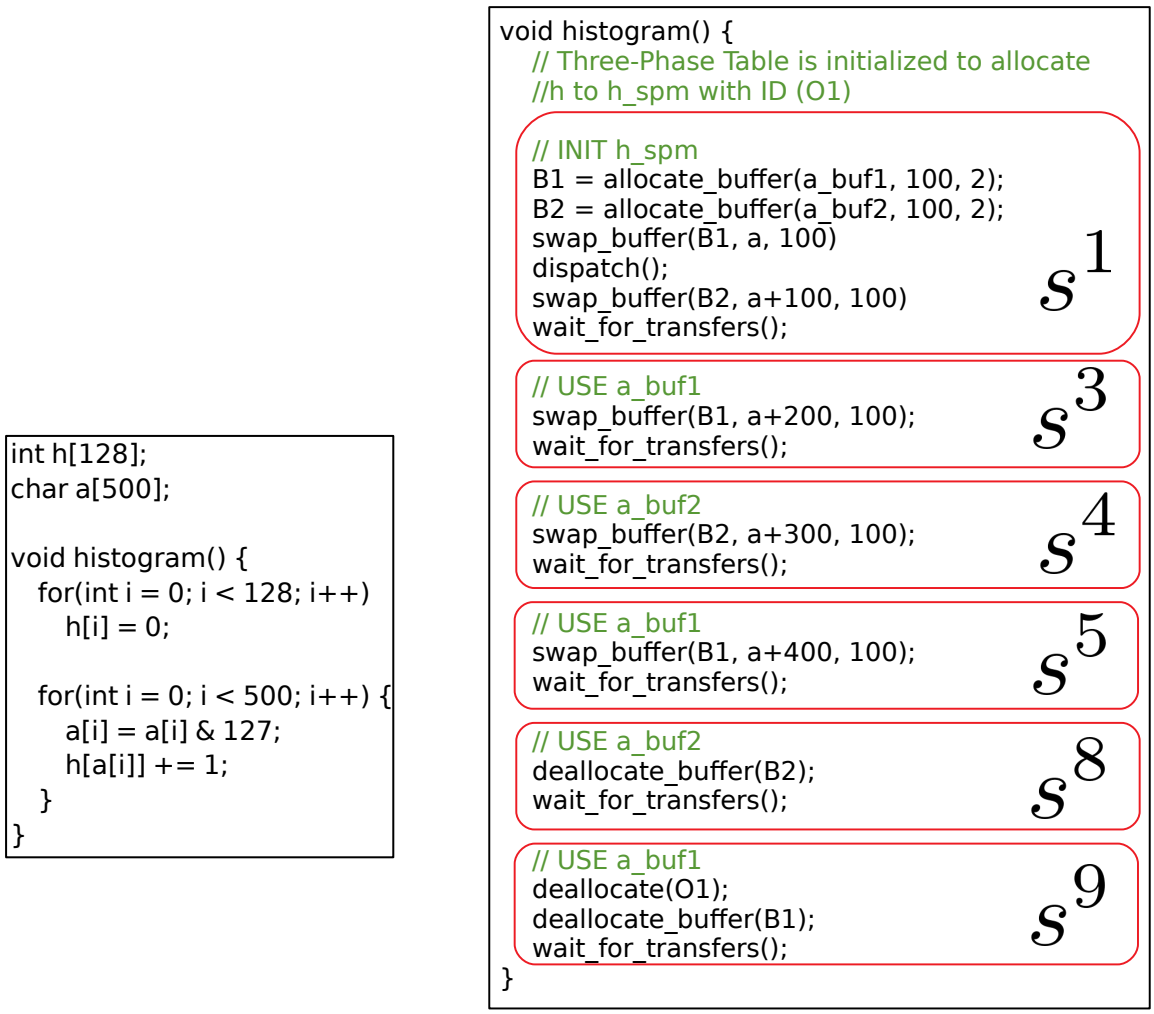
We describe how the schedule from Figure 5.2 is accomplished using the proposed API and implementation with an example program. Figure 5.3a shows the source code of `histogram` function that uses two arrays h and a. It starts with an initialization of all elements of h, then it iterates over elements of a masking their values and then incrementing the corresponding histogram bin. Our goal is to do the initialization of h in a segment and break the histogram loop into 5 segments such that each segment processes 100 element of array a. The code in Figure 5.3b represents the segmented function after adding the API calls[4] and Figure 5.4 shows the OS tables and queues for the task; for each scheduling interval in Figure 5.2, we show the content for the tables and queues after the scheduler logic and the code of the segment is executed, but before the DMA interrupt removes any request.

Since the first segment for the task s^1 uses array h, h has to be allocated in the SPM

[4]We unrolled the outer loop that iterates over segments to illustrate the details of the execution and replaced the actual loops with representative comments.

Algorithm 3 Scheduler logic in Interval$_i$

Determine segment s^{i+1} (if any)
$\tau_i = \tau(s_i)$ if a segment s_i is scheduled in Interval$_i$, else $\tau_i = \emptyset$
$\tau_{i+1} = \tau(s_{i+1})$ if a segment s_{i+1} will be scheduled in Interval$_{i+1}$, else $\tau_{i+1} = \emptyset$
if $\tau_i \neq \emptyset$ **then**
    τ_i: Dispatch requests from SWQ to SDQ.
if $\tau_i \neq \emptyset$ **and** $\tau_{i+1} \neq \emptyset$ **and** $\tau_i = \tau_{i+1}$ **then** (swap-out/swap-in)
    τ_i: Send write requests in SDQ to DMA.
    τ_{i+1}: Send read requests in SDQ to DMA.
else
    if $\tau_p \neq \emptyset$ **then** (unload)
        τ_p: Add read/write requests to 3PQ for objects in the 3PT.
        τ_p: Send write requests in 3PQ to DMA.
        τ_p: Send write requests in SDQ to DMA.
        τ_p: Send write requests in SWQ to DMA.
    if $\tau_{i+1} \neq \emptyset$ **then** (load)
        if τ_{i+1} has not started **then**
            $\tau(s^{i+1})$: Copy initial state of 3PT from TCB.
            $\tau(s^{i+1})$: Add read requests to 3PQ for objects in 3PT.
        τ_{i+1}: Send read requests in 3PQ to DMA.
        τ_{i+1}: Send read requests in SDQ to DMA.
if $\tau_i = \emptyset$ **or** $\tau_i \neq \tau_{i+1}$ **then**
    $\tau_p = \tau_i$



(a) Original code

(b) Segmented code

Figure 5.3: API usage example

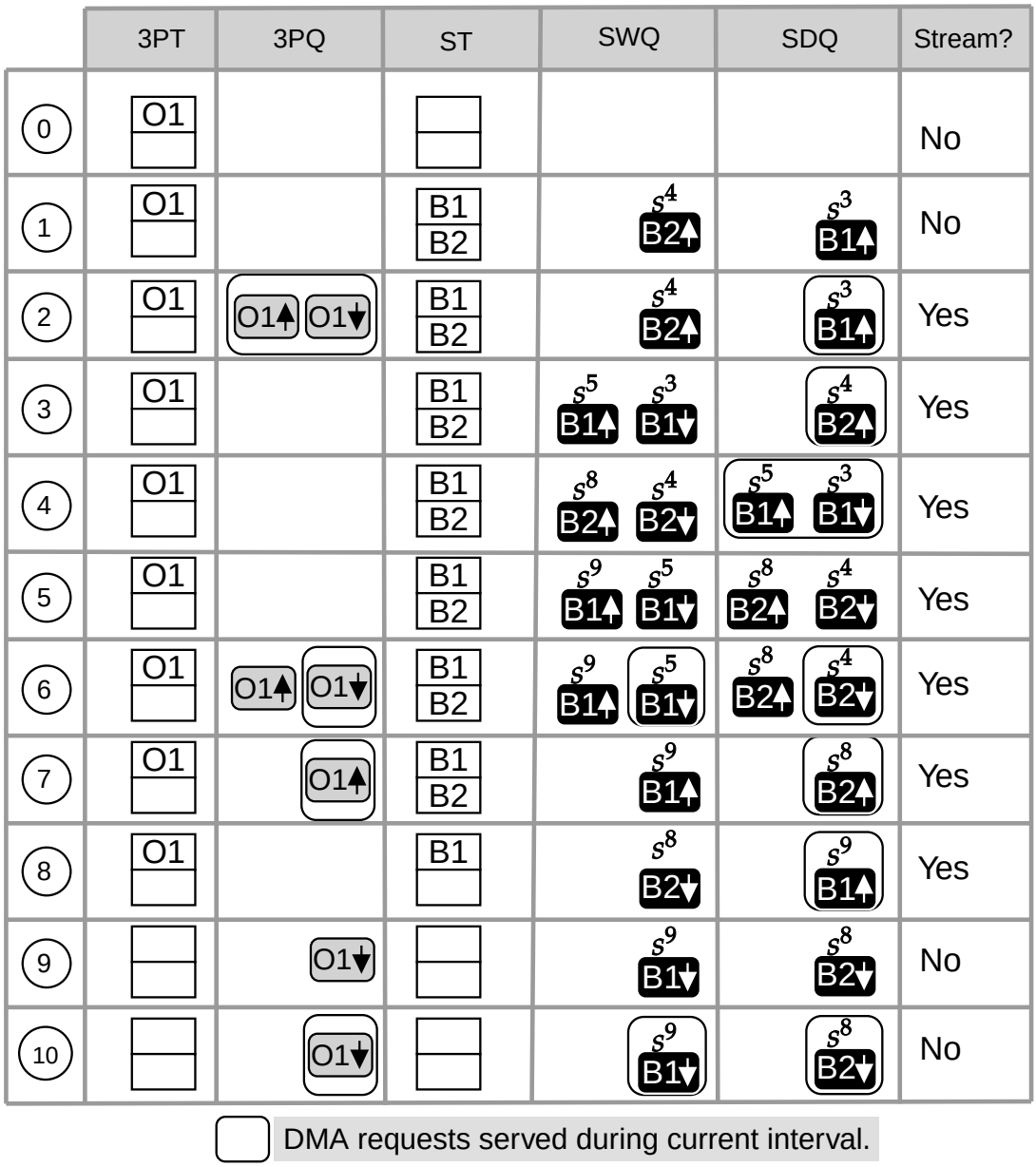


Figure 5.4: SPM management example

before s^1 starts. Accordingly, the 3PT is initialized with entry O1 as shown in Figure 5.3 corresponding to allocating h to h_spm with a write-only status. The OS processes this allocation in interval ⓪ before the segmented function starts. Note that O1 does not trigger a read request as the object is write-only. However, the code and other data, e.g. the stack, are transferred in this interval. During s^1 execution in interval ①, h_spm is initialized and two read-write buffers B1 and B2 are allocated for array a. The first 100 bytes of a are swapped-in B1 and a read request B1↑ is added to the SWQ, then moved to the SDQ once `dispatch` function is called. After that, the next 100 bytes of a are swapped-in B2 and a load transfer B2↑ is added to the SWQ. Since s^1 is a terminal segment, a different task executes in interval ②. Hence, a write request O1↓ and read request O1↑ for O1 are added to the 3PQ by the scheduler. In this example, segment s^3 of the task is assumed to resume in interval ③. Therefore, both O1↓ and O1↑ are processed [5] during interval ② as well as B1↑. In interval ③, s^3 executes the histogram loop on the first 100 bytes of a from B1, then invokes a swap for B1 for the third 100 bytes of a. This adds two transfers in the SWQ to write back the current data B1↓ and then read the new data B1↑. The entry for B1 is modified to reflect the swap result. While s^3 is executing, B2 is filled with the second 100 bytes of a as B2↑ is processed from the SDQ. In interval ④, swapping B1 proceeds from the SDQ and a swap for B2 is added to the SWQ. At the beginning of ⑤, the OS schedules segment s^6 from another task to be executed in interval ⑥ and no DMA requests are processed for the task as the other partition for s^6. A swap for B1 is also added to the SWQ in interval ⑤. At the beginning of interval ⑥, write/read requests for O1 are added to the 3PQ. Then, B2↓ is processed from the SDQ, B1↓ is processed from the SWQ, and O1↓ is processed from the 3PQ. The OS prepares the task in interval ⑦ to resume execution. So, B2↑ is processed from the SDQ along with O1↑ from the 3PQ. In interval ⑧, the task resumes executing s^8 while B1↑ is processed. Since s^8 is the last segment to use B2, B2 is deallocated and B2↓ is added to the SWQ. In interval ⑨, the segment stream is concluded with s^9 in which all the remaining objects and buffers are deallocated; and therefore the tables are cleared. The deallocation triggers B1↓ and O1↓ to copy back the modified data to main memory. The requests B2↓, B1↓, and O1↓ are processed in interval ⑩ as shown in Figure 5.3.

The OS identifies the execution mode (streaming or three-phase, as needed for Rule 3) based on the current mode and the API calls. That is, when B1 and B2 are allocated in the terminal segment s^1, the OS knows that it will switch to streaming mode in the next segment. The task remains in the streaming mode until all buffers are deallocated and the ST is empty.

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

5.4 Scheduling Analysis for the Fixed-size DMA Model

In this section, we develop a sufficient schedulability analysis for a task set scheduled according to Rules 1 - 5 in Section 5.2.1 on one processor, where scheduling decisions in Rule 2 are based on fixed per-task priorities and the platform employs the fixed-size DMA model. In essence, we extend the analysis in [136] for fixed-size DMA systems to support conditional, streaming task execution. Since we use fixed priority scheduling, without loss of generality we assume that tasks in $\Gamma = \{\tau_1, \ldots, \tau_N\}$ are ordered by decreasing and distinct priorities. Before detailing the critical instant (the task arrival pattern that leads to the worst case response time for the task under analysis), we begin by introducing some properties for the DAG model under fixed-size DMA.

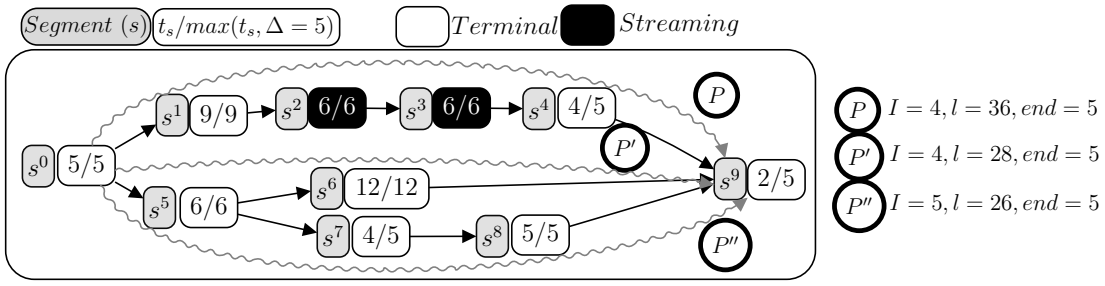


Figure 5.5: Example segment DAG (s^0 is s^{begin} and s^9 is s^{end}).

Figure 5.5 shows an example DAG with three maximal paths: $P = \{s^0, s^1, s^2, s^3, s^4, s^9\}$, $P' = \{s^0, s^5, s^6, s^9\}$, and $P'' = \{s^0, s^5, s^7, s^8, s^9\}$. Note that we have $P.L = 36, P.I = 4, P'.L = 28, P'.I = 4, P''.L = 26, P''.I = 5$, and $P.end = P'.end = P''.end = 5$ (recall that $p.L$ counts the length of all segments in the path, but $p.I$ counts only terminal segments). 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 3. *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 [6] of all maximal paths in G. Intuitively, for a task τ_i, we

[6] 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.

show in this section that the set $G_i.C$ replaces the concept of worst-case execution time. For example, for Figure 5.5, $G.C$ 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 3 to compare two DAGs for the same program.

Definition 4. *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.

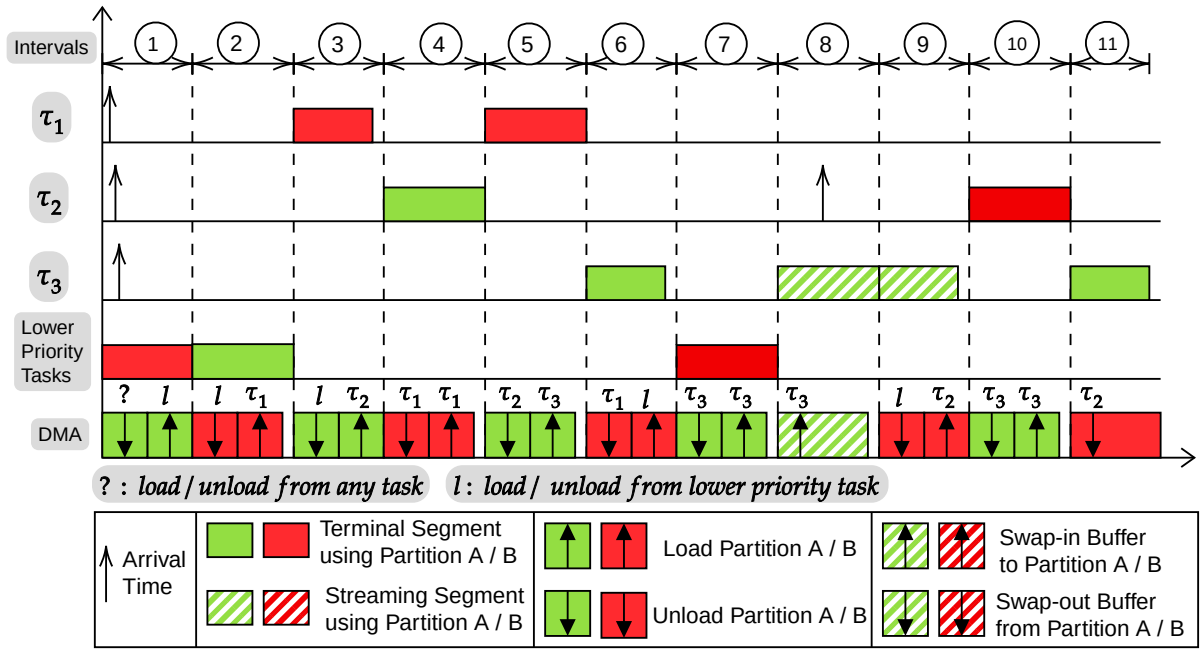


Figure 5.6: Example critical instant for fixed-priority scheduling in the fixed-sized DMA model.

The critical instant for a task under analysis τ_3, as derived in [136, 152], is depicted in Figure 5.6. 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:

$$l_i^{l\max} = \max(\Delta, \max_{j=i+1,N} \max_{s \in S_j} s.l) \tag{5.3}$$

$$B_i = \begin{cases} 2 \cdot l_i^{l\max}, & \text{if } i \leq N-1. \\ \Delta, & \text{if } i = N. \end{cases} \tag{5.4}$$

For the lowest priority task τ_N, the maximum blocking time is Δ as there can only be one initial blocking interval consisting of memory only (a load, possibly preceded by an unload to free a partition). 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. Finally, note that τ_i suffers one extra blocking time of length $l_i^{l\max}$ every time it executes a terminal segment, with the exception of the last terminal segment s^{end}: a terminal segment cannot be followed by another segment of τ_i, hence in the worst case a segment of a lower priority task can execute instead, as shown in interval ⑦ in the figure. Since for a maximal path P of τ_i there are $P.I$ terminal segments, such blocking time is bounded by $(P.I-1) \cdot l_i^{l\max}$.

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

$$\text{Inter}_i(t) = \sum_{j=1}^{i-1} \lceil t/T_j \rceil \cdot L_j, \tag{5.5}$$

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:

$$L_j^{\max} = \max\{P.L \mid P \in G_j.C\}. \tag{5.6}$$

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 3 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:

$$R_i(P) = B_i + (P.I - 1) \cdot l_i^{l\max} + P.L - P.end + \text{Inter}_i\big(R_i(P)\big), \tag{5.7}$$

and the task is schedulable along that path if:

$$R_i(P) \leq D_i - P.end. \tag{5.8}$$

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 the key property of the analysis with respect to path domination.

Property 1. *Consider two paths P, P' with $P' \succeq P$. If Equation 5.8 holds for P', then it also holds for P.*

Proof. Note that Equation 5.5 is increasing in t, and Equation 5.7 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 5. *Task τ_i is schedulable if:*

$$\bigwedge_{P \in G_i.C} R_i(P) \leq D_i - P.end. \tag{5.9}$$

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 5.9 for all tasks in the order $\tau_1, \ldots, \tau_N$. However, as we show in Chapter 6, each program can be segmented in many different ways, resulting in

different DAGs for the task. Hence, the real problem that we are interested in solving is how to find a set of "best" DAGs for the tasks in Γ, that is, a set of DAGs that make the task set schedulable according to our derived analysis. To discuss how we proceed, we prove two more properties of the analysis.

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

Proof. Part (A): by definition of Equations 5.7, 5.9. 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. □

Based on Property 2, we can proceed as follows: we again iterate on the tasks in inverse priority order. At each step, we 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 a value is then used by our segmentation algorithm working on τ_{i+1}: as we detail in Section 6.1, the algorithm considers a segmentation of τ_{i+1} to be valid only if its maximum segment length is no larger than a specified value $l^{\max}$. Note that in theory, one could determine $\overline{l_i^{l\max}}$ by performing a binary search over Equation 5.9. However, we show in the next Section 5.4.1 that an alternative formulation based on the concept of scheduling points used in [19] 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 5.9 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 5.9 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 segmentation algorithm can use Definition 4 to determine the set of "best" DAGs for a task. In fact, in

Section 6.2 we will prove that our segmentation algorithm is optimal, in the sense that if there are segmentations that result in a schedulable task set according to the analysis, the algorithm will find one such segmentation for each task in Γ.

5.4.1 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 [19]. In details, define the set of points:

$$\begin{aligned}&\mathcal{S}_i(P.end) = (D_i - P.end) \cup \\ &\{k \cdot T_j \mid j = 1 \dots i-1, k = 1 \dots \lfloor (D_i - P.end)/T_j \rfloor\}.\end{aligned} \tag{5.10}$$

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:

$$\bigwedge_{P \in G_i.C} \bigvee_{t \in \mathcal{S}_i(P.end)} B_i + (P.I - 1) \cdot l_i^{l\max} + P.L - P.end + \text{Inter}_i(t) \leq t, \tag{5.11}$$

where $\bigvee$ represents the or of the conditions. In practice, the test can be efficiently evaluated because, as shown in [19], 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 5.11 as a condition on the maximum value of $l_i^{l\max}$ through simple algebraic manipulation. For a task τ_i with $i \leq N-1$ we obtain:

$$\begin{aligned}&\bigwedge_{P \in G_i.C} \bigvee_{t \in \bar{\mathcal{S}}_i(P.end)} (P.I + 1) \cdot l_i^{l\max} + P.L - P.end + \text{Inter}_i(t) \leq t \\ \Leftrightarrow &\bigwedge_{P \in G_i.C} \bigvee_{t \in \bar{\mathcal{S}}_i(P.end)} l_i^{l\max} \leq \frac{t - P.L + P.end - \text{Inter}_i(t)}{P.I + 1} \\ \Leftrightarrow &\overline{l_i^{l\max}} = \min_{P \in G_i.C} \max_{t \in \bar{\mathcal{S}}_i(P.end)} \frac{t - P.L + P.end - \text{Inter}_i(t)}{P.I + 1}.\end{aligned} \tag{5.12}$$

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

$$0 \leq \min_{P \in G_i.C} \max_{t \in \bar{\mathcal{S}}_i(P.end)} t - P.L + P.end - P.I \cdot \Delta - \text{Inter}_i(t). \tag{5.13}$$

5.5 Schedulability Analysis for the Variable-Size DMA Model

We next provide a sufficient schedulability analysis for a multi-segment streaming conditional task set based on the variable-size DMA model. As in Section 5.4, we assume fixed per-task priorities, where tasks are indexed by decreasing, distinct priorities, and each task τ_i has an associated segment DAG $G_i = (S_i, E_i)$. Similarly to how Section 5.4 extends the analysis in [136] to the case of conditional, streaming tasks, here we do the same with respect to the original 3-phase analysis in [151], which is also based on a variable-size model.

The same critical instant as in Figure 5.6 applies, where the task under analysis τ_i suffers an initial blocking time by two intervals where lower priority tasks execute. We assume that the intervals in the busy period (the window of time where the task under analysis is active) are numbered by increasing indexes, starting with interval ①. Hence, by definition for the critical instant, intervals ① and ② represent the initial blocking time. As before, we are interested in computing the response time $R_i(P_i)$ of τ_i up to its last segment, assuming that the task executes along maximal path P_i; the task is then schedulable under the condition:

$$\bigwedge_{P_i \in G_i} R_i(P_i) \leq D_i - P_i.end. \tag{5.14}$$

Note that in this case we need to check all maximal paths in G_i, rather than just the paths in the pareto frontier $G_i.C$, as the more complex DMA model does not allow us to define a simple dominance relation between paths.

We compute $R_i(P_i)$ iteratively, after decomposing it in two terms: 1) a constant time B_i, which represents the length of the first blocking interval ①. 2) Term $H_i\big(R_i(P_i)\big)$, which represents the cumulative length of all intervals from interval ②, to the last segment s_i^{end} of τ_i excluded (intervals ② to ⑩ in the example in Figure 5.6). Note that H_i depends on the response time $R_i(P_i)$ computed at the previous iteration, since the value of $R_i(P)$ determines the number numInter of interfering jobs of higher priority tasks in H:

$$\text{numInter}_j(t) = \lceil t/T_j \rceil, \tag{5.15}$$

$$\text{numInter}(t) = \sum_{j=1}^{i-1} \text{numInter}_j(t). \tag{5.16}$$

Let $\mathcal{P} = \{P_1, ..., P_{\text{numInter}(R_i(P_i))}\}$ denote the maximal paths of the $\text{numInter}\big(R_i(P_i)\big)$ interfering jobs, where each higher priority task τ_j has $\text{numInter}_j\big(R_i(P_i)\big)$ paths/jobs. Then

the number of intervals numH_i in H_i is equal to

$$\text{num}H_i\big(R_i(P_i)\big) = P_i.S + P_i.I - 1 + \sum_{\forall P \in \mathcal{P}} P.S : \tag{5.17}$$

we have blocking interval ②, plus one lower priority blocking interval for each terminal segment of τ_i expect the last, that is $P_i.I - 1$ intervals (interval ⑦ in the example), plus $P_i.S - 1$ intervals for segments of τ_i except the last (intervals ⑥, ⑧, ⑨), plus $\sum_{\forall P \in \mathcal{P}} P.S$ intervals of higher priority jobs (intervals ③, ④, ⑤, ⑩).

We begin with two simple observations based on the scheduling rules in Section 5.2.1. For simplicity, we shall say that a streaming segment is preempted (or for short, a p-streaming segment) if the segment executed immediately after it in the schedule belongs to a different task. If instead the following segment belongs to the same task, we say that the streaming segment is non-preempted (or for short, a np-streaming segment). For task τ_3 in Figure 5.6, the segment in interval ⑧ is np-streaming, while the segment in ⑨ is p-streaming.

Observation 6. *According to Rule 5, in the worst case an interval executing a terminal or p-streaming segment s^l also requires unloading a segment s^k and loading a segment s^j. Hence, the interval length is bounded by* $\max\big(s^l.c, \delta + \rho \cdot (s^k.ul + s^j.ld)\big)$. *$s^k$ can be a terminal or p-streaming segment; s^j can be an initial segment, or another segment where the previous segment of the same job was p-streaming.*

As an example, note that for task τ_3 in Figure 5.6 there are three load phases: for the initial segments executed in interval ⑥ and ⑧, and for the segment in interval ⑪, which follows the p-streaming segment in ⑨. There are also two unloads: the one for the terminal segment executed in interval ⑥, and for the p-streaming segment in ⑨ (the download for the last terminal segment is not shown, since it does not affect the response time).

Observation 7. *Based on Rule 4, in the worst case an interval executing a np-streaming segment s requires a swap operation of duration $s.st$. Hence, the interval length is bounded by $s.l$.*

Remember that by definition, for a streaming segment $s.l = \max(s.c, \delta + \rho \cdot s.st)$. Also note that for a streaming segment s^l that is not initial (meaning, the previous segment is also streaming), $s^l.st$ includes the time to swap-out the previous segment s^k and swap-in the next segment s^j of its job. If in the schedule, s^k is p-streaming and s^l is np-streaming

(meaning that s^l is not executed immediately after s^k), then only a swap-out is required during the interval of s^l based on Rule 4, as the data of s^k has already been unloaded. This means that the swap phase is shorter than $s^l.st$; hence, the upper bound in Observation 7 is still valid.

