Automatic Loop Nest Parallelization for the Predictable Execution Model

by

Zhao Gu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2023

Author's Declaration

Statement of Contributions

Zhao Gu was the sole author for Section 2.2, 2.3, 5.2, 5.3, 6.3 and 7 which were written under the supervision of Prof. Rodolfo Pellizzoni and were not written for publication.

Research presented in rest Sections This research was conducted at the University of Waterloo by Zhao Gu under the supervision of Prof. Rodolfo Pellizzoni. Zhao Gu designed the study, completed the coding, data analysis and wrote the draft manuscripts with assistance from Prof. Rodolfo Pellizzoni.

Citations: Zhao Gu and Rodolfo Pellizzoni. 2022. Optimizing parallel PREM compilation over nested loop structures. In Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC '22). Association for Computing Machinery, New York, NY, USA, 1249–1254. https://doi.org/10.1145/3489517.3530610 [21]

As lead author of these sections, I was responsible for contributing to conceptualizing study design, carrying out coding, data collection and analysis, and drafting and submitting manuscripts. My supervisor provided guidance during each step of the research and provided feedback on draft manuscripts.

Abstract

Currently, embedded real-time systems still widely use single-core processors. A major challenge in the adoption of multicore processors is the presence of shared hardware resources such as main memory. Contention between threads executing on different cores for access to such resources makes it difficult to tightly estimate the Worst-Case Execution Time (WCET) of applications. To safely employ multicore processors in real-time systems, previous work has introduced a PRedictable Execution Model (PREM) for embedded Multi-Processor Systems-on-a-Chip (MPSoCs). Under PREM, each thread is divided into memory phases, where the code and data required by the thread are moved from main memory to a local memory (cache or scratchpad) or vice versa, and execution phases, where the thread computes based on the code and data available in local memory. Memory phases are then scheduled by the Operating System (OS) to avoid contention among threads, thus resulting in tight WCET bounds. The main challenge in applying the model is to automatically generate optimized PREM-compliant code instead of rewriting programs manually. Note that many programs of interests, such as emerging AI and neural network kernels, comprise both compute-intensive and memory-intensive deeply nested loops. Hence, PREM code generation and optimization should be applicable to nested loop structures and consider whether performance is constrained by computation or memory transfers.

In this thesis, we address the problem of automatically parallelizing and optimizing nested loop structure programs by presenting a workflow that automatically generates PREM-compliant optimized code. To correctly model the structure of nested loop programs, we leverage existing polyhedral compilation tools that analyze the original program and generate optimized executables. Two main techniques are adopted for optimization: loop tiling and parallelization. We build a timing model to estimate the length of execution and memory phases, and then construct a Directed Acyclic Graph (DAG) of program phases to estimate its makespan. During this process, our framework searches for the combination of tile sizes and thread numbers that minimize the makespan of the program; given the complexity of the optimization problem, we design a heuristic algorithm to find solutions close to the optimal. Finally, to show its usefulness, we evaluate our technique based on the Gem5 architectural simulator on computational kernels from the PolyBench-NN benchmark.

Acknowledgements

I would like to thank my advisor, Professor Rodolfo Pellizzoni, who gives me his maximum support and toleration during the days when progress stuck for my mental struggles.

I also want to thank my family who always care for me during this time.

Dedication

The thesis is dedicated to my grandparents, the childhood memories I had with you are always precious to me.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Novel embedded applications in domains such as autonomous cars and unmanned vehicles are driving the adoption of Multi-Processor Systems-on-a-Chip (MPSoC) to satisfy their performance requirements. Many such applications, such as Deep Neural Networks (DNN), consist of computational kernels that can be parallelized across several processing cores. At the same time, due to safety considerations, applications in such domains also typically require real-time guarantees. Unfortunately, this is problematic in MPSoCs since the presence of shared hardware resources between cores makes it difficult to estimate the timing of concurrent threads.

In this thesis, we focus on the issue of predictably sharing access to main memory. In recent years, the community has proposed memory-aware execution models, such as the PRedictable Execution Model (PREM) [31] and its extensions [14, 27, 37, 16], which address the memory contention problem from a software perspective. The PREM method involves compiling each thread in a way that allows it to switch between two distinct phases: memory and execution. During the memory phase, the thread accesses main memory to retrieve necessary code and data and saves it into a local, private memory, in the form of a cache or ScratchPad Memory (SPM). The thread also writes back any modified data from the local memory to main memory during this phase. In contrast, the execution phase involves the thread performing useful computation using only the code and data in its local memory. The Operating System (OS) can then schedule memory phases in such a way as to prevent saturating main memory. This ensures that the length of both memory and execution phases of a thread remains unaffected by concurrent activity on other cores, simplifying timing estimation.

A main complexity in PREM is how to divide a thread into memory and execution phases. In general, the data used by a thread might be too large to entirely fit in its local memory. To solve this issue, the thread can often be divided into multiple *segments*, where each segment accesses only a portion of the total data footprint of the thread. In the case of loop structures, this can be achieved by tiling the loop [14, 29, 37, 16], i.e., partitioning the loop iterations into a set of disjoint ranges and executing the iterations in each range as part of a different segment.

1.1 Objective

The overall objective of this thesis is to facilitate the adoption of PREM by studying how to automatically generate optimized PREM-compliant application code. Specifically, we are concerned with compiling and scheduling a single application on a multicore system, where each core has a private SPM. We consider computational kernels written as sequential C programs and consisting of a structure of nested loops. Our PREM compiler employs data and loop analysis to understand the loop structure, and divides the program into segments by employing loop tiling on multiple loop levels. Whenever possible, segments are assigned to different threads and executed in parallel on multiple cores to minimize the worst-case makespan of the application (total time required to run it once, including memory phases).

A key observation is that, outside the constraint on local memory size, properly selecting tile/segment sizes is essential to optimize the makespan of the application. This is because the segment size affects the ability of scheduling memory phases in parallel with execution phases, thus "hiding" the overhead of memory operations; while the choice of which loop to tile affects the data footprint, and thus the memory phase length, for any given segment. For this reason, we seek not only to transform the program in a legal way, but also to optimize such transformation by tiling the program in the optimal way. However, finding the best tile sizes is in general difficult due to the size of the problem space: for example, the convolutional DNN kernel we study in our evaluation consists of 7 nested loops.

1.2 Contributions and Structure

In summary, we provide the following main contributions: **(1)** We consider a system where memory phases are performed by a dedicated DMA component, and we extend the streaming PREM model[36] which was previously employed to schedule independent, sequential tasks, to the case of an application consisting of parallel threads. Dependencies are encoded in a DAG, which is then traversed to determine the makespan of the schedule.

(2) Due to the complexity of the tile optimization problem, we introduce a heuristic that efficiently searches for optimized tile sizes and tile-to-thread assignments, even for kernels comprising deeply-nested loop structures. **(3)** We demonstrate the applicability of our technique by applying it to the kernels in the PolyBench-NN benchmark suite [42] for DNN applications. Compared to previous work on PREM compilation [14, 29, 37, 16], our solution is the first to target parallel applications with an arbitrary number of nested loop levels.

The rest of the thesis is structured as following:

- In Chapter 2, we introduce the problem background and how it can be addressed using PREM. We also introduce the basic concepts of Polyhedral Model which is used for program analysis.
- In Chapter 3, we show step by step how we model the nested loop program and how it is transformed and executed in a manner compliant with PREM.
- In Chapter 4, we demonstrate how to calculate the makespan of transformed program and the algorithm of finding tiling and parallelization sizes that gives makespan close to minimal.
- In Chapter 5, we show the whole compilation flow of transforming a nested loop program into an optimized PREM-compliant program. Then we explain the key steps in this compilation flow, including the validity of loop transformation and how we calculate the bounding box of data transfer statements.
- In Chapter 6, we evaluate this approach on PolyBench-NN and compare it with a greedy approach.
- Finally, in Chapter 7, we present the conclusions and directions for future research.

Note that most of the content in Chapters 3 and 4, and some of the content in Chapters 5 and 6 have been previously published in conference paper [21], of which I am the first author and sole student author.

Chapter 2

Background and Related Work

In this chapter, we cover the required background to understand the rest of the thesis, and we discuss how our contribution compares to related work. We first provide a thorough discussion of the Predictable Execution Model in Section 2.1. In particular, we detail the execution model and related scheduling API used in the streaming model introduced in [36], since they are most related to the approach used in this thesis. We also discuss related work in PREM compilers for automatic program transformation and optimization. Since our optimizing compiler relies on the polyhedral model for program analysis and transformation, in Section 2.2 we introduce key concepts in the polyhedral loop model, and we show how data dependencies are modeled using these concepts.

2.1 Predictable Execution Model

Modern MPSoCs are characterized by a variety of hardware resources, such as caches, buses and main memory, that are shared among processing elements / cores. Interference for access to such shared resources can significantly increase the execution time of tasks. Main memory, which is typically implemented as DRAM, represent a potential significant performance bottleneck and a major source of unpredictability [23, 22], making it difficult to derive tight bounds on the worst-case execution time of tasks. For this reason, over the past decade, the real-time community has spent significant effort in devising memory-aware allocation, partitioning and scheduling mechanisms for shared resources. One such approach is the PRedictable Execution Model (PREM) first introduced in [31]. Such work attempts to solve the predictability issue of accesses to main memory by dividing the execution of software tasks into two parts. First, a memory phase is used to load the data required by the task from main memory into cache by performing suitable prefetch instructions. During this phase, dirty cache lines evicted from cache are also written back to main memory. Then, during the computation phase, the task executes using the data prefetched to cache. Since the task does not access main memory during the computation phase, either I/O devices or tasks executing on other cores are free to use main memory without causing contention.

Following works [49, 3] have refined the approach proposed by [31] into a three-phase model. In this extended model, the task is divided into three phases: first, a load or acquisition memory phase is used to move required data from main memory into a local memory; then, during the execution phase, the core computes using the content of the local memory; and finally during an unload or restitution phase, modified data in local memory is written back to main memory. The Operating System (OS) is responsible for scheduling memory phases in such a way that main memory is not overutilized; for most approaches, this means that only one memory phase is allowed to proceed at a time. In turn, this prevents contention and slowdown in main memory, ensuring that it is possible to tightly bound the length of both memory phases and execution phases.

The three-phase model has been employed by many published works [51, 50, 49, 3, 2, 1, 40, 41, 52, 26, 27, 14, 10, 30, 11, 12, 34, 4, 35, 19, 33]. Existing approaches differ on several aspects, including: (1) whether they target sequential tasks or parallel tasks; (2) depending on the employed scheduling discipline, that can follow either a partitioned or global scheme, and either allow or disallow task preemption; (3) whether they require loading data for the whole task, or support loading only a portion to allow it to fit into limited local memory space. In the latter case, the task is generally divided into a set of sequential segments, where every segment comprises one load, execution and unload phase.

Among the cited works, the approaches in [4, 10, 11, 12, 14, 27, 40, 41, 50, 19, 33] have been implemented on actual MPSoC platforms. Employed platforms differ on two main aspects: (1) while most of the works target general purpose CPU cores, some of them are implemented on GPU cores; (2) some platforms use caches as local memory, while others employ a ScratchPad Memory (SPM) which is private to each core. On platforms that employ cache memory, memory phases must be executed on the CPU to prefetch / flush the required data to / from cache. On the other hand, platforms employing SPM can use a dedicated DMA component to execute the memory phases. Such approaches attempt to hide the latency of memory operations by performing the load/unload of one task in parallel with the execution of a different task. This is achieved by using a double-buffering technique [49, 15, 14, 40], so that the DMA performs memory phases targeting one buffer while the core executes using the other buffer.