We can now derive the length of B_i. Note that for the lowest priority task τ_N, $B_N = 0$ since there is no lower priority task; instead, as already noticed in Section 5.4, τ_N suffers a single blocking interval ② (included in H_i) where no segment executes, but an unload and load might be required based on Rule 5. Equations 5.19-5.22 are used to derive the maximum value of any segment length, computation time, load and unload, respectively, for either any task in the system or just lower priority tasks (the latter is indicated by a superscript l).

$$l^{l\max} = \max_{\forall s \in S_{i+1},\ldots,S_N} s.l, \tag{5.18}$$

$$c^{l\max} = \max_{\forall s \in S_{i+1},\ldots,S_N} s.c, \tag{5.19}$$

$$ld^{l\max} = \max_{\forall s \in S_{i+1},\ldots,S_N} s.ld, \tag{5.20}$$

$$ul^{l\max} = \max_{\forall s \in S_{i+1},\ldots,S_N} s.ld. \tag{5.21}$$

$$ul^{\max} = \max_{\forall s \in S_1,\ldots,S_N} s.ld. \tag{5.22}$$

Lemma 8. *The length of the first blocking interval in the schedule is upper bounded by:*

$$B_i = \begin{cases} \max\left(l^{l\max}, \delta + \rho \cdot (ul^{\max} + ld^{l\max})\right), & \text{if } i \leq N-1. \\ 0, & \text{if } i = N. \end{cases} \tag{5.23}$$

Proof. If $i = N$, then there are no lower priority tasks than the task under analysis τ_N; in this case, there can only be one interval of initial blocking time (consisting of an unload of a partition, plus the load of one of the tasks executed in H_N), which is already included in H_i. Hence, in this case $B_N = 0$.

Otherwise, B_i can comprise one interval where the segment s^l of a lower priority task executes. We have two cases: 1) If s^l is np-streaming, then the segment length is $s^l.l$ by Observation 7. 2) If s^l is terminal or p-streaming, then by Observation 6 the length of the segment is bounded by $\max\left(s^l.c, \delta + \rho \cdot (s^k.ul + s^j.ld)\right)$; note that here s^k can be a segment of any task, but s^j must be a segment of a lower priority task executed in interval ② (which is included in H_i). Since for any segment, $s.l$ is either equal (for a terminal) or larger or equal (for a streaming) than $s.c$, and furthermore we obtain $l^{l\max}, ld^{l\max}$ as the

Algorithm 4 Composing Execution and Memory Times

1: **function** ComposeTimes(numH, NPS, E, UL, LD)
2: $HNP = \sum_{\forall s \in NPS} s.l$
3: sort E, UL, LD in non-increasing order
4: **for all** $j = 1...\text{num}H - |NPS|$ **do**
5: $DMA_j = \delta + \rho \cdot (UL_j + LD_j)$
6: $M = DMA \cup E$ in non-increasing order
7: $HTP = \sum_{j=1}^{\text{num}H-|NPS|} M_j$
8: **return** $HNP + HTP$

maximum $s.l, s.ld$ for any segment of lower priority, and $ul^{\max}$ as the maximum $s.ul$ for any segment, then $\max(l^{l\max}, \delta + \rho \cdot (ul^{\max} + ld^{l\max}))$ is an upper bound to the interval length for both cases. □

We next discuss how to compute the length H_i of the remaining numH_i intervals. Due to the complexity of the analysis, we shall do it in steps. Let us start by assuming that the following four multisets are given: the multiset NPS containing all np-streaming segments executed in H_i; the multiset E containing the computation time of all terminal and p-streaming segments in H_i; the multisets LD, UL containing the time of all load and unload phases in H_i. Note that for a multiset A, we use $|A|$ to denote its cardinality, and A_j with $j = 1...|A|$ to denote its $j-th$ element (with repetitions). Function ComposeTimes in Algorithm 4 then bounds H_i based on NPS, E, LD and UL, and the number of intervals numH_i. The function bounds the cumulative length of the $|NPS|$ intervals executing np-streaming segments in line 2 based on Observation 7. To determine the length of the remaining $NH_i - |NPS|$ intervals, the function proceeds similarly to Algorithm 2 in [151]: the length is maximized without making any assumption on the order in which the various segments and memory phases are executed. First, multisets E, LD and UL are sorted in non-increasing order. Then, LD and UL are combined to determine a new multiset DMA of memory times based on the DMA parameters δ, ρ. Finally, based on Observation 7, the lengths of the intervals are obtained by taking the maximum values in either E or DMA.

Lemma 9. *Given multisets NPS, E, UL, LD and number of intervals numH_i, Algorithm 4 computes a valid upper bound to H_i.*

Proof. We show that the algorithm computes H_i as the sum of upper bounds to the length of each interval in H_i executing an np-streaming segment, and an upper bound to the

cumulative length of intervals executing either terminal or p-streaming segments; hence, the computed value of H_i must be a valid upper bound.

NP-streaming intervals: by Observation 7, the length of an interval executing an np-streaming segment is bounded by the length of the segment. Hence, $HNP = \sum_{\forall s \in NPS} s.l$ is an upper bound to the cumulative length of all such intervals.

Terminal and p-streaming intervals: by definition of NPS and num$H$, the number of such intervals is num$H - |NPS|$. By Observation 6, the length of each interval is bounded by $\max(c, dma)$, where $c \in E$ is the execution time of a segment, and $dma = \delta + \rho \cdot (ul + ld)$ is the memory time, where $ul \in UL$ and $ld \in LD$ are load / unload phase lengths.

On line 7, the algorithm computes the cumulative length of the intervals by selecting the maximum num$H - |NPS|$ values out of all $c \in E$, and values in set DMA. If we can thus show that the values in DMA represent upper bounds to the lengths of the memory times $dma = \delta + \rho \cdot (ul + ld)$, the cumulative length computed by the algorithm must be an upper bound. Assume that the maximum cumulative length is found by selecting k elements in E and k' in DMA, with $k + k' =$ num$H - |NPS|$. We then have to show that $\sum_{j=1}^{k'} DMA_j$ in indeed an upper bound to the cumulative length of k' memory times: $\sum_{j=1}^{k'} dma_j$. We can rewrite the cumulative memory time as: $\sum_{j=1}^{k'} dma_j = \sum_{j=1}^{k'} \left(\delta + \rho \cdot (ul_j + ld_j)\right) = k' \cdot \delta + \rho \cdot (\sum_{j=1}^{k'} ul_j + \sum_{j=1}^{k'} ld_j)$, where ul_j and ld_j are some unload and load phases in UL and LD. But by construction at lines 3 and 5 of the algorithm, we must have $\sum_{j=1}^{k'} DMA_j = k' \cdot \delta + \rho \cdot (\sum_{j=1}^{k'} UL_j + \sum_{j=1}^{k'} LD_j)$ where $\sum_{j=1}^{k'} UL_j$ is the maximum sum of any k' loads in UL, and $\sum_{j=1}^{k'} LD_j$ is the maximum sum of any k' loads in LD. Hence, $\sum_{j=1}^{k'} DMA_j$ upper bounds $\sum_{j=1}^{k'} dma_j$, concluding the proof. □

We also state some further properties of Algorithm 4 which will be helpful later on.

Observation 10. *Adding an extra element to E, LD or UL, or increasing the value of an element in either E, LD or UL, cannot decrease the result of Algorithm 4.*

Lemma 11. *Removing a segment s from NPS and adding an execution time equal to $s.l$ to E cannot decrease the result of Algorithm 4.*

Proof. Removing s from NPS results in decreasing HNP by $s.l$ and increasing num$H - |NPS|$ by one. Since the algorithm computes HTP by summing the num$H - |NPS|$ highest values in E and DMA, and we added a value $s.l$ to E, it follows that HTP must increase by at least $s.l$. Hence, the returned value $HNP + HTP$ cannot decrease. □

Lemma 12. *Consider two segments $s', s'' \in NPS$ with $s'.l \leq s''.l$. Let H' be the result of Algorithm 4 after removing s' from NPS and adding $s'.l$ to E, and H'' be the result after removing s'' from NPS and adding s'' to E. Then $H' \geq H''$.*

Proof. Let HNP, HTP denote the values originally computed by Algorithm 4 at lines 2, 7, and let HNP', HTP' (HNP'', HTP'') be the corresponding values computed after moving s' from NPS to E (respectively, after moving s''). Then we have $HNP' = HNP - s'.l, HNP'' = HNP - s''.l$, and $HTP' = HTP + e', HTP'' = HTP + e''$, where e' (e'') is the extra element summed to HTP' (respectively, HTP'') compared to HTP, due to the fact that num$H - |NPS|$ increases by one after removing s' (s'').

Since in line 7 the algorithm picks the largest num$H - |NPS|$ values out of E or DMA, it follows that $e' \geq s'$, and if $e' \geq s''.l$, then $e'' = e'$, otherwise $e'' = s''.l$. We consider both cases. 1) Case $e' \geq s''.l$: since $s'.l \leq s''.l$ and $e'' = e'$, we have $HNP' + HTP' = HNP - s'.l + HTP + e' \geq HNP - s''.l + HTP + e'' = HNP'' + HTP''$, proving the lemma. 2) Case $e' < s''.l$: since $e' \geq s'.l$ and $e'' = s''.l$, we have $HNP' + HTP' = HNP - s'.l + HTP + e' \geq HNP + HTP = HNP - s''.l + HTP + e'' = HNP'' + HTP''$, again proving the lemma. □

We next discuss how to compute the multisets NPS, E, UL, LD in a safe manner. The key complexity is how to divide the streaming segments into p-streaming and np-streaming, since preemptions are a function of the schedule. We thus use the following idea: we start by putting all streaming segments into NPS, and determining the maximum number of p-streaming segments. Then, we remove such number of segments from NPS and add their corresponding lengths to E based on Lemmas 11 and 12. To determine the number of p-streaming segments, we reason about preemption in H_i.

Observation 13. *Based on Rules 2, 4 under fixed priorities, the scheduler will continue executing segments of the same job until it encounters a terminal segment, or the job is preempted by a higher priority job. Hence, the number of preemptions caused by a job with maximal path P to lower priority jobs is bounded by $P.I$.*

The number of preemptions for the highest priority task τ_1 is obviously 0. Based on Observation 13, the maximum number of preemptions numP_j, and thus p-streaming segments, suffered by jobs of tasks $\tau_2, ..., \tau_j$ for any $2 \leq j \leq N$ is bounded by the number of terminal segments of jobs of tasks $\tau_1, ..., \tau_{j-1}$:

$$\text{num}P_j = \sum_{\forall 1 \leq k < j, \forall P: P \in \mathcal{P} \wedge P \in G_k} P.I. \tag{5.24}$$

Based on the constraints expressed by Equation 5.24, Function ComputeH in Algorithm 5 first determines the multisets NPS, E, UL and UD, and then invokes ComposeTimes to compute H_i. The algorithm begins by considering the initial blocking interval ② in line 4, which includes the execution of a segment of a lower priority task. The interval also includes an unload of a previous task, and the load of s^{begin} for either τ_i or a higher priority job, which will be added later on. Finally, we have to consider the unload of the segment executed in interval ②, which will happen within H_i (in interval ③ in Figure 5.6). Then, at line 5 the algorithm adds to E, LD and UD computation, load and unload phases for the other $P_i.I - 1$ blocking intervals, selecting the maximum such values among all lower priority tasks. The algorithm then adds to NPS all streaming segments of τ_i and higher priority tasks; and based on Observation 6, it adds to E all executions of terminal segments, to UL all unload phases of terminal segments (except the last segment s_i^{end} of τ_i, since such unload is executed outside H_i), and to LD all load phases of initial segments. Finally, based on the preemption constraints, the algorithm adds load phases of non-initial segments to LD (as non-initial segments that follow a p-streaming segment, based on Observation 6); unload phases of streaming segments to UL (as p-streaming segments, again based on Observation 6); and finally moves streaming segments from NPS to E.

Lemma 14. *Algorithm 5 compute a valid upper bound H_i based on paths $P_i, \mathcal{P}$.*

Proof. Since a segment of τ_i cannot be executed immediately after a terminal segment of τ_i, in the worst case each terminal segment of τ_i can induce an interval of a lower priority task. Since H_i does not include the execution of the last segment s_i^{end}, the number of such blocking intervals is bounded by $P_i.I - 1$; in addition, there is the initial blocking interval ②. Hence, the value of numH_i is by construction a valid upper bound on the number of intervals in H_i.

We will next show that the way we construct the multisets E, UL, LD, and NPS cannot result in a lower value of ComposeTimes(numH_i, NPS, E, UL, LD) compared to the value that ComposeTimes would compute given the actual multisets for any valid schedule. Since furthermore by Lemma 9 the result of ComposeTimes upper bounds H_i, this will complete the proof. We consider two types of values / segments in E, UL, LD, NPS: those belonging to segments in P_i and $\mathcal{P}$ (that is, the job under analysis and higher priority jobs) and those belonging to other jobs.

We start with the latter, which must belong to segments executed in a blocking interval. As noted above, such intervals include the initial blocking interval ②, and $P_i.I - 1$ further intervals. All such intervals must be terminal or p-streaming, since they are followed by a

Algorithm 5 Computing H_i

1: **function** ComputeH($\Gamma, i, P_i, \mathcal{P}$)
2: Compute numH_i based on Equation 5.17
3: $NPS = E = LD = UL = \emptyset$
4: add $c^{l\max}$ to E, add $ul^{\max}$ and $ul^{l\max}$ to UL
5: for $P_i.I - 1$ times: add $c^{l\max}$ to E, add $ld^{l\max}$ to LD, add $ul^{l\max}$ to UL
6: $\forall P \in \mathcal{P} \cup P_i, \forall$ streaming $s \in P$: add s to NPS
7: $\forall P \in \mathcal{P} \cup P_i, \forall$ terminal $s, s \in P \wedge s \neq s_i^{end}$: add $s.c$ to E and $s.ul$ to UL
8: $\forall P \in \mathcal{P} \cup P_i, \forall$ initial $s \in P$: add $s.ld$ to LD
9: select the maximum possible number of non-initial segments $s \in \mathcal{P} \cup P_i$ with maximum value of $s.ld$ based on the constraints numP_2...numP_i from Equation 5.24 and add such $s.ld$ values to LD
10: select the maximum possible number of segments $s \in NPS$ with maximum value of $s.ul$ based on the constraints numP_2...numP_i from Equation 5.24 and add such $s.ul$ values to UL
11: select the maximum possible number of segments $s \in NPS$ with minimum value of $s.l$ based on the constraints numP_2...numP_i from Equation 5.24; remove each selected segment s from NPS and add $s.l$ to E
12: **return** ComposeTimes(numH_i, NPS, E, UL, LD)

segment of τ_i or a higher priority job. We considering E. Since the algorithm adds $P_i.I$ times to E the maximum computation of any lower priority task, by Observation 10, this cannot decrease the result of ComposeTimes. We next consider UL. In interval ②, an unload can be performed for a previous segment (note that if the segment executed in interval ① is np-streaming, this unload would not belong to such segment); the algorithm adds to UL the longest unload of any task. Both interval ② and each of the other $P_i.I - 1$ blocking intervals can further induce an unload for a lower priority segment; since again the algorithm adds to UL the longest unload of any lower priority task for each such interval, by Observation 10 this cannot decrease ComposeTimes. Similarly for LD, $P_i.I - 1$ blocking intervals can induce a load of a lower priority segment (note that the load of interval ② is performed in interval ① outside of H_i), and for each such interval, the algorithm adds the longest load of any lower priority segment.

We next consider values / segments in E, UL, LD, NPS for jobs of P_i and $\mathcal{P}$. We first discuss loads and unloads. Based on Observation 6, the set of loads includes all initial segments, and all (non-initial) segments that follow a p-streaming segment. The algorithm inserts all loads of initial segments in LD at line 8. The number of p-streaming

segments is bounded by the number of preemptions, and hence the constraints expressed by Equation 5.24; and based on the constraints, the algorithm selects the largest load phases and adds them to LD. In summary, the algorithm adds to LD a number of load phases that is larger or equal than the number of segments that follow a p-streaming segment in any valid schedule; and the values added to LD are larger or equal than the length of the load phases of segments that follow a p-streaming segment. Hence, by Observation 10, this cannot decrease ComposeTimes. The same argument applies for unloads, where the algorithm first adds to UL all unload phases of terminal segments (with the exception of s_i^{end}, since its unload is performed outside H_i), then maximizes the number and lengths and unload phases of p-streaming segments. It remains to discuss E and NPS. The algorithm first adds to E all terminal segments (again, except s_i^{end}), and adds to NPS all streaming segments. To obtain the actual E and NPS for an valid schedule, each p-streaming segment s must be removed from NPS and its computation time $s.c$ added to E. The algorithm moves segments from NPS to E at line 11. Note that for each selected segment, the algorithm adds to E the length $s.l$ of the interval, rather than its computation time $s.c$; however, since $s.l \geq s.c$, this is safe by Observation 10. The number of moved segments is bound by the constraints expressed by Equation 5.24; hence the algorithm moves a number of segments that is higher or equal than the actual number of p-streaming segments in any schedule. By Lemma 11, moving more segments than the actual number cannot decrease ComposeTimes. Finally, the algorithm selects the segments with the minimum length; hence, the algorithm might move a segment s' instead of a segment s'' with $s'.l \leq s''.l$, while in the actual schedule s' was np-streaming and s'' was p-streaming. However, by Lemma 12, again this cannot decrease ComposeTimes. This concludes the proof. □

Based on Algorithm 5 and Lemma 8, the response time of τ_i can then be computed based on the iteration:

$$R_i(P_i) = B_i + H_i\big(R_i(P_i)\big), \tag{5.25}$$

where at each step of the iteration, $R_i(P_i)$ is used to obtain $\text{numInter}\big(R_i(P_i)\big)$, and H_i is computed as the maximum value of $\text{ComposeTimes}(\Gamma, i, P_i, \mathcal{P})$ for all possible path sets $\mathcal{P}$. If the number of maximal paths for each task is sufficiently small, then such an approach can be computationally feasible; this is indeed the case for the benchmarks used in our evaluation in Section 6.4. Otherwise, we argue that one could still use the analysis after reducing the DAG G_j for each higher priority task τ_j to a single path P that is worse than all paths originally in G_j; intuitively, this could be performed by taking the maximum number of terminal, streaming and initial segments over any path in G_j, and maximizing their length and load / unload phases. We reserve a formal description of the reduction procedure as part of our future work.

5.6 Summary

In this chapter, we reviewed the background and the related work for the 3-phase model on which we base our segmentation approach in the next chapter. We identified the limitations with the model and proposed an extension to a new multi-segment conditional streaming model and detailed the schedulability analysis based on fixed and variable-size DMA model. We also presented an OS-interface to realize our new model. In the next chapter, we utilize our framework proposed in Chapter 2 and the schedulability analysis to produce automated segmentation algorithms using the new 3-phase model.

Chapter 6

Program Segmentation

In this chapter, we show how a task is compiled into segments. A program segmentation represents a partition of the regions of the program into a set of segments. We start by discussing the concept of a valid segmentation in Section 6.1: intuitively, the segmentation must obey a set of constraints deriving from the code structure, and furthermore, the code and data of each segment must fit in the available SPM space. We also add a length constraint, which forces the length of all segments to be no larger than a provided value $l^{\max}$. This is done to limit the maximum amount of blocking time suffered by higher priority tasks, as detailed in the schedulability analyses in Sections 5.4 and 5.5.

Based on the concept of valid segmentation, we then introduce algorithms to segment a task in the fixed-size and variable-size DMA models in Sections 6.2 and 6.3. In particular, for the fixed-size case, we show that we can segment a task set in an optimal manner; meaning that if there exists a set of valid segmentations that result in a schedulable task set according to the analysis in Section 5.4, then our algorithm will find one such set. For the more complex variable-size case, we are not able to find an optimal algorithm. Hence, we instead propose a heuristic segmentation algorithm. We evaluate both algorithms in terms of schedulability on synthetic task sets based on actual benchmarks in Section 6.4. Results show that both approaches greatly outperform a naive, greedy segmentation approach, with limited loss of schedulability compared to an ideal (non implementable) scheme which does not suffer from any overhead or constraints.

6.1 Valid Segmentation

Program segmentation is the process of assigning each part of the program code to a segment. In this work, we rely on the refined region structure discussed in Section 2.2; hence we restrict the parts of the program that can be assigned to a segment to be a region or a sequence of regions. Note that 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 as discussed in Chapter 2.

A segmentation is valid if it satisfies the *footprint constraint*, the (optional) *length constraint* and the *compilation constraints*. The footprint constraint for a terminal segment states that the footprint of the segment, i.e. the code and data of regions assigned to the segment must fit in the available SPM size. For a streaming segment, the footprint constraint implies that the union of the code and data of the segment as well as the next segment that it streams into must fit the available SPM size. The length constraint states that the length of each segment must be at most $l^{\max}$; 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 $s.l = \max(\Delta, (+t_{seg})t_r + t_{seg})$. If multiple regions in sequence are assigned to a segment s, then $s.l = \max(\Delta, (+t_{seg})(\sum_r t_r) + t_{seg})$. We further assume that the regions' WCETs satisfy the following property, which we argue is required for the WCET values to be sound:

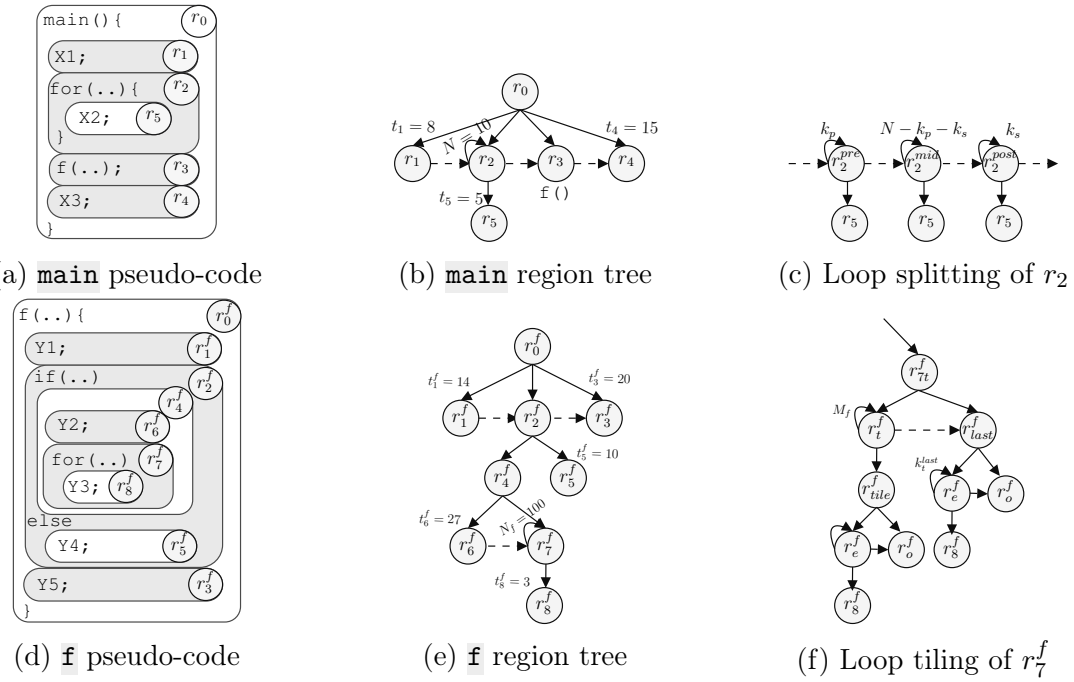
Property 4. *If r is a conditional region, then t_r is equal to the WCET of its longer children. If r is a sequential region or tiled loop, then its WCET is less than or equal to the sum of the WCETs of its children or tiles.*

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. 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.
- All the segments in the program are terminal segments except for tiled loops which can be streaming or non-streaming. A non-streaming loop comprises of terminal segments only while a streaming loop has a set of streaming segments that are ended with a terminal segment.

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) `main` pseudo-code (b) `main` region tree (c) Loop splitting of r_2

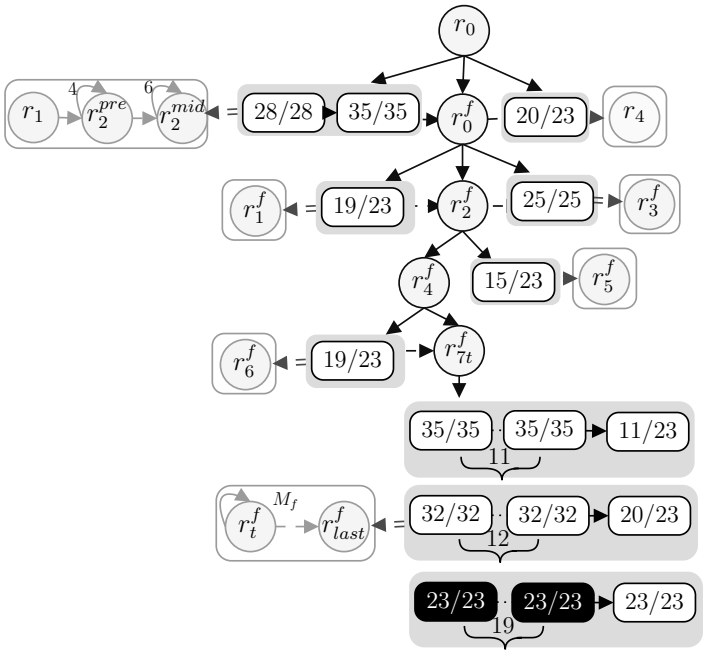
(d) `f` pseudo-code (e) `f` region tree (f) Loop tiling of r_7^f

Figure 6.1: Region representation ($\rightarrow \equiv$ parent-child / $\dashrightarrow \equiv$ sequential regions)

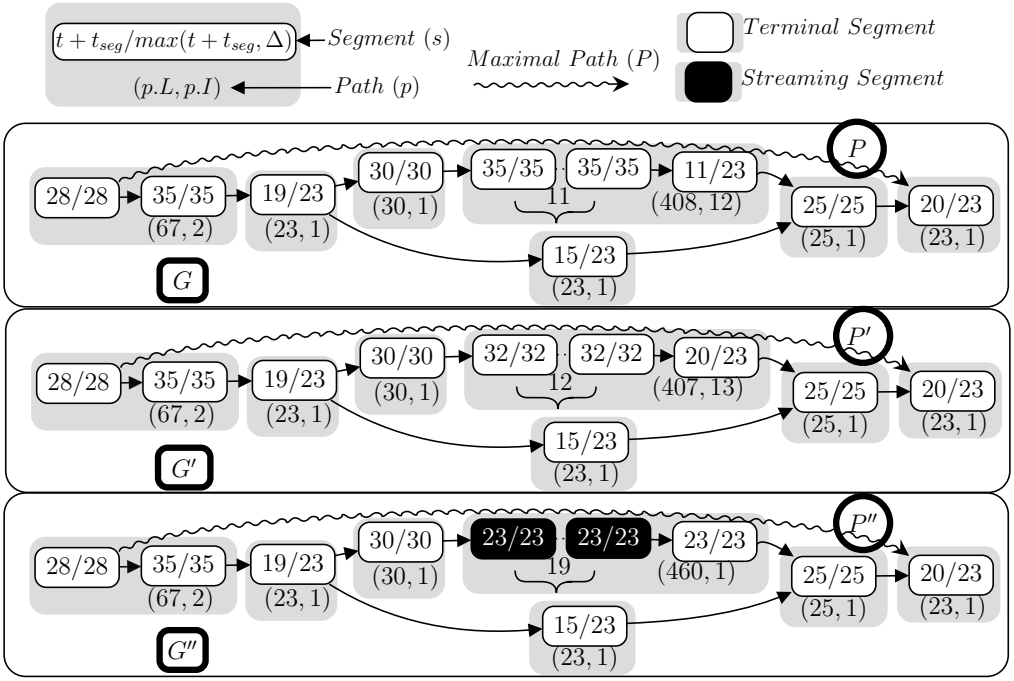
A segmented tree $\mathcal{T}$ implicitly generates a set $\mathcal{G}$ of segment DAGs: each DAG in $\mathcal{G}$ is constructed by taking one path out of each path set and joining them according to the segmented tree hierarchy. A maximal path in the DAG thus comprises a sequence of paths $\{p_1, p_2, ..., p_n\}$ 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.

6.1.1 Segmentation Example

We now show an example of valid segmentation for the program in Figure 6.1; which consists of two functions `main()` and `f()`. For `main`, the pseudo-code is shown in Figure 6.1a and the regions in Figure 6.1b. 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 6.1d. Figure 6.1d is the pseudo-code for `f` with its region tree



(a) Segmented Tree



(b) DAGs

Figure 6.2: Segmentation Example

in Figure 6.1e. Region 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. The true path has two regions in sequence, r_6^f and the loop r_7^f.

Let the maximum segment length be $l_{max} = 35$, the segmentation overhead $t_{seg} = 5$, and the tiling overhead $t_{tiling} = 3$. We assume for this example that the footprint constraint is satisfied for all regions except the loop r_7^f which can satisfy the footprint constraint if the tile size ≤ 10 iterations if the loop is not streaming or if the tile size ≤ 5 iterations if the loop is streaming. Given the times for each basic block t in Figure 6.1b and Figure 6.1d, regions $\{r_1, r_4, r_1^f, r_3^f, r_5^f, r_6^f, r_8^f\}$ are mergeable regions. Loop regions $\{r_2, r_7^f\}$ are splittable and tileable, but not mergeable. Figure 6.2a shows the segmented tree of the program which represents three possible DAGs.

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 6.2a, 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. 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 has two regions: r_6^f and the loop region r_f^7 with 100 iterations. We assume that the loop is only tiled but not split; and hence r_6^f is wrapped solely in a segment. There are many possible tiling options for r_f^7 that would satisfy the max segment length. We choose two possible non-streaming tilings and one streaming tiling as following:

- With tile size $k_t = 9$, there are 11 terminal segments that are complete tiles with computation time $9 * 3 + t_{tiling} + t_{seg} = 35$. The last terminal segment is the last tile $k_t^{last} = 100 - 11 * 9 = 1$ with computation time $1 * 3 + t_{tiling} + t_{seg} = 11$.
- With tile size $k_t = 8$, there are 12 terminal segments that are complete tiles with computation time $8 * 3 + t_{tiling} + t_{seg} = 32$. The last terminal segment is the last tile $k_t^{last} = 100 - 12 * 8 = 4$ with computation time $4 * 3 + t_{tiling} + t_{seg} = 20$.
- With tile size $k_t = 5$, there are 19 streaming segments that are complete tiles with computation time $5 * 3 + t_{tiling} + t_{seg} = 23$. The last terminal segment is the last tile $k_t^{last} = 100 - 19 * 5 = 5$ with computation time $5 * 3 + t_{tiling} + t_{seg} = 23$.

Figure 6.2b shows the three DAGs that result from the segmented tree. In the figure, we show the segment computation time as well as the segment length for a fixed DMA slot

$\Delta = 23$. The three DAGs G, G', G'' have a dominant maximal paths P, P', P'' with lengths $L = 576, L' = 575, L'' = 628$ and number of terminal segments $I = 18, I' = 19, I'' = 7$ respectively.

6.2 Segmentation for the Fixed-size DMA Model

In this section, we show how to produce an optimal segmentation for a task set Γ, assuming the fixed-size DMA model. Based on Property 3 in Section 5.4, we first present an algorithm that explores the set of all valid DAGs for a program, but quickly cuts dominating (i.e., worse) DAGs. Then, in Section 6.2.3 we show that, based on Property 2, we can invoke the algorithm on each task in priority order and obtain a set of DAGs (one for each task) that is optimal from a schedulability perspective.

The example in Section 6.1 shows that different segmentation decisions can result in incomparable maximal paths according to Definition 3 as in Figure 6.2b: for the path P, we have $P.L = 576$, $P.I = 18$ and $P.end = 23$; while for the path P', we have $P'.L = 575$, $P'.I = 18$ and $P'.end = 23$; finally for the path P'', we have $P''.L = 628$, $P''.I = 7$ and $P'.end = 23$. Since a DAG generated from the segmented tree $\mathcal{T}$ includes either P, P' or P'', the resulting three DAGs G, G' and G'' are also incomparable. This means that without considering the other tasks in the system, we cannot determine whether G, 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 all three DAGs. On the other hand, if for example, $G' \succeq G$, we can safely ignore G' based on Property 3. This is formally captured by the following definition.

Definition 15. *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 proceed 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 6 that preserves optimality according to Definition 4 based on the constraints in Section 6.1, but removes dominating DAGs without enumerating $\mathcal{G}$; instead, the algorithm explores the segmented tree recursively and removes unneeded

Algorithm 6 Segmentation Algorithm

```
1: function SegmentTask(τ)
2:     if r_0 is mergeable then
3:         Create DAG G with a single segment comprising r_0, return 𝒢 = {G}
4:     Generate DAG set 𝒢 from 𝒯 = Segment(r_0), return 𝒢
5: function Segment(r)
6:     Initialize R = ∅                                  ▷ A set of sequential regions.
7:     Initialize 𝒯 to be the subtree whose root is r
8:     for all r_c ∈ 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 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 ≠ ∅, replace regions in R with SegmentSequence(R)
22:    return 𝒯
```

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 6 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 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 6 depends on SegmentSequence and Tile, we first state a key property of both functions, which will be implemented in Algorithms 7 and 8. Since the functions return a path set $\mathcal{P}$, we begin by defining a concept of domination among paths and path sets.

Definition 16. *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 16 is similar to Definition 3 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 17. *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 3 we have $P' \succeq P$. □

Definition 18. *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 22 that Algorithm 6 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 19. *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 has a single segment, it must hold $p.I \leq 1$. On the other hand, since by compilation constraint only tiled loops can be streamed and must finish with a terminal segment, it must hold $p'.I \geq 1$, and hence we have $p'.I \geq p.I$. Based on Property 4, there must also exist one path p' with $p'.L \geq p.L$. By Lemma 17, 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 20. *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 19, 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 15, this means that generating DAGs from $\mathcal{T}$ preserves optimality. □

Lemma 20 shows that to preserve optimality, it is sufficient to return a single segmented tree with maximal region sequences, which is what Algorithm 6 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 21. *Consider a segmented tree $\mathcal{T}$ as in Lemma 20. 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 17, 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 20 $\mathcal{T}$ preserves optimality, it thus follows that the DAG set generated from $\overline{\mathcal{T}}$ also preserves optimality according to Definition 15. □

Theorem 22. *If Properties 5, 6 hold, Algorithm 6 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 $\mathcal{P}'$ 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 21, in both cases $\mathcal{P} \subseteq \mathcal{P}'$ and $\mathcal{P}' \succeq \mathcal{P}$ hold. In summary, Lemma 21 applies to all maximal regions, hence the algorithm preserves optimality. □

6.2.1 Tiling Algorithm

In this section, we discuss our Algorithm 7 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.

As discussed in Section 2.3, a 2-level tiling results in the tiles in Figure 2.9 with four tile timings: 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 such 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:

$$(M_1 + 1)(M_2 + 1), \tag{6.1}$$

and length:

$$\begin{aligned} p.L = {} & M_2 * M_1 * \max(\Delta, t_1^2 + t_{seg}) \\ & + M_2 * \max(\Delta, t_{1l}^2 + t_{seg}) \\ & + M_1 * \max(\Delta, t_1^{2l} + t_{seg}) \\ & + \max(\Delta, t_{1l}^{2l} + t_{seg}). \end{aligned} \tag{6.2}$$

For a streaming loop, $p.I = 1$ as only the last segment is terminal. For a non-streaming loop, all segments are terminal, hence $p.I = (M_1 + 1)(M_2 + 1)$. We can next rewrite the length as $p.L = t_{loop} + t_{overhead} + t_\Delta$ such that t_{loop} is the original loop time and does not depend on the tile size, $t_{overhead}$ is the tiling and segmentation overhead, and t_Δ is the total segment under-utilization:

$$t_{loop} = N_2 * (N_1 * t_1 + t_2), \tag{6.3}$$

$$t_{overhead} = N_2 * M_1 * t_2 + (M_2 + 1) * \big((M_1 + 1) * (t_{tile}^1 + t_{seg}) + t_{tile}^2\big), \tag{6.4}$$

$$\begin{aligned} t_\Delta = {} & M_1 * M_2 * \max(\Delta - (t_1^2 + t_{seg}), 0) + \\ & + M_2 * \max(\Delta - (t_{1l}^2 + t_{seg}), 0) \\ & + M_1 * \max(\Delta - (t_1^{2l} + t_{seg}), 0) \\ & + \max(\Delta - (t_{1l}^{2l} + t_{seg}), 0). \end{aligned} \tag{6.5}$$

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

Algorithm 7 2-Level Tiling

```
1: function TILE(r)
2:     P = TILELOOP(r, ∅, 0)
3:     P = TILELOOP(r, P, 1)
4:     return P
5: function TILELOOP(r, P, stream)
6:     Compute k_2^max based on stream
7:     for all k_2 ≤ k_2^max do
8:         Compute k_1^max(k_2) based on stream
9:         k_1^Δ(k_2) = max{k_1 | t_1^2 + t_tile^1 + t_seg ≤ Δ}
10:        k_1 = k_1^max, t̂_Δ = ∞, t̂_overhead = 0
11:        repeat
12:            Generate p(k_2, k_1) based on stream
13:            if p(k_2, k_1) is valid based on stream then
14:                Add p(k_2, k_1) to P
15:                Compute t_overhead, t_Δ based on Equations 6.4, 6.5
16:                if t_Δ < t̂_Δ then
17:                    t̂_Δ = t_Δ, t̂_overhead = t_overhead
18:            k_1 = k_1 − 1
19:        until k_1 = k_1^Δ or t̂_Δ = 0 or t_overhead ≥ t̂_overhead + t̂_Δ
20:    Filter P by removing dominating paths based on Definition 16
21:    return P
```

Algorithm 7 takes as input a region r and returns a set of valid paths $\mathcal{P}$ for r. The function TILE calls TILELOOP twice for the case of non-streaming loop ($stream = 0$) and the case of streaming loop ($stream = 1$). The final path set is the union of the returned paths of both cases. Note that the choice to create a streaming or non-streaming loop affects the footprint of the generated segments; hence, the value of $stream$ must be considered in TILELOOP every time the footprint constraint is evaluated. Function TILELOOP 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 $s.\mathcal{D} \leq \mathcal{D}_{SPM}$. 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 23, 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 the 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 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 7. We prove in Lemma 25 that the algorithm preserves Property 6.

Lemma 23. *All segments in a path $p(k_2, k_1)$ with $k_1 \leq k_1^{\Delta}(k_2)$ have length Δ.*

Proof. Note that based on the tiling formulation in Section 2.3, t_1^2 is increasing in k_1. Hence, by definition of $k_1^{\Delta}(k_2)$, it must hold for k_1: $t_1^2 + t_{tile}^1 + t_{seg} \leq \Delta$. By definition, we also have $k_1^l \leq k_1$ and $k_2^l \leq k_2$. 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 under the fixed-size DMA assumption, 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 24. *Consider two valid solutions (k_2, k_1) with overhead $t_{overhead}$ and (k_2, k_1') with overhead $t'_{overhead}$; assume that both solutions are of the same type (either both streaming, or both non-streaming). If $k_1' \leq k_1$ then $p(k_2, k_1').I \geq p(k_2, k_1).I$, $t'_{overhead} \geq t_{overhead}$, and the number of segments in $p(k_2, k_1')$ is larger than or equal to the one in $p(k_2, k_1)$*

Proof. The properties for $t_{overhead}$ and the number of segments follow directly by noticing

that both $t_{overhead}$ in Equation 6.4 and the number of segments in Equation 6.1 depend on M_1, which is non-increasing in k_1. For what concerns the number of terminal segments, note that if both solutions are streaming, then $p(k_2, k_1').I = p(k_2, k_1).I = 1$, while if both solutions are streaming, then the number of terminal segments is equal to the number of segments and hence $p(k_2, k_1').I \geq p(k_2, k_1).I$. □

Lemma 25. *Property 6 holds for Algorithm 7.*

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 streaming paths $p(k_2, k_1)$ such that $k_2 \leq k_2^{max}$ and $k_1 \leq k_1^{max}(k_2)$ (where k_2^{max} and $k_1^{max}(k_2)$ are computed based on the footprint for streaming segments), and all valid non-streaming paths $p(k_2, k_1)$ such that $k_2 \leq k_2^{max}$ and $k_1 \leq k_1^{max}(k_2)$ (where k_2^{max} and $k_1^{max}(k_2)$ are computed based on the footprint for non-streaming segments).

For a given value of k_2, define $\bar{k}_1$ as the value of k_1 for which the algorithm breaks 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 $p(k_2, k_1') \succeq p(k_2, k_1)$. Since furthermore the filtering on line 20 based on Definition 16 respects Definition 18 (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 19 evaluates to true. Note that we have $k_1 < \bar{k}_1 \leq \hat{k}_1$. 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 17), 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 23, all segments in both $p(k_2, k_1^\Delta)$ and $p(k_2, k_1')$ have length Δ. Given $k_1' < \bar{k}_1$, by Lemma 24 the number of segments in $p(k_2, k_1')$ is larger than or equal to the number of segments in $p(k_2, k_1^\Delta)$; hence, $p(k_2, k_1').L \geq p(k_2, k_1^\Delta).L$. Again by $k_1' < \bar{k}_1$ and Lemma 24, we also have $p(k_2, k_1').I \geq p(k_2, k_1^\Delta).I$. Hence, $p(k_2, k_1') \succeq p(k_2, k_1^\Delta)$.
- If $\hat{t}_\Delta = 0$, then it must hold $t'_\Delta \geq \hat{t}_\Delta$. By Lemma 24, we also have $t'_{overhead} \geq \hat{t}_{overhead}$; therefore, $p(k_2, k_1').L \geq p(k_2, \hat{k}_1).L$. Also by Lemma 24 we have $p(k_2, k_1').I \geq p(k_2, \hat{k}_1).I$. Therefore $p(k_2, k_1') \succeq p(k_2, \hat{k}_1)$.
- If $t_{overhead} \geq \hat{t}_{overhead} + \hat{t}_\Delta$, then $t'_{overhead} + t'_\Delta \geq t_{overhead} \geq \hat{t}_{overhead} + \hat{t}_\Delta$; this implies $p(k_2, k_1').L \geq p(k_2, \hat{k}_1).L$. Since again by Lemma 24 we have $p(k_2, k_1').I \geq p(k_2, \hat{k}_1).I$, it holds $p(k_2, k_1') \succeq p(k_2, \hat{k}_1)$.

□

6.2.2 Region Sequence Segmentation

Next, we consider Algorithm 8 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 $\bar{p}_n.t_{end} = t_r$, and $\bar{p}_n.L = \bar{p}.L + \bar{p}_n.end$. Furthermore, if r is a streaming tile, then $\bar{p}_n.I = \bar{p}.I$, otherwise $\bar{p}_n.I = \bar{p}.I + 1$. Note that $\bar{p}_n$ is always valid, since r is mergeable (or a tile).

2. Add r to the last segment of $\bar{p}$, resulting in a new partial path $\bar{p}_m$. Note that $\bar{p}_m$ might not be a valid path according to the constraints; in particular, tiles cannot be merged with other regions. Hence, it is only added to the new set of partial paths if valid. We then have $\bar{p}_m.I = \bar{p}.I$, $\bar{p}_m.t_{end} = \bar{p}.t_{end} + t_r$, and $\bar{p}_m.L = \bar{p}.L - \bar{p}.end + \bar{p}_m.end$.

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 26. *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 $p' \succeq p$ 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 $\bar{p}' \succeq \bar{p}$. 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 $(\bar{p}_m, \bar{p}_n)$ / $(\bar{p}'_m, \bar{p}'_n)$ 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 6.6 holds:

$$\bar{p}'_m.L - \bar{p}'_m.end \geq \bar{p}_m.L - \bar{p}_m.end \tag{6.6}$$

  And since $\bar{p}'.I \geq \bar{p}.I$, then Equation 6.7 holds:

$$\bar{p}'_m.I \geq \bar{p}_m.I \tag{6.7}$$

  And since $\bar{p}'.t_{end} \geq \bar{p}.t_{end}$, then by adding t_r to both sides Equation 6.8 holds:

$$\bar{p}'_m.t_{end} \geq \bar{p}_m.t_{end} \tag{6.8}$$

  Since Equations 6.6, 6.7 and 6.8 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 6.9 holds:

$$\bar{p}'_n.L - \bar{p}'_n.end \geq \bar{p}_n.L - \bar{p}_n.end \tag{6.9}$$

  And since $\bar{p}'.I \geq \bar{p}.I$, then Equation 6.10 holds:

$$\bar{p}'_n.I \geq \bar{p}_n.I \tag{6.10}$$

  And since $\bar{p}'_n.t_{end} = \bar{p}_n.t_{end}$ and Equations 6.9 and 6.10 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 6.7 holds.
     Since $\bar{p}'.t_{end} \geq \bar{p}.t_{end} > \Delta - t_{seg}$, Equation 6.8 holds. Since $\bar{p}'.t_{end} \geq \bar{p}.t_{end} > \Delta - t_{seg}$, then Equations 6.11 and 6.12 hold:

$$\bar{p}'_m.end - \bar{p}'.end = t_r \tag{6.11}$$

$$\bar{p}_m.end - \bar{p}.end = t_r \tag{6.12}$$

From Equations 6.11 and 6.12 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 6.13 holds:

$$\bar{p}'_m.L \geq \bar{p}_m.L \tag{6.13}$$

Since Equations 6.7, 6.8 and 6.13 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 6.10 holds. And since $\bar{p}'.L \geq \bar{p}.L$, then Equation 6.9 holds. Since $\bar{p}'_n.t_{end} = \bar{p}_n.t_{end}$ and Equations 6.9 and 6.10 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 6.14
     $$\bar{p}'_m.L \geq \bar{p}_n.L \tag{6.14}$$
     Since $\bar{p}'.I > \bar{p}.I$, then Equation 6.15 holds:
     $$\bar{p}'_m.I \geq \bar{p}_n.I \tag{6.15}$$
     And since, $\bar{p}'.t_{end} + t_r > t_r$, then Equation 6.16 holds:
     $$\bar{p}'_m.t_{end} \leq \bar{p}_n.t_{end} \tag{6.16}$$
     Since Equations 6.14, 6.15 and 6.16 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 6.9 holds. And since $\bar{p}'.I > \bar{p}.I$, then Equation 6.10 holds. From Equations 6.9 and 6.10 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 6.17 holds:

$$\bar{p}'_m.L - \bar{p}'_m.end + \bar{p}'.end \geq \bar{p}_n.L - \bar{p}_n.end + \Delta \tag{6.17}$$

  And since $\bar{p}'.t_{end} \geq \Delta - t_{seg}$, then Equation 6.18 holds:

$$\Delta - \bar{p}'.t_{end} \leq 0 \tag{6.18}$$

  From Equations 6.17 and 6.18, then Equation 6.19 holds:

$$\bar{p}'_m.L - \bar{p}'_m.end \geq \bar{p}_n.L - \bar{p}_n.end \tag{6.19}$$

  And since, $\bar{p}'.I > \bar{p}.I$, then Equation 6.14 holds.
  From Equations 6.19 and 6.14 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 6.9 holds. And since $\bar{p}'.I > \bar{p}.I$, then Equation 6.10 holds. From Equations 6.9 and 6.10 and since $\bar{p}'_n.t_{end} \geq \bar{p}'_n.t_{end}$, then Condition 1 applies between $\bar{p}'_n$ and $\bar{p}_n$.

□

Based on Lemma 26, if any of the conditions apply to two partial paths $\bar{p}$ and $\bar{p}'$, then we can safely cut $\bar{p}'$ as one of the complete paths obtained from $\bar{p}$ is guaranteed to be better than any complete path that we can obtain from $\bar{p}'$. We next present the complete region sequence segmentation algorithm.

Algorithm 8 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 26 before becoming the set of partial paths $\bar{\mathcal{P}}$ at the next iteration. If r is a splittable loop, then before generating a 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

Algorithm 8 Segment a Sequence of Regions

Input: A set of sequential regions R and the set of last segment regions R_{last}

function SEGMENTSEQUENCE(R)
  $\bar{\mathcal{P}} = [\bar{p} = \emptyset]$, $\bar{\mathcal{P}}_{last} = \emptyset$, $\mathcal{P}_{next} = \emptyset$, $\bar{R}_{last} = R_{last}$
  for all $r \in R$ **do** ▷ Traverse the sequence in topological order.
    if r is a splittable loop **then**
      for all k_p, k_s **do**:
        Split r to r_p, r_t and r_s
        $\bar{\mathcal{P}}_{loop}$ = CREATEPARTIALPATHS($r_p, \bar{\mathcal{P}}$)
        Filter $\bar{\mathcal{P}}_{loop}$ using Lemma 26
        $\bar{\mathcal{P}}_{loop}$ = all path by joining $\bar{\mathcal{P}}_{loop}$ with TILE(r_t, $N_r - k_p - k_s$)
        $\bar{\mathcal{P}}_{loop}$ = CREATEPARTIALPATHS($r_s, \bar{\mathcal{P}}_{loop}$)
        if $r_s \in \bar{R}_{last}$ **then**
          Create s^{end} from all regions in $\bar{R}_{last}$
          For each $\bar{p} \in \mathcal{P}_{loop}$, create $\bar{p}_{last}$ by adding s^{end} to $\bar{p}$, add $\bar{p}_{last}$ to $\bar{\mathcal{P}}_{last}$
        $\bar{\mathcal{P}}_{next} = \bar{\mathcal{P}}_{next} \bigcup \mathcal{P}_{loop}$
    else ▷ r is a mergeable region that is not a splittable loop
      $\bar{\mathcal{P}}_{next}$ = CREATEPARTIALPATHS($r, \bar{\mathcal{P}}$)
      if $r \in \bar{R}_{last}$ **then**
        Create s^{end} from all regions in $\bar{R}_{last}$
        For each $\bar{p} \in \bar{\mathcal{P}}$, create $\bar{p}_{last}$ by adding s^{end} to $\bar{p}$, add $\bar{p}_{last}$ to $\bar{\mathcal{P}}_{last}$
    Filter $\bar{\mathcal{P}}_{next}$ using Lemma 26, $\bar{\mathcal{P}} = \bar{\mathcal{P}}_{next}$, $\mathcal{P}_{next} = \emptyset$, $\bar{R}_{last} = \bar{R}_{last} \setminus r$
  Filter $\bar{\mathcal{P}}$ by removing dominating paths based on Definition 16
  return $\mathcal{P} = (\mathcal{P}_{last}$ if $R \supseteq R_{last}$ else $\bar{\mathcal{P}})$
function CREATEPARTIALPATHS($r, \bar{\mathcal{P}}$)
  $\bar{\mathcal{P}}_{tmp} = \emptyset$
  for all $\bar{p}$ in $\bar{\mathcal{P}}$ **do**
    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
    Create $\bar{p}_n$ by adding new segment using r to $\bar{p}$, add $\bar{p}_n$ to $\bar{\mathcal{P}}_{tmp}$
  return $\bar{\mathcal{P}}_{tmp}$

paths in $\bar{\mathcal{P}}_{loop}$; note this is equivalent to adding each tile region to a segment and adding the segment to each partial path, i.e., following the rule for constructing new partial paths $\bar{p}_n$. Then, partial paths are created using r_s for all paths in $\bar{\mathcal{P}}_{loop}$. All paths $\bar{\mathcal{P}}_{loop}$ are finally accumulated in $\bar{\mathcal{P}}_{next}$.

The final complexity regards the case where $R \supseteq R_{last}$. In this case, Definition 18 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 27. *Algorithm 8 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 7 (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 $p' \succeq p$, and if $R \supseteq R_{last}$, then $p'.end = p.end$. A path can be discarded for three reasons: (1) Algorithm 7 removes a tiling solution; (2) a partial path is discarded based on the conditions in Lemma 26; (3) a complete path is filtered based on Definition 16.

Case (1): Assume that Algorithm 7 removes a path p'_t from the returned path set; by Property 6, it must return another path p_t such that $p'_t \succeq p_t$. 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 17, it must hold $p' \succeq p$. 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 26, there must be a path $p \in \mathcal{P}$ such that $p' \succeq p$. 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 26 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 $p'_1 \succeq p_1$. This implies that we can find a complete path $p = \{p_1, \{s^{end}\}\}$ in $\mathcal{P}$, where by Lemma 17 it holds $p' \succeq p$, 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 16 $p' \succeq p$ must hold. □

6.2.3 Optimal Task Set Segmentation

Based on the analysis Properties 2, 3 introduced in Section 5.4 and segmentation Algorithm 6, we now show that we can obtain an optimal task set segmentation using Algorithm 9. 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 6 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 $\{G_1, \ldots, G_N\}$. If no solution can be found, the algorithm eventually terminates on Line 2.

We now prove the optimality of Algorithm 9 for a program segmentation obeying the footprint and compilation constraints in Section 6.1. We start with a corollary.

Corollary 1. *Consider two DAGs G_j, G'_j for task τ_j where $1 \leq j \leq i$ 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 $\overline{l_i^{l\max}} \geq \overline{l_i^{l\max}}'$.*

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 $\overline{l_i^{l\max}} \geq \overline{l_i^{l\max}}'$. □

Algorithm 9 Task Set Segmentation

Input: Task set Γ, source code for each task in Γ

1: SegmentTaskSet(Γ, $i, +\infty, \emptyset$)
2: Terminate with FAILURE
3: **function** SegmentTaskSet($\Gamma, i, l^{\max}, \{G_1, \ldots, G_{i-1}\}$)
4: Generate $\mathcal{G}_i$ = SegmentTask(τ_i) using Algorithm 6 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($\Gamma, i+1, \min\left(l^{\max}, \overline{l_i^{l\max}}\right), \{G_1, \ldots, G_i\}$)
9: **else**
10: **for all** $G_N \in \mathcal{G}_i$ **do**
11: If analysis returns schedulable on $\{G_1, \ldots, G_N\}$, terminate with SUCCESS

Theorem 28. *Algorithm 9 is an optimal segmentation algorithm for a multi-segment conditional streaming 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 9 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 SegmentTaskSet that results in a DAG set $G_1, \ldots G_i$ such that $G'_j \succeq G_j$ 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 6. Hence, by Definition 15 the algorithm must find a DAG $G_1 \in \mathcal{T}_1$ such that $G'_1 \succeq G_1$. 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 $G'_j \succeq$

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 $G_i \in \mathcal{G}_i$ 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 $G'_1, \ldots, G'_N$, 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 15, this implies that Algorithm 6 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 1. □

Complexity: since it iterates over all $G_i \in \mathcal{G}_i$, Algorithm 9 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 5.6. 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 6.4, in practice this results in an acceptable runtime considering the algorithm is an offline optimization.

Composability and Generality: note that the proposed conditional, streaming execution model is a generalization of the previous 3-phase model in [136, 152] and related papers; hence, it is also optimal for such approaches. In fact, our algorithm is optimal for any schedulability analysis satisfying Properties 2, 3. As we (re-)compile all tasks, our approach requires the source code of all applications in the system. Since Algorithm 9 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 4.9 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.

6.3 Segmentation for the Variable-size DMA Model

In the previous section, we pursued an optimal approach by segmenting the tasks one at a time and propagating a maximum length for a segment from the higher priority to the lower

priority tasks to control the blocking time. This was possible as the DMA time is constant and hence it is not part of the optimization problem. So, the optimization was focused on the segment length; and the approach was to maximize the segment length for the task, as this incurs less overhead and less blocking from lower priority tasks. For the variable-size DMA model, both the segment length and the DMA time of the segments affect the schedulability of the task set. In this case, we cannot rely on maximizing the segment length as this may increase the memory length also creating more interference on the lower priority tasks. Also, having a limit on the segment length does not necessarily reflect the limit on the memory time. Therefore, a global optimization problem that segments all tasks together is required. However, such an optimization problem is too complicated to formulate and solve. Therefore, we propose a set of heuristic algorithms to tackle this complexity. The intuition behind these heuristics is as following:

- Minimize the number of segments whenever possible to avoid the overhead of segmentation, especially terminal segments to limit the blocking from lower priority tasks. This means that streaming segments are better than terminal segments unless they introduce high under-utilization. That is, streaming segments cause less blocking from lower priority tasks; but if the streaming memory time is larger than the segment computation time, then the task suffers under-utilization.

- Avoid small segments and large memory times as this can lead to segment under-utilization, i.e. if a small segment executes in parallel with a large DMA transfer, the segment has to wait for the DMA time to finish before switching to the next segment.

- Segments with similar sizes are better than segments with different sizes. The variations in the segment sizes can be quantified using the standard deviation. This follows the interference computation in Algorithm 5 where segment lengths are listed along with DMA lengths and the highest elements are chosen. So, if two paths have similar lengths (path length is the summation of segment lengths), then segments with more balanced lengths are better than unbalanced lengths as they avoid being dominated by DMA lengths.

In this section, we start with the proposed iterative algorithm for the task set segmentation. Then, we detail the heuristics for the program segmentation including loop tiling and region sequence segmentation.

The heuristics are controlled with some parameters that work as knobs to tune how the segmentation space is explored. More specifically, we use a maximum number of attempts for segmentation $N_{attempts}$, a memory limit factor $\alpha < 1.0$, and a step for splitting and

tiling loops k_{step}. We explain the role of each parameter in the context of their usage in the algorithms.