2.1.1 Streaming Model

```
int a[300];
memset(a, 0, sizeof(a));

for (i = 0; i < 300; i++)
{
  a[i] += 2 * i + 1;
}
```

(a) Origin code of a simple loop

```
int a[300];
memset(a, 0, sizeof(a));

B1 = allocate_buffer(a_buf1, RW);
B2 = allocate_buffer(a_buf2, RW);
swap_buffer(B1, a, 100);
dispatch();
swap_buffer(B2, a+100, 100);
end_segment();

for (i = 0; i < 100; i++)
{
  a_buf1[i] += 2 * i + 1;
}
swap_buffer(B1, a+200, 100);
end_segment();

for (i = 0; i < 100; i++)
{
  a_buf2[i] += 2 * i + 1;
}
deallocate_buffer(B2);
end_segment();

for (i = 0; i < 100; i++)
{
  a_buf1[i] += 2 * i + 1;
}
deallocate_buffer(B1);
end_segment();
```

(b) After PREM program transformation

Figure 2.1: PREM API usage example adapted from [36]

In this section, we discuss in more detail the PREM streaming model introduced in [36]. The goal of [36] is to schedule a set of sequential programs (real-time tasks) based on the three-phase model; each program is divided into multiple segments allocated on one core. Memory phases are performed by a single programmable DMA engine, under the control of the OS. The operating system schedules DMA operations of different cores based on a Time-Division Multiple Access (TDMA) scheme. Compared to previous PREM approaches, the

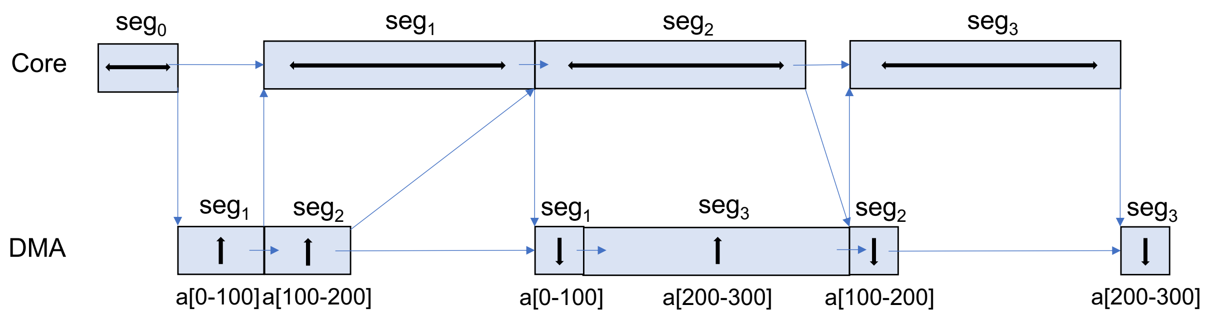


Figure 2.2: PREM Execution on single core

Table 2.1: Proposed PREM API to manage SPM data from [36]

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

streaming model allows segments of the same program to be executed consecutively by alternating between the two allocated buffers.

To control the execution of segments and data load/unload, the work in [36] introduces a PREM API, which we detail in Table 2.1; the program must employ suitable API calls in each segment to define required data buffers, specify the data to be transferred to/from main memory, and determine the end of the segment. To illustrate the usage of the PREM API, we introduce the example of a simple program, adapted from [36]; we provide the code of the program in Figure 2.1, while Figure 2.2 shows the corresponding PREM execution for the transformed program. Note that in the figure, up arrows represent load phases and down arrows represent unload phases; furthermore, arrows linking different segments represent precedence constraints between segments. Specifically, the execution phase of segment seg_i can only start after the end of the load phase for seg_i, and the unload phase of seg_i can only start after its execution phase. Finally, note that for simplicity we are only showing the execution of a single program on one core; in reality, a program can be preempted by another, higher priority program, leading to segment interleaving, and furthermore as previously mentioned, the DMA services different cores based on assigned TDMA slots.

The original program in Figure 2.1a performs a simple task that assigns a value to every element of array `a`. In Figure 2.1b, we execute the same task. But instead of executing the loop on line 4 of Figure 2.1a, we execute this loop in three separate loops, each of which assigns a value to 100 elements of array `a`. Each loop is put into a segment to be executed in streaming mode.

seg_0, which comprises the execution in lines 4 to 9 of Figure 2.1b, is responsible for buffer allocation and firing data transfer instructions for segments seg_1 and seg_2. On lines 4-5, `allocate_buffer` is called to allocate two buffers for the array. Since the array `a` is both read from and written to in the original program, these two buffers are allocated as `RW` (read-write mode). After the streaming buffers are allocated on SPM, on line 6 a `swap_buffer` is called to schedule a load operation for data required by seg_1. The design of `swap_buffer` is that if it is called with a buffer allocated as `WO/RW`, it checks whether this buffer is bound to an address in main memory. If it is not bound to any address in main memory, this `swap_buffer` would bind this buffer to the input memory address, otherwise it schedules a unload operation during the execution of next segment that copy the data in this buffer to the previously bound address in main memory and bind the buffer to the input address after unload operation completes. If `swap_buffer` is called with a buffer allocated as `RO/RW`, it schedules a load operation during the execution of the next segment (possibly after the unload operation for a bound `RW` buffer). Since in this case buffer `B1` is `RW` and not bound to any address yet, this call would bind the buffer to the address corresponding to the beginning of array `a` in main memory and schedule a load operation, but not an unload one.

On line 7, we call `dispatch` to enforce an immediate load operation because seg_1 cannot start executing before its data load completes. The OS will schedule the load operations required by the `swap_buffer` calls before the `dispatch` after the end of seg_0, and will not start the following segment seg_1 until such load phases have completed. On the other hand, load operations for `swap_buffer` calls performed after the `dispatch` will be scheduled by the OS in parallel with the execution of seg_1. Specifically, on line 8, another load operation for data required by seg_2 is scheduled by `swap_buffer` to be performed during seg_1. It also binds the address of `a+100` to `a_buf2`. Finally, on line 9, `end_segment` is called to end the execution of the current segment.

Table 2.2: Comparison of PREM Compilers

	DMA	task model	optimization
[14]	yes	parallel, single	greedy, 1 level
[29]	no	parallel, single	greedy, 1 level
[37]	yes	sequential, multi	optimal, 2 levels
[16]	no	sequential, multi	heuristic, 1 level
This work	yes	parallel, single	heuristic, any levels

seg_1 comprises the execution from line 11 to line 16. On line 11-14, it executes a tiled version of the original loop program with array `a` replaced by its corresponding data structure in SPM, array `a_buf1`. On line 15, this time the `swap_buffer` first schedules an unload operation in the next segment from `a_buf1` to `a` in main memory and rebind `a_buf1` to `a+200`, then it also schedules a load operation in the next segment from `a+200` to `a_buf1`. Then seg_1 ends execution.

In seg_2 from line 18 to line 23, line 18-21 are similar tiled computation code. The `deallocate_buffer` on line 22 schedules an unload operation from `a_buf2` to its last bound address `a+100` and deallocates the buffer on SPM after the unload operation completes during the execution of the next segment seg_3.

seg_3 from line 25 to line 30 is similar to seg_2. The `deallocate_buffer` on line 29 schedules an unload operation from `a_buf2` to `a+200` because it is bound by `swap_buffer` on line 15. After seg_3 ends, the unload operation and buffer deallocation is performed, and the whole program ends afterward.

2.1.2 PREM Compilers

The seminal work in [31] simply assumed that a program could be manually modified to be made PREM-compliant. However, we argue that, for realistic programs, segmenting the code and implementing the required memory and execution phases manually is not practical. For this reason, previous work [14, 29, 37, 16] has introduced PREM compilers that are able to automatically segment a program and introduce the required code to create memory phases, as shown in the example of Figure 2.1.

Table 2.2 summarizes the key characteristics of the existing PREM compilers: whether they use a dedicated DMA for memory phases, their assumed system model (sequential or parallel applications and single or multitasking) and the employed optimization strategy.

All presented compilers employ loop tiling to break a program into segments so that they fit in the available local memory space. Loop tiling partitions the iterations of a loop level into a set of iteration ranges. Multiple loop levels can be simultaneously tiled; a tile is a combination of iteration ranges, one for each tiled loop. The execution of a segment corresponds to the execution of all iterations within a given tile. "Greedy" approaches simply select the largest tile size that fits; in case of nested loops, only one loop level is tiled. The approaches in [37, 16], which deal with multitasking real-time scheduling of sequential applications, are more refined. Since both memory and execution phases are executed non-preemptively to preserve local memory content, reducing the size of memory phases by varying tile sizes helps decreasing the blocking time suffered by high-priority tasks. [16] presents a fast heuristic that can only tile on one loop level. [36] provides a slower, optimal approach that is limited to tiling the first two levels in a nested loop. In contrast, this thesis targets *optimized parallel scheduling of a single application with an arbitrary number of nested loop levels.* The parallel nature of the application makes it harder to analytically compute its makespan, while tiling over many loop levels greatly increases the search space for tile sizes. Therefore, an efficient search heuristic is needed.

2.2 Polyhedral Loop Model

The approach we adopt for modeling nested loops is to represent them in the *polyhedral model*. In a traditional program representation, using the control flow graph, program statements are placed in basic blocks and the whole program is represented by a directed graph with basic blocks as its nodes. In contrast to the control flow graph, which represents the whole program, the polyhedral model only represents the restricted loop structure part of the program. In it, each program statement is modeled as a $\mathcal{Z}$-polyhedron. A big advantage of using the polyhedral model is that, not only does it capture the semantics of statement, but also it gives enough information of statement execution in each iteration. In a typical control flow graph representation of loop nest, each statement appears only once, though it would be executed for many times. This makes it difficult or even impossible to reason about information like dependencies between particular iterations, execution order of statements in different loop nests, etc.

The polyhedral model solves this limitation by treating each loop iteration within nested loops as lattice points inside a multidimensional polyhedron. With such a model, combinatorial and geometrical optimizations can be applied on these objects to analyze and optimize the programs. The research community has developed many tools to apply this model to program analysis and transformation. The PLuTo compiler [9] uses the polyhedral model to generate automatic parallel programs with good locality. The PPCG compiler [47] transforms nested loop programs to CUDA-compliant C code. The ISL library [45] is a C library for manipulating sets and relations between integer points; it provides a framework for polyhedral model related operations. Widely used compilers GCC and LLVM also have polyhedral model based optimization components called Graphite [32] and Polly [20].

Yet, the polyhedral model still has its own limitations. The condition of *for* statement must be an affine combination of loop indexes. This means all control flow must be known during compile time. Conditions like $a[i][j] < N$ are not legal because this information is only known at run time, and it is impossible to know the execution order of statements given such conditions.

2.2.1 Basic Concepts

The program segment that could be analyzed by polyhedral model is called *static control part of program* which also referred to as *SCoP*. A SCoP contains multiple statements contained in a sequence of nested loops. The reason why it is called SCoP is that it could only handle program with compile time static control flow.

For a particular statement, we can use the values of its indexes to specify specific iterations. To distinguish this notation from *schedule*, notice these indexes are used for convenience and do not imply its execution order. This iteration is called an *instance* of the statement, and these index values compose a vector named *iteration vector*. With statement $Stmt_i$ and iteration vector v, we denote its corresponding instance $Stmt_i[v]$. In Figure 2.3, $Stmt_2$ is represented by an iteration vector $[i]$, then $Stmt_2[0]$ is one of its instances. And $Stmt_3$ has instances of $Stmt_3[0][0], Stmt_3[0][1], Stmt_3[0][2]...Stmt_3[99][99]$

Figure 2.3: Sample C program of vector multiplication

```c
int i, j, a[100][100], b[100], c[100];
int n = 100; //Stmt1
for (i = 0; i < n; i++) {
  c[i] = 0; //Stmt2
  for (j = 0; j < n; j++) {
    c[i] = c[i] + a[i][j] * b[j]; //Stmt3
  }
}
```

We call the set that contains all possible iteration vectors of this statement the *domain*.

The domain represents the range this statement lives in. For example, the *domain* of $Stmt_3$ is $\mathcal{D}_{Stmt_3} = \{(i, j) \mid 0 \leq i < 100 \wedge 0 \leq j < 100\}$.

The *schedule* is an integer tuple that corresponds to the execution order of each instance in the domain. This tuple represents the lexicographic order in which they execute. We use Φ to denote the function that maps one instance of a statement to its corresponding schedule. In this example, the schedule of $Stmt_2$ is $\Phi(Stmt_2[i]) = (1, i, 0, 0)$, schedule of $Stmt_3$ is $\Phi(Stmt_3[i, j]) = (1, i, 1, j)$. We can tell that $Stmt_2[10]$ executes after $Stmt_3[8][40]$ because the order of $Stmt_2[10]$ is $(1, 10, 0, 0)$ and the order of $Stmt_3[8][40]$ is $(1, 8, 1, 40)$ which is lexicographically smaller than schedule of $Stmt_2[10]$. The concept of schedule is crucial because we can formalize data dependence between statement instances and then encode the program transformation as a transformation of the schedule. The program transformation would be valid as long as the new schedule meets all the data dependencies. For example, we can transform the program in Figure 2.3 by splitting the for loop on line 3 into 2 loops.

Data access is also modeled as a map from domain to array elements. In most cases, the map between domain to array element is the identity. For example, in $Stmt_3$, the access to array a can be modeled as $\mathcal{A}_a^{Stmt_3} = \{Stmt_3(i, j) \to a(i, j)\}$. However, sometimes the space of mapped elements can be different from the original space. If we change the access of a from a[i][j] to a′[i+j][j], we could rename the index of *a'* from (i+j, j) to (m, n). In this case, the data access is $\mathcal{A}_{a'}^{Stmt_3} = \{Stmt_3(i, j) \to a'(m, n) \mid m = i + j \wedge n = j\}$. The procedure of calculating the corresponding mapped elements given input domain is called applying the domain to the map, which is noted as $\mathcal{D}_{Stmt} \times \mathcal{A}_a^{Stmt}$. The set of mapped elements is called *range*. If we apply domain $\mathcal{D}_{Stmt_3}$ to map $\mathcal{A}_{a'}^{Stmt_3}$, the resulting range is $\mathcal{R}_{a'} = \mathcal{D}_{Stmt_3} \times \mathcal{A}_{a'}^{Stmt_3} = \{(m, n) \mid 0 \leq n \leq 99 \wedge n \leq m \leq n + 99\}$ and this is the area of array *a'* that is accessed by $Stmt_3$.