6.3.1 Task Set Segmentation

In order to optimize the segmentation of the task set, we need to optimize the components that contribute to the response time of the tasks: segment lengths and the memory times of the segments. So, we follow an iterative approach in Algorithm 10 by tuning the memory times through iterations and optimizing the segment lengths in each iteration. Each iteration, we try to segment the task set giving a limit on the memory time, i.e. load or unload time, of each segment m^{max}. Note that each task segmentation generates a single DAG from the segmented tree, unlike Algorithm 6 that returns a set of DAGs. We start with unlimited m^{max} and try to segment the tasks. If the segmentation fails at task i, it returns m as the maximum load or unload time from the tasks up to task i. Then, m^{max} is computed as a fraction α of m and the segmentation is repeated giving the new m^{max} until the task set is schedulable or until $N_{attempts}$ attempts are made. If no segmentation is found, the algorithm declares failure.

In each iteration, each task of the task set is segmented with a maximum segment length l^{max}. Computing l^{max} is not straightforward as in the fixed-size DMA model. So, we derive an estimate for l^{max} using Algorithm 11. The algorithm uses the points of interest as computed in Section 5.4.1. Then, it iterates over each point t in $\mathcal{S}_i$ computing l^t_{max}. The final l^{max} is the maximum l^t_{max} over all points. For each point, the algorithm iterates over the combinations of all paths P_i of the task's DAG and all paths P of interfering jobs. For each combination, we compute l^{max}_P; then l^{max}_t is obtained as the maximum over all combinations. To estimate l^{max}_P, Algorithm 5 is used to get the M list after assuming 0 for all the lower priority computation and memory times. This implies that only the higher priority task and the task under analysis will contribute to M. After that, the elements in the list are used to compute H' which ignores the lower priority blocking. Then, we use $H'_i + B_i + x * l^{max}_P \leq t$ to compute l^{max}_P, such that x is the number of blocking intervals and $B_i = l^{max}_P$ if $i < N$ or 0 otherwise. This assumes that lower priority blocking is $x * l^{max}_P$ with $x = P.I$ as initial value. The computed l^{max}_P might not be correct if there are elements of M after the first num$H - |NPS| - x$ elements that are higher than l^{max}_P and hence will push the l^{max}_P elements after them in the list and hence they contribute to H'_i. The number of elements x of l^{max}_P is decreased for each element that is larger than it. The process continues until either x becomes 0 and hence l^{max}_P becomes the last element in the complete M list; or until the list is stable, i.e. no more elements can push l^P_{max} down the list.

Algorithm 10 Task Set Segmentation

Input: Task set Γ, source code for each task in Γ

1: $m^{max} = +\infty$, $attempt = 1$
2: **while** $attempt \leq N_{attempts}$ **and** $m^{max} > 0$ **do**
3: m = SegmentTaskSet(Γ, $1, +\infty, m^{max}, \emptyset$)
4: $m^{max} = \alpha * m$, $attempt = attempt + 1$
5: Terminate with FAILURE
6:
7: **function** SegmentTaskSet($\Gamma, i, l^{\max}, m^{max}, \{G_1, \ldots, G_{i-1}\}$)
8: Generate G_i = SegmentTask(τ_i) using Algorithm 12 based on length constraint $l^{\max}$ and memory transfer limit m^{max}
9: **if** $i == N$ **then**
10: **if** The analysis returns schedulable on $\{G_1, \ldots, G_N\}$ **then**
11: Terminate with SUCCESS
12: **else**
13: **return** m = maximum load or unload time
14: **else**
15: $l_i^{l\max}$ = ComputeLmax(G_i)
16: **if** $l_i^{l\max} \leq 0$ **then**
17: **return** m = maximum load or unload time
18: SegmentTaskSet($\Gamma, i+1, \min\left(l^{\max}, l_i^{l\max}\right), \{G_1, \ldots, G_i\}$)

Algorithm 11 l^{max} Computation

1: **function** ComputeLmax(G_i)
2: $l^{max} = 0$
3: Compute the set of points $\mathcal{S}_i$ as in Section 5.4.1
4: **for all** $t \in \mathcal{S}_i$ **do**
5: $l_t^{max} = +\infty$
6: **for all** $P_i \in G_i \cup P \in \mathcal{P}$ **do**
7: Assume that $c^{l\max} = 0$, $ul^{l\max} = ld^{l\max} = 0$ in Algorithm 5 to compute H_i
8: In ComposeTimes in Algorithm 4, obtain list M.
9: Compute HTP' as $HTP' = \sum_{j=1}^{\text{num}H-|NPS|-x} M_j$
10: Compute $H_i' = HNP + HTP'$
11: Set $x = P_i.I$, $k = 0$
12: **repeat**
13: $stable = 1$
14: Compute $l_P^{max} = \frac{t-H_i'}{x+1}$ if $i < N$, otherwise $l_P^{max} = \frac{t-H_i'}{x}$
15: **while** $M_{\text{num}H-|NPS|+k} > l_t^{max}$ **and** $x > 0$ **do**
16: $H_i' = H_i' + M_{\text{num}H-|NPS|+k}$, $x = x - 1$
17: $l_P^{max} = M_{\text{num}H-|NPS|+k}$
18: $k = k + 1$, $stable = 0$
19: **until** $stable = 1$
20: $l_P^{max} = \min(l_P^{max}, l_t^{max})$
21: $l^{max} = \max(l_t^{max}, l^{max})$
22: **return** l^{max}

6.3.2 Segmentation Algorithm

Algorithm 12 Segmentation Algorithm

1: **function** SEGMENTTASK(τ)
2: **if** r_0 is mergeable **and** (r_0 is a basic block **or** satisfies m^{max} constraint) **then**
3: Create DAG G with a single segment comprising r_0, **return** G
4: Generate a single DAG G from $\mathcal{T}$ = SEGMENT(r_0), **return** 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 **and** satisfies m^{max} constraint) **or** r_c is a splittable loop **then**
10: Add r_c to R
11: **else if** r_c is mergeable **and** (r_c is a basic block **or** satisfies m^{max} constraint) **then** ▷ r is not sequential
12: Replace r_c single-segment path p
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** $\mathcal{T}$

Algorithm 12 is the task segmentation algorithm. It is similar to Algorithm 6 with the following differences: 1) a mergeable region has to be either a basic block or to satisfy the memory transfer limit m^{max} to be replaced with a path of a single segment or to be added to a region sequence; 2) functions TILE and SEGEMENTSEQUENCE return a single path; 3) it returns a single DAG from the segmented tree. For loop tiling and region sequence segmentation, we propose two heuristics, Algorithm 13 and Algorithm 14.

2-Level Loop Tiling

In Algorithm 13, our purpose is to generate a single path for the loop by applying 2-level tiling. The algorithm is invoked by calling the TILE function for the loop region. The function TILE starts by calling TILELOOP to generate a path for the loop by streaming the tile segments. If no path is found, TILELOOP is called again to segment the loop without streaming the segments. Then, the final path is returned. In function TILELOOP, we iterate over the two loop levels similar to Algorithm 7. For each k_2 and k_1 values, a path is generated and added to the path set $\mathcal{P}$ if it is valid. Then, the tile size is decremented by a step value k_{step} which is an arbitrary value to control the algorithm speed. The path set $\mathcal{P}$ is then filtered and one path is chosen if $\mathcal{P}$ is not empty. The filtration criteria is based on three factors:

- If the function is called with $stream = 1$, then $\mathcal{P}$ is filtered using a streaming threshold. The streaming threshold $th_{seg} = s.c/s.l$ is a threshold on the ratio between the streaming segment computation time $s.c$ and the streaming segment length $s.l$ of a complete tile segment. We found by experiment that the streaming case performs better for $0.6 \geq th_{seg} \leq 0.7$. The intuition for this threshold is that if the computation time $s.c$ is lower than the streaming time $s.st$, then the segment is underutilized. This implies that the total time of the path also increases revoking the benefit of streaming.
- The path set is filtered to conform to the memory transfer limit m^{max} used in the iterative task set algorithm. That is, each segment with unload time $s.ul$ and load time $s.ld$ should be less than or equal to m^{max}. This constraint is not strict as violating it does not necessarily lead to the failure of scheduling the task set. This implies that if no paths are found to satisfy this constraint, the closest path to this limit is chosen.
- The last filter for the path set is based on the number of segments and the variations of the segment lengths in the path. This means that the paths with the lowest number of segments are kept. If multiple paths have the same number of segments, then the paths with the least variation in the lengths of the segments are kept.

Region Sequence Segmentation

Algorithm 14 takes a region sequence as an input and generates a single path. The heuristic works similar to Algorithm 8 by iterating over the regions in the sequence and generating partial paths. However, it differs in the following aspects:

Algorithm 13 2-Level Tiling

```
1: function TILE(r)
2:     p = TILELOOP(r, 1)                                   ▷ Try to stream first
3:     if p does not exist then
4:         p = TILELOOP(r, 0)                               ▷ Try without stream
5:     return p
6: function TILELOOP(r, stream)
7:     P = ∅
8:     Compute k_2 = k_2^max based on stream
9:     repeat
10:        Compute k_1 = k_1^max(k_2) based on stream
11:        repeat
12:            Generate p(k_2, k_1) based on stream
13:            if p(k_2, k_1) is valid based on stream then
14:                Add p(k_2, k_1) to P
15:            k_1 = k_1 − k_step
16:        until k_1 ≤ 0
17:        k_2 = k_2 − k_step
18:    until k_2 ≤ 0
19:    if stream then
20:        Filter P using streaming threshold th_seg.
21:    Filter P using memory transfer limit.
22:    Filter P based on number of segments, then minimum segment variation.
23:    return First p ∈ P
```

- Tiling a loop returns a single path which results in a reduced number of partial paths to propagate through the sequence.
- The partial path set is filtered at two positions in the algorithm: the path set generated for each value of k_p before joining them with the loop path, and the final path set is filtered after the last region from which a single path is chosen and returned.
- Filtering a path set in this algorithm depends on: number of segments, path length and the segment length variation. That is, a path with fewer segments is better; if two paths have the same number of segments, then the path with lower sum of segment lengths is better; finally if two paths have the same number of segments and the same path length, then the path with less variation in the segment length is better.

6.4 Evaluation

The evaluation of the segmentation algorithms target a simple MIPS processor model with 5-stage pipeline and no branch prediction similar to the model used in Chapter 4. Note that the WCET of each region in a program is statically estimated using the simple MIPS processor model. We assume that there are data ScratchPad Memory (SPM), and code SPM and that the task code fits in the code SPM.

Benchmark	**Description**	**Suite**	**LOC**	**Data (Bytes)**	**WCET (cycles)**
adpcm_dec	ADPCM decoder	TACLeBench	476	404	176947
cjpeg_transupp	JPEG image transcoding routines	TACLeBench	474	3459	12083696011
fft	1024-point FFT, 13 bits per twiddle	TACLeBench	173	24572	89540809
compress	Compresses a 128 x 128 pixel image	UTDSP	131	136448	168984645
lpc	Linear predictive coding (LPC) encoder	UTDSP	249	8744	233390
spectral	Calculates the power spectral estimate	UTDSP	340	4584	109074793
disparity	Compute depth information using dense stereo	CortexSuite	87	2704641	339361377

Table 6.1: Evaluation Benchmarks

Algorithm 14 Segment a Sequence of Regions

Input: A set of sequential regions R and the set of last segment regions R_{last}

function SEGMENTSEQUENCE(R)
  $\bar{\mathcal{P}} = [\bar{p} = \emptyset]$, $\mathcal{P}_{next} = \emptyset$
  for all $r \in R$ **do** ▷ Traverse the sequence in topological order.
    if r is a splittable loop **then**
      for all k_p, k_s with step k_{step} **do**:
        Split r to r_p, r_t and r_s
        $\bar{\mathcal{P}}_{loop}$ = CREATEPARTIALPATHS($r_p, \bar{\mathcal{P}}$)
        Filter $\bar{\mathcal{P}}_{loop}$ based on number of segments, then path length, then segment length variation.
        $\bar{\mathcal{P}}_{loop}$ = all path by joining $\bar{\mathcal{P}}_{loop}$ with TILE(r_t)
        $\bar{\mathcal{P}}_{loop}$ = CREATEPARTIALPATHS($r_s, \bar{\mathcal{P}}_{loop}$)
        $\bar{\mathcal{P}}_{next} = \bar{\mathcal{P}}_{next} \bigcup \bar{\mathcal{P}}_{loop}$
    else ▷ r is a mergeable region that is not a splittable loop
      $\bar{\mathcal{P}}_{next}$ = CREATEPARTIALPATHS($r, \bar{\mathcal{P}}$)
    $\bar{\mathcal{P}} = \bar{\mathcal{P}}_{next}$
  Filter $\bar{\mathcal{P}}$ based on number of segments, then path length, then segment length variation.
  return First $p \in \mathcal{P}$
function CREATEPARTIALPATHS($r, \bar{\mathcal{P}}$)
  $\bar{\mathcal{P}}_{tmp} = \emptyset$
  for all $\bar{p}$ in $\bar{\mathcal{P}}$ **do**
    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
    Create $\bar{p}_n$ by adding new segment using r to $\bar{p}$, add $\bar{p}_n$ to $\bar{\mathcal{P}}_{tmp}$
  return $\bar{\mathcal{P}}_{tmp}$

We evaluate the segmentation and scheduling algorithms using a set of synthetic and real benchmarks. We used applications from UTDSP [140], TACLeBench [48] and CortexSuite [138] benchmark suites. The applications are chosen to represent a variety of sizes, complexities and data footprints (see Table 6.1). The applications are used to generate sets of random tasks. Each task set is composed of a random number of tasks between 5 and 15 tasks where each task is an application from the chosen benchmarks. Given a system utilization and the number of tasks, the utilization of each task is generated with uniform distribution [20], 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 100 task sets.

We report the results in terms of the system schedulability and the weighted schedulability metric. The *system schedulability* is the proportion of the schedulable task sets out of the total tested task sets. We define the *weighted schedulability* 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. In this metric, weighting individual schedulability results by u reflects the intuition that high-utilization task systems have higher "value" since they are more difficult to schedule [17].

6.4.1 Fixed-size DMA Model

For the fixed-size DMA case, we assume that the DMA needs 1 cycle per word (4 bytes). We vary the size of the SPM from 2 kB to 512 kB. The segmentation overhead t_{seg} includes the DMA intialization and the context switching, and it is assumed to be 100 cycles.

We compare our optimal algorithm with ideal, greedy and heuristic algorithms. The tests are done with and without multi-segment streaming to highlight its merit to the system schedulability. 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 9 [1]. The *greedy* and *heuristic* algorithms do not depend on Algorithm 9 to drive the segmentation of each task based on the schedulability analysis. The greedy algorithm resembles the algorithm used in [99] 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 schedulability.

[1] Note that the ideal algorithm is still compliant with the 3-phase model, i.e. the next segment has to be decided and loaded while the current segment is executing.

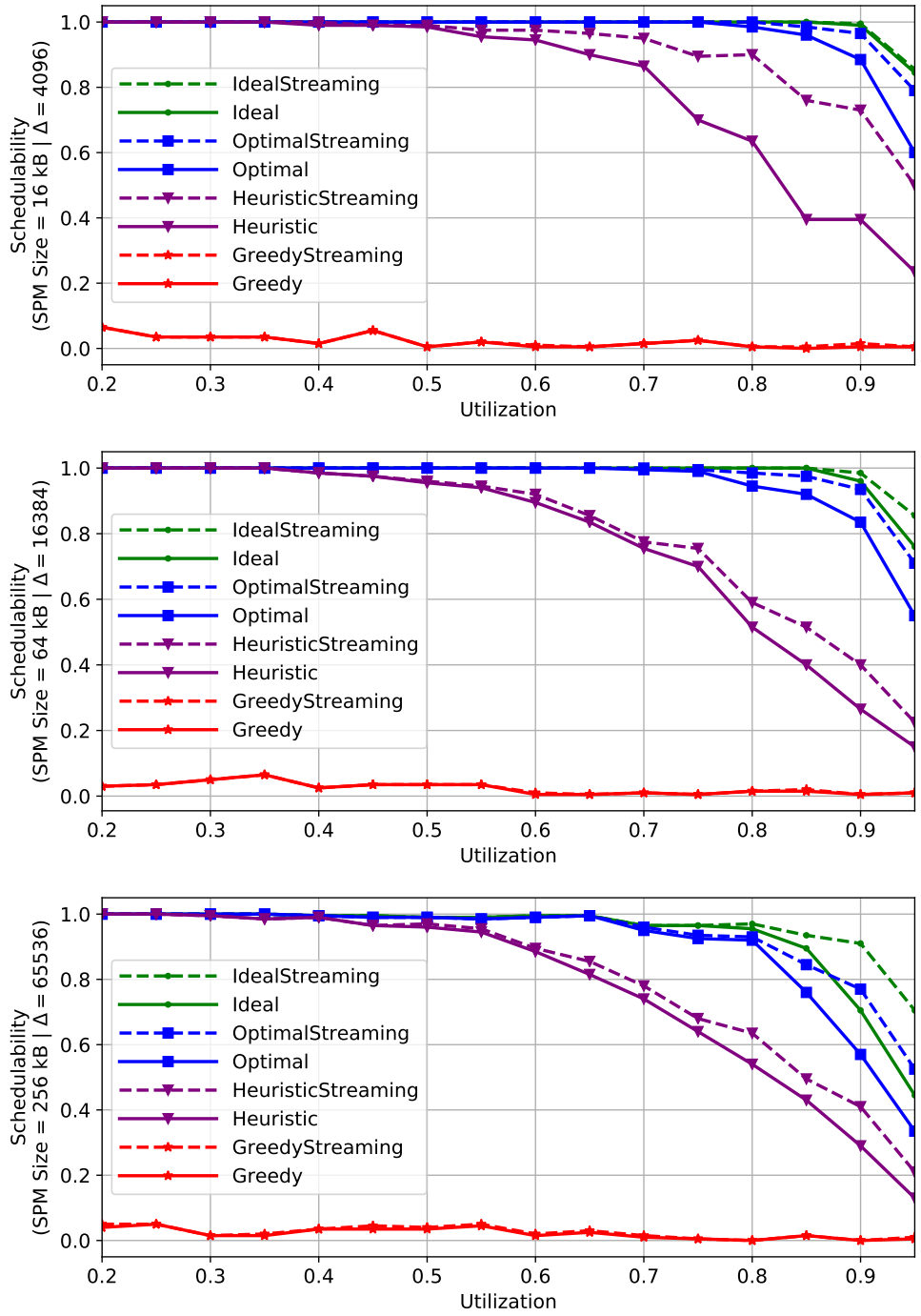


Figure 6.3: Fixed-size DMA: Schedulability vs Utilization

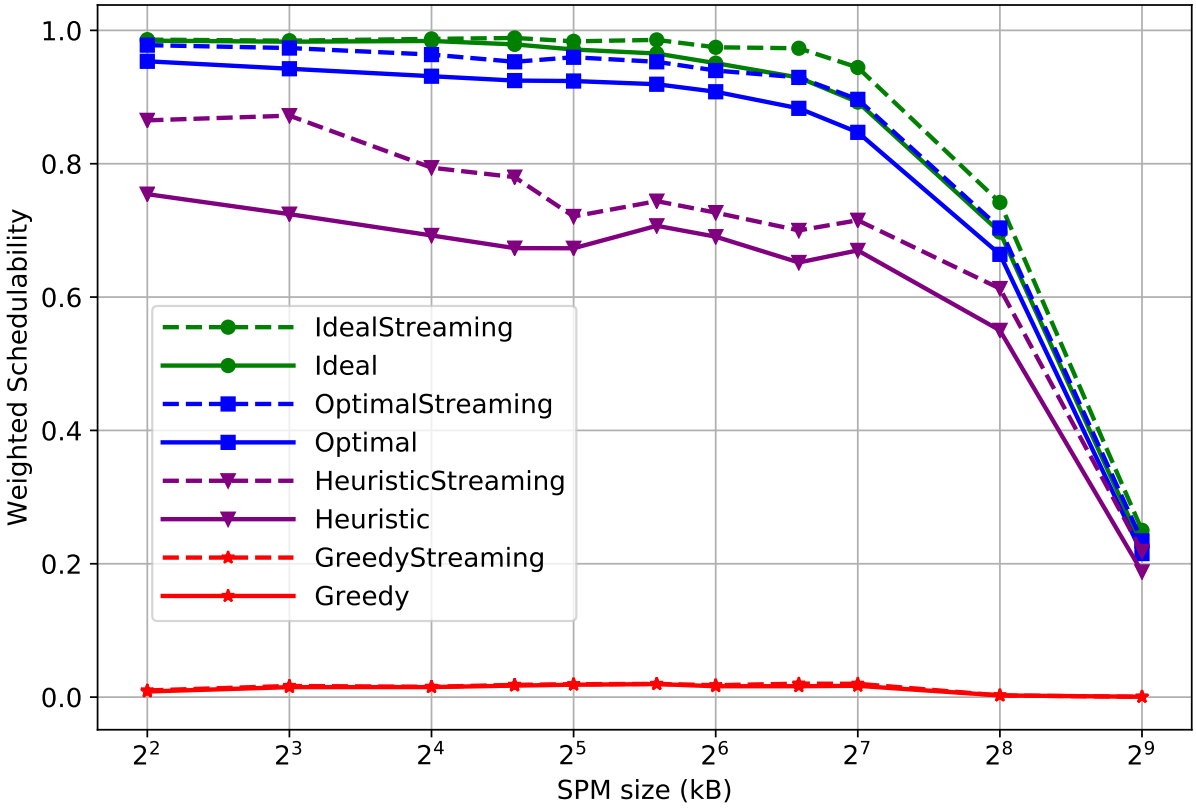


Figure 6.4: Fixed-size DMA: Weighted Schedulability VS SPM Size ($t_{seg} = 100$)

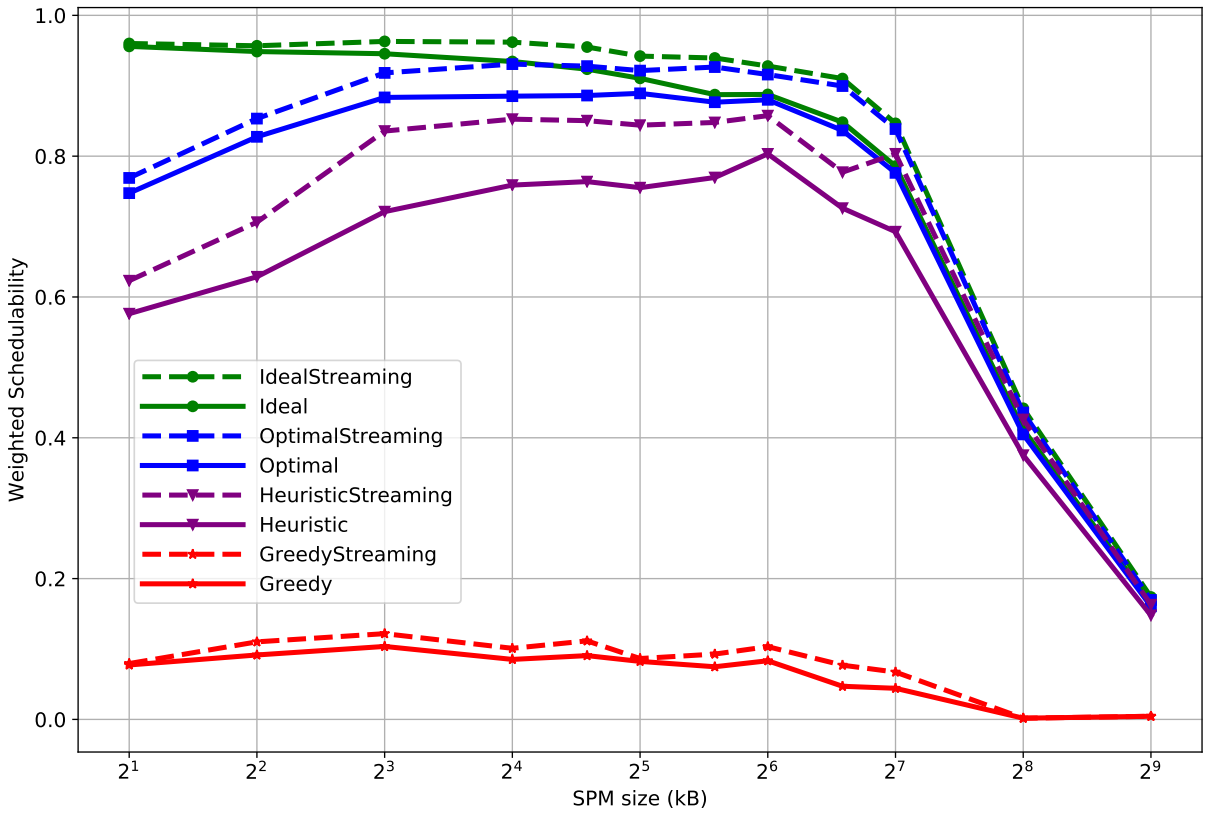


Figure 6.5: Fixed-size DMA: Weighted Schedulability VS SPM Size
($t_{seg} = 1000$, footprint > 24 kB)

Figure 6.3 shows the system schedulability for the four algorithms with and without streaming for SPM sizes of 16, 64 and 256 kB. The graphs show that the greedy algorithm performs significantly worse than the heuristic and the optimal algorithms, and that the greedy algorithm does not benefit from segment streaming. The optimal algorithms are prominently superior to the heuristic algorithms with and without streaming for different SPM sizes. The graphs also show that using segment streaming can improve the system schedulability for the ideal, optimal and heuristic algorithms compared to the case without streaming. This is more significant for high system utilization. This is confirmed in Figure 6.4 that shows the weighted schedulability for the compared algorithms for different SPM sizes . 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 6.4 that the weighted schedulability 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 8 kB. For this reason, we show in Figure 6.5 the weighted schedulability using only applications with data footprint greater than 24 kB and $t_{seg} = 1000$. The figure shows that the system schedulability ascends at first and then declines around SPM size of 48 kB.

The DMA speed is a main factor in the schedulability of the system. In order to illustrate its effect, we show in Figure 6.6 the change of the weighted schedulability vs the DMA speed factor when SPM size is 64 kB. The DMA slowdown factor is relative to the base speed of 1 cycle per word, i.e. a factor of 2 means the DMA speed is 2 cycles per word. We can see that the weighted schedulability decreases as the DMA slowdown factor increases which is related to the segment under-utilization. The figure also shows that the optimal algorithm is superior to the greedy and heuristic algorithms for all the tested speed factors.

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 6.7 the min/mean/max time in seconds to segment a task set with number of tasks per set varying between 5 and 40. The numbers were obtained by collecting the time of segmentation 100 task sets for each number of tasks.

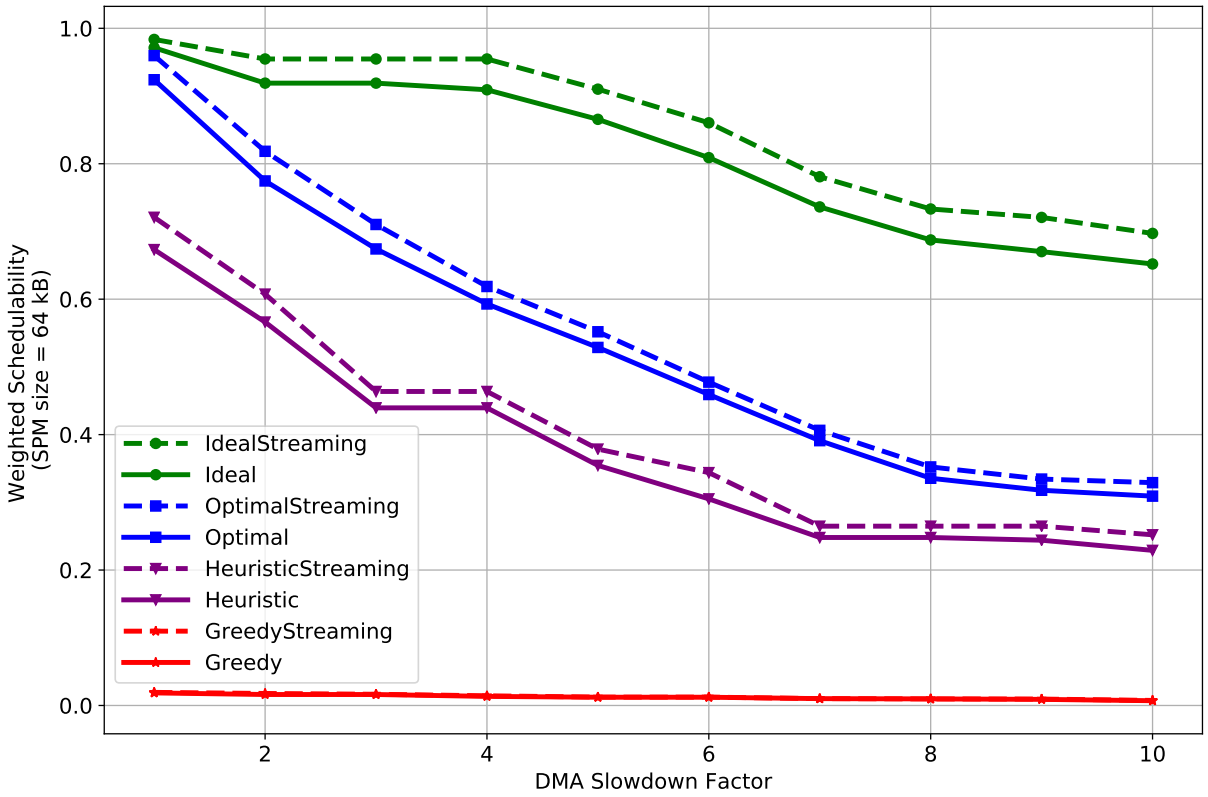


Figure 6.6: Fixed-size DMA: Weighted Schedulability VS DMA Slowdown Factor

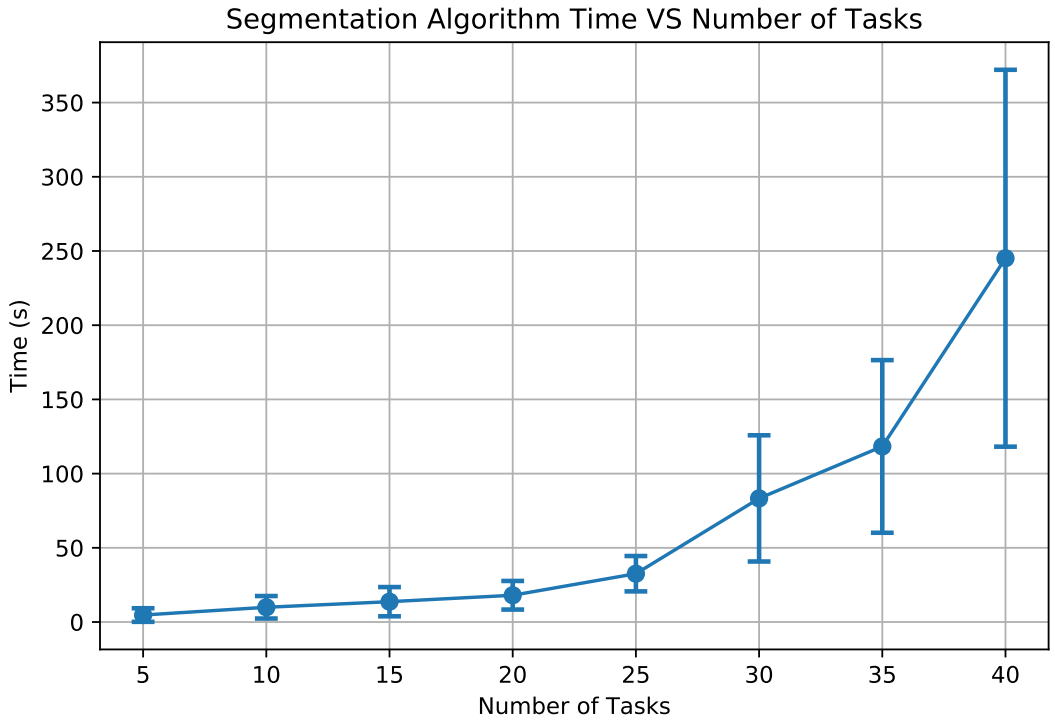


Figure 6.7: Fixed-size DMA: Segmentation Time VS Number of Tasks

6.4.2 Variable-size DMA Model

The evaluation of the segmentation using the variable-size DMA model considers the following points: the effect of the SPM size and DMA parameters, ρ and δ, on the system schedulability, and the performance of the variable-size vs the fixed-size DMA models, and the performance of the streaming vs no streaming models.

We first evaluate the effect of the SPM size by fixing $\rho = 0.5$ and $\beta = 1000$ and varying the SPM size between 4 kB to 256 kB. Figure 6.8 shows the system schedulability vs the system utilization for SPM sizes of 16, 64, and 256 kB. We can see that the variable-size model is significantly better than the fixed-length model. The figure also shows that multi-segment streaming improves the system schedulability over the non-streaming model. Figure 6.9 shows the weighted schedulability. The graph depicts how the variable-size model can adapt to the SPM size by using only the space it needs while the fixed-size model suffers segment under-utilization as the SPM size increases.

Second, we show in Figure 6.10 the effect of varying δ by fixing the SPM size to 32 kB and $\rho = 0.5$ and varying δ between and 1000 and 50000.

Finally, the effect of varying ρ is shown in Figure 6.11 by fixing the SPM size to 32 kB and $\delta = 5000$ and varying ρ between and 0.1, 5.0.

We can notice that the weighted schedulability in Figures 6.10 and Figure 6.11 decreases as δ/ρ increases and that effectiveness of the variable-size model over the fixed-size model also degrades as the δ/ρ increases. This happens due to the greater impact of the memory time on the schedulability.

6.5 Summary

In this chapter, our goal was to develop an automated compilation flow to segment a set of real-time tasks considering the limitations on the local SPM size and taking into account the system schedulability. We presented a segmentation approach based on the region-tree program structure. We defined a set of constraints to construct a valid segmented DAG. Then, we addressed the program segmentation considering two models for the DMA, fixed and variable-size. We derived an optimal approach for the segmentation for the fixed-size model and presented a heuristic-based approach for the variable-size model. We evaluated our segmentation approach and showed that the system schedulability can be improved significantly using our segmentation approach.

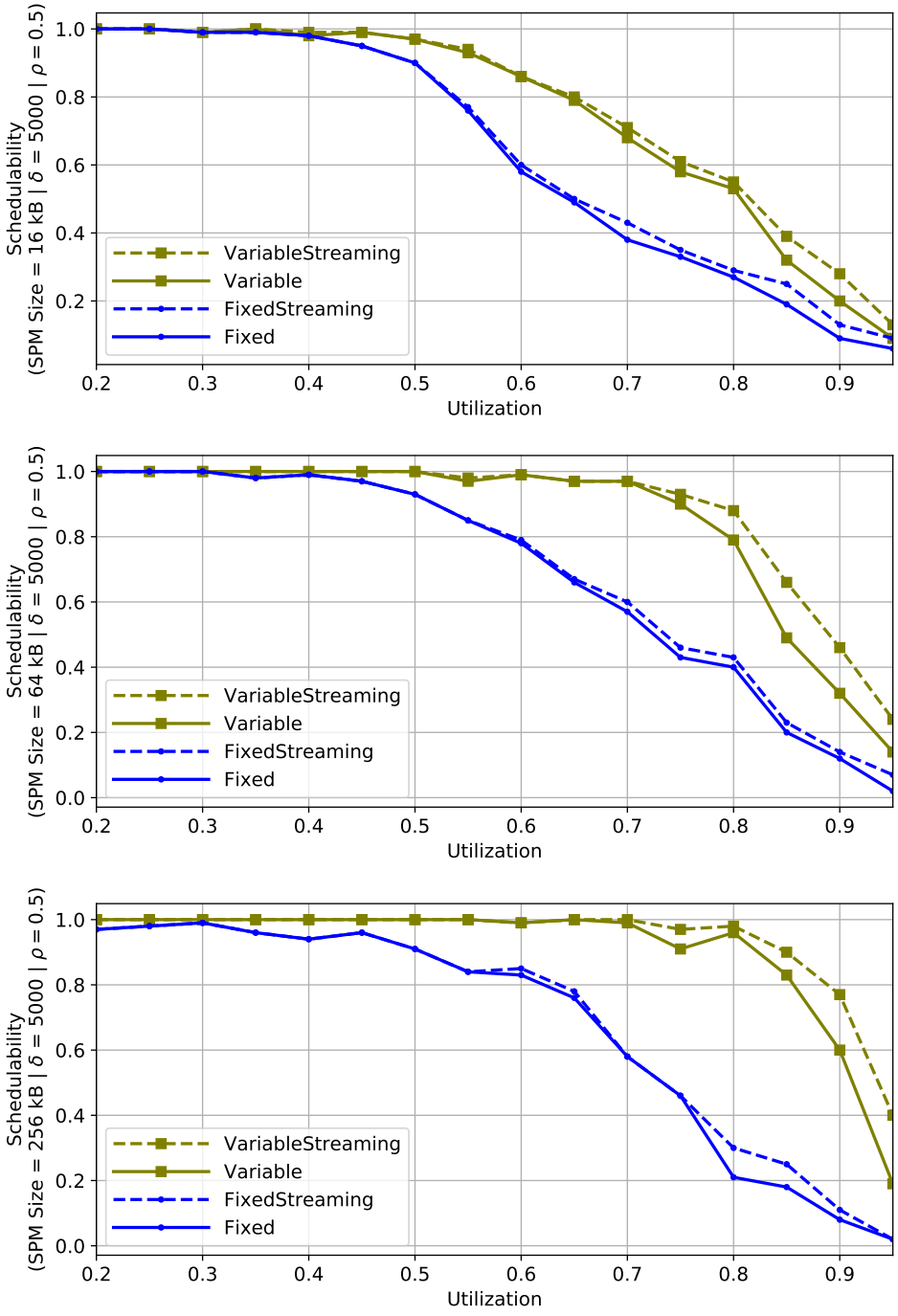


Figure 6.8: Variable-size DMA: Schedulability vs Utilization
(SPM = 16 / 64 / 256 KB, $\delta = 5000, \rho = 0.5$)

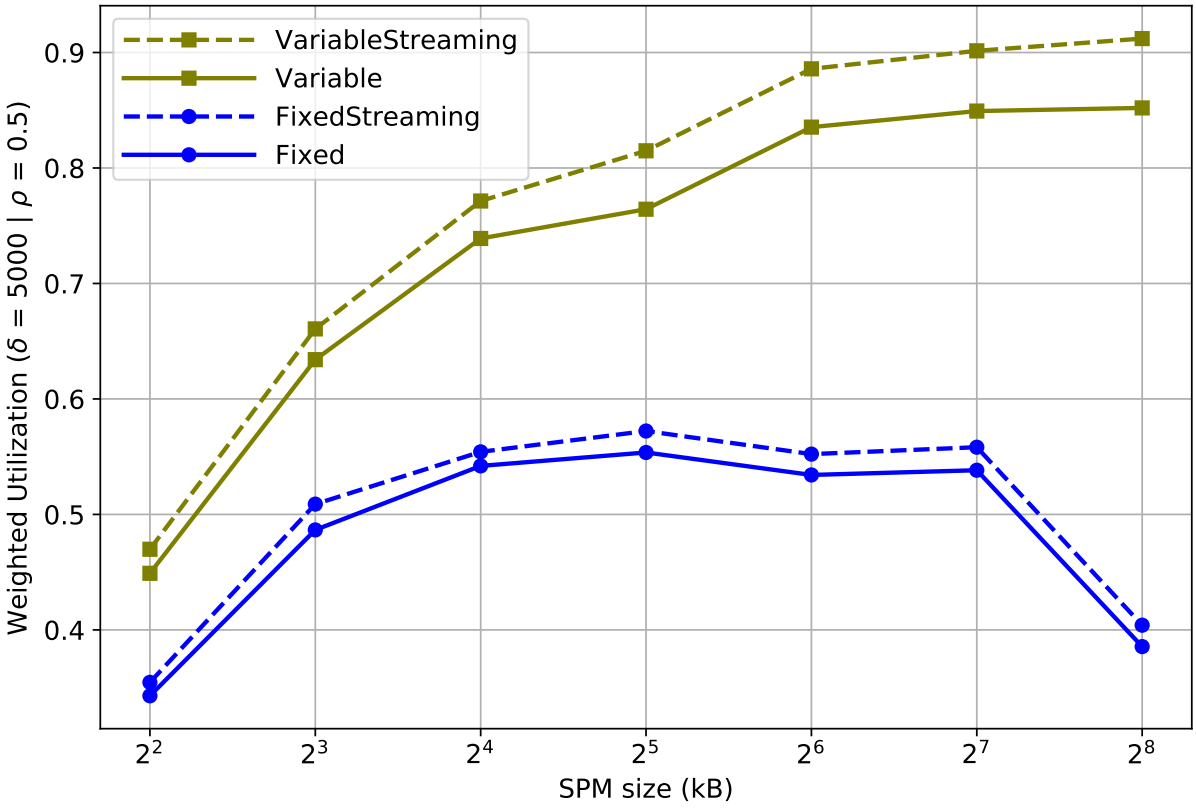


Figure 6.9: Variable-size DMA: Weighted Schedulability VS SPM Size
($\delta = 5000, \rho = 0.5$)

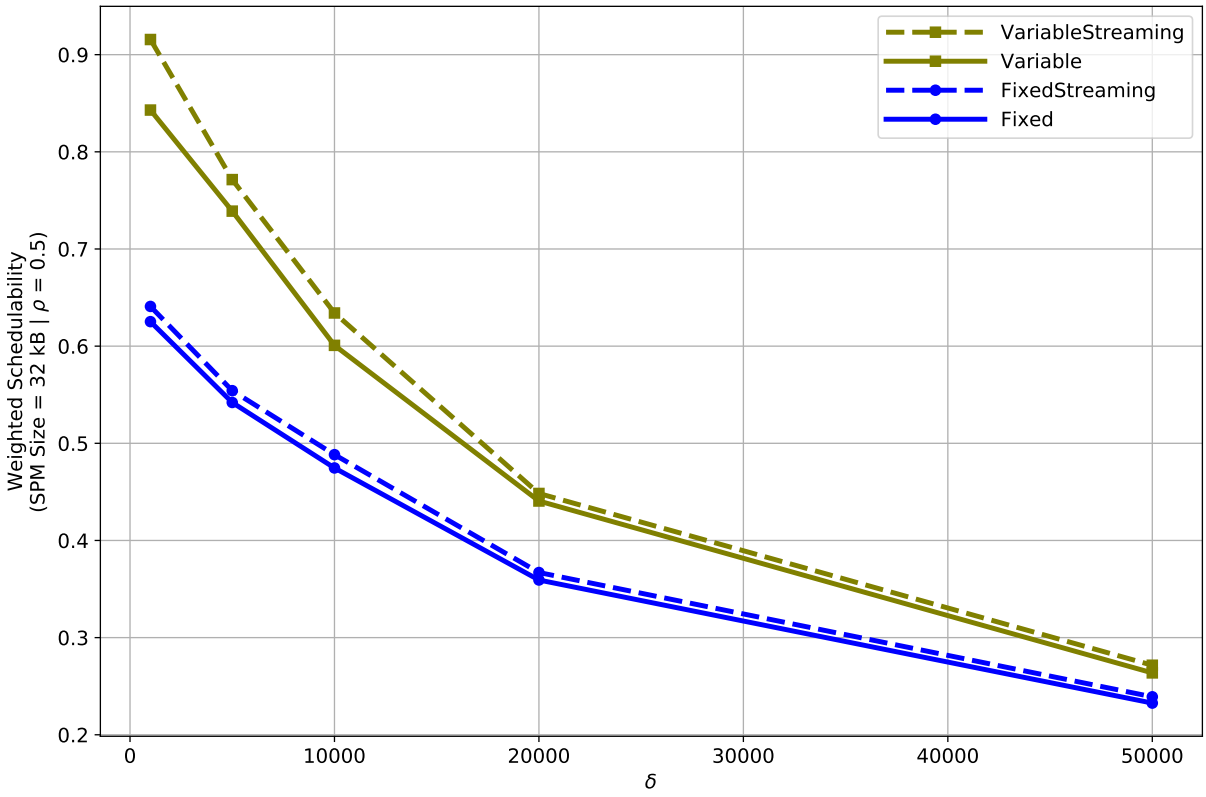


Figure 6.10: Variable-size DMA: Weighted Schedulability VS δ
(SPM size = 32 kB, $\rho = 0.5$)

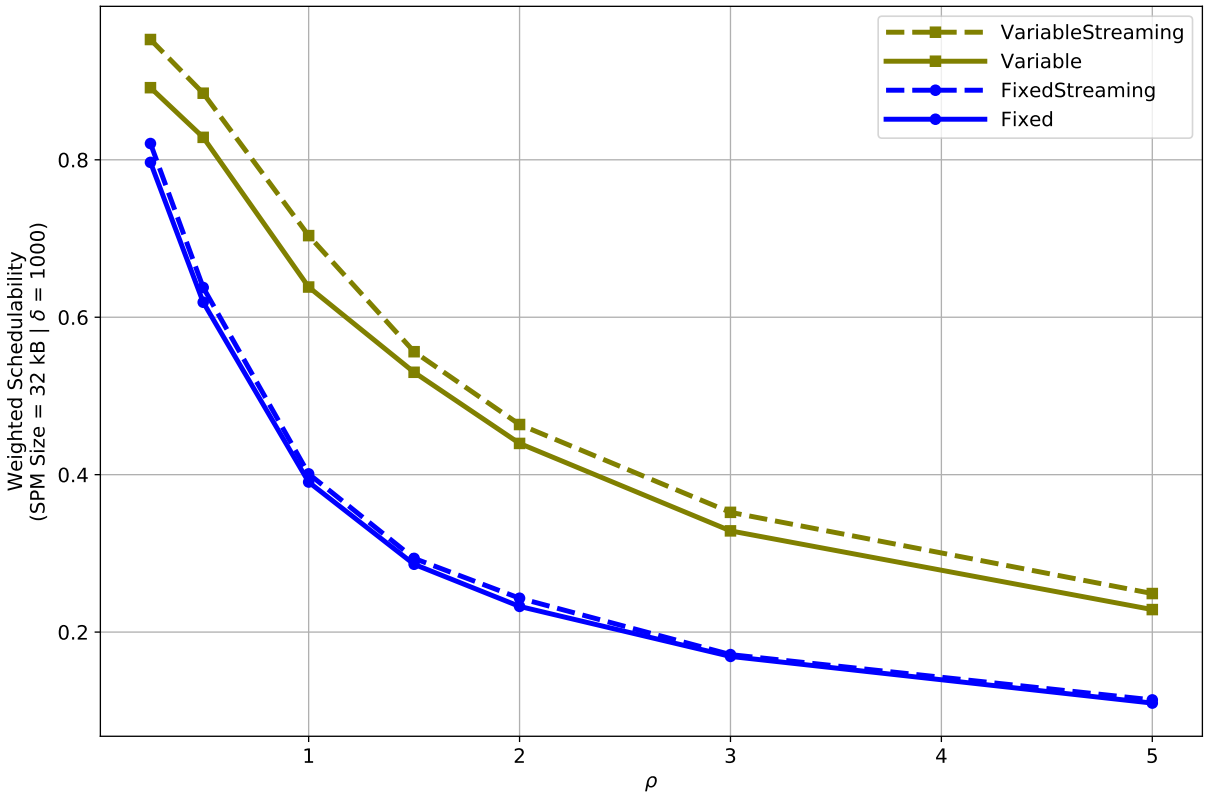


Figure 6.11: Variable-size DMA: Weighted Schedulability VS ρ
(SPM size = 32 kB, $\delta = 5000$)

Chapter 7

Conclusion and Future Work

High-performance real-time systems are gaining prominent importance in today's endeavors for automation. The design of real-time systems has to cope with the requirements for performance without losing guarantees for safety and predictability. Employing more complex hardware architectures and providing analysis techniques to model and test such systems is a necessity. On the other hand, software compilation needs to be integrated with the design process in order to create more optimized systems. In this thesis, we introduced an automated compilation framework as a step in this direction. The framework presents a compilation flow that is integrated with real-time algorithms for single-task and multi-tasking systems. We can summarize the work in this thesis as follows:

1. We presented a compilation framework based on LLVM, a well-known compiler that is open-source and commercially used on a large scale. The framework is equipped with multiple analysis tools to collect the information of the application with refinements to serve the purposes of real-time algorithms. The framework is also extendable as it is designed in a modular way following the LLVM structure.

2. We developed a ScratchPad Memory (SPM) management approach that employs software prefetching to optimize the execution of real-time applications. The approach is a step in the direction of exploiting such techniques in high-performance real-time systems. With algorithms for allocation, WCET analysis, and address assignment, the approach fulfills the automation of data SPM management.

3. We extended the 3-phase model, a widely adopted task execution model, to achieve contentionless execution of the shared resources in MPSoC systems. The extension

provided the theoretical bases for scheduling applications with conditional DAG representation and supporting multi-segment streaming.

4. We proposed a set of algorithms to automatically segment tasks to conform with the 3-phase model with limitation on the local memory size and to further enhance the system schedulability by imposing constraints on the segmentation length. The algorithms targeted two models for the DMA: fixed-size and variable-size. An optimal segmentation technique was provided for the fixed-size model and a heuristic-based technique was proposed for the variable-size model. The evaluation of the 3-phase model extensions and the developed algorithms shows significant improvements in the system schedulability.

The compilation flow of our framework allows easy integration of more algorithms to benefit from the analysis and transformation capabilities, which enables multiple possible extensions. This includes:

1. In this work, we relied on a simple processor model for the evaluation of our algorithms. The integration of WCET analysis tools such as Heptane [65] and Chronos [91] will allow the analysis of more complex processor models. Hence, this extends the applicability of our techniques for more platforms.

2. The compilation framework is used for the analysis of loops using the polyhedral model. However, the polyhedral model is more capable than only analyzing the code. It can be used for the optimization of the loops for data locality, array compaction, and minimization of inter-core communication. Several works [85, 120, 121] take advantage of the model to provide high-performance compilation for single core, multi-core, and distributed systems. The integration of such techniques has the potential of enhancing the performance of the target real-time applications.

3. We provided an allocation and prefetching approach for data SPM. Supporting the allocation and prefetching of program code will make it more viable to automate the program compilation and widens the scope of optimization. Also, the proposed allocation and prefetching technique can be improved by using software pipelining similar to the proposed segment streaming for multi-tasking systems. This will enable the allocation and prefetching of data blocks with finer granularity and hence more overlapping opportunities between computation and memory transfer.

4. The scheduling and segmentation algorithms can be extended for parallel task execution and accelerator offloading. This will take advantage of the current and future

embedded system platforms with multiple heterogeneous processing elements as well as programmable logic. The segmentation process will have to handle challenges of data coherency and communication between processing elements as well as the management of the shared resources.

Software compilation is a vital component to the efficiency of real-time systems. The integration of real-time algorithms with a powerful compiler like LLVM can have a significant impact on providing more control on the performance and the behaviour of real-time applications. This thesis shows the benefit of such integration and the potential for more optimized high-performance real-time systems.

References

[1] Hybrid Memory Cube. http://www.hybridmemorycube.org, 2017.

[2] LLVM analysis and transform passes. http://www.openmp.org, 2017.

[3] Static Value-Flow Analysis in LLVM. http://unsw-corg.github.io/SVF/, 2017.

[4] M. J. Absar and F. Catthoor. Compiler-based approach for exploiting scratch-pad in presence of irregular array access. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*, DATE '05, pages 1162–1167, Washington, DC, USA, 2005. IEEE Computer Society.

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

[6] 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.

[7] 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.

[8] 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.

[9] L. Alvarez, M. Moretó, M. Casas, E. Castillo, X. Martorell, J. Labarta, E. Ayguadé, and M. Valero. Runtime-guided management of scratchpad memories in multicore

architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 379–391, Oct 2015.

[10] F. Angiolini, L. Benini, and A. Caprara. An efficient profile-based algorithm for scratchpad memory partitioning. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 24(11):1660–1676, November 2006.

[11] Luis C. Aparicio, Juan Segarra, Clemente Rodríguez, and Víctor Viñals. Improving the wcet computation in the presence of a lockable instruction cache in multitasking real-time systems. *J. Syst. Archit.*, 57(7):695–706, August 2011.

[12] Luis C. Aparicio, Juan Segarra, Clemente Rodriguez, and Victor Vinals. Combining prefetch with instruction cache locking in multitasking real-time systems. In *Proceedings of the 2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '10, pages 319–328, Washington, DC, USA, 2010. IEEE Computer Society.

[13] Oren Avissar, Rajeev Barua, and Dave Stewart. Heterogeneous memory management for embedded systems. In *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '01, pages 34–43, New York, NY, USA, 2001. ACM.

[14] Oren Avissar, Rajeev Barua, and Dave Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Trans. Embed. Comput. Syst.*, 1(1):6–26, November 2002.

[15] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/-Software Codesign*, CODES '02, pages 73–78, New York, NY, USA, 2002. ACM.

[16] Shamik Bandyopadhyay, Thomas Huining Feng, Hiren D. Patel, and Edward A. Lee. A scratchpad memory allocation scheme for dataflow models. Technical Report UCB/EECS-2008-104, EECS Department, University of California, Berkeley, Aug 2008.

[17] Andrea Bastoni, Björn B. Brandenburg, and James H. Anderson. Cache-related preemption and migration delays : Empirical approximation and impact on schedulability. In *Proceedings of OSPERT*, 2010.

[18] 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.

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

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

[21] Uday Bondhugula, J Ramanujam, and P Sadayappan. PLuTo: A Practical and Fully Automatic Polyhedral Program Optimization System. In *PLDI 2008 - 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

[22] 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.

[23] José V. Busquets-Mataix, Carlos Catalá, and Antonio Martí-Campoy. Architecture extensions for efficient management of scratch-pad memory. In *Proceedings of the 21st International Conference on Integrated Circuit and System Design: Power and Timing Modeling, Optimization, and Simulation*, PATMOS'11, pages 43–52, Berlin, Heidelberg, 2011. Springer-Verlag.

[24] Giorgio C. Buttazzo. Hard real-time computing systems: Predictable scheduling algorithms and applications, third edition. In *Real-Time Systems Series*, 2004.

[25] Surendra Byna, Yong Chen, and Xian-He Sun. A taxonomy of data prefetching mechanisms. In *Proceedings of the The International Symposium on Parallel Architectures, Algorithms, and Networks*, ISPAN '08, pages 19–24, Washington, DC, USA, 2008. IEEE Computer Society.

[26] Marco Caccamo, Marco Cesati, Rodolfo Pellizzoni, Emiliano Betti, Roman Dudko, and Renato Mancuso. Real-time cache management framework for multi-core architectures. In *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, RTAS '13, pages 45–54, Washington, DC, USA, 2013. IEEE Computer Society.

[27] L. Caccetta and A. Kulanoot. Computational aspects of hard knapsack problems. *Nonlinear Analysis: Theory, Methods & Applications*, 47(8):5547 – 5558, 2001.

[28] 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.

[29] Gregory Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Not.*, 39(4):66–74, April 2004.

[30] D. W. Chang, I. C. Lin, Y. S. Chien, C. L. Lin, A. W. Y. Su, and C. P. Young. Casa: Contention-aware scratchpad memory allocation for online hybrid on-chip memory management. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(12):1806–1817, Dec 2014.

[31] G. Chen, O. Ozturk, M. Kandemir, and M. Karakoy. Dynamic scratch-pad memory management for irregular array access patterns. In *Proceedings of the Design Automation Test in Europe Conference*, volume 1, pages 1–6, March 2006.

[32] Yu Chen, Wenlong Li, Changkyu Kim, and Zhizhong Tang. Efficient shared cache management through sharing-aware replacement and streaming-aware insertion policy. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.

[33] Zhong-Ho Chen and Alvin W. Y. Su. A hardware/software framework for instruction and data scratchpad memory allocation. *ACM Trans. Archit. Code Optim.*, 7(1):2:1–2:27, May 2010.

[34] D. Cho, S. Pasricha, I. Issenin, N. D. Dutt, M. Ahn, and Y. Paek. Adaptive scratch pad memory management for dynamic behavior of multimedia applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(4):554–567, April 2009.

[35] Bekim Cilku, Daniel Prokesch, and Peter Puschner. A time-predictable instruction-cache architecture that uses prefetching and cache locking. In *Proceedings of the 2015 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, ISORCW '15, pages 74–79, Washington, DC, USA, 2015. IEEE Computer Society.

[36] Jason Cong, Hui Huang, Chunyue Liu, and Yi Zou. A reuse-aware prefetching scheme for scratchpad memory. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 960–965, New York, NY, USA, 2011. ACM.

[37] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[38] M. Dasygenis, E. Brockmeyer, B. Durinck, F. Catthoor, D. Soudris, and A. Thanailakis. A combined dma and application-specific prefetching approach for tackling the memory latency bottleneck. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(3):279–291, March 2006.