2.2.2 Dependence Analysis

Data dependence is crucial in program analysis because it provides constraints for program transformation. In order to preserve the semantics of the original program, we must preserve its data dependencies.

The algorithms of dependence analysis in polyhedral loop model have been studied since Lamport in 1974 [25]. Dependence analysis in the polyhedral loop model is much more fine-grained than in control flow graph representation. In control flow graph representation, the minimal unit of dependence analysis is usually a reference to a variable or an array. On the other hand, in the polyhedral loop model, the minimal unit can be a reference to a particular array cell. A typical polyhedral transformation can reorder not only statements that contain dependent references, but also particular statement instances. In the work of PPCG [47], which is a polyhedral compiler that automatically compiles plain loop code into optimized CUDA code that executes efficiently on GPGPU, they optimize the plain loop code by reordering the execution order of statement instances.

In this thesis, we use the dependence analysis provided by PPCG [47], which is also using Lamport's dependency testing algorithm. It is a value based approach of dependence analysis, which is also applied in popular polyhedral compilation frameworks like Pluto and Candl [18]. If an instance $\vec{s}$ of a statement $Stmt_i$ precedes another instance $\vec{t}$ of a statement $Stmt_j$ and $Stmt_j[\vec{t}]$ reads/writes the data produced by $Stmt_i[\vec{s}]$, then we call $Stmt_i[\vec{s}]$ the *source* of this dependence and $Stmt_j[\vec{t}]$ the *sink* of the dependence.

The data dependencies in a nested loop are often represented by a data dependence graph. The nodes are statements in the program, and the directed edges are dependence relations. One data dependence can be represented using a pair of source instance and sink instance. We call this a dependent pair. In this thesis, we represent the data dependencies graph with a set Dep, which contains all the dependent pairs of instances $Stmt_i[\vec{s}] \rightarrow Stmt_j[\vec{t}]$ in the whole SCoP.

$$Dep = \{Stmt_i[\vec{s}] \rightarrow Stmt_j[\vec{t}] \mid Stmt_i[\vec{t}] \text{ depends on } Stmt_j[\vec{s}]\} \tag{2.1}$$

Note that in Equation 2.1, $Stmt_i$ and $Stmt_j$ can be the same statement. We get this set from dependency analysis result. The dependency analysis is conservative, which means as long as we satisfy all the dependent pairs in Dep, the program transformation is guaranteed to be legal. This is used to verify the legality of our program transformed procedure, as we will discuss in Section 5.2.1.

2.3 Loop Transformation

In this thesis, our goal is to enable parallel execution of nested loops by program transformations, constrained constrained under a limited SPM size. To achieve the above goal, we have to transform the source program. Many similar studies have been published in the field of polyhedral optimization. For example, the work in [9] applied combined loop transformations in the Pluto compiler, including loop tiling, fusion, interchanging, etc.

However, these schemes do not meet our requirements well. First, they assume a cache-based system, where the tiling transformation is only an optional approach to improve the cache utilization of the program. However, in our approach, tiling is required because we assume the usage of an SPM with limited space.

Second, although the polyhedral model libraries can support non-uniform nested loops, due to the overly aggressive exploitation of data locality within nested loops, the code generated by these schemes tends to have irregular execution order and complex shapes. One example is the diamond tiling transformation proposed by Bertolacci [6], which changes the structure of the code significantly compared to the original program. These transformations make it difficult to evaluate the transformed program's execution time: if segments execute widely different number of loop iterations, then the execution time of segments similarly varies significantly, making WCET estimation difficult and bringing additional complexity to segment scheduling. Thus, it is hard to guarantee their real-time performance through static scheduling.

Finally, the main goal of these schemes is to reduce the communication overhead when executing the programs in parallel, which is also an important aspect for our model. While reducing the communication overhead allows us to reduce the data transfer cost and bring us a greater performance improvement, it does not enable us to provide guarantees on the program validity.

To address these issues, we use a program transformation template to generate optimized code while respecting all the discussed constraints. We assume that each level of the nested loop program is a constant iteration range that has fixed iteration stride. Under this assumption, the polyhedron have a fixed rectangular shape, and the number of instances is uniformly distributed. The nested loops can be partitioned into segments, most of which have the same number of instances, by the transformation of loop tiling. We will further discuss how the transformation is performed in Section 5.2.2.

Chapter 3

System Model and Parallel PREM Schedule

In this chapter, we show how our approach leverages PREM to predictably execute a parallel application on an SPM-based multicore system. We begin by stating our assumptions on the required computing architecture in Section 3.1. Our approach targets computational kernels comprising a nested loop structure. Therefore, in Section 3.3 we show how we formally model such loop structure as a *loop tree*. Our approach does not construct a PREM schedule for the whole loop tree; instead, we first decompose the loop tree into a set of "linear" subtrees which we call *tilable components*, and then construct a schedule for each component. Section 3.4 formally defines the concept of a tilable component and shows how the component can be tiled and parallelized based on a set of tiling and parallelization parameters (also called an *optimization solution*); note that for the sake of clarity, we defer to Chapter 4 to show how to decompose the loop tree into components and select optimization solutions. Finally, in Section 3.5, we show how the PREM schedule is constructed for each tiled and parallelized component. Our solution is inspired by the streaming model for sequential tasks introduced in Section 2.1.1; in particular, we reuse the same scheduling API. Hence, we also provide a comprehensive example to show how API calls are inserted in the application's code to schedule DMA transfers.

3.1 Target Architecture

We consider the problem of compiling of a single sequential C application (a computational kernel) to execute on a set of P processing cores, with the objective of minimizing the

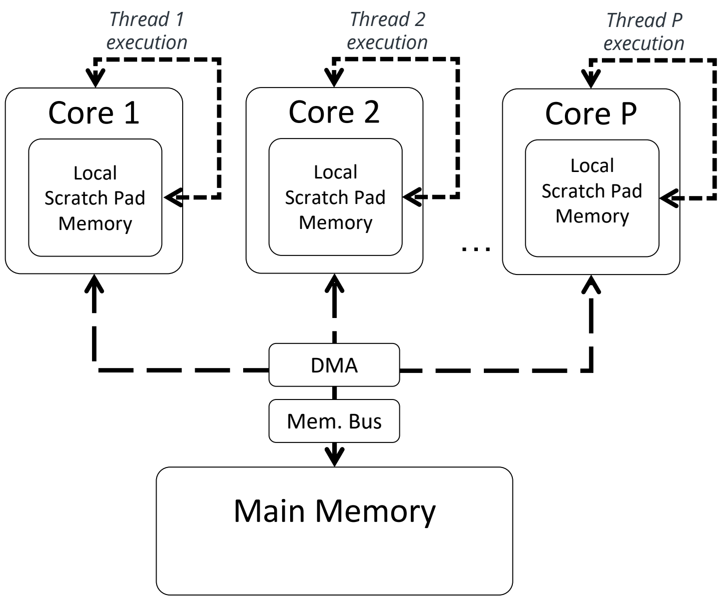


Figure 3.1: Target hardware architecture

kernel's makespan. Like previous work discussed in Section 2.1, we adopt an SPM-based memory hierarchy. As shown in Figure 3.1, each processing core has a local SPM and all the data transfer operations on these SPMs are executed by a single dedicated DMA. To execute the kernel program in parallel, the kernel is divided into up to P threads, with one thread assigned per core. Each thread can be further divided into multiple segments, executed according to the PREM streaming model discussed in Section 2.1.1. Following the streaming model, we assume that the SPM is divided into two partitions, one for the current segment execution and one for data transfers of the previous/next segment.

3.2 Code Assumptions

Before we introduce our application model, first we discuss the limitations of our approach. We adopt the polyhedral model for program analysis, program transformation and code generation. Our tool is designed to parallelize and transform the code in a single SCoP; if the program comprises other constructs and/or multiple SCoPs, those have to be transformed manually. Our tool has the same limitations of other polyhedral compilation tools: the execution order of each loop iteration must be known at compile time. We also have a more strict assumption on the program, the loops all have constant iteration ranges.

To perform data analysis on arrays, memory access relations must be affine functions (a memory access relation is a map from loop iterator variables to the position in the array of each element that is read or written in a loop iteration). Furthermore, we assume no pointer aliasing in input code, since this would invalidate the analysis result of memory accesses. To simplify the tiling transformation, we only handle loops with uniform-stride, meaning the stride of each loop is a constant number.

3.3 Application Model

After performing data and loop analysis (see Section 5.1), our PREM compiler builds a model for the kernel that can be used for scheduling optimization. Specifically, the kernel is modeled as a loop tree $\mathcal{T}$. Let $root(\mathcal{T})$ be an ordered list of the first-level loops in the kernel (possibly only one). We use $l \in \mathcal{T}$ to denote any one loop in the tree, where $l.\mathcal{C}$ is an ordered list of children loops (or $\emptyset$ if the loop is a leaf). $l.N$ is the number of iterations of loop level l, note that this is different from the range of loop index because loop stride could be larger than 1. $l.begin$ is the beginning index of loop level l and $l.S$ is the stride, thus the last index of loop level l is $l.begin + l.S \cdot (l.N - 1)$. $l.I$ is the number of times the loop is executed (where $l.I = 1$ if $l \in root(\mathcal{T})$, otherwise, $l.I$ depends on the number of iterations of predecessor loops in the tree). For each loop level from top to bottom, we perform a validity check that is introduced later in Section 5.2.1 to determine if the current loop level satisfies the dependency constraints after tiling. If not, we fold all sub-loop levels including this node to the parent of the current node to make it a leaf node. Finally, based on data dependencies, we define a parallelization attribute: if tiles over different iteration ranges of l can be executed in parallel in different threads, then $l.parallel = true$, otherwise, $l.parallel = false$.

We say that an ordered sequence of $L \geq 1$ loops $\mathcal{L} = (l_1, ..., l_j, ..., l_L)$ in the loop tree is a *tilable component* if it is a perfectly nested loop, formally: $\forall j = 1...L-1, l_j.\mathcal{C} = \{l_{j+1}\}$. We denote the set that contains all the arrays that are accessed in this tilable component $\mathcal{L}$ to be $\mathcal{L}.A$. Our framework constructs a parallel schedule for each tilable component by tiling its loops and assigning them to different cores.

Example: Listing 3.1 shows the code of LSTM benchmark. The corresponding loop tree is shown in Figure 3.2. Note that $l_{s1\_1}.I = l_{b\_0}.I = l_t.N - 1$ rather than $l_t.N$ because loops l_{s1_1}, l_{b_0} are not executed during the $0-th$ iteration of l_t.

```
for (int t = 0; t < NT; t++)
{
    for (int s1_0 = 0; s1_0 < NS; s1_0++)
    {
      for (int p = 0; p < NP; p++)
      {
        if(p==0)
        {
          i[s1_0] = 0.0;
          f[s1_0] = 0.0;
          o[s1_0] = 0.0;
          g[s1_0] = 0.0;
        }
        i[s1_0] += U_i[s1_0][p] * inp_F[t][p];
        f[s1_0] += U_f[s1_0][p] * inp_F[t][p];
        o[s1_0] += U_o[s1_0][p] * inp_F[t][p];
        g[s1_0] += U_g[s1_0][p] * inp_F[t][p];
      }
    }

    if (t > 0)
    {
      for (int s1_1 = 0; s1_1 < NS; s1_1++)
      {
          for (int s2 = 0; s2 < NS; s2++)
          {
            i[s1_1] += W_i[s1_1][s2] * s_F[t - 1][s2];
            f[s1_1] += W_f[s1_1][s2] * s_F[t - 1][s2];
            o[s1_1] += W_o[s1_1][s2] * s_F[t - 1][s2];
            g[s1_1] += W_g[s1_1][s2] * s_F[t - 1][s2];
          }
      }
    }

    if (t > 0)
    {
      for (int b_0 = 0; b_0 < NS; b_0++)
        c_F[t][b_0] = c_F[t - 1][b_0] * f[b_0] + g[b_0] * i[b_0];
    }

    for (int b_1 = 0; b_1 < NS; b_1++)
      s_F[t][b_1] = c_F[t][b_1] * o[b_1];
}
```
Listing 3.1: Original LSTM benchmark code