[39] Ning Deng, Weixing Ji, Jiaxin Li, Feng Shi, and Yizhuo Wang. A novel adaptive scratchpad memory management strategy. In *Proceedings of the 2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '09, pages 236–241, Washington, DC, USA, 2009. IEEE Computer Society.

[40] Jean-Francois Deverge and Isabelle Puaut. Wcet-directed dynamic scratchpad memory allocation of data. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, ECRTS '07, pages 179–190, Washington, DC, USA, 2007. IEEE Computer Society.

[41] Huping Ding, Yun Liang, and Tulika Mitra. Integrated instruction cache analysis and locking in multitasking real-time systems. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 147:1–147:10, New York, NY, USA, 2013. ACM.

[42] Huping Ding, Yun Liang, and Tulika Mitra. Wcet-centric dynamic instruction cache locking. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pages 27:1–27:6, 3001 Leuven, Belgium, Belgium, 2014. European Design and Automation Association.

[43] Angel Dominguez, Nghi Nguyen, and Rajeev K. Barua. Recursive function data allocation to scratch-pad memory. In *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '07, pages 65–74, New York, NY, USA, 2007. ACM.

[44] Angel Dominguez, Sumesh Udayakumaran, and Rajeev Barua. Heap data allocation to scratch-pad memory in embedded systems. *J. Embedded Comput.*, 1(4):521–540, December 2005.

[45] 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.

[46] Bernhard Egger, Jaejin Lee, and Heonshik Shin. Dynamic scratchpad memory management for code in portable systems with an mmu. *ACM Trans. Embed. Comput. Syst.*, 7(2):11:1–11:38, January 2008.

[47] Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. VDM Verlag, 2008.

[48] 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.

[49] Heiko Falk and Jan C. Kleinsorge. Optimal static wcet-aware scratchpad allocation of program code. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 732–737, New York, NY, USA, 2009. ACM.

[50] Christian Ferdinand and Reinhard Wilhelm. On predicting data cache behavior for real-time systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, LCTES '98, pages 16–30, London, UK, UK, 1998. Springer-Verlag.

[51] Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-timesystems. *Real-Time Syst.*, 17(2-3):131–181, December 1999.

[52] 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.

[53] 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.

[54] Poletti Francesco, Paul Marchal, David Atienza, Luca Benini, Francky Catthoor, and Jose M. Mendias. An integrated hardware/software approach for run-time scratchpad management. In *Proceedings of the 41st Annual Design Automation Conference*, DAC '04, pages 238–243, New York, NY, USA, 2004. ACM.

[55] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[56] Giovani Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. A survey on cache management mechanisms for real-time embedded systems. *ACM Comput. Surv.*, 48(2):32:1–32:36, November 2015.

[57] Giovani Gracioli, Rohan Tabish, Renato Mancuso, Reza Mirosanlou, Rodolfo Pellizzoni, and Marco Caccamo. Designing mixed criticality applications on modern heterogeneous mpsoc platforms. In *Proceedings of the 31th Euromicro Conference on Real-Time Systems*, ECRTS '19, 2019.

[58] Tobias Grosser. Enabling polyhedral optimizations in LLVM. Diploma thesis, University of Passau, 2011.

[59] Daniel Grund and Jan Reineke. Toward precise plru cache analysis. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15, pages 23–35. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010.

[60] Nan Guan, Mingsong Lv, Wang Yi, and Ge Yu. Wcet analysis with mru cache: Challenging lru for predictability. *ACM Trans. Embed. Comput. Syst.*, 13(4s):123:1–123:26, April 2014.

[61] Nan Guan, Xinping Yang, Mingsong Lv, and Wang Yi. Fifo cache analysis for wcet estimation: A quantitative approach. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 296–301, San Jose, CA, USA, 2013. EDA Consortium.

[62] Y. Guo, Q. Zhuge, J. Hu, J. Yi, M. Qiu, and E. H. M. Sha. Data placement and duplication for embedded multicore systems with scratch pad memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(6):809–817, June 2013.

[63] Erik G. Hallnor and Steven K. Reinhardt. A fully associative software-managed cache design. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 107–116, New York, NY, USA, 2000. ACM.

[64] 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.

[65] Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. The Heptane Static Worst-Case Execution Time Estimation Tool. In *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, volume 8 of *International Workshop on Worst-Case Execution Time Analysis*, page 12, Dubrovnik, Croatia, June 2017.

[66] 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.

[67] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of wcet tools. *Proceedings of the IEEE*, 91(7):1038–1054, July 2003.

[68] J. Herter, P. Backes, F. Haupenthal, and J. Reineke. Cama: A predictable cache-aware memory allocator. In *2011 23rd Euromicro Conference on Real-Time Systems*, pages 23–32, July 2011.

[69] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM.

[70] J. Hu, C. J. Xue, Q. Zhuge, W. C. Tseng, and E. H. M. Sha. Towards energy efficient hybrid on-chip scratch pad memory with non-volatile memory. In *2011 Design, Automation Test in Europe*, pages 1–6, March 2011.

[71] S. W. Huang, Y. C. Chiu, Z. H. Chen, C. K. Shieh, A. W. Y. Su, and T. Y. Liang. A region-based allocation approach for page-based scratch-pad memory in embedded systems. In *Computational Science and Engineering, 2009. CSE '09. International Conference on*, volume 2, pages 9–16, Aug 2009.

[72] Maha Idrissi Aouad, René Schott, and Olivier Zendra. A Tabu Search Heuristic for Scratch-Pad Memory Management. In WASET, editor, *ICSET 2010 - International Conference on Software Engineering and Technology*, volume 64, pages 386–390, Rome, Italy, April 2010. WASET - World Academy of Science, Engineering and Technology, WASET.

[73] Maha Idrissi Aouad and Olivier Zendra. A Survey of Scratch-Pad Memory Management Techniques for low-power and -energy. In Olivier Zendra, Eric Jul, and Michael Cebulla, editors, *2nd ECOOP Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2007)*, pages 31–38, Berlin, Germany, July 2007. ECOOP. ICOOOLPS'2007 was co-located with the 21st European Conference on Object-Oriented Programming (ECOOP'2007).

[74] Ilya Issenin, Erik Brockmeyer, Miguel Miranda, and Nikil Dutt. Drdu: A data reuse analysis technique for efficient scratch-pad memory management. *ACM Trans. Des. Autom. Electron. Syst.*, 12(2), April 2007.

[75] Andhi Janapsatya, Sri Parameswaran, and A. Ignjatovic. Hardware/software managed scratchpad memory for embedded system. In *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, pages 370–377, Nov 2004.

[76] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 171–185, New York, NY, USA, 1994. ACM.

[77] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 690–695, New York, NY, USA, 2001. ACM.

[78] M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. A compiler-based approach for dynamically managing scratch-pad memories in embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(2):243–260, Feb 2004.

[79] Mahmut Kandemir, Ismail Kadayif, and Ugur Sezer. Exploiting scratch-pad memory using presburger formulas. In *Proceedings of the 14th International Symposium on Systems Synthesis*, ISSS '01, pages 7–12, New York, NY, USA, 2001. ACM.

[80] S. Kang and A. G. Dean. Darts: Techniques and tools for predictably fast memory using integrated data allocation and real-time task scheduling. In *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 333–342, April 2010.

[81] Sangyeol Kang and Alexander G. Dean. Leveraging both data cache and scratchpad memory through synergetic data allocation. In *Proceedings of the 2012 IEEE 18th*

Real Time and Embedded Technology and Applications Symposium, RTAS '12, pages 119–128, Washington, DC, USA, 2012. IEEE Computer Society.

[82] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[83] 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.

[84] Sungjun Kim. Using scratchpad memory for stack data in hard real-time embedded systems. In *Proceedings of the Memory Architecture and Organization Workshop*, 2011.

[85] Michael Kruse. *Lattice QCD Optimization and Polytopic Representations of Distributed Memory.* PhD thesis, 9 2014.

[86] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[87] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. When prefetching works, when it doesn’t, and why. *ACM Trans. Archit. Code Optim.*, 9(1):2:1–2:29, March 2012.

[88] Lian Li, Hui Feng, and Jingling Xue. Compiler-directed scratchpad memory management via graph coloring. *ACM Trans. Archit. Code Optim.*, 6(3):9:1–9:17, October 2009.

[89] Lian Li, Hui Wu, Hui Feng, and Jingling Xue. Towards data tiling for whole programs in scratchpad memory allocation. In *Proceedings of the 12th Asia-Pacific Conference on Advances in Computer Systems Architecture*, ACSAC'07, pages 63–74, Berlin, Heidelberg, 2007. Springer-Verlag.

[90] Lian Li, Jingling Xue, and Jens Knoop. Scratchpad memory allocation for data aggregates via interval coloring in superperfect graphs. *ACM Trans. Embed. Comput. Syst.*, 10(2):28:1–28:42, January 2011.

[91] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1):56 – 67, 2007. Special issue on Experimental Software and Toolkits.

[92] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. Predictable programming on a precision timed architecture. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '08, pages 137–146, New York, NY, USA, 2008. ACM.

[93] Jochen Liedtke, Hermann Haertig, and Michael Hohmuth. Os-controlled cache predictability for real-time systems. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, RTAS '97, pages 213–, Washington, DC, USA, 1997. IEEE Computer Society.

[94] Thomas Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, School of Computer Science and Engineering, Chalmers University of Technology, Sweden, 2002.

[95] Thomas Lundqvist and Per Stenström. A method to improve the estimated worst-case performance of data caching. In *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, RTCSA '99, pages 255–, Washington, DC, USA, 1999. IEEE Computer Society.

[96] Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. A survey on static cache analysis for real-time systems. *Leibniz Transactions on Embedded Systems*, 3(1):05–1–05:48, 2016.

[97] 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.

[98] Peter Marwedel. *Embedded system design: Embedded systems foundations of cyber-physical systems*. Springer Science & Business Media, 2010.

[99] 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.

[100] Ross McIlroy, Peter Dickman, and Joe Sventek. Efficient dynamic heap allocation of scratch-pad memory. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 31–40, New York, NY, USA, 2008. ACM.

[101] 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.

[102] Stefan Metzlaff, Irakli Guliashvili, Sascha Uhrig, and Theo Ungerer. A dynamic instruction scratchpad memory for embedded processors managed by hardware. In *Proceedings of the 24th International Conference on Architecture of Computing Systems*, ARCS'11, pages 122–134, Berlin, Heidelberg, 2011. Springer-Verlag.

[103] Sparsh Mittal. A survey of recent prefetching techniques for processor caches. *ACM Comput. Surv.*, 49(2):35:1–35:35, August 2016.

[104] Csaba Andras Moritz, Matthew Frank, and Saman P. Amarasinghe. Flexcache: A framework for flexible compiler generated data caching. In *Revised Papers from the Second International Workshop on Intelligent Memory Systems*, IMS '00, pages 135–146, London, UK, UK, 2001. Springer-Verlag.

[105] Frank Mueller. Compiler support for software-based cache partitioning. *SIGPLAN Not.*, 30(11):125–133, November 1995.

[106] T. R. Mück and A. A. Fröhlich. Run-time scratch-pad memory management for embedded systems. In *IECON 2011 - 37th Annual Conference on IEEE Industrial Electronics Society*, pages 2833–2838, Nov 2011.

[107] Nghi Nguyen, Angel Dominguez, and Rajeev Barua. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. *ACM Trans. Embed. Comput. Syst.*, 8(3):21:1–21:32, April 2009.

[108] Nir Oren. A survey of prefetching techniques. Technical Report CS-2000-10, University of the Witwatersrand, July 2000.

[109] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. On-chip vs. off-chip memory: The data partitioning problem in embedded processor-based systems. *ACM Trans. Des. Autom. Electron. Syst.*, 5(3):682–704, July 2000.

[110] Soyoung Park, Hae-woo Park, and Soonhoi Ha. A novel technique to use scratch-pad memory for stack management. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '07, pages 1478–1483, San Jose, CA, USA, 2007. EDA Consortium.

[111] Hiren D Patel, Ben Lickly, Bas Burgers, and Edward A Lee. A Timing Requirements-Aware Scratchpad Memory Allocation Scheme for a Precision Timed Architecture. Technical Report UCB/EECS-2008-115, EECS Department, University of California, Berkeley, 2008.

[112] David Patterson and John L Hennessy. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[113] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for cots-based embedded systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279, April 2011.

[114] 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.

[115] Louis-Noël Pouchet. Iterative Optimization in the Polyhedral Model. 2010.

[116] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, Nicolas Vasilache, Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. Loop Transformations: Convexity, Pruning and Optimization. *ACM SIGPLAN Notices*, 46(1), 2011.

[117] Aayush Prakash and Hiren D. Patel. An instruction scratchpad memory allocation for the precision timed architecture. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, pages 659–664, San Jose, CA, USA, 2012. EDA Consortium.

[118] Robert Pyka, Christoph Fassbach, Manish Verma, Heiko Falk, and Peter Marwedel. Operating system integrated energy aware scratchpad allocation strategies for multi-process applications. In *Proceedingsof the 10th International Workshop on Software &Amp; Compilers for Embedded Systems*, SCOPES '07, pages 41–50, New York, NY, USA, 2007. ACM.

[119] M. Qiu, Z. Chen, J. Niu, Z. Zong, G. Quan, X. Qin, and L. T. Yang. Data allocation for hybrid memory with genetic algorithm. *IEEE Transactions on Emerging Topics in Computing*, 3(4):544–555, Dec 2015.

[120] Thejas Ramashekar, Uday Bondhugula, Thejas Ramashekar, and Uday Bondhugula. Automatic data allocation and buffer management for multi-GPU machines. *ACM Transactions on Architecture and Code Optimization*, 10(4), 2013.

[121] Chandan Reddy and Uday Bondhugula. Effective Automatic Data Allocation for Parallelization of Affine Loop Nests. (March), 2014.

[122] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007.

[123] 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.

[124] Abhik Sarkar, Frank Mueller, and Harini Ramaprasad. Static task partitioning for locked caches in multi-core real-time systems. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '12, pages 161–170, New York, NY, USA, 2012. ACM.

[125] Martin Schoeberl. *A Time Predictable Instruction Cache for a Java Processor*, pages 371–382. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[126] Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-case execution time analysis for a java processor. *Softw. Pract. Exper.*, 40(6):507–542, May 2010.

[127] Aviral Shrivastava, Arun Kannan, and Jongeun Lee. A software-only solution to use scratch pads for stack data. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(11):1719–1727, November 2009.

[128] Froderberg B. Lindgren T. Sjodin, J. Allocation of global data objects in on-chip ram. 1998.

[129] Jan Sjödin and Carl von Platen. Storage allocation for embedded processors. In *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '01, pages 15–23, New York, NY, USA, 2001. ACM.

[130] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Found. Trends Program. Lang.*, 2(1):1–69, April 2015.

[131] Muhammad Refaat Soliman and Rodolfo Pellizzoni. Wcet-driven dynamic data scratchpad management with compiler-directed prefetching. In *29th Euromicro Conference on Real-Time Systems, ECRTS 2017, June 27-30, 2017, Dubrovnik, Croatia*, pages 24:1–24:23, 2017.

[132] Muhammad Refaat Soliman and Rodolfo Pellizzoni. Prem-based optimal task segmentation under fixed priority scheduling. In *31st Euromicro Conference on Real-Time Systems, ECRTS 2019, July 9-12, 2019, Stuttgart, Germany.*, pages 4:1–4:23, 2019.

[133] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '02, pages 409–, Washington, DC, USA, 2002. IEEE Computer Society.

[134] V. Suhendra, T. Mitra, A. Roychoudhury, and Ting Chen. Wcet centric data allocation to scratchpad memory. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 10 pp.–232, Dec 2005.

[135] Yulei Sui and Jingling Xue. SVF: Interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 265–266, New York, NY, USA, 2016. ACM.

[136] 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.

[137] Stephan Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Universität des Saarlandes, 2004.

[138] 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.

[139] Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.*, 5(2):472–511, May 2006.

[140] UTDSP Benchmark Suite, http://www.eecg.toronto.edu/ corinna/dsp/infrastructure/utdsp.html.

[141] 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.

[142] Jussi Vanhatalo, Hagen Völzer, and Jana Koehler. The refined process structure tree. In *Proceedings of the 6th International Conference on Business Process Management*, BPM '08, pages 100–115, Berlin, Heidelberg, 2008. Springer-Verlag.

[143] Xavier Vera, Björn Lisper, and Jingling Xue. Data cache locking for higher program predictability. *SIGMETRICS Perform. Eval. Rev.*, 31(1):272–282, June 2003.

[144] Xavier Vera, Björn Lisper, and Jingling Xue. Data caches in multitasking hard real-time systems. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, RTSS '03, pages 154–, Washington, DC, USA, 2003. IEEE Computer Society.

[145] Xavier Vera, Björn Lisper, and Jingling Xue. Data cache locking for tight timing calculations. *ACM Trans. Embed. Comput. Syst.*, 7(1):4:1–4:38, December 2007.

[146] Manish Verma and Peter Marwedel. Overlay techniques for scratchpad memories in low power embedded processors. *IEEE Trans. Very Large Scale Integr. Syst.*, 14(8):802–815, August 2006.

[147] Manish Verma, Stefan Steinke, and Peter Marwedel. Data partitioning for maximal scratchpad usage. In *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, ASP-DAC '03, pages 77–83, New York, NY, USA, 2003. ACM.

[148] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Cache-aware scratchpad allocation algorithm. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*, DATE '04, pages 21264–, Washington, DC, USA, 2004. IEEE Computer Society.

[149] Z. Wang, Z. Gu, and Z. Shao. Wcet-aware energy-efficient data allocation on scratchpad memory for real-time embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(11):2700–2704, Nov 2015.

[150] 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 *Proceedings of the 2013 25th Euromicro Conference on Real-Time Systems*, ECRTS '13, pages 157–167, Washington, DC, USA, 2013. IEEE Computer Society.

[151] 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.

[152] 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.

[153] L. Wehmeyer and P. Marwedel. Influence of onchip scratchpad memories on wcet prediction. In *Proceedings of the 4th International Workshop on Worst-Case Execution Time (WCET) Analysis*, New York, 2004. ACM.

[154] Lars Wehmeyer, Urs Helmig, and Peter Marwedel. Compiler-optimized usage of partitioned memories. In *Proceedings of the 3rd Workshop on Memory Performance Issues: In Conjunction with the 31st International Symposium on Computer Architecture*, WMPI '04, pages 114–120, New York, NY, USA, 2004. ACM.

[155] Lars Wehmeyer and Peter Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '05, pages 600–605, Washington, DC, USA, 2005. IEEE Computer Society.

[156] J. Whitham and N. Audsley. Investigating average versus worst-case timing behavior of data caches and data scratchpads. In *2010 22nd Euromicro Conference on Real-Time Systems*, pages 165–174, July 2010.

[157] J. Whitham and N. Audsley. Studying the applicability of the scratchpad memory management unit. In *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 205–214, April 2010.

[158] J. Whitham and M. Schoeberl. Wcet-based comparison of an instruction scratchpad and a method cache. In *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 301–308, June 2014.

[159] Jack Whitham and Neil Audsley. Implementing time-predictable load and store operations. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, pages 265–274, New York, NY, USA, 2009. ACM.

[160] Jack Whitham and Neil Audsley. The scratchpad memory management unit for microblaze: Implementation, testing, and case study. *University of York, Tech. Rep. YCS-2009-439*, 2009.

[161] Reinhard Wilhelm et al. The worst-case execution-time problem : Overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[162] L. Wu and W. Zhang. Reducing worst-case execution time of hybrid spm-caches. In *2013 IEEE 32nd International Performance Computing and Communications Conference (IPCCC)*, pages 1–9, Dec 2013.

[163] Xuejun Yang, Li Wang, Jingling Xue, Tao Tang, Xiaoguang Ren, and Sen Ye. Improving scratchpad allocation with demand-driven data tiling. In *Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '10, pages 127–136, New York, NY, USA, 2010. ACM.

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

[165] Yanqin Yang, Haijin Yan, Zili Shao, and Minyi Guo. Compiler-assisted dynamic scratch-pad memory management with space overlapping for embedded systems. *Software: Practice and Experience*, 41(7):737–752, 2011.

[166] 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.

[167] 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.

[168] Taylan Yemliha, Shekhar Srikantaiah, Mahmut Kandemir, and Ozcan Ozturk. Spm management using markov chain based data access prediction. In *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '08, pages 565–569, Piscataway, NJ, USA, 2008. IEEE Press.

[169] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *2008 IEEE International Symposium on Circuits and Systems*, pages 1192–1195, May 2008.

[170] Wei Zhang and Yiqiang Ding. Hybrid spm-cache architectures to achieve high time predictability and performance. In *Proceedings of the 2013 IEEE 24th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, ASAP '13, pages 297–304, Washington, DC, USA, 2013. IEEE Computer Society.

[171] Z. Zhou, L. Ju, Z. Jia, and X. Li. Fast and accurate code placement of embedded software for hybrid on-chip memory architecture. In *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS), 2014 IEEE Intl Conf on*, pages 1008–1015, Aug 2014.

Appendices

Appendix A

SPM Controller

A.1 Allocation Command Encoding

The allocation commands are implemented in LLVM Intermediate Representation (IR) as load/store instructions.The address and data are used to encode the command as shown in Figure A.1. The command consists of all or some of the following fields:

1. *CMD_TYPE*: the op-code for the command.
2. *TBL_IDX*: the address for the table entry of the object/pointer.
3. *PTR*: a flag to indicate that the command is for a pointer.
4. *SIZE*: the size of an object.
5. *MEM_ADDR*: the main memory or scratchpad address.

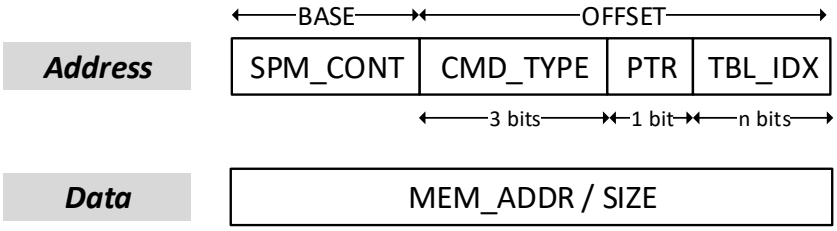


Figure A.1: Encoding of the allocation commands

The first three fields are embedded in the address offset. The MEM_ADDR and the $SIZE$ are passed using the data of the store instruction.

Command	CMD_TYPE	PTR
ALLOC	000	0/1
ALLOCP	001	0/1
ALLOCW	010	0/1
ALLOCPW	011	0/1
DEALLOC	100	0/1
GETADDR	101	0/1
SETPTR	110	1
SETMM	110	0
SETSIZE	111	0

Table A.1: Commands encodings

Table A.1 shows the binary encoding for each of the allocation commands. The command type CMD_TYPE uses 3 bits, the pointer flag PTR uses 1 bit, and the table index TBL_IDX uses n bits which is the number of address bits required to access either the object table or the pointer table. n depends on the implementation of the ScratchPad Memory (SPM) controller.

A.1.1 SPM Controller Abstraction

To be able to address and initialize the SPM controller, we abstract it as a two dimensional array added to the program IR: $SPM\_CONT[N][8]$. The first dimension of the array represents the entries for both the object and pointer tables. In terms of command fields, $N = PTR, TBL\_IDX$. The second dimension is the command type CMD_TYPE. Hence, there are 8 possible commands for each table entry. However, not all of them are valid, *e.g.* **SETSIZE** does not apply for pointers.

Based on the SPM controller abstraction, any allocation command can be represented in LLVM IR as an access to a two dimensional array. For instance, **ALLOC**P $5, 0, 100$ allocates the object at entry 5 to the SPM address 100. In C, this can be written as: $SPM\_CONT[5][1] = 100$ which will be translated in LLVM IR to:

```
1 %alloc_ptr = getelementptr [32 x [8 x i32]], [32 x [8 x i32]]* ↩
    @SPM_CONT, i32 0, i32 5, i32 1
2 store i32 100, i32* %alloc_ptr
```

Similarly, **GETADDR** $1, 4$ requests the current address of the pointee of the pointer in entry 4 of the pointer table. It can represented to C as: $addr = SPM\_CONT[12][5]$. Then, compiled in LLVM IR as:

```
1 %getaddr_ptr = getelementptr [32 x [8 x i32]], [32 x [8 x i32]]* ↩
    @SPM_CONT, i32 0, i32 12, i32 5
2 %addr = load i32, i32* %getaddr_ptr
```

This representation can also be used to manually insert the allocation commands in the high level source code.

SPM Controller Initialization

SPM controller is initialized as a global array in the program. For instance, to initialize entry 0 in the SPM controller to the main memory address a_x of object x and size s_x, we initialize two elements in the array: $SPM\_CONT[0][6] = a_x$, $SPM_CONT[0][7] = s_x$. The element $SPM\_CONT[0][6]$ will be decoded by the SPM controller to **SETMM** command and $SPM\_CONT[0][7]$ will be decoded to **SETSIZE** command. The initialization can be inserted in the beginning of the code or the array initialization is stored in the data segment and copied from the main memory before the program execution.

A.2 Control Unit

The control unit is responsible for executing the allocation commands and managing the Direct Memory Access (DMA). It updates the object and pointer tables and the command and allocation queues accordingly.

A.2.1 Command Execution

Allocate

This command allocates an object or a pointer to the scratchpad and schedules a prefetch from the main memory if needed. The command passes hints to the control unit about prefetching (P) and write-back (W) of the allocated object.

ALLOC*XX* *TBL_IDX, PTR, MEM_ADDR*

After the command is decoded, it is pushed to the command queue as it is non-blocking. Then, it is executed as following:

- If *PTR* flag is set:
  - *TBL_IDX* is used to access the pointer table.
  - The pointer entry is checked for *ALIASED* flag. If the flag is not set, the command is dismissed as the pointer is not referring to a valid object entry. Otherwise, the *OBJ_TBL_IDX* is used to access the object table.
- If *PTR* flag is not set, *TBL_IDX* is used to access the object table directly.
- According to the status flags of the object entry, the object entry is updated and an allocation operation can be scheduled:
  - *A = 0, PF_OP = 0, WB_OP = 0*: If the command has the *P*-hint, a prefetch operation is pushed to the allocation queue and *PF_OP = 1*, otherwise *A = 1*
  - *A = 1, PF_OP = 0, WB_OP = 0/1*: No change →the object is already allocated.
  - *A = 0, PF_OP = 1, WB_OP = 0*: No change →the object is scheduled for a prefetch.
  - *A = 0, PF_OP = 1, WB_OP = 1*: No change →a previous copy of the object will be written back and then the object will be prefetched to a new address in the scratchpad.
  - *A = 0, PF_OP = 0, WB_OP = 1*: if *MEM_ADDR* matches the *SPM_ADDR* in the object entry, then the write-back operation will be canceled as the object will be allocated in its current address space and *A = 1, WB_OP = 0*. If the addresses do not match, a prefetch operation will be scheduled it the command has the *P*-hint and *PF_OP = 1, WB_OP = 1*, otherwise *A = 1, WB_OP = 1*.
- The number of users of the object in the scratchpad will be incremented by one (*Users++*).
- If the command has the *W*-hint, the *WB* flag is set.

De-allocate

This command de-allocates an object or a pointer from the SPM and schedules a write-back to the main memory if needed.

DEALLOC *TBL_IDX, PTR*

The command is non-blocking, so it is pushed to the command queue to be executed.

- If *PTR* flag is set:
  - *TBL_IDX* is used to access the pointer table.
  - The pointer entry is checked for *ALIASED* flag. If the flag is not set, the command is dismissed as the pointer is not referring to a valid object entry. Otherwise, the *OBJ_TBL_IDX* is used to access the object table.
- If *PTR* flag is not set, *TBL_IDX* is used to access the object table directly.
- According to the status flags of the object entry, the object entry is updated and an allocation operation can be scheduled:
  - *A = 1, PF_OP = 0, WB_OP = 0*: if *USERS = 1*, a write-back operation is scheduled and *A = 0, WB_OP = 1* when *toWB_OP = 1, Dirty = 1*, otherwise *A = 0* and the object is de-allocated with copying back to the main memory.
  - *A = 0, PF_OP = 1, WB_OP = 0/1*: if *USERS = 1*, the prefetch operation is canceled and *A = 0, PF_OP = 0*.
  - *A = 1, PF_OP = 0, WB_OP = 1*: if *USERS = 1*, then *A = 0* as a previous write-back operation is already scheduled.
- If *USERS = 1*, the two flags *toWrite-back, Dirty* are reset.
- The number of users of the object in the scratchpad will be decremented by one (*Users–*).

Get Address

This command is used to obtain the current address of an object or a pointer. It is a blocking command that stalls the execution if there is a scheduled memory transfer (prefetch or write-back).

GETADDR *TBL_IDX, PTR*

- If *PTR* flag is set:
  - *TBL_IDX* is used to access the pointer table.
  - The pointer entry is checked for *ALIASED* flag. If the flag is not set, *MM_ADDR* is returned as the pointer has no match in the object table. Otherwise, OBJ_TBL_IDX is used to access the object table.
- If *PTR* flag is not set, *TBL_IDX* is used to access the object table directly.
- According to the status flags of the object entry, the object entry is updated and an address is returned:
  - *A = 0, PF_OP = 0, WB_OP = 0*: the object is not allocated and no memory transfer is scheduled. *MM_ADDR* is returned as it the current address of the object.
  - *A = 0, PF_OP = 1, WB_OP = 0/1*: the last scheduled memory transfer is a prefetch, so the command has to wait till the transfer is done and *A = 1, PF_OP = 0, WB_OP = 0*. Then, the *SPM_ADDR* is returned and *Dirty = 1* to indicate that the object has been used.
  - *A = 1, PF_OP = 0, WB_OP = 0/1*: the object is allocated in the scratchpad, so the *SPM_ADDR* is returned and *Dirty = 1* to indicate that the object has been used.
  - *A = 0, PF_OP = 0, WB_OP = 1*: the last scheduled memory transfer is a write-back, so the command has to wait till the transfer is done and *A = 0, PF_OP = 0, WB_OP = 0*. Then, the *MM_ADDR* is returned.

Set Pointer

This command is essential to be able to handle the pointer aliasing during run-time. The command is inserted before the **ALLOC/DEALLOC/GETADDR** commands of a pointer to check if there is an aliasing object in the object table.

SETPTR *TBL_IDX, MEM_ADDR*

It sets the entry at TBL_IDX in the pointer table with the main memory address (*MEM_ADDR*). Then, it starts a comparison with all the objects in the object table.

The command is non-blocking, so it is pushed to the command queue till executed by the control unit.

- *MEM_ADDR* is compared to the range and *[MM_ADDR:MM_ADDR+SIZE-1]* of the valid entries of the object table until a match is found or all entries are checked. The comparison can be implemented in one cycle using a comparator for each entry or over multiple-cycles using less number of comparators. The one cycle implementation is preferable if the size of the object table is small.
- If a match is found for *MEM_ADDR*, the fields in the entry at *TBL_IDX* in the pointer table are set to *MM_ADDR* = *MEM_ADDR*, *TBL_IDX* = object table entry that aliases with the pointer and *ALIASED* = 1.
- If no object in the object table is found that matches the *MEM_ADDR*, the fields in the entry at *TBL_IDX* in the pointer table are set to *MM_ADDR* = *MEM_ADDR* and *ALIASED* = 0.

Set Main Memory Address

The command sets the main memory address field (MM_ADDR) of the entry at TBL_IDX in the object table.

SETADDR *TBL_IDX, MEM_ADDR*

Set Size

The command sets the size field ($SIZE$) of the entry at TBL_IDX in the object table. If *SIZE* is 0, this invalidates the object entry and resets the flag V, other wise it sets V to mark the entry as valid.

SETSIZE *TBL_IDX, SIZE*

A.2.2 DMA Management

The control unit reads the operations from the allocation queue in FIFO order and starts a DMA transfer if required. The object table at index OBJ_TBL_IDX from the operation entry to be executed is first checked for the scheduled operation. If OP_TYPE is a prefetch and the flag $PF\_OP = 0$, the operation is canceled. Similarly, if OP_TYPE is a write-back and the flag $WB\_OP = 0$, the operation is canceled. Otherwise, the DMA is configured with the source, destination and size of the object. When the transfer is done, the control unit updates the object entry according to the operation type:

- Prefetch: $A = 1$, $PF\_OP = 0$.
- Write-back: $WB\_OP = 0$.

Appendix B

WCET Analysis

We discussed in Section 4.7 the intuition of using abstract interpretation to integrate our prefetching scheme in the Worst-Case Execution Time (WCET) analysis. In the appendix, we first provide required preliminaries on the underlying mathematical principles in Section B.0.1. We then formally introduce our abstraction and prove it correct in Section B.0.2.

B.0.1 Preliminaries

The theory of abstract interpretation [37] provides a formal way to describe a mathematical model for the state of the program. In this section, we base our discussion on the formulation of DFA with abstract interpretation for WCET analysis proposed in [137].

Definition 29 (Bounds for Partially Ordered Set)**.** *Consider a set A with partial order $\leq_A$. We say that an element $a \in A$ is an upper bound (lower bound) for a subset Y of A iff $\forall y \in Y : y \leq_A a$ (respectively, $a \leq_A y$). We further say that a is the unique least upper bound (greatest lower bound) for Y, and write $a = \vee_A Y$ (respectively, $a = \wedge_A Y$) iff for all other upper bounds b of Y it holds $a \leq_A b$ (respectively, $b \leq_A a$).*

For simplicity, for a set $Y = \{a, b\}$, we shall write $a \vee_A b$ ($a \wedge_A b$) as a shorthand for $\vee_A Y$ ($\wedge_A Y$).

Definition 30 (Complete Lattice)**.** *A partially ordered set $(A, \leq_A)$ is said to be a complete lattice if any subset Y of A admits both a least upper bound and a greatest lower bound.*

Observation 31 (Concrete State Set)**.** *The set $\mathcal{P}(\Sigma)$ together with the subset partial relation $\subseteq$ is a complete lattice, where $S \vee S' = S \cup S'$ and $S \wedge S' = S \cap S'$.*

The complete lattice $\big(\mathcal{P}(\Sigma), \subseteq\big)$ is used to model the "real" (concrete) state of the system; in this sense, the partial order $\subseteq$ represents a relation of generality, in the sense that if $S \subseteq S'$, we can say that S' is more general (since it contains more program states), or equivalently less precise, compared to S.

Definition 32 (Monotone Function)**.** *Let $(A, \leq_A)$ and $(B, \leq_B)$ be partially ordered sets. A function $f : A \to B$ is said to be monotone iff: $\forall a, a' \in A : a \leq_A a' \Rightarrow f(a) \leq_B f(b)$.*

Definition 33 (Abstraction)**.** *We say that a complete lattice $(D, \leq_D)$ is an abstraction for the concrete state $\big(\mathcal{P}(\Sigma), \subseteq\big)$ iff there exists a monotone function $\gamma : D \to \mathcal{P}(\Sigma)$ such that:*

$$\forall S \in \mathcal{P}(\Sigma) : \exists d \in D : S \subseteq \gamma(d). \tag{B.1}$$

γ is also called the *concretization* function of the abstraction. Since the concretization function is monotone, for every $d \leq_D d'$, it must hold: $\gamma(d) \subseteq \gamma(d')$. In other words, the partial order $\leq_D$ on D must express a relation of generality similar to the one for the concrete state. Furthermore, Equation B.1 ensures that for every concrete state S, there exists an abstract state d that "contains" S.

Based on the described framework, the MOP DFA is then carried out as follows: we first obtain an initial abstract state d_{entry} such that $S_{entry} \subseteq \gamma(d_{entry})$. We then traverse the DFG using an abstract transfer function $\hat{\mathcal{T}}_{e,\sigma} : D \to D$, which represents the abstraction of the transfer function $\mathcal{T}_{e,\sigma}$ to the abstract state D. Whenever we need to join paths for two abstract states d, d', we compute a new join state $d'' = d \vee_D d'$. After obtaining a final abstract state d_{exit} for the program, we then determine the WCET as the largest elapsed time in $\gamma(d_{exit})$. There are two fundamental advantages to this approach: 1) as discussed in the example in Section 4.7, we can represent states that cannot occur in the concrete execution of the system. 2) Since the abstract state set D is a model of the system, we can ignore program and architectural details that are too complex to handle in the analysis, albeit at the cost of decreased analysis precision. Overall, the goal is to obtain an abstract transfer function $\hat{\mathcal{T}}_{e,\sigma}$ that can be computed in a reasonable amount of time, rather than evaluating $\mathcal{T}_{e,\sigma}$ on all program states contained in a concrete state set S, which is generally computationally intractable. The following theorem states the fundamental sufficient condition on the abstract transfer function that we use in this work.

Theorem 34 (MOP II Correctness; Theorem 3.3.5 in [137])**.** *Let D be an abstraction for $\mathcal{P}(\Sigma)$. If for every edge e, the transfer function $\hat{\mathcal{T}}_{e,\sigma}$ satisfies the following property:*

$$S \subseteq \gamma(d) \Rightarrow \mathcal{T}'_{e,\sigma}(S) \subseteq \gamma(\hat{\mathcal{T}}_{e,\sigma}(d)), \tag{B.2}$$

then the MOP analysis over D using an initial state $d_{entry} : S_{entry} \subseteq \gamma(d_{entry})$ is a correct analysis for the program, meaning that $S_{exit} \subseteq \gamma(d_{exit})$.

Intuitively, Equation B.2 means that applying the abstract transfer function $\hat{\mathcal{T}}_{e,\sigma}(d)$ results in a state that is more general compared to applying the concrete transfer function $\mathcal{T}'_{e,\sigma}$. In turn, this implies that if we start with an initial abstract state d_{entry} that is more general than the initial concrete state S_{entry}, we will obtain a final abstract state d_{exit} that is still more general (hence, a safe approximation) than the final concrete state S_{exit}.

B.0.2 Abstract State Model

We detail our abstraction for WCET analysis in this section. Since our goal is to show how to handle the scratchpad controller, for the sake of simplicity we will consider the simplest possible model for the rest of the hardware system, namely, an in-order CPU where the number of clock cycles required to process the instructions in each basic block does not depend on previous block (i.e., no pipelining effects between blocks), and memory accesses stall the CPU. Under this model, we let t_{comp} be the maximum computation time for a code block without considering the stall time due to load/store operations, t_{spm} be the maximum time for SPM accesses, and t_{mm} the maximum time for main memory accesses; the total execution time of the basic block can then be bounded as $t_{comp} + t_{spm} + t_{mm}$ plus the GETADDR blocking time. We will also not include any memory state (i.e., value assigned to variables) in the abstract state. However, please note that both memory state and other architectural states could be included in the abstract state following well-established WCET analysis techniques [137]. Finally, again for simplicity and to match our implementation, we will assume that all scratchpad commands can be executed in one clock cycle, i.e. we do not handle the command queue. However, if the alias check takes multiple clock cycles, the effects of the command queue could be handled by adding an additional timer to the abstract state, as it will become clearer in the rest of the discussion.

Based on Theorem 34, in the rest of the section we provide the following steps:

- define abstract state set D and its partial order $\leq_D$. This is done in Definitions 37 and 40;
- prove that $(D, \leq_D)$ is a complete lattice (Lemma 41);
- define concretization function γ (Definition 44), prove that it is monotone and it satisfies Equation B.1 (Lemma 45);

- define $\hat{\mathcal{T}}_{e,\sigma}(d)$ (Definition 51);
- finally, prove that Equation B.2 holds (Theorem 57).

This ensures that all assumptions in Theorem 34 hold, hence proving that the MOP analysis over the described abstraction is correct.

We begin by providing a definition for the program state $\mathbf{s}$ that will be used throughout the section. In what follows, let t^x_{dma} denote the time required for the DMA operation (prefetch or write-back) for an object x, while for a pointer x it denotes the maximum DMA operation time of any object pointed to by x.

Definition 35 (Trailing DMA time)**.** *We define the* trailing length *of any DMA operation in the allocation queue as follows:*

- *the trailing length of the operation at the front of the queue is the time remaining to complete the operation;*
- *the trailing length for any other operation on an object v is t^v_{dma} plus the trailing length of the operation immediately ahead in the queue.*

Essentially, the trailing length for an operation represents the maximum DMA time required to complete it, considering that operations in the allocation queue are served in FIFO order.

Definition 36 (Abstract Timers)**.** *Let $\mathcal{V}$ be the set of all objects and $\mathcal{A}$ be the set of all addresses assigned to objects/pointers by the address assignment algorithm. We define the following set of* abstract timers*:*

- *For an object v, the abstract prefetch timer $\mathbb{T}^{pr}_v$ is a single value $\mathbb{T}^{pr}_v.t \in \mathbb{N}$.*
- *For an object v, the abstract write-back timer $\mathbb{T}^{wb}_v$ is a tuple $\{\mathbb{T}^{pr}_v.t, \mathbb{T}^{pr}_v.A\}$ with $\mathbb{T}^{wb}_v.t \in \mathbb{N}$ and $\mathbb{T}^{wb}_v.A \in \mathcal{P}(\mathcal{A})$.*
- *For a pointer p, the abstract prefetch timer $\mathbb{T}^{pr}_p$ is a tuple $\{\mathbb{T}^{pr}_p.t, \mathbb{T}^{pr}_p.V\}$ with $\mathbb{T}^{pr}_p.t \in \mathbb{N}$ and $\mathbb{T}^{pr}_p.V \in \mathcal{P}(\mathcal{V})$.*
- *For a pointer p, the abstract write-back timer $\mathbb{T}^{wb}_p$ is a tuple $\{\mathbb{T}^{wb}_p.t, \mathbb{T}^{wb}_p.A, \mathbb{T}^{wb}_p.V\}$ with $\mathbb{T}^{wb}_p.t \in \mathbb{N}$, $\mathbb{T}^{wb}_p.V \in \mathcal{P}(\mathcal{V})$ and $\mathbb{T}^{wb}_p.A \in \mathcal{P}(\mathcal{A})$.*

For simplicity, we use the symbol $\mathbb{T}$ *to denote any abstract timer, defining* $\mathbb{T}.V = \{v\}$ *for the timers of object* v*, and* $\mathbb{T}.A = \emptyset$ *for prefetch timers. We call the value* t *the timer's* trailing length*,* A *its* address set*, and* V *its* points-to set*. We write* $\mathbb{T} = 0$ *to mean* $\mathbb{T}.t = 0, \mathbb{T}.A = \emptyset, \mathbb{T}.V = \emptyset$*.*

Definition 37 (DMA Abstraction)**.** *An abstract state* d *is a tuple* $d = \{d.t, \ldots, d.\mathbb{T}^{pr}_{v_i}, d.\mathbb{T}^{wb}_{v_i}, \ldots, d.\mathbb{T}^{pr}_{p_k}, d.\mathbb{T}^{wb}_{p_k}, \ldots, \}$*, comprising one prefetch and one write-back timer for each object* v_i *and each pointer* p_k*. We call* $d.t \in \mathbb{N}$ *the* abstract elapsed time*. Let* D *be the set of all abstract states.*

Intuitively, an abstract state is composed of an elapsed time, which is an upper bound to the time elapsed since the beginning of the program, and a prefetch and write-back timer for every object and every pointer. For all timers, the trailing length $\mathbb{T}.t$ models the trailing length of any prefetch or write-back operation for that object/pointers. In essence, our abstraction models the cumulative DMA time required for the operations of a given object/pointer, rather than the ordered list of DMA operations. Since the same pointer can point to different objects during its lifetime, pointer timers must also store the point-to list $\mathbb{T}.V$. Finally, write-back timers additionally store the address at which the object/pointer was allocated. As explained in Section 4.4.3, this is required to cancel a write-back operation if the same object is allocated at the same address. To allow the MOP procedure, $\mathbb{T}.A$ must be defined as a set of addresses (i.e., an element of the powerset of $\mathcal{A}$) so that the union over different paths can be computed.

To simplify notation, we further define the following intuitive operations on timers.

Definition 38 (Operations on Timers)**.** *We define the following operations, where* $\Delta \in \mathbb{N}$*.*

- $\mathbb{T}' = \mathbb{T} + \Delta$ *returns the timer* $\mathbb{T}'$ *where* $\mathbb{T}.t$ *is incremented by* Δ*.*
- $\mathbb{T}' = \mathbb{T} - \Delta$ *returns the timer* $\mathbb{T}'$ *where* $\mathbb{T}.t$ *is decremented by* Δ *if* $\Delta < \mathbb{T}.t$*; otherwise,* $\mathbb{T}' = 0$*.*
- $\mathbb{T}' = \mathbb{T} \setminus v$ *returns the timer* $\mathbb{T}'$ *where* $\mathbb{T}'.V = \mathbb{T}.V \setminus \{v\}$ *if* $\mathbb{T}.V \neq \{v\}$*; otherwise,* $\mathbb{T}' = 0$*.*
- $\mathbb{T}'' = \mathbb{T} \vee \mathbb{T}'$ *where* $\mathbb{T}''.t = \max(\mathbb{T}.t, \mathbb{T}'.t), \mathbb{T}''.A = \mathbb{T}.A \cup \mathbb{T}'.A, \mathbb{T}''.V = \mathbb{T}.V \cup \mathbb{T}'.V$*.*
- $\mathbb{T}'' = \mathbb{T} \wedge \mathbb{T}'$ *where* $\mathbb{T}''.t = \min(\mathbb{T}.t, \mathbb{T}'.t), \mathbb{T}''.A = \mathbb{T}.A \cap \mathbb{T}'.A, \mathbb{T}''.V = \mathbb{T}.V \cap \mathbb{T}'.V$*.*

Definition 39 (Partial Order for Abstract Timers)**.** *We define the partial order* $\leq$ *on abstract timers such that for any two timers* $\mathbb{T}, \mathbb{T}'$ *for the same object/pointer and operation (prefetch or writeback):*

$$\mathbb{T} \leq \mathbb{T}' \Leftrightarrow \mathbb{T}.t \leq \mathbb{T}'.t \text{ and } \mathbb{T}.V \subseteq \mathbb{T}'.V \text{ and } \mathbb{T}.A \subseteq \mathbb{T}'.A. \tag{B.3}$$

Definition 40 (Partial Order on DMA Abstraction)**.** *We define the partial order* $\leq_D$ *on the abstraction* D *such that for any two abstract states* d, d'*:*

$$d \leq_D d' \Leftrightarrow d.t \leq d'.t \text{ and } \forall \text{ timer } \mathbb{T} : d.\mathbb{T} + d.t \leq d'.\mathbb{T} + d'.t. \tag{B.4}$$

Since $\subseteq$ is a partial order on any set, and $\leq$ is a total order on $\mathbb{N}$, it is trivial to see that $\leq_D$ is also a partial order. Intuitively, d' is larger than d if and only if it has both a larger elapsed time, and a larger value of elapsed time plus timer for every object and pointer; following the example in Section 4.7, this implies that d' is guaranteed to cause a larger delay on successive basic blocks compared to d. We next show that $(D, \leq_D)$ is a complete lattice.

Lemma 41. $(D, \leq_D)$ *is a complete lattice, where for any two abstract states* d, d' *with* $t_{\max} = \max(d.t, d'.t)$ *and* $t_{\min} = \min(d.t, d'.t)$*:*

- *for* $d'' = d \vee_D d'$ *it holds* $d''.t = t_{\max}$ *and for any timer* $\mathbb{T} : d''.\mathbb{T} = \big(d.\mathbb{T} - (t_{\max} - d.t)\big) \vee \big(d'.\mathbb{T} - (t_{\max} - d'.t)\big)$*;*
- *for* $d'' = d \wedge_D d'$ *it holds* $d''.t = t_{\min}$ *and for any timer* $\mathbb{T} : d''.\mathbb{T} = \big(d.\mathbb{T} + (d.t - t_{\min})\big) \wedge \big(d'.\mathbb{T} + (d'.t - t_{\min})\big)$*.*

Proof. We formally prove that $(D, \leq_D)$ is a lattice, i.e., for any two elements d and d', $d \vee_D d'$ is the least upper bound to d, d' and $d \wedge_D d'$ is the greatest lower bound to d, d'; the completeness of the lattice (i.e., the fact that we can find a least upper bound and greatest lower bound for any subset Y of D) then follows from the completeness of the sets $\mathbb{N}$, $\mathcal{P}(\mathcal{A})$, $\mathcal{P}(\mathcal{V})$ used to represent times and address/object sets.

Consider $d'' = d \vee_D d'$. Since $d''.t = \max(d.t, d'.t)$, $d''.t$ is the smallest value that satisfies the partial order constraints $d.t \leq d''.t$ and $d'.t \leq d''.t$. Similarly, since $d''.\mathbb{T}.A = d.\mathbb{T}.A \cup d'.\mathbb{T}.A$, it is the smallest set that satisfies $d.\mathbb{T}.A \subseteq d''.\mathbb{T}.A$ and $d'.\mathbb{T}.A \subseteq d''.\mathbb{T}.A$; the same argument applies to $\mathbb{T}.V$. Next, assume without loss of generality that $d.t \leq d'.t$. Based on Definition 38 we then obtain: $d''.\mathbb{T}.t + d''.t = \max\big(d.\mathbb{T}.t - (t_{\max} - d.t), d'.\mathbb{T}.t - (t_{\max} - d'.t)\big) + d''.t = \max(d.\mathbb{T}.t - d'.t + d.t, d'.\mathbb{T}.t) + d'.t = \max(d.\mathbb{T}.t + d.t, d'.\mathbb{T}.t + d'.t)$;

hence, $d''.\mathbb{T}.t$ is the smallest value that satisfies the partial order constraint for timer trailing length (Equation B.4), concluding the proof for the least upper bound.

We omit the proof for the greatest lower bound as it is specular to the least upper bound. □

Note that the operator $d \vee_D d'$ computes the same upper bound as in Equations 4.19, 4.20.

Definition 42 (Generation of DMA Operations)**.** *Given an abstract state d and a DMA operation for object v in the allocation queue for a program state $\mathbf{s}$, we say that a timer $d.\mathbb{T}$ can* generate *the operation if it is of the same type (prefetch or write-back) as the timer and its trailing length is less than or equal to $d.\mathbb{T}.t$; additionally, v must be contained in $d.\mathbb{T}.V$; finally, for a write-back timer, the SPM address of the operation must be contained in $d.\mathbb{T}.A$.*

Observation 43. *By definition, if a timer $\mathbb{T}$ can generate a DMA operation, then any timer $\mathbb{T}' : \mathbb{T} \leq \mathbb{T}'$ can also generate that operation.*

Definition 44 (Concretization Function)**.** *Given any abstract state d, the concrete state $S = \gamma(d)$ is the set of all feasible program states $\mathbf{s}$ for which:*

- *the elapsed time t since the beginning of the program is less than or equal to $d.t$; let $\Delta = d.t - t$;*
- *for any DMA operation in the allocation queue with trailing length greater than Δ, there is at least one timer $d.\mathbb{T}$ such that $d.\mathbb{T} + \Delta$ can generate the operation.*

Definitions 42, 44 are key to understand how the abstraction works. In essence, the key idea is that adding Δ units of time to the elapsed time is always worse than increasing the trailing lengths of timers by the same amount Δ. Hence, if the difference between elapsed times for the abstract and program state is Δ, the program state can contain any DMA operation with trailing length up to Δ; while for operations with larger trailing length $k > \Delta$, a timer of the correct type/address/points-to set is required with $k \leq d.\mathbb{T}.t + \Delta$.

Lemma 45. *The DMA Abstraction D is a valid abstraction for $\mathcal{P}(\Sigma)$.*

Proof. We first show that Equation B.1 holds. Given a concrete state S, we construct the abstract state d such that $d.t$ is an upper bound to the elapsed time of any program state $\mathbf{s} \in S$, and for any object v: $d.\mathbb{T}_v^{pr}.t$ is an upper bound to the trailing length of any prefetch

operation for v in $\mathbf{s}$; $d.\mathbb{T}_v^{wb}.t$ is an an upper bound to the trailing length and $d.\mathbb{T}_v^{wb}.A$ is the union of the SPM addresses of any write-back operation for v in $\mathbf{s}$. It then immediately follows that for any $\mathbf{s} \in S$, the elapsed time for $\mathbf{s}$ is less than or equal to $d.t$ and every DMA operation is generated by a timer in d; hence, based on Definition 44, we have $\mathbf{s} \in \gamma(d)$ and thus $S \subseteq \gamma(d)$.

It remains to show that γ is monotone. Consider two abstract states $d \leq_D d'$; we have to show that $\mathbf{s} \in \gamma(d) \Rightarrow \mathbf{s} \in \gamma(d')$. Let t be the elapsed time of $\mathbf{s}$; then it must hold $t \leq d.t \leq d'.t$. Define $\Delta = d.t - t$; since $\Delta \geq 0$, based on Definition 40 it must hold for any timer: $d.\mathbb{T} + \Delta \leq d'.\mathbb{T} + \Delta + (d.t' - d.t)$. Hence, if an operation of $\mathbf{s}$ can be generated by timer $d.\mathbb{T} + \Delta$, it can also be generated by timer $d'.\mathbb{T} + \Delta + (d.t' - d.t)$. This concludes the proof. □

It now remains to define the abstract transfer function $\hat{\mathcal{T}}_{e,\sigma}$, and prove Equation B.2. We start by defining a set of helper functions. For simplicity of notation, we will consider three-valued logic variables which can assume one of the following values: {True, False, Unknown}. In particular, for each ALLOC/DEALLOC command on an object/pointer x we define an *exec* flag with the following meaning: if *exec* = True, then the value of the USERS field for the object pointed to by x is guaranteed to be 0 before an ALLOC and 1 before a DEALLOC; this implies that the corresponding command is effectively executed. If instead *exec* = False, USERS is guaranteed to be greater than 0/1 for an ALLOC/DEALLOC; hence, the command does not cause any state change. Finally, if *exec* = Unknown, then no assumptions on the value of the USERS can be made. In our approach, the *exec* flags are statically computed by the allocation algorithm: for a given allocation, if there is no enclosing allocation (in an ancestor region) on the same object, then *exec* = True. If there is an enclosing allocation which is guaranteed to be on the same object, then *exec* = False. Otherwise, *exec* = Unknown; note this case is required to handle pointers where the value of USERS can only be determine at run-time.

Definition 46 (ALLOC function)**.** *The function $d' = ALLOC(d, x, a, BB, pr, exec)$, where x is an object or pointer, a an address, BB a basic block, pr a binary flag and exec a three-valued flag, modifies the abstract state d into d' by performing the following steps:*

1. *if exec = True and $d.\mathbb{T}_x^{wb}.A = \{a\}$ and x points to a single object v in BB, then $d'.\mathbb{T}_x^{wb} = d.\mathbb{T}_x^{wb} \setminus v$;*
2. *then if $pr = 1$ and exec $\neq$ False, $d'.\mathbb{T}_x^{pr}.t$ is set to the maximum trailing length of any timer plus t_{dma}^x and $d'.\mathbb{T}_x^{pr}.V$ is the union of $d.\mathbb{T}_x^{pr}.V$ and the points-to list of x in BB.*

Definition 47 (DEALLOC function)**.** *The function* $d' = DEALLOC(d, x, a, BB, wb)$*, where* wb *is a binary flag, modifies the abstract state* d *into* d' *by performing the following steps:*

1. *if* $exec =$ *True and* x *points to a single object* v *in* BB*, then* $d'.\mathbb{T}_x^{pr} = d.\mathbb{T}_x^{pr} \setminus v$*;*
2. *then if* $wb = 1$ *and* $exec \neq$ *False,* $d'.\mathbb{T}_x^{wb}.t$ *is set to the maximum trailing length of any timer plus* t_{dma}^x*;* $d'.\mathbb{T}_x^{wb}.A = d.\mathbb{T}_x^{wb}.A \cup \{a\}$*; and* $d'.\mathbb{T}_x^{wb}.V$ *is the union of* $d.\mathbb{T}_x^{wb}.V$ *and the points-to list of* x *in* BB*.*

Functions $ALLOC$ and $DEALLOC$ are applied every time an ALLOC or DEALLOC command is encountered in a basic block. Based on the discussion in Section 4.4.4, the ALLOC command is guaranteed to cancel a write-back operation on the same object if the two allocations target the same address in the SPM. This is performed in the $ALLOC$ function by checking that the address of the write-back timer coincides with the address of the ALLOC, and removing the pointed-to object from the points-to set of the write-back timer. Note that for an object timer, this is equivalent to resetting the timer to 0, since by definition every object points to itself only; however, for a pointer we can do so only if there is no ambiguity in the points-to list (i.e., the pointer points to a single object in b). Then, if $pr = 1$, meaning that a prefetch operation must be scheduled, the function intuitively "appends" a new operation of length t_{dma}^x to the end of the allocation queue by setting the prefetch timer to the maximum trailing length in the queue plus t_{dma}^x. The behavior of the $DEALLOC$ function is equivalent. Finally, all steps are dependent on the value of $exec$: to conservatively capture the worst case, we add a timer if the command could be executed ($exec =$ True or Unknown), but we remove a timer only if we are certain that the command is executed ($exec =$ True).