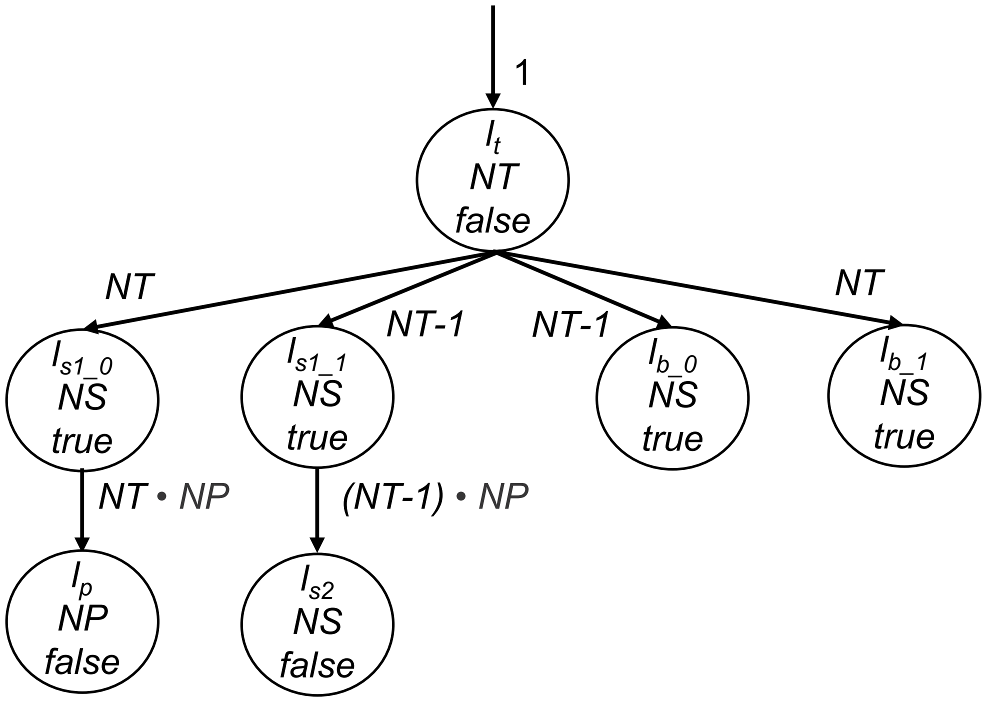


Figure 3.2: Loop tree for LSTM. $l.N$ and $l.parallel$ are shown within the node, while $l.I$ is shown on the incoming edge. $root(\mathcal{T}) = (l_t)$.

3.4 Tilable Components

We next discuss how each tilable component $\mathcal{L} = (l_1, ..., l_j, ..., l_L)$ is broken into a set of tiles, each of which is executed in a different PREM segment, and how segments are mapped to the P application threads.

Our optimization algorithm, which we will detail in Chapter 4, selects a *scheduling solution* consisting of tile size $l_j.K$ and number of *thread groups* $l_j.R$ for each loop l_j in the component, where $l_j.R = 1$ if $l_j.parallel = \mathit{false}$. Based on the assigned parameters, the loop l_j is divided into $l_j.M = \lceil l_j.N/l_j.K \rceil$ iteration ranges. The total number of tiles/segments is thus $\prod_{j=1...L} l_j.M$. The iteration ranges of l_j are partitioned among the $l_j.R$ thread groups, where each thread group is assigned at most $l_j.Z = \lceil l_j.M/l_j.R \rceil$ ranges. The total number of required threads/cores to execute the component is $\prod_{j=1...L} l_j.R$; therefore, for a thread group assignment to be valid, $\prod_{j=1...L} l_j.R \leq P$ must hold.

The resulting code after transformation has two parts: tiled loop and element loop. Both of them come from original loop level l_j and their loop ranges come from parameter set $l_j.N$, $l_j.K$, $l_j.M$ and $l_j.R$. Tiled loop refers to the nested *for* statements that iterate over the iteration ranges. Element loops are loops that are contained inside the tiled loop; each element loop iterates over each instance element in the current iteration range.

If $l_j.parallel = false$, then by definition the number of thread groups of this loop level is 1, meaning that the iteration ranges of l_j cannot be partitioned among cores. Thus, its tiling variable jt which ranges from 0 to $l_j.M - 1$ is used as the current iteration range number. Specifically, on each thread/core it executes at most $l_j.Z = \lceil l_j.M/l_j.R \rceil$ tiles (the last thread/core would execute fewer tiles). We assume we have access to an API call $threadID()$ to return the ID of the current thread. Its possible return values are $0, ..., P-1$. We can calculate for parallelizable loop level $l_j \in \mathcal{L}$ that its thread group ID is $\lfloor threadID()\%\ (\prod_{k=j...L} l_k.R)\ /\ (\prod_{k=j+1...L} l_k.R) \rfloor$ when $l_j \neq l_L$ and when $l_j = l_L$, its thread group ID is $\lfloor threadID()\% l_L.R \rfloor$.

Example: consider the $\mathcal{L} = (l_{s1\_0}, l_p)$ tiling component in the LSTM example in Listing 3.1 with $l_{s1\_0}.N = NS = 650$ and $l_p.N = NP = 700$ (`LARGE` problem size), together with an example (non-optimal) scheduling solution $l_{s1\_0}.K = 109, l_p.K = 350, l_{s1\_0}.R = 3, l_p.R = 1$. The resulting code after transformation is shown in Listing 3.2. The detailed transformation process will be discussed in Section 5.2.2; here we just show how this program is tiled and executed based on the example scheduling solution.

There are $l_{s1\_0}.M = \lceil l_{s1\_0}.N/l_{s1\_0}.K \rceil = \lceil 650/109 \rceil = 6$ iteration ranges for l_{s1_0} and $l_p.M = \lceil l_p.N/l_p.K \rceil = \lceil 700/350 \rceil = 2$ for l_p; the corresponding ranges are indexed by tiling variables $s1t$ and pt, which take values in 0...5 and 0...1, respectively. The resulting $6 \cdot 2 = 12$ tiles are partitioned across $l_{s1\_0}.R \cdot l_p.R = 3 \cdot 1 = 3$ threads with IDs from 0 to 2.

Each of the three thread groups for l_{s1_0} (consisting of $l_p.R = 1$ threads each) is assigned $l_{s1\_0}.Z = \lceil l_{s1\_0}.M/l_{s1\_0}.R \rceil = \lceil 6/3 \rceil = 2$ iteration ranges. For l_p, it only has one thread group (consisting of $l_{s1\_0}.R = 3$ threads). The only one thread group is assigned $l_p.Z = \lceil l_p.M/l_p.R \rceil = \lceil 2/1 \rceil = 2$ iteration ranges. For any given thread, its thread group ID on l_{s1_0} is computed as $threadID()\%(l_{s1\_0}.R \cdot l_p.R)/l_p.R = threadID()\%(3 \cdot 1)/1 = threadID()$, while its thread group on l_p is computed as $threadID()\%l_p.R = threadID()\%1 = 0$. The thread group IDs are used to determine the values for $s1\_0\_t$ and p_t used by each thread.

After transformation, we also insert three macro statements in Listing 3.2. These three macro statements would expand to data transfer APIs that schedule data transfer from main memory to SPM, the insertion of these macros are discussed later in Section 3.5.

```
for (int t = 0; t < NT; t++)
{
    /* transformed code of component (s1_0, p) start*/
    BUFFER_ALLOC_APIS //Macro Stmt
    //Tiled loops
    for (int s1_0_t = threadID()*2; s1_0_t < (threadID()+1)*2; s1_0_t += 1)
    {
        for (int p_t = 0; p_t < 2; p_t += 1)
        {
            DATA_SWAP_APIS //Macro Stmt
            for (int s1_0 = s1_0_t*109; s1_0 < MIN(NS, s1_0_t*109+109); s1_0
++)
            //Element loops
            {
              for (int p = p_t*350; p < MIN(NP, p_t*350+350); p++){
                if(p==0){
                  i[s1_0] = 0.0;
                  f[s1_0] = 0.0;
                  o[s1_0] = 0.0;
                  g[s1_0] = 0.0;
                }
                i[s1_0] += U_i[s1_0][p]*inp_F[t][p];
                f[s1_0] += U_f[s1_0][p]*inp_F[t][p];
                o[s1_0] += U_o[s1_0][p]*inp_F[t][p];
                g[s1_0] += U_g[s1_0][p]*inp_F[t][p];
              }
            }
        }
    }
    BUFFER_DEALLOC_APIS //Macro Stmt
    /* transformed code component (s1_0, p) end */
    ... ...
}
```

Listing 3.2: Component (l_{s1_0}, l_p) in LSTM code after tiling and core placement

3.5 Parallel Streaming PREM Schedule

In the previous section, we showed how to tile a tilable component and map the resulting tiles to up to P application threads. From the perspective of scheduling, each tile is mapped to a PREM segment. To execute these segments on the target architecture according to PREM, we use a set of well-defined APIs to control data transfers and segment boundaries.

Our design of PREM API is similar to [36, 33]. We assume the OS provides functionalities to schedule the segments on each core and program DMA transfers. The j-th segment executing on core i is denoted as $seg_{i,j}$. After tiling, the compiler inserts the required API

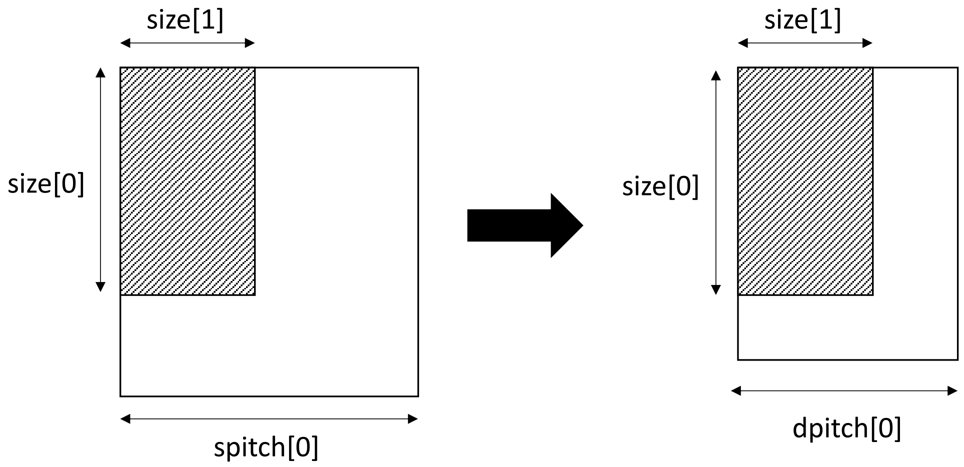


Figure 3.3: N-dimensional data transfer with N=2, adapted from [36]

calls throughout the kernel code. Before the execution of tiles on each core, an *initialization segment* must be executed. The initialization segment $seg_{i,0}$ on core i contains API calls to allocate buffers for the data elements used by the thread, and to start the DMA load for the segment $seg_{i,1}$. In addition, before the initialization segment can start, the code and initial stack content for the thread must be loaded in SPM (we assume such information is contained in the process control block).

The API we employ has three main differences compared to the one in [36] that we reviewed in Section 2.1.1: first, we assume that the OS now provides support for multi-threaded applications. Since threads are statically created and bound to cores when the application starts executing, this minimally impacts the API; we only need to add a function $threadID()$ to return the thread ID. Second, the schedule of memory phases changes. Specifically, in [36] the memory phases of a core are executed within assigned per-core TDMA slots, while in this work, following the scheme in Figure 3.4, we assume a round-robin schedule among threads. That is, we enforce a round-robin schedule of memory phases across cores, except that we combine the unload phase for segment $seg_{i,j}$ with the load phase for segment $seg_{i,j+2}$, since we want to perform both memory phases in parallel with the execution of $seg_{i,j+1}$. Finally, to support data structures with any number of dimensions, we assume that the following additional swap API `swapnd_buffer` is available. It is defined as:

```
void swapnd_buffer(int id, uint64_t *src, size_t dimension, int size[],
int spitch[], int dpitch[])
```

The design of `swapnd_buffer` is similar to the design of `swap2d_buffer` that is explained in [36]. The `swapnd_buffer` is used to swap N-dimensional data (N is greater than 1) with ID `id` by writing-back the current data for `WO/RW` buffer and copying data from `src` for `RO/RW` buffer. An N-dimensional buffer is the sub-array of the corresponding array in main memory. The `dimension`, `size`, `spitch` and `dpitch` determines this N-dimensional data transfer. The `dimension` parameter is restricted to be larger than 1. As is shown in Figure 3.3 which is adapted from [36], this is an example when `dimension` is 2 for a 2D data transfer. The value of `dimension` restricts the length of `size` parameter to be 2 and the length of `spitch` and `dpitch` parameters to be 1. The value of `size` parameter determines the size of data that is transferred. The last elements of these parameter arrays refer to the innermost dimension. The `spitch` parameter specifies the shape of the source N-dimensional array in main memory, notice it contains $N-1$ values as the outermost dimension is skipped. And similarly, `dpitch` specifies the shape of the destination N-dimensional array in SPM. All the values in `size`, `spitch` and `dpitch` are in bytes, e.g., `spitch[0]` of an `int32_t` 2D-array with 4×5 elements is $5 \cdot (32/8) = 20$. For the load operation in this example when `dimension` is 2, it transfers data from main memory to SPM, the DMA reads `size[1]` bytes every `spitch[0]` in main memory starting from the address `src`. Then, the DMA writes `size[1]` bytes every `dpitch[0]` to the buffer on SPM associated with ID `id`.