Definition 48 (ELAPSE function)**.** *The function* $d' = ELAPSE(d, \Delta, \Lambda)$*, with* $\Delta, \Lambda \in \mathbb{N}$*, modifies the abstract state* d *into* d' *such that:* $d'.t = d.t + \Delta$ *and* $\forall$ *timer* $\mathbb{T}$*:* $d'.\mathbb{T} = d.\mathbb{T} - \Lambda$*.*

Intuitively, the function $ELAPSE$ is used to increment time: the elapsed time is increased by Δ and every abstract timer is decreased by an amount Λ. Note that $\Lambda \leq \Delta$, since the DMA unit is stalled while the CPU accesses main memory.

Definition 49 (GETADDR stall)**.** *Given an abstract state* d*, we say that a GETADDR command on object/pointer* x *in basic block* BB stalls *on a timer* $d.\mathbb{T}$ *iff the intersection of the points-to list of* x *in* BB *and* $d.\mathbb{T}.V$ *is not empty.*

Definition 50 (Depending ALLOC/DEALLOC)**.** *We say that an ALLOC/DEALLOC command for object/pointer x in basic block BB* depends *on a GETADDR command for object/pointer y in the same basic block iff the intersection of the points-to lists of x and y in BB is not empty.*

Intuitively, if a GETADDR stalls on a timer, then in the worst case we need to wait until that timer elapses before the GETADDR can proceed. Similarly, if an ALLOC/DEALLOC depends on GETADDR, then in the worst case the GETADDR will stall on any DMA operation added by the ALLOC/DEALLOC.

Definition 51 (Abstract Transfer Function)**.** *Consider a CFG edge $e : BB \to BB'$. Let the execution for BB along e be divided into a set of consecutive* intervals*, such that the set of intervals cover all executed instructions but any change to the state of the SPM controller (including stalling the core due to a blocking command) only happens between one interval and the next. Then abstract transfer function $\hat{\mathcal{T}}_{e,\sigma}(d)$ is computed by applying an iterative set of transformations of the abstract state d using functions $ELAPSE$, $ALLOC$, $DEALLOC$ based on the order of intervals, ALLOC, DEALLOC and GETADDR commands in BB:*

- *For each interval, let t_{comp}, t_{mm} and t_{spm} be the maximum computation time, main memory and SPM time for the interval, assuming that all load/stores to any object v_i (pointer p_k) access main memory iff $spm_{r_j}^{v_i} = 0$ (respectively, $spm_{r_j}^{p_k} = 0$) for all regions that contain BB. Then transform the state into $ELAPSE(d, t_{comp}^{BB} + t_{mm}^{BB} + t_{spm}^{BB}, t_{comp}^{BB} + t_{spm}^{BB})$.*
- *For a GETADDR command on object/pointer x, transform the state into $ELAPSE(d, \Delta, \Delta)$, where Δ is the maximum trailing length of any timer on which the GETADDR stalls.*
- *For an ALLOC command on object/pointer x, transform the state into $ALLOC(d, x, a, BB, pr, exec)$, where a is the SPM address of the ALLOC and $pr = 1$ if the P flag is set.*
- *For a DEALLOC command on object/pointer x, transform the state into $DEALLOC(d, x, a, BB, wb, exec)$, where a is the SPM address of the DEALLOC and $wb = 1$ if the W flag is set in the ALLOC command corresponding to this DEALLOC.*

Note that in Definition 51, the execution of the code within the basic block is modeled by advancing elapsed time by the maximum execution time $t_{comp}+t_{mm}+t_{spm}$ and decreasing

all timers by $t_{comp} + t_{spm}$, which is the time that DMA operations can proceed in parallel with the CPU assuming a dual-ported SPM. If the SPM is single-ported, we amend the definition to instead decrease the timers by t_{comp} only.

We are now ready to prove our main Theorem 57, which shows that Equation B.2 holds for the described abstraction, hence concluding our proof obligations. Due to its complexity, we first present the intuition behind the proof and introduce several supporting lemmas. We first prove that the equation holds assuming that the order of ALLOC/DEALLOC/GETADDR commands and other instructions in the basic block is known. Intuitively, we construct a chain of abstract and program states, starting at the beginning of the basic block until its end; each successive pairs of states $d^i, \mathbf{s}^i$ and $d^{i+1}, \mathbf{s}^{i+1}$ represent the state changes caused by the execution of an SPM commands, or time elapsed executing instructions. In particular, in Lemmas 53-56 we prove that at each step in the chain $\mathbf{s}^i \in \gamma(d^i) \Rightarrow \mathbf{s}^i \in \gamma(d^i)$; this ensures that the abstract state always remains more general than the concrete state, as required in Equation B.2.

Lemma 52. *Consider a DEALLOC command in basic block BB for object/pointer x, and let v be the object pointed to by x in BB. If for v it holds $USERS = 1$ before executing the DEALLOC, then the WB flag for v is equal to the W flag for the corresponding ALLOC command.*

Proof. Since allocations for objects/pointers that might point to the same object must be fully nested, if $USERS = 1$ before the DEALLOC, then it must have hold $USERS = 0$ before the corresponding ALLOC; hence, the value of the WB flag after the ALLOC command is equal to the W flag. Furthermore, any nested allocation on the same object v cannot modify the WB flag, given that after the original ALLOC it holds $USERS = 1$ for v. Hence, the value of the WB flag before the DEALLOC must still be equal to the W flag for the corresponding ALLOC. □

Lemma 53. *Let $\mathbf{s}$ be the program state after the instruction(s) for an ALLOC command has been decoded and processed, but before any change to the state of the SPM controller is made, and let $\mathbf{s}'$ be the state after the changes (if any). Furthermore, let d' be computed based on abstract state d according to Definition 46, where $x, a, BB, pr, exec$ are determined based on the ALLOC command. Then $\mathbf{s} \in \gamma(d) \Rightarrow \mathbf{s}' \in \gamma(d')$.*

Proof. By definition, no instruction is processed between $\mathbf{s}, \mathbf{s}'$, hence no time elapses and $\mathbf{s}.t = \mathbf{s}'.t$. Furthermore by Definition 46, we have $d.t = d'.t$; hence, $\Delta = d.t - \mathbf{s}.t = \Delta' = d.t - \mathbf{s}'.t$. Therefore, to show $\mathbf{s}' \in \gamma(d')$, we only need to prove that the timers in d' generate all DMA operations in $\mathbf{s}'$ with trailing length greater than $\Delta = \Delta'$. Hence,

consider changes to the list of DMA operations between $\mathbf{s}$ and $\mathbf{s}'$ and to the values of timers between d and d'. If a DMA operation is removed, then all DMA operations in $\mathbf{s}'$ must also be in $\mathbf{s}$, except that operations in $\mathbf{s}'$ might have smaller trailing length (if the removed operation was ahead in the queue). Hence, they can still be generated by the abstract state. Similarly, if a timer $\mathbb{T}$ is changed such that $d.\mathbb{T} \leq d'.\mathbb{T}$, then all operations generated by $\mathbb{T}$ in $\mathbf{s}$ can also be generated in $\mathbf{s}'$ (Observation 43). In summary, we only need to prove that the inclusion $\mathbf{s}' \in \gamma(d')$ is maintained for the following two changes to the program and abstract state: a DMA operation is added, or a timer $\mathbb{T}$ is changed and $d.\mathbb{T} \not\leq d'.\mathbb{T}$; we call the second case a *timer removal*.

Timer removal: Note that for step 2 in Definition 46, it holds $d.\mathbb{T} \leq d'.\mathbb{T}$ by construction. Hence, we only consider step 1, where $d'.\mathbb{T}^{wb}_x = d.\mathbb{T}^{wb}_x \setminus v$ if $exec = \text{True}$ and $d.\mathbb{T}^{wb}_x.A = \{a\}$ and x points to a single object v in BB. To prove that the inclusion $\mathbf{s}' \in \gamma(d')$ is maintained, we show that any DMA operation on object v generated by $d.\mathbb{T}^{wb}_x$ in $\mathbf{s}$ must be removed in $\mathbf{s}'$. By assumption, any such operation must be a write-back at the same address a as the ALLOC, the ALLOC command is for the same object v as the operation, and the command is executed ($exec = \text{True}$); hence, based on Section 4.4 the ALLOC command will indeed cancel the DMA operation.

Operation insertion: Assume that a prefetch operation for v is inserted in the allocation queue (potentially after canceling a write-back). Based on Section 4.4, the following must then be true: the P flag is set, $USERS = 0$ for v before the ALLOC, and the points-to list of x in BB must include v. This implies $pr = 1$ and $exec \neq \text{False}$, hence, $\mathbb{T}^{pr}_x$ is modified in step 2 of Definition 48. Now let k be the maximum trailing length of any DMA operation in $\mathbf{s}$ before the write-back removal (if any), and K be the maximum trailing length of any timer in d. Based on Definition 44, it must hold: $k \leq K + \Delta$. Similarly, let $\bar{k}, \bar{K}$ be the maximum trailing lengths after the write-back removal: based on the previous timer removal case, if a timer is reset in the abstract state, then the corresponding operation is removed from the program state, thus it also holds $\bar{k} \leq \bar{K} + \Delta$. The trailing length of the appended prefetch operation for v in $\mathbf{s}'$ is then $\bar{k} + t^v_{dma}$, while based on Definition 46 for d' we set the timer $d'.\mathbb{T}^{pr}_x.t = \bar{K} + t^x_{dma}$, where $d'.\mathbb{T}^{pr}_x.V$ is union of $d'.\mathbb{T}^{pr}_x.V$ and the points-to set for x. Since x can point to v, then $t^x_{dma} \geq t^v_{dma}$, implying $\bar{k} + t^v_{dma} \leq \bar{K} + \Delta + t^x_{dma} = d'.\mathbb{T}^{pr}_x.t + \Delta'$. Hence, the added prefetch operation in $\mathbf{s}'$ is generated by $d'.\mathbb{T}^{pr}_x$, concluding the proof. □

Lemma 54. *Let* $\mathbf{s}$ *be the program state after the instruction(s) for a DEALLOC command has been decoded and processed, but before any change to the state of the SPM controller is made, and let* $\mathbf{s}'$ *be the state after the changes (if any). Furthermore, let* d' *be computed based on abstract state* d *according to Definition 47, where* $x, a, BB, wb, exec$ *are determined based on the DEALLOC command. Then* $\mathbf{s} \in \gamma(d) \Rightarrow \mathbf{s}' \in \gamma(d')$.

Proof. Similarly to the proof of Lemma 53, we have $\Delta = d.t - \mathbf{s}.t = \Delta' = d.t - \mathbf{s}'.t$ and we only need to prove that the inclusion $\mathbf{s}' \in \gamma(d')$ is maintained for any DMA operation insertion and timer removal.

As in Lemma 53, a timer $\mathbb{T}_x^{pr}$ can only be removed in step 1; but since $exec$ = True and x points to a single object v in BB, this guarantees that any DMA operation on object v generated by $d.\mathbb{T}_x^{pr}$ in $\mathbf{s}$ must be removed in $\mathbf{s}'$. The only operation that can be inserted is a write-back in step 2. Assuming the operation is for object v, it must hold that before the DEALLOC, the WB flag for v is set, $USERS = 1$ and x points to v. Based on Lemma 52, this implies that the P flag for the corresponding ALLOC is set, hence $wb = 1$ in Definition 47. Following the same reasoning as in Lemma 53, it then follows that the added write-back operation in $\mathbf{s}'$ is generated by $d'.\mathbb{T}_x^{pr}$. □

Lemma 55. *Let* $\mathbf{s}$ *be the program state after the instruction(s) for a GETADDR command has been decoded and processed, but before any change to the state of the SPM controller (including stalling the CPU) is made, and let* $\mathbf{s}'$ *be the state after the changes (if any). Furthermore, let* d' *be computed based on abstract state* d *according to Definition 48, where* $\Delta = \Lambda$ *is the maximum trailing length of any timer in* d *on which the GETADDR stalls. Then* $\mathbf{s} \in \gamma(d) \Rightarrow \mathbf{s}' \in \gamma(d')$.

Proof. Let $\bar{\Delta}$ be the amount of time that the program stalls due to the GETADDR command. Then $\mathbf{s}'.t = \mathbf{s}.t + \bar{\Delta}$, any DMA operation with trailing length less than or equal to $\bar{\Delta}$ in $\mathbf{s}$ is removed from $\mathbf{s}'$, while all other operations have a trailing length reduced by $\bar{\Delta}$. Let also $K = d.t - \mathbf{s}.t \geq 0$. Based on the SPM controller behavior in Section 4.4, $\bar{\Delta}$ is the maximum trailing length of any DMA operation that stalls the GETADDR. We consider two cases: 1) $\bar{\Delta} \leq K$; then, there might be no timer in d that generates the maximum length operation, hence we can only assert $\Delta \geq 0$. 2) $\bar{\Delta} > K$; then, there must a timer $\mathbb{T}$ in d such that $\bar{\Delta} \leq d.\mathbb{T}.t + K$. Since this timer can generate the operation, by definition GETADDR stalls on the timer. Hence, we have $\bar{\Delta} \leq \Delta + K$. Combining the two cases we obtain:

$$\Delta \geq (\bar{\Delta} - K)^+. \tag{B.5}$$

Now consider $K' = d'.t - \mathbf{s}'.t$; to prove the inclusion of $\mathbf{s}'$ in d', we have to show that K' is non-negative. Note that based on Definition 48, we have $d'.t = d.t + \Delta$, and for each timer: $d'.\mathbb{T} = d.\mathbb{T} - \Delta$. Hence, we obtain: $K' = d.t + \Delta - \mathbf{s}.t - \bar{\Delta} = K + \Delta - \bar{\Delta}$. Substituting Equation B.5 then yields: $K' \geq K + (\bar{\Delta} - K)^+ - \bar{\Delta} \geq K + (\bar{\Delta} - K) - \bar{\Delta} = 0$. It then remains to prove that operations in $\mathbf{s}'$ can be generated by d'.

Therefore, consider any operation in $\mathbf{s}'$ with trailing length k' greater than K'; we have to prove that the operation is generated by a timer in d'. Let k be the trailing length

of the operation in $\mathbf{s}$, then $k = k' + \bar{\Delta}$. We then obtain: $k = k' + \bar{\Delta} > K' + \bar{\Delta} = K + \Delta - \bar{\Delta} + \bar{\Delta} = K + \Delta \geq K$. Since $k > K$, then there must exist a timer $\mathbb{T}$ in d that generates the operation, with $k \leq d.\mathbb{T}.t + K$. Note this implies $d.\mathbb{T}.t \geq k - K = k' + \bar{\Delta} - (K' - \Delta + \bar{\Delta}) > K' + \bar{\Delta} - K' + \Delta - \bar{\Delta} = \Delta$; since $d.\mathbb{T}.t > \Delta$, it thus holds $d.\mathbb{T}.t = d'.\mathbb{T}.t + \Delta$ (i.e., timer $\mathbb{T}$ is not reset in d'). We then obtain: $k' = k - \bar{\Delta} \leq d.\mathbb{T}.t + K - \bar{\Delta} = (d'.\mathbb{T}.t + \Delta) + (K' - \Delta + \bar{\Delta}) - \bar{\Delta} = d'.\mathbb{T}.t + K'$. Therefore, the trailing length of $d'.\mathbb{T}$ is sufficient to generate the DMA operation in $\mathbf{s}'$, completing the proof. □

Lemma 56. *Consider an interval of time where the program executes with no change to the state of the SPM controller (including stalling the CPU) during the interval, and let t_{comp}, t_{mm} and t_{spm} be the maximum computation time, main memory and SPM time for the interval, assuming that all load/stores to any object v_i (pointer p_k) access main memory iff $spm^{v_i}_{r_j} = 0$ (respectively, $spm^{p_k}_{r_j} = 0$) for all regions that contain the interval. Furthermore, let $\mathbf{s}$ be the program state at the beginning of the interval, $\mathbf{s}$ the state at the end of the interval, and d' be computed based on abstract state d according to Definition 48 with $\Delta = t_{comp} + t_{mm} + t_{spm}$ and $\Lambda = t_{comp} + t_{spm}$. If the latency for access to main memory is greater than or equal to the latency for access to the SPM, then $\mathbf{s} \in \gamma(d) \Rightarrow \mathbf{s}' \in \gamma(d')$.*

Proof. Let $\bar{t}_{comp}, \bar{t}_{mm}$ and $\bar{t}_{spm}$ denote the actual computation, main memory and SPM times for the interval, rather than the upper bounds. Then by definition we have $t_{comp} \geq \bar{t}_{comp}$ and $t_{mm} \geq \bar{t}_{mm}$. Note that for the SPM time it might hold $\bar{t}_{spm} \geq t_{spm}$, since some load/stores operations that are assumed to access main memory might access the SPM in the actual program execution; however, since memory latency is at least equal to SPM latency, it must still hold $t_{mm} + t_{spm} \geq \bar{t}_{mm} + \bar{t}_{spm}$. Now define $\bar{\Delta} = \bar{t}_{comp} + \bar{t}_{mm} + \bar{t}_{spm}$ and $\bar{\Lambda} = \bar{t}_{comp} + \bar{t}_{spm}$; note we must have $\bar{\Delta}, \bar{\Lambda} \geq 0$. Finally, let $\delta = \Delta - \bar{\Delta}$ and $\lambda = \bar{\Lambda} - \Lambda$. Note $\delta \geq 0$, and furthermore: $\delta + \lambda = \Delta - \Lambda - (\bar{\Delta} - \bar{\Lambda}) = t_{mm} - \bar{t}_{mm} \geq 0$.

By assumption on the behavior of the interval, $\mathbf{s}'.t = \mathbf{s}.t + \bar{\Delta}$, any DMA operation with trailing length less than or equal to $\bar{\Lambda}$ in $\mathbf{s}$ is removed from $\mathbf{s}'$, while all other operations have a trailing length reduced by $\bar{\Lambda}$. Also based on Definition 48: $d'.t = d.t + \Delta$. Let $K = d.t - \mathbf{s}.t \geq 0$, we then have $K' = d'.t - \mathbf{s}' = d.t + \Delta - \mathbf{s}.t - \Delta + \delta = K + \delta$; thus $K' \geq K \geq 0$, and to satisfy the inclusion $\mathbf{s}' \in \gamma(d')$ it remains to show that operations in $\mathbf{s}'$ can be generated by d'.

Therefore, consider any operation in $\mathbf{s}'$ with trailing length k' greater than K'; we have to prove that the operation is generated by a timer in d'. The trailing length of that operation in $\mathbf{s}$ must be $k = k' + \bar{\Lambda} > K' \geq K$; hence, there must be a timer $\mathbb{T}$ in d that generates that operation, such that:

$$k \leq d.\mathbb{T}.t + K. \tag{B.6}$$

This implies $d.\mathbb{T}.t + K \geq k' + \bar{\Lambda} > K' + \Lambda + \lambda$, and thus $d.\mathbb{T}.t > (K' - K) + \Lambda + \lambda = \Lambda + \delta + \lambda \geq \Lambda$. Based on Definition 48, we have $d'.\mathbb{T}.t = d.\mathbb{T}.t - \Lambda$, and since $d.\mathbb{T}.t > \Lambda$, it thus holds $d.\mathbb{T}.t = d'.\mathbb{T}.t + \Lambda$ (i.e., timer $\mathbb{T}$ in not reset in d'). Substituting the expression for $d.\mathbb{T}.t$ in Equation B.6 yields: $d'.\mathbb{T}.t + \Lambda + K = d'.\mathbb{T}.t + \Lambda + K' - \delta \geq k = k' + \Lambda + \lambda$, which is equivalent to: $d'.\mathbb{T}.t + K' \geq k' + \delta + \lambda \geq k'$. Therefore, the trailing length of $d'.\mathbb{T}$ is sufficient to generate the DMA operation in $\mathbf{s}'$, completing the proof. □

Note that while we proved Lemma 56 for the dual-ported SPM case, the Lemma is also valid for the single-port case where $\Lambda = t_{comp}$, $\bar{\Lambda} = \bar{t}_{comp}$, since it still holds $\delta + \lambda = t_{mm} + t_{spm} - \bar{t}_{mm} - \bar{t}_{spm} \geq 0$.

Theorem 57. *Equation B.2 holds for the described DMA Abstraction $(D, \leq_D)$ with abstract transfer function $\hat{\mathcal{T}}_{e,\sigma}(d)$.*

Proof. We need to show $S \subseteq \gamma(d) \Rightarrow \mathcal{T}'_{e,\sigma}(S) \subseteq \gamma(\hat{\mathcal{T}}_{e,\sigma}(d))$ for every edge $e : BB \rightarrow BB'$. Since by definition $\mathcal{T}'_{e,\sigma}(S) = \cup_{\mathbf{s}\in S}\mathcal{T}_{e,\sigma}(\mathbf{s})$, this is equivalent to showing that $\forall d \in D, \forall \mathbf{s} \in \gamma d$, if the execution can flow along edge e from state $\mathbf{s}$ with $d' = \hat{\mathcal{T}}_{e,\sigma}(d)$ and $\mathbf{s}' = \mathcal{T}_{e,\sigma}(\mathbf{s})$, it must hold: $\mathbf{s}' \in \gamma(d')$.

Since in Definition 51 we have described $\hat{\mathcal{T}}_{e,\sigma}(d)$ as an iterative transformation based on the scratchpad commands within basic block BB, we apply the same technique to $\mathcal{T}_{e,\sigma}$, and describe the transformation of the program state $\mathbf{s}$ based on a sequence of instruction intervals and ALLOC/DEALLOC/GETADDR commands. Note that since basic blocks in the extended CFG do not contain branches or function calls, every execution of BB along e has the same sequence of intervals/commands as the one considered by $\hat{\mathcal{T}}_{e,\sigma}(d)$.

Without loss of generality, let N be the total number of intervals and commands. Let us define a set of abstract states $\{d^0, \ldots, d^N\}$ and program states $\{\mathbf{s}^0, \ldots, \mathbf{s}^N\}$, where $d^0 = d, \mathbf{s}^0 = \mathbf{s}$ and for $0 < i \leq N$, d^i and $\mathbf{s}^i$ represent the abstract and program state after the N^{th} interval/command in the sequence. Then by definition: $d' = d^N, \mathbf{s}' = \mathbf{s}^N$. Now note that based on Lemma 56 for intervals and Lemmas 53, 54, 55 for commands, it holds $\mathbf{s}^{i-1} \in \gamma(d^{i-1}) \Rightarrow \mathbf{s}^i \in \gamma(d^i)$. Hence, by induction on i, it also holds: $\mathbf{s}^0 \in \gamma(d^0) \Rightarrow \mathbf{s}' = \mathbf{s}^N \in \gamma(d^N)$, concluding the proof. □

Applying Definition 51 requires a precise knowledge of the position of each command in basic block BB. For simplicity of implementation, it can also be useful to formulate an analysis where the only available timing information are upper bounds to the computation, memory and SPM times t^{BB}_{comp}, t^{BB}_{mm} and t^{BB}_{spm} for the entire basic block, rather than individual intervals, and only the relevant ordering of SPM commands in the basic block is known.

Definition 58 (Imprecise Abstract Transfer Function)**.** *Consider a CFG edge $e : BB \to BB'$, and let t^{BB}_{comp}, t^{BB}_{mm}, t^{BB}_{spm} represent the maximum computation time, main memory and SPM time for basic block BB, assuming that all load/stores to object v_i (pointer p_k) access main memory iff $spm^{v_i}_{r_j} = 0$ (respectively, $spm^{p_k}_{r_j} = 0$) for all regions that contain BB. We can then compute an abstract transfer function $\tilde{\mathcal{T}}_{e,\sigma}(d)$ by applying an iterative set of transformations of the abstract state d using functions $ELAPSE$, $ALLOC$, $DEALLOC$ based on the order of $ALLOC$, $DEALLOC$ and $GETADDR$ commands in BB:*

- *Order the set of transformations as follows: first, apply transformations for each $GETADDR$ command and each $ALLOC$/$DEALLOC$ command that depends on a $GETADDR$ or is followed by another $ALLOC$/$DEALLOC$ that depends on a $GETADDR$, in the order in which the commands appear in BB; then, apply the transformation for BB's execution time; then, apply transformations for each $ALLOC$/$DEALLOC$ commands that has not been considered yet, in the order in which they appear in BB.*
- *For a $GETADDR$ command on object/pointer x, transform the state into $ELAPSE(d, \Delta, \Delta)$, where Δ is the maximum trailing length of any timer on which the $GETADDR$ stalls.*
- *For an $ALLOC$ command on object/pointer x, transform the state into $ALLOC(d, x, a, BB, pr, exec)$, where a is the SPM address of the $ALLOC$ and $pr = 1$ if the P flag is set.*
- *For a $DEALLOC$ command on object/pointer x, transform the state into $DEALLOC(d, x, a, BB, wb, exec)$, where a is the SPM address of the $DEALLOC$ and $wb = 1$ if the W flag is set in the $ALLOC$ command corresponding to this $DEALLOC$.*
- *To transform the state based on BB's execution time, apply function $ELAPSE(d, t^{BB}_{comp} + t^{BB}_{mm} + t^{BB}_{spm}, t^{BB}_{comp} + t^{BB}_{spm})$.*

Intuitively, the imprecise transfer function works as follows: we assume that all ALLOC/DEALLOC commands that do not depend on a GETADDR are "pushed" to the end of the basic block, since doing so adds prefetch and write-back operations at the last possible time, hence maximizing the blocking that can be suffered by following basic block. On the other hand, GETADDR commands (and depending ALLOC/DEALLOC) are "pulled" to the beginning of the basic block, since this maximizes the amount of blocking that the GETADDR suffers due to DMA operations started in preceding basic blocks.

Theorem 59. *Equation B.2 holds for the described DMA Abstraction $(D, \leq_D)$ with abstract transfer function $\tilde{\mathcal{T}}_{e,\sigma}(d)$.*

Proof Sketch. Consider $e : BB \to BB'$, and let $d' = \hat{\mathcal{T}}_{e,\sigma}(d)$ and $d'' = \tilde{\mathcal{T}}_{e,\sigma}(d)$. As in the proof of Theorem 57, we have to show that $\mathbf{s} \in \gamma(d) \Rightarrow \mathbf{s}' \in \gamma(d'')$, where $\mathbf{s}'$ is the program state after the execution of BB along e starting from program state $\mathbf{s}$.

Next note that the only case in which commands can be reordered in Definition 58 is when an ALLOC/DEALLOC that does not depend on any GETADDR is pushed to the end of the basic block. By definition, the ALLOC/DEALLOC command cannot operate of any timer that are checked by a GETADDR; hence, reordering the commands in this way cannot change the behavior of the SPM controller. Therefore, it remains to argue that the following three changes will maintain the order $d' \leq_D d''$: 1) moving a GETADDR to the beginning of the basic block; 2) moving a non-dependent ALLOC/DEALLOC to the end of the basic block; 2) moving a dependent ALLOC/DEALLOC (or an ALLOC/DEALLOC followed by a dependent one) to the beginning of the basic block.

GETADDR. Let Δ be the blocking time of the GETADDR for the precise abstraction ($\hat{\mathcal{T}}_{e,\sigma}(d)$). Since in $\tilde{\mathcal{T}}_{e,\sigma}(d)$ the GETADDR is moved at the beginning of the interval, the trailing length of any timer on which GETADDR can stall must be greater than or equal to the trailing length in the precise abstraction; hence, $\tilde{\Delta} \geq \Delta$, where $\tilde{\Delta}$ is the blocking time for the imprecise abstraction. Following the same argument as in Lemma 55, we have $\tilde{\Delta} \geq \Delta \geq \bar{\Delta}$, where $\bar{\Delta}$ is the actual blocking time for $\mathbf{s}$, which then implies $\mathbf{s}' \in \gamma(d'')$.

Non-dependent ALLOC/DEALLOC. Note that moving an ALLOC/DEALLOC while keeping the same order of dependent commands does not change which timers are removed (if any). Hence, consider any timer $\mathbb{T}$ added by the ALLOC/DEALLOC. Since the command is moved to the end of the basic block, it must hold $d'.\mathbb{T}.t \leq d''.\mathbb{T}.t$, and hence $d' \leq_D d''$; since $\mathbf{s}' \in \gamma(d')$ by Theorem 57, then $\mathbf{s}' \in \gamma(d'')$ by monotonicity of γ.

Dependent ALLOC/DEALLOC. Any timer added by a dependent ALLOC/DEALLOC will by definition cause a GETADDR in BB to stall. Similarly, any ALLOC/DEALLOC followed by a dependent command will increase the maximum trailing length of any timer, hence increasing the trailing length of the dependent timers. Therefore, moving these commands to the beginning of the basic block immediately before the GETADDR cannot decrease the program stall time compared to the precise abstraction, meaning $\tilde{\Delta} \geq \Delta$ and $\mathbf{s}' \in \gamma(d'')$ from Lemma 55.

□