To generate the corresponding API calls, the required information is: for each array used in this entire tilable component, which sub-array area is possibly accessed in each segment. To model this, we introduce the concept of *canonical data element range*. For a segment $seg_{i,j}$ on core i, the canonical data element range of array a is $\hat{\mathcal{R}}_a(seg_{i,j})$. It is a set of data elements of array a with rectangular shape. The set is computed by finding, for each dimension of the array a, the minimum and maximum index for that dimension of any data element of a that might be accessed during the segment. When we transfer the data that is required for segment $seg_{i,j}$, we transfer the canonical data element ranges $\hat{\mathcal{R}}_a(seg_{i,j})$ of all arrays accessed in $seg_{i,j}$. The canonical data element ranges for all segments of all arrays are provided by compiler analysis using polyhedral model, the details are discussed in Section 5.3.1.

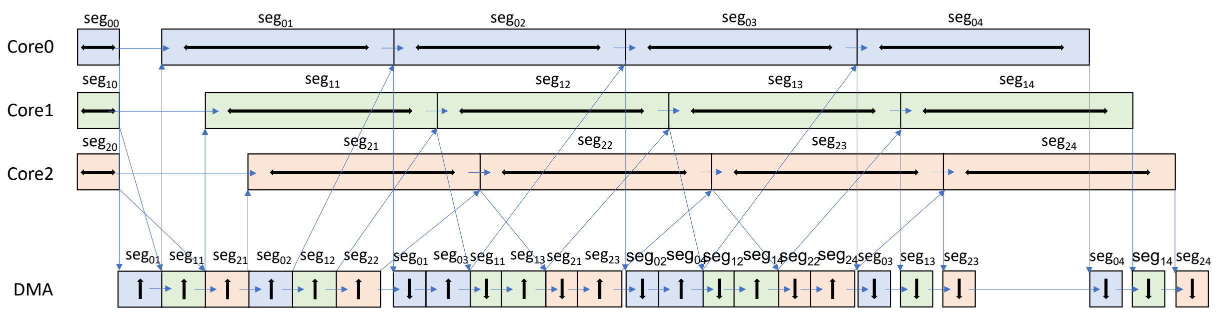

Core0
Core1
Core2
DMA
seg_{00} seg_{01} seg_{02} seg_{03} seg_{04}
seg_{10} seg_{11} seg_{12} seg_{13} seg_{14}
seg_{20} seg_{21} seg_{22} seg_{23} seg_{24}
seg_{01} seg_{11} seg_{21} seg_{02} seg_{12} seg_{22} seg_{01} seg_{03} seg_{11} seg_{13} seg_{21} seg_{23} seg_{02} seg_{04} seg_{12} seg_{14} seg_{22} seg_{24} seg_{03} seg_{13} seg_{23} seg_{04} seg_{14} seg_{24}

Figure 3.4: Example parallel streaming PREM schedule on 3 cores. Up arrows represent load memory phases and down arrows unload memory phases.

execution segment	loop variable values	PREM API calls	load/unload in parallel	SPM status at the end of segment

$seg_{0,0}$	s1_0_t=0, p_t=0	**alloc RO** U_ifog_buf$_1$, U_ifog_buf$_2$;
		alloc RO inp_F_buf$_1$, inp_F_buf$_2$;
		alloc WO ifog_buf$_1$, ifog_buf$_2$;
		swap $\hat{\mathcal{R}}_{\text{U\_ifog}}(seg_{0,1})$ **with** U_ifog_buf$_1$;
		swap $\hat{\mathcal{R}}_{\text{inp\_F}}(seg_{0,1})$ **with** inp_F_buf$_1$;
		swap $\hat{\mathcal{R}}_{\text{ifog}}(seg_{0,1})$ **with** ifog_buf$_1$;
		dispatch;
		swap $\hat{\mathcal{R}}_{\text{U\_ifog}}(seg_{0,2})$ **with** U_ifog_buf$_2$;
		swap $\hat{\mathcal{R}}_{\text{inp\_F}}(seg_{0,2})$ **with** inp_F_buf$_2$;
		end_segment;
				U_ifog_buf$_1$ = **empty**
				U_ifog_buf$_2$ = **empty**
				inp_F_buf$_1$ = **empty**
				inp_F_buf$_2$ = **empty**
				ifog_buf$_1$ = **empty**
				ifog_buf$_2$ = **empty**

after $seg_{0,0}$
before $seg_{0,1}$
	s1_0_t=0, p_t=0		// Load data for $seg_{0,1}$
			load $\hat{\mathcal{R}}_{\text{U\_ifog}}(seg_{0,1})$ **to** U_ifog_buf$_1$;
			load $\hat{\mathcal{R}}_{\text{inp\_F}}(seg_{0,1})$ **to** inp_F_buf$_1$;
				U_ifog_buf$_1$ = $\hat{\mathcal{R}}_{\text{U\_ifog}}(seg_{0,1})$
				U_ifog_buf$_2$ = **empty**
				inp_F_buf$_1$ = $\hat{\mathcal{R}}_{\text{inp\_F}}(seg_{0,1})$
				inp_F_buf$_2$ = **empty**
				ifog_buf$_1$ = **empty**
				ifog_buf$_2$ = **empty**

$seg_{0,1}$	s1_0_t=0, p_t=0	**swap** $\hat{\mathcal{R}}_{\text{U\_ifog}}(seg_{0,3})$ **with** U_ifog_buf$_1$;
		swap $\hat{\mathcal{R}}_{\text{inp\_F}}(seg_{0,3})$ **with** inp_F_buf$_1$;
		swap $\hat{\mathcal{R}}_{\text{ifog}}(seg_{0,3})$ **with** ifog_buf$_2$;
		end_segment;
			// Load data for $seg_{0,2}$
			load $\hat{\mathcal{R}}_{\text{U\_ifog}}(seg_{0,2})$ **to** U_ifog_buf$_2$;
			load $\hat{\mathcal{R}}_{\text{inp\_F}}(seg_{0,2})$ **to** inp_F_buf$_2$;
				U_ifog_buf$_1$ = $\hat{\mathcal{R}}_{\text{U\_ifog}}(seg_{0,1})$
				U_ifog_buf$_2$ = $\hat{\mathcal{R}}_{\text{U\_ifog}}(seg_{0,2})$
				inp_F_buf$_1$ = $\hat{\mathcal{R}}_{\text{inp\_F}}(seg_{0,1})$
				inp_F_buf$_2$ = $\hat{\mathcal{R}}_{\text{inp\_F}}(seg_{0,2})$
				ifog_buf$_1$ = $\hat{\mathcal{R}}_{\text{ifog}}(seg_{0,1})$
				ifog_buf$_2$ = **empty**

$seg_{0,2}$	s1_0_t=0, p_t=1	**swap** $\hat{\mathcal{R}}_{\text{U\_ifog}}(seg_{0,4})$ **with** U_ifog_buf$_1$;
		swap $\hat{\mathcal{R}}_{\text{inp\_F}}(seg_{0,4})$ **with** inp_F_buf$_1$;
		dealloc ifog_buf$_1$;
		end_segment;
			// Load data for $seg_{0,3}$
			load $\hat{\mathcal{R}}_{\text{U\_ifog}}(seg_{0,3})$ **to** U_ifog_buf$_1$;
			load $\hat{\mathcal{R}}_{\text{inp\_F}}(seg_{0,3})$ **to** inp_F_buf$_1$;
				U_ifog_buf$_1$ = $\hat{\mathcal{R}}_{\text{U\_ifog}}(seg_{0,3})$
				U_ifog_buf$_2$ = $\hat{\mathcal{R}}_{\text{U\_ifog}}(seg_{0,2})$
				inp_F_buf$_1$ = $\hat{\mathcal{R}}_{\text{inp\_F}}(seg_{0,3})$
				inp_F_buf$_2$ = $\hat{\mathcal{R}}_{\text{inp\_F}}(seg_{0,2})$
				ifog_buf$_1$ = $\hat{\mathcal{R}}_{\text{ifog}}(seg_{0,2})$
				ifog_buf$_2$ = **empty**

$seg_{0,3}$	s1_0_t=1, p_t=0	**dealloc** U_ifog_buf$_1$;
		dealloc inp_F_buf$_1$;
		end_segment;
			// Unload data for $seg_{0,1}$ and $seg_{0,2}$
			unload $\hat{\mathcal{R}}_{\text{ifog}}(seg_{0,2})$ **from** ifog_buf$_1$;
			// Load data for $seg_{0,4}$
			load $\hat{\mathcal{R}}_{\text{U\_ifog}}(seg_{0,4})$ **to** U_ifog_buf$_2$;
			load $\hat{\mathcal{R}}_{\text{inp\_F}}(seg_{0,4})$ **to** inp_F_buf$_2$;
				U_ifog_buf$_1$ = $\hat{\mathcal{R}}_{\text{U\_ifog}}(seg_{0,3})$
				U_ifog_buf$_2$ = $\hat{\mathcal{R}}_{\text{U\_ifog}}(seg_{0,4})$
				inp_F_buf$_1$ = $\hat{\mathcal{R}}_{\text{inp\_F}}(seg_{0,3})$
				inp_F_buf$_2$ = $\hat{\mathcal{R}}_{\text{inp\_F}}(seg_{0,4})$
				ifog_buf$_2$ = $\hat{\mathcal{R}}_{\text{ifog}}(seg_{0,3})$

$seg_{0,4}$	s1_0_t=1, p_t=1	**dealloc** U_ifog_buf$_2$;
		dealloc inp_F_buf$_2$;
		dealloc ifog_buf$_2$;
		end_segment;
				U_ifog_buf$_2$ = $\hat{\mathcal{R}}_{\text{U\_ifog}}(seg_{0,4})$
				inp_F_buf$_2$ = $\hat{\mathcal{R}}_{\text{inp\_F}}(seg_{0,4})$
				ifog_buf$_2$ = $\hat{\mathcal{R}}_{\text{ifog}}(seg_{0,1})$

after $seg_{0,4}$			// Unload data for $seg_{0,4}$ **unload** $\hat{\mathcal{R}}_{\text{ifog}}(seg_{0,4})$ **from** ifog_buf$_2$;	

Table 3.1: API operation and SPM status in each segment on Core 0

To illustrate the usage of the PREM API to schedule the data transfer and creation of segments, we again consider the example of the LSTM kernel in Listings 3.1 and 3.2. Table 3.1 details the API calls inserted in each segment on core 0, the load/unload phases performed in parallel which each segment, and the state of the SPM at the end of each segment. Note that for simplicity, the arrays used in the tilable component are divided into three groups, and we show API calls performed on all arrays within each group: specifically, group `U_ifog` comprises arrays `U_i` , `U_f` , `U_o` , `U_g` , group `ifog` comprises arrays `i` , `f` , `o` , `g` , while `inp_F` represents the single array with the same name. The corresponding PREM schedule is shown in Figure 3.4. We next discuss the usage of the API and how API calls in Table 3.1 correspond to the three macro statements in Listing 3.2.

We have the initialization segment $seg_{0,0}$ to allocate the buffers on each core's SPM and perform data transfer required by the first execution segment $seg_{0,1}$. As is shown in the first row of Table 3.1, first we allocate buffers with given `R/W` types, then we call swap to transfer the canonical data element range required by the segment $seg_{0,1}$. After that, we call `dispatch` to order these data transfers for $seg_{0,1}$ to be executed after the end of $seg_{0,0}$. These API calls are generated and inserted to the macro `BUFFER_ALLOC_APIS` in Listing 3.2.

After the `dispatch` call, we generate the API calls of data transfers for segments other than the first segment. These APIs are placed in the macro `DATA_SWAP_APIS` . This includes `swap` calls, `dealloc` calls and `end_segment` calls. Notice, the segment where we execute `swap` call is different from the segment that requires the data of the `swap` call. We determine the segment where to place each `swap` call based on the following constraints: (1) For the first two segments that requires data transfer, if data is accessed in $seg_{i,j}$, then the `swap` call of this data must be executed in $seg_{i,j-2}$ (or $seg_{i,0}$ before the dispatch if $j-2=-1$) because this would make the actual data transfer happen at $seg_{i,j-1}$, right before the execution of $seg_{i,j}$. (2) For other segments, if the same data is accessed in segments $seg_{i,j}$ to $seg_{i,k}$, then the `swap` call to unload data of segments $seg_{i,j}$-$seg_{i,k}$ and load data for the 2nd segment afterward that requires data transfer is executed in $seg_{i,k}$. This makes the actual data transfer happen during $seg_{i,k+1}$. (3) For the last two segments that require data transfer, after the segment that last accesses the data, the `dealloc` call would be executed to deallocate the buffers and unload the data if there is written access.

In the example of Table 3.1, `DATA_SWAP_APIS` includes the two `swap` calls and `end_segment` call after `dispatch` call in $seg_{0,0}$ of Table 3.1. It also includes all other `swap` calls, `end_segment` calls and `dealloc` calls from $seg_{0,1}$ to $seg_{0,3}$ except the `dealloc` and `end_segment` calls in $seg_{0,4}$.

In the end, we generate the `dealloc` and `end_segment` calls in $seg_{0,4}$ in the macro `BUFFER_DEALLOC_APIS`. After this `end_segment` call, the write-out data of $seg_{0,4}$ would be unloaded to main memory and all the buffers allocated on SPM are deallocated.

We generate API calls for these three different macros because we want to perform data transfer operations at different positions. The APIs in `BUFFER_ALLOC_APIS` are executed before all execution segments. That is why we place the array allocation process, data transfer for data required by the first execution segment and dispatch in the initialization segment into this macro. The second macro `DATA_SWAP_APIS` does the major job of data transferring for segments other than the first segment. We put the part of the initialization segment that transfer data for the second segment in this macro. The macro `DATA_SWAP_APIS` also performs buffer deallocation for segment $seg_{0,2}$ and segment $seg_{0,3}$ because we use two buffers for streaming execution. Segment $seg_{0,2}$ is the second last segment that accesses the first buffers of `ifog` and segment $seg_{0,3}$ is the second last segment that accesses the first buffer of `U_ifog` and `inp_F`. The third macro `BUFFER_DEALLOC_APIS` is only for the last segment because the macro `DATA_SWAP_APIS` is placed at the beginning of the tiled loop. Without the `end_segment` call in macro `BUFFER_DEALLOC_APIS` after all segments, the buffer deallocation would not be performed for the last segment.

Next, we discuss the logic of generating these API calls and adding them to the code. The challenges in this task mainly lie in the following five points:

1. Before the execution of any segment, we must allocate two buffers on SPM for each array that is accessed in the tilable component.

2. Different cores might execute different numbers of segments. In this case, we must generate different API calls for each core.

3. We need to determine in which segments to add each `swap` call, as well as the required `deallocate` calls, for every array.

4. Since we allocate two buffers on the SPM for every array, we have to modify the code so that for every original statement that accessed the array in main memory, we instead access one of the two buffers; note that the accessed buffer must further depend on the segment.

5. Finally, we also need to determine the parameters for each `swap` call.

To address all these challenges, we rely on the information provided by compiler analysis. Specifically, for each array a used in this tilable component and every segment $seg_{i,j}$, we rely on its canonical data element range $\hat{\mathcal{R}}_a(seg_{i,j})$. Then, we decide what are the segments that access a different part of the array from the previous segment. We use $SegmentToSwap_a(i)$ to denote the tuple of segments that requires data transfer for array a on core i. For segment $seg_{i,j}$ starting from the segment $seg_{i,1}$ to last segment on core i, if $\hat{\mathcal{R}}_a(seg_{i,j-1}) \neq \hat{\mathcal{R}}_a(seg_{i,j})$, then the segment $seg_{i,j}$ is added to tuple $SegmentToSwap_a(i)$. Segment $seg_{i,1}$ is always in $SegmentToSwap_a(i)$ because its canonical data element range is not on SPM before its execution. We use $ST^i_a(x)$ to denote the segment index of x-th element in $SegmentToSwap_a(i)$. For example, if $SegmentToSwap_a(i) = \{seg_{i,1}, seg_{i,3}, seg_{i,4}, seg_{i,7}\}$, then $ST^i_a(1) = 1$, $ST^i_a(2) = 3$, $ST^i_a(3) = 4$, $ST^i_a(4) = 7$.

In the example of Table 3.1, core 0 executes 4 segments (the other 2 cores also execute 4 segments each). These four segments on core 0 are $seg_{0,1}$, $seg_{0,2}$, $seg_{0,3}$, $seg_{0,4}$. $seg_{0,1}$ executes l_{s1} from 0 to 108 and $l.p$ from 0 to 349, $seg_{0,2}$ executes l_{s1} from 0 to 108 and $l.p$ from 350 to 699, $seg_{0,3}$ executes l_{s1} from 109 to 317 and $l.p$ from 0 to 349, $seg_{0,3}$ executes l_{s1} from 109 to 317 and $l.p$ from 350 to 699.

Arrays `U_i`, `U_f`, `U_o`, `U_g` in group `U_ifog` are only read in the tilable component. We denote the canonical data element range of each segment as $\hat{\mathcal{R}}_{\text{U\_ifog}}(seg_{0,j})$. The canonical data element range of $seg_{1,1}$ is `U_ifog[0-108][0-349]` according to code in Listing 3.2. Similarly, $\hat{\mathcal{R}}_{\text{U\_ifog}}(seg_{0,2})$ is `U_ifog[0-108][350-699]`, $\hat{\mathcal{R}}_{\text{U\_ifog}}(seg_{0,3})$ is `U_ifog[109-217][0-349]` and $\hat{\mathcal{R}}_{\text{U\_ifog}}(seg_{0,4})$ is `U_ifog[109-217][350-699]`. We can see that all these canonical data element ranges are not the same for each segment. Hence, $SegmentToSwap_{\text{U\_ifog}}(0)$ contains all the segments on core 0 which are $\{seg_{0,1}, seg_{0,2}, seg_{0,3}, seg_{0,4}\}$.

Array `inp_F`, is also only read in the component. Similar to array group `U_ifog`, $SegmentToSwap_{\text{inp\_F}}(0)$ contains all the segments on core 0 which are $\{seg_{0,1}, seg_{0,2}, seg_{0,3}, seg_{0,4}\}$.

Arrays `i`, `f`, `o`, `g` in group `ifog` are both read and written in the component. However, since every data element in the array is written in the tilable component, and it is written before being read, we allocate them as `WO` in this tilable component. Different from previous two groups, this group of arrays have some canonical data element ranges equal to each other. $\hat{\mathcal{R}}_{\text{ifog}}(seg_{0,1}) = \hat{\mathcal{R}}_{\text{ifog}}(seg_{0,2})$ and $\hat{\mathcal{R}}_{\text{ifog}}(seg_{0,3}) = \hat{\mathcal{R}}_{\text{ifog}}(seg_{0,4})$. Hence, $SegmentToSwap_{\text{ifog}}(0) = \{seg_{0,1}, seg_{0,3}\}$.

With the information of *SegmentToSwap* and canonical data element ranges, next we demonstrate how to address the five points discussed above. The code of Listing 3.2 with API calls inserted is shown in Listing 3.3. For each tilable component, we use the variable *segCount* to keep track of which segment is currently under execution. In this example shown in Listing 3.3, the name of *segCount* is `s1_0_p_seg_count` to prevent naming conflict.

For the first point, we allocate two buffers on SPM for each array accessed in tilable component. The name of the pointers for these buffers are constructed by taking the name of the array and adding either `_buf1` or `_buf2`. For example, in Table 3.1, the buffer pointers for array `inp_F` are named `inp_F_buf1` and `inp_F_buf2`. These two buffers are allocated on two different partitions on SPM. Assume the starting address of the first partition on core i is $addr_1(i)$ and the starting address of the second partition on core i is $addr_2(i)$. Assume `inp_F_buf1` is the first array that has buffer allocated on SPM, the pointer to the first buffer for the array `inp_F_buf1` on core i is assigned to $addr_1(i)$ and the pointer to the second buffer is $addr_2(i)$. The pointer to the first buffer for the next array would be $addr_1(i)$ plus the size of `inp_F_buf1` and this is the same for the pointers for all the successive arrays. The size of `inp_F_buf1` would be the maximum size of canonical data element ranges among all the segments executed on core i, we call this size *bounding box*. The detailed process of computing the bounding box of a given array will be discussed in Section 5.3.1. The code that sets the value of `buf1`s and `buf2`s is inserted in macro `BUFFER_ALLOC_APIS` so that these pointers are available during the execution of the current tilable component.

For the second point, we compare if the index of segments in $SegmentToSwap_a(i)$ is the same for all cores, formally,

$$\forall 0 \le i, j \le P-1, \forall k, seg_{i,k} \in SegmentToSwap_a(i) \iff seg_{j,k} \in SegmentToSwap_a(j) \tag{3.1}$$

If Equation 3.1 holds, then the API calls we generate are the same for all cores (albeit with different parameters for the `swap` calls). Otherwise, we generate a different set of API calls for each core, and at run-time select among them based on the threadID. In our example, for arrays `i`, `f`, `o`, `g`, $SegmentToSwap_{\text{ifog}}(0) = \{seg_{0,1}, seg_{0,3}\}$, $SegmentToSwap_{\text{ifog}}(1) = \{seg_{1,1}, seg_{1,3}\}$, $SegmentToSwap_{\text{ifog}}(2) = \{seg_{2,1}, seg_{2,3}\}$. Equation 3.1 holds true for array `i`, `f`, `o`, `g` and also for the other two groups of arrays. Thus, the same set of API calls as in Table 3.1 apply to all cores.

For the third point, the information of segments that require data transfer are stored in $SegmentToSwap_a(i)$. To ensure that `swap` calls are executed in the correct segments, we use conditional statements in `DATA_SWAP_APIS` such that a swap call is only executed when a corresponding condition holds. Such condition is constructed differently under two situations. We say that an array a has a *constant change stride* if $ST_a^i(x+1) - ST_a^i(x)$ is constant for all x; in this case, we call such constant the *change stride* of the array. For example, in Table 3.1, the array group `U_i`, `U_f`, `U_o`, `U_g` and array `inp_F` all have a change stride of 1 because in $SegmentToSwap_{\text{U\_ifog}}(0)$ and $SegmentToSwap_{\text{inp\_F}}(0)$, the distance between the segments is 1. And array group `i`, `f`, `o`, `g` have a change stride of 2 because in $SegmentToSwap_{\text{ifog}}(i) = \{seg_{i,1}, seg_{i,3}\}$ the distance between the segments is 2. We next first discuss the case of constant change stride arrays, and then discuss the case of arrays that do not have a constant change stride.

In the former case, we can compute $ST_a^i(x)$ given the value of change stride $changeStride$. Since the first segment $seg_{i,j}$ is always in $SegmentToSwap_a(i)$, we have $ST_a^i(1) = 1$. Because $changeStride = ST_a^i(x+1) - ST_a^i(x)$, we can compute $ST_a^i(x) = 1 + changeStride \cdot (x-1)$.

For the first two segments in $SegmentToSwap_a(i)$, which are $seg_{i,ST_a^i(1)}$ and $seg_{i,ST_a^i(2)}$, the segment that performs the `swap` call for them is two segments before. Since $seg_{i,ST_a^i(1)} = seg_{i,1}$, the `swap` call for it is executed in $seg_{i,0}$ before the dispatch as part of macro `BUFFER_ALLOC_APIS`, and the first buffer is used for data swap. The second segment in $SegmentToSwap_a(i)$ is $seg_{i,ST_a^i(2)} = seg_{i,1+changeStride}$, the `swap` call for it is executed in $seg_{i,changeStride-1}$ (note that is $seg_{i,0}$ after the dispatch if $changeStride = 1$). To select this segment $seg_{i,changeStride-1}$, we simply use condition $segCount = changeStride - 1$ in a conditional statement in the macro `DATA_SWAP_APIS`, and use the second buffer for data swap.

For each other segment $seg_{i,ST_a^i(x)}$ in $SegmentToSwap_a(i)$, the `swap` call for it is executed at the last segment that accesses the data used in $seg_{i,ST_a^i(x-2)}$, which is segment $seg_{i,ST_a^i(x-1)} - 1$. This ensures the write-out data be unloaded from SPM to main memory immediately after execution of segments using such data is completed. To select this segment $seg_{i,ST_a^i(x-1)-1}$, we have $segCount = ST_a^i(x-1) - 1$ when $x = 3, 4, \ldots$ $length(SegmentToSwap_a(i))$. Since $ST_a^i(x) = 1 + changeStride \cdot (x-1)$, $ST_a^i(x-1) - 1 = changeStride \cdot (x-2)$. Thus, we have $segCount = changeStride \cdot (x-2)$ and $x - 2 < length(SegmentToSwap_a(i)) - 1$. We can also compute $x = segCount/changeStride + 2$. When $x = 3, 5, 7, \ldots$, the first buffer is used for data swap and when $x = 4, 6, 8, \ldots$, the second buffer is used. In summary, we use condition $segCount \bmod changeStride = 0 \wedge segCount < changeStride \cdot (length(SegmentToSwap_a(i)) - 1)$ to select these segments, and when $(segCount/changeStride + 2) \bmod 2 = 1$, the first buffer is used, otherwise the second buffer is used.

For the last two segments in $SegmentToSwap_a(i)$, in addition to the `swap` call described above, we also need to use `dealloc` call to deallocate the buffer and unload the data from SPM to main memory. Notice the `dealloc` call for last segment in $SegmentToSwap_a(i)$ is inserted in macro `BUFFER_DEALLOC_APIS`, so there is no need to generate a conditional statement for it. We just check $length(SegmentToSwap_a(i)) \bmod 2$, if it is 1, we deallocate the first buffer, otherwise we deallocate the second buffer. For the `dealloc` call for second last segment, it is selected by condition $segCount = changeStride \cdot (length(SegmentToSwap_a(i)) - 1)$. This is inserted in macro `DATA_SWAP_APIS`.

For arrays without a constant change stride, a different approach is required. In this case, we use bit vectors to store the segments when `swap` and deallocate calls must be made in `DATA_SWAP_APIS`, as well as which buffer to use. For example, when $SegmentToSwap_a(i) = \{seg_{i,1}, seg_{i,2}, seg_{i,4}, seg_{i,5}\}$ and core i executes a total of 8 segments, then the bit vector encoding the `swap` calls is `0b00001011`, i.e., `swap` calls must be performed in $seg_{i,0}$ (after dispatch, for segment $seg_{i,2}$), $seg_{i,1}$ (for segment $seg_{i,4}$), and $seg_{i,3}$ (for segment $seg_{i,5}$) - in addition to $seg_{i,0}$ before the dispatch for segment $seg_{i,1}$, which we remind is performed in `BUFFER_ALLOC_APIS`.

For the fourth point, to avoid modifying statements inside the loop that use the array a, we employ the following strategy: we redefine a to be a pointer with the same type as the original array. I.e., if a was declared as `double a[3][4]`, then we redefine it as a pointer `double (*a)[4]`. To associate to the pointer the value of either `a_buf1` / `a_buf2` at the correct time, in the macro `DATA_SWAP_APIS`, we execute statements `a=a_buf1` or `a=a_buf2` whenever a segment in $SegmentToSwap_a(i)$ is encountered, using similar conditions as the one discussed for point two.

For the fifth point, each data swap has different input parameters, specifically the starting address in main memory and the data size (pitches remain equal, since they depend on the shape of the original array in main memory, and the allocated buffer in SPM). Such parameters can be calculated given the canonical data element range $\hat{\mathcal{R}}_a(seg_{i,j})$, as we will detail later in Section 5.3.2. Once we have these values of input parameters, we create a global array to store the values of input parameters of `swap` call for each segment that requires data swap. Then, the index x of the current segment in $SegmentToSwap_a(i)$ is used to select the corresponding input parameters. As an example, for array group `ifog` in Table 3.1, we create the array as is shown in Table 3.2 to store the parameters of swap calls for corresponding segments. The table is inserted in the macro `BUFFER_ALLOC_APIS`. In Listing 3.3, the table that stores the `swap` call parameters for the array i is named as `i_swap_params`.

execution segment	starting address in main memory	data size
$seg_{0,0}$	(int32_t*)ifog	109*4
$seg_{0,1}$	(int32_t*)ifog+109	109*4
$seg_{1,0}$	(int32_t*)ifog+218	109*4
$seg_{1,1}$	(int32_t*)ifog+327	109*4
$seg_{2,0}$	(int32_t*)ifog+436	109*4
$seg_{2,1}$	(int32_t*)ifog+545	105*4

Table 3.2: `seg_count` to swap input parameters

```
static int s1_0_p_seg_count = 0;
/* Assume each array element has length of 4 bytes */
#define DTYPE_LEN 4

struct param_1d {
  uint64_t* starting_addr;
  int size;
};

struct param_2d {
  uint64_t* starting_addr;
  int width;
  int height;
};

for (int t = 0; t < NT; t++) {
  /* transformed code of component (s1_0, p) start */
  /* content of swap_params shown in Table 3.2 */
  struct param_1d i_swap_params[3][2] = {...};
  struct param_1d f_swap_params[3][2] = {...};
  struct param_1d o_swap_params[3][2] = {...};
  struct param_1d g_swap_params[3][2] = {...};
  struct param_2d U_i_swap_params[3][4] = {...};
  struct param_2d U_f_swap_params[3][4] = {...};
  struct param_2d U_o_swap_params[3][4] = {...};
  struct param_2d U_g_swap_params[3][4] = {...};
  struct param_1d inp_F_swap_params[3][4] = {...};
  /* set values for buffer pointers on SPM */
  int32_t *i_buf1 = ...;
  int32_t *i_buf2 = ...;
  int32_t *f_buf1 = ...;
  int32_t *f_buf2 = ...;
  int32_t *o_buf1 = ...;
  int32_t *o_buf2 = ...;
  int32_t *g_buf1 = ...;
  int32_t *g_buf2 = ...;
  int32_t (*U_i_buf1)[350] = ...;
  int32_t (*U_i_buf2)[350] = ...;
  int32_t (*U_f_buf1)[350] = ...;
  int32_t (*U_f_buf2)[350] = ...;
  int32_t (*U_o_buf1)[350] = ...;
  int32_t (*U_o_buf2)[350] = ...;
  int32_t (*U_g_buf1)[350] = ...;
  int32_t (*U_g_buf2)[350] = ...;
  int32_t *inp_F_buf1 = ...;
  int32_t *inp_F_buf2 = ...;
  /* allocate buffers */
  I1 = allocate_buffer(i_buf1, WO);
  I2 = allocate_buffer(i_buf2, WO);
  F1 = allocate_buffer(f_buf1, WO);
  F2 = allocate_buffer(f_buf2, WO);
  O1 = allocate_buffer(o_buf1, WO);
  O2 = allocate_buffer(o_buf2, WO);
  G1 = allocate_buffer(g_buf1, WO);
  G2 = allocate_buffer(g_buf2, WO);
  U_I1 = allocate_buffer(U_i_buf1, RO);
  U_I2 = allocate_buffer(U_i_buf2, RO);
  U_F1 = allocate_buffer(U_f_buf1, RO);
  U_F2 = allocate_buffer(U_f_buf2, RO);
  U_O1 = allocate_buffer(U_o_buf1, RO);
  U_O2 = allocate_buffer(U_o_buf2, RO);
  U_G1 = allocate_buffer(U_g_buf1, RO);
  U_G2 = allocate_buffer(U_g_buf2, RO);
  INP_F1 = allocate_buffer(inp_F_buf1, RO);
```

```
  INP_F2 = allocate_buffer(inp_F_buf2, RO);
  swap_buffer(I1, i_swap_params[threadID()][0].starting_addr, i_swap_params[threadID()][0].size);
  swap_buffer(F1, f_swap_params[threadID()][0].starting_addr, f_swap_params[threadID()][0].size);
  swap_buffer(O1, o_swap_params[threadID()][0].starting_addr, o_swap_params[threadID()][0].size);
  swap_buffer(G1, g_swap_params[threadID()][0].starting_addr, g_swap_params[threadID()][0].size);
  swap2d_buffer(U_I1, U_i_swap_params[threadID()][0].starting_addr, U_i_swap_params[threadID()][0].width, U_i_swap_params[threadID()][0].height, NP * DTYPE_LEN, 350 * DTYPE_LEN);
  swap2d_buffer(U_F1, U_f_swap_params[threadID()][0].starting_addr, U_f_swap_params[threadID()][0].width, U_f_swap_params[threadID()][0].height, NP * DTYPE_LEN, 350 * DTYPE_LEN);
  swap2d_buffer(U_O1, U_o_swap_params[threadID()][0].starting_addr, U_o_swap_params[threadID()][0].width, U_o_swap_params[threadID()][0].height, NP * DTYPE_LEN, 350 * DTYPE_LEN);
  swap2d_buffer(U_G1, U_g_swap_params[threadID()][0].starting_addr, U_g_swap_params[threadID()][0].width, U_g_swap_params[threadID()][0].height, NP * DTYPE_LEN, 350 * DTYPE_LEN);
  swap_buffer(INP_F1, inp_F_swap_params[threadID()][0].starting_addr, inp_F_swap_params[threadID()][0].size);
  dispatch();
  for (int s1_0_t = threadID() * 2; s1_0_t < (threadID() + 1) * 2; s1_0_t += 1) {
    for (int p_t = 0; p_t < 2; p_t += 1) {
      /* API call executed at the end of each segment start */
      {
        /* set alias variable for current tile based on tile count */
        /* When segments are seg1, seg2 */
        if ((s1_0_p_seg_count / 2) % 2 == 0) {
          i = i_buf1;
          f = f_buf1;
          o = o_buf1;
          g = g_buf1;
        } else {
          i = i_buf2;
          f = f_buf2;
          o = o_buf2;
          g = g_buf2;
        }
        /* When segments are seg1, seg3 */
        if (s1_0_p_seg_count % 2 == 0) {
          U_i = U_i_buf1;
          U_f = U_f_buf1;
          U_o = U_o_buf1;
          U_g = U_g_buf1;
          inp_F = inp_F_buf1;
        } else {
          U_i = U_i_buf2;
          U_f = U_f_buf2;
          U_o = U_o_buf2;
          U_g = U_g_buf2;
          inp_F = inp_F_buf2;
        }
        if (s1_0_p_seg_count == 0) {
          swap2d_buffer(U_I2, U_i_swap_params[threadID()][1].starting_addr, U_i_swap_params[threadID()][1].width, U_i_swap_params[threadID()][1].height, NP * DTYPE_LEN, 350 * DTYPE_LEN);
          swap2d_buffer(U_F2, U_f_swap_params[threadID()][1].starting_addr, U_f_swap_params[threadID()][1].width, U_f_swap_params[threadID()][1].height, NP * DTYPE_LEN, 350 * DTYPE_LEN);
          swap2d_buffer(U_O2, U_o_swap_params[threadID()][1].starting_addr, U_o_swap_params[threadID()][1].width, U_o_swap_params[threadID()][1].height, NP * DTYPE_LEN, 350 * DTYPE_LEN);
          swap2d_buffer(U_G2, U_g_swap_params[threadID()][1].starting_addr, U_g_swap_params[threadID()][1].width, U_g_swap_params[threadID()][1].height, NP * DTYPE_LEN, 350 * DTYPE_LEN);
          swap_buffer(INP_F1, inp_F_swap_params[threadID()][1].starting_addr, inp_F_swap_params[threadID()][1].size);
        }
        if (s1_0_p_seg_count == 1) {
          swap_buffer(I2, i_swap_params[threadID()][1].starting_addr, i_swap_params[threadID()][1].size);
          swap_buffer(F2, f_swap_params[threadID()][1].starting_addr, f_swap_params[threadID()][1].size);
          swap_buffer(O2, o_swap_params[threadID()][1].starting_addr, o_swap_params[threadID()][1].size);
          swap_buffer(G2, g_swap_params[threadID()][1].starting_addr, g_swap_params[threadID()][1].size);
        }
        if (s1_0_p_seg_count % 2 == 0 && s1_0_p_seg_count < 2 && (s1_0_p_seg_count / 2) % 2 == 1) {
          swap_buffer(I1, i_swap_params[threadID()][s1_0_p_seg_count / 2 + 1].starting_addr, i_swap_params[threadID()][(s1_0_p_seg_count + 2) / 2].size);
          swap_buffer(F1, f_swap_params[threadID()][s1_0_p_seg_count / 2 + 1].starting_addr, f_swap_params[threadID()][(s1_0_p_seg_count + 2) / 2].size);
```

```
            swap_buffer(O1, o_swap_params[threadID()][s1_0_p_seg_count / 2 + 1].starting_addr, o_swap_params[threadID()][(s1_0_p_seg_count + 2) / 2].size);
            swap_buffer(G1, g_swap_params[threadID()][s1_0_p_seg_count / 2 + 1].starting_addr, g_swap_params[threadID()][(s1_0_p_seg_count + 2) / 2].size);
        }
        if (s1_0_p_seg_count % 2 == 0 && s1_0_p_seg_count < 2 && (s1_0_p_seg_count / 2) % 2 == 0) {
            swap_buffer(I2, i_swap_params[threadID()][s1_0_p_seg_count / 2 + 1].starting_addr, i_swap_params[threadID()][(s1_0_p_seg_count + 2) / 2].size);
            swap_buffer(F2, f_swap_params[threadID()][s1_0_p_seg_count / 2 + 1].starting_addr, f_swap_params[threadID()][(s1_0_p_seg_count + 2) / 2].size);
            swap_buffer(O2, o_swap_params[threadID()][s1_0_p_seg_count / 2 + 1].starting_addr, o_swap_params[threadID()][(s1_0_p_seg_count + 2) / 2].size);
            swap_buffer(G2, g_swap_params[threadID()][s1_0_p_seg_count / 2 + 1].starting_addr, g_swap_params[threadID()][(s1_0_p_seg_count + 2) / 2].size);
        }
        if (s1_0_p_seg_count < 3 && s1_0_p_seg_count % 2 == 1) {
            swap2d_buffer(U_I1, U_i_swap_params[threadID()][s1_0_p_seg_count + 1].starting_addr, U_i_swap_params[threadID()][s1_0_p_seg_count + 1].width, U_i_swap_params[threadID()][s1_0_p_seg_count + 1].height, NP * DTYPE_LEN, 350 * DTYPE_LEN);
            swap2d_buffer(U_F1, U_f_swap_params[threadID()][s1_0_p_seg_count + 1].starting_addr, U_f_swap_params[threadID()][s1_0_p_seg_count + 1].width, U_f_swap_params[threadID()][s1_0_p_seg_count + 1].height, NP * DTYPE_LEN, 350 * DTYPE_LEN);
            swap2d_buffer(U_O1, U_o_swap_params[threadID()][s1_0_p_seg_count + 1].starting_addr, U_o_swap_params[threadID()][s1_0_p_seg_count + 1].width, U_o_swap_params[threadID()][s1_0_p_seg_count + 1].height, NP * DTYPE_LEN, 350 * DTYPE_LEN);
            swap2d_buffer(U_G1, U_g_swap_params[threadID()][s1_0_p_seg_count + 1].starting_addr, U_g_swap_params[threadID()][s1_0_p_seg_count + 1].width, U_g_swap_params[threadID()][s1_0_p_seg_count + 1].height, NP * DTYPE_LEN, 350 * DTYPE_LEN);
            swap_buffer(INP_F1, inp_F_swap_params[threadID()][s1_0_p_seg_count + 1].starting_addr, inp_F_swap_params[threadID()][s1_0_p_seg_count + 1].size);
        }
        if (s1_0_p_seg_count < 3 && s1_0_p_seg_count % 2 == 0) {
            swap2d_buffer(U_I2, U_i_swap_params[threadID()][s1_0_p_seg_count +
1].starting_addr, U_i_swap_params[threadID()][s1_0_p_seg_count + 1].width, U_i_swap_params[threadID()][s1_0_p_seg_count + 1].height, NP * DTYPE_LEN, 350 * DTYPE_LEN);
        swap2d_buffer(U_F2, U_f_swap_params[threadID()][s1_0_p_seg_count + 1].starting_addr, U_f_swap_params[threadID()][s1_0_p_seg_count + 1].width, U_f_swap_params[threadID()][s1_0_p_seg_count + 1].height, NP * DTYPE_LEN, 350 * DTYPE_LEN);
        swap2d_buffer(U_O2, U_o_swap_params[threadID()][s1_0_p_seg_count + 1].starting_addr, U_o_swap_params[threadID()][s1_0_p_seg_count + 1].width, U_o_swap_params[threadID()][s1_0_p_seg_count + 1].height, NP * DTYPE_LEN, 350 * DTYPE_LEN);
        swap2d_buffer(U_G2, U_g_swap_params[threadID()][s1_0_p_seg_count + 1].starting_addr, U_g_swap_params[threadID()][s1_0_p_seg_count + 1].width, U_g_swap_params[threadID()][s1_0_p_seg_count + 1].height, NP * DTYPE_LEN, 350 * DTYPE_LEN);
        swap_buffer(INP_F2, inp_F_swap_params[threadID()][s1_0_p_seg_count + 1].starting_addr, inp_F_swap_params[threadID()][s1_0_p_seg_count + 1].size);
      }
      if (s1_0_p_seg_count == 2) {
        deallocate(I1);
        deallocate(F1);
        deallocate(O1);
        deallocate(G1);
      }
      if (s1_0_p_seg_count == 3) {
        deallocate(U_I1);
        deallocate(U_F1);
        deallocate(U_O1);
        deallocate(U_G1);
        deallocate(INP_F1);
      }
      s1_0_p_seg_count++;
      end_segment();
    }
    /* API call executed at the end of each segment end */
    for (int s1_0 = s1_0_t * 109; s1 < MIN(NS, s1_0_t * 109 + 109); s1_0++) {
      for (int p = p_t * 350; p < MIN(NP, p_t * 350 + 350); p++) {
        if (p == 0) {
          i[s1_0 - s1_0_t * 109] = 0.0;
          f[s1_0 - s1_0_t * 109] = 0.0;
          o[s1_0 - s1_0_t * 109] = 0.0;
          g[s1_0 - s1_0_t * 109] = 0.0;
        }
```

```
            i[s1_0 - s1_0_t * 109] += U_i[s1_0 - s1_0_t * 109][p - p_t * 350]
  * inp_F[p - p_t * 350];
            f[s1_0 - s1_0_t * 109] += U_f[s1_0 - s1_0_t * 109][p - p_t * 350]
  * inp_F[p - p_t * 350];
            o[s1_0 - s1_0_t * 109] += U_o[s1_0 - s1_0_t * 109][p - p_t * 350]
  * inp_F[p - p_t * 350];
            g[s1_0 - s1_0_t * 109] += U_g[s1_0 - s1_0_t * 109][p - p_t * 350]
  * inp_F[p - p_t * 350];
          }
        }
      }
    }
    deallocate(I2);
    deallocate(F2);
    deallocate(O2);
    deallocate(G2);
    deallocate(U_I2);
    deallocate(U_F2);
    deallocate(U_O2);
    deallocate(U_G2);
    deallocate(INP_F2);
    end_segment();
    /* transformed code of component (s1_0, p) end */
    // ......
  }
}
```
Listing 3.3: Component (l_{s1_0}, l_p) in LSTM code after inserting PREM APIs

Chapter 4

Schedule Optimization

We now show how our framework can be employed to minimize the makespan of an application scheduled according to the model in Chapter 3. Specifically, following the discussion in the previous chapter, we need to automatically perform two steps: (1) decompose the loop tree into a set of disjoint tilable components; (2) for each component, find the optimization solution that minimizes the length of the PREM schedule for that component. We note that neither of the steps is trivial, since there are potentially multiple different ways in which the loop tree could be decomposed, and many more different optimization solutions for each component.

We begin by motivating why picking the best optimization solution is important to minimize the schedule length in Section 4.1 through a clarifying example. We then discuss how the schedule length is computed based on a given optimization solution in Section 4.2. This procedure is used by the component optimizer in Section 4.3, which performs step (2) above. Finally, in Section 4.4 we show how to perform step (1).

4.1 Motivation

To better understand how the optimization solution affects the makespan of the application, we again consider the schedule for the tiled component $\mathcal{L} = (l_{s1\_0}, l_p)$ shown in Figure 3.4, where the total number of tiles is $l_{s1\_0}.M \cdot l_p.M = 12$. Let e, ld and ul denote the total length of execution, load and unload phases, respectively. Note for this execution-bound schedule, the makespan is equal to the length of the load phases for $seg_{0,1}, seg_{1,1}$ and $seg_{2,1}$, plus the four execution phases on core 2, plus the unload phase of $seg_{2,4}$. Assuming that all phases of the same type have equal length, the makespan is thus equal to $3 \cdot (ld/12) + 4 \cdot (e/12) + ul/12 = ld/4 + e/3 + ul/12$. Next, assume that tile sizes are reduced, so that the application is divided in 15 segments. Following the same scheduling logic, if the values of e, ld, ul remain the same, then the makespan decreases to $ld/5+e/3+ul/15$. In other words, increasing the number of segments decreases the overhead of memory transfers. In reality, the problem is more complex. Because tiling adds extra code overhead, in the example above the value of e would actually increase, rather than stay the same. After a certain number of segments, the overhead becomes larger than any memory gain. Furthermore, certain data elements might be reused across tiles. Hence, ld and ul might also increase, negating any makespan reduction from improved parallelism. Since data reuse across tiles is strongly dependent on which loop level is tiled, in general it is necessary to explore all levels to find a good solution.

4.2 Schedule Length

Once a optimization solution is selected, our framework would transform the code as discussed previously in Section 3.5. It also calculates the makespan of the tiling component by constructing a DAG representation of the schedule. In this DAG representation, nodes represent either segments or memory phases, and edges represent precedence constraints either for sequential phases on the same core/DMA, or for data constraints between memory and execution phases.

Note that Figure 3.4 shows the required precedence constraints for the depicted schedule. The length of each node is determined by the compiler based on data and timing analysis, and by incorporating the overhead of API calls. The makespan of the component is equal to the longest path over the DAG. To quickly compute the makespan of the schedule, we have to compute the length of execution and memory phases.

Length of execution phases. The length of an execution phase is a composite of two parts: the API calls overhead and the length of the tiled program execution. For the API call overhead, after we finish the dependency analysis of the segment, the API calls used in each tile are determined. We simply sum the time required to execute all employed API calls.

For program execution, worst-case execution time estimation can be performed by either static analysis or through a measurement-based approach; here we use the latter. Note that, as we will discuss in Section 4.3, our optimization heuristic needs to compute the DAG makespan multiple times based on different scheduling solutions. Executing the kernel to measure phase lengths every time would be too expensive. Instead, we first use measurements to build an analytical timing model that is parametric in the tile sizes, and then use this analytical model to quickly estimate execution times. Specifically, we use parameters $(l_1.O, ..., l_L.O)$ to denote the loop iteration overhead at each loop level, and W for the worst-case execution time of innermost code. We can then model the execution time of the phase as: $\sum_{j=1...L}(l_j.O \cdot \prod_{k=1...j} l_k.K) + W \cdot \prod_{j=1...L} l_j.K$. We profile the program to obtain multiple samples for the execution time under different $(l_1.K, ..., l_L.K)$ values, and perform parametric fitting on $(l_1.O, ..., l_L.O, W)$ using the least square methods along with the constraint that for each sample, the measured value of the execution time cannot be larger than the estimated one.

Length of memory phases The length of memory phases is determined by the size of data that is accessed in the corresponding segment. For a particular segment, its load-/unload phases transfer multiple multi-dimensional arrays. The details of computing the size of data arrays with tile sizes is discussed in Section 5.3.1. Next, we introduce how to estimate the time of transferring an array given its sizes.

We have used $\hat{\mathcal{R}}_a(seg_{i,j})$ to denote the canonical element range of array a that is accessed in segment $seg_{i,j}$ in Section 3.5. Here, we use $\hat{\mathcal{R}}_a$ to denote a general canonical element range of array a. We denote $Shape(\hat{\mathcal{R}}_a)$ as its shape. The shape of canonical data element range is calculated by computing the distance between maximum index and minimum index on loop level l. $Shape(\hat{\mathcal{R}}_a)_1$ is the size of its outermost dimension. If it has n dimensions meaning $n = length(Shape(\hat{\mathcal{R}}_a))$, then $Shape(\hat{\mathcal{R}}_a)_n$ is the size of its innermost dimension. The formal definition of shape will be provided in Section 5.3.1.

We compute the data transfer time of this sub-area of array a with two parts:

1. $T_{DMA}(\hat{\mathcal{R}}_a)$: DMA overhead, which increases with number of data lines.
2. $T_{BUS}(\hat{\mathcal{R}}_a)$: Bus transfer time, which is proportional to total amount of transferred data.

Data line refers to a consecutive range of locations in main memory. When the data to be transferred in inner dimensions take up the whole dimension, e.g. $Shape(\hat{\mathcal{R}}_a)_n = Shape(a)_n$, those consecutive dimensions composite a single data line. Let α denote the index of the first dimension (from 1 to n) such that, for all dimensions from the αth (included) to nth dimension, $\hat{\mathcal{R}}_a$ always takes up the whole dimension of the array a; or $\alpha = n+1$ if no such dimension exists. Then, the number of data lines contained in $\hat{\mathcal{R}}_a$ is:

$$DataLineNum(\hat{\mathcal{R}}_a) = \max\Big(1, \prod_{j=1}^{\alpha-2} Shape(\hat{\mathcal{R}}_a)_j\Big),$$

while the size of each data line in number of elements is:

$$DataLineSize(\hat{\mathcal{R}}_a) = \prod_{j=\max(1,\alpha-1)}^{n} Shape(\hat{\mathcal{R}}_a)_j.$$

Example: consider a two-dimensional array a with shape $Shape(a) = \langle 3, 5\rangle$, i.e., 3 rows and 5 columns, and assume $Shape(\hat{\mathcal{R}}_a) = \langle 2, 5\rangle$. Then $\alpha = 2$, and the transfer would require one line of size $2 \cdot 5 = 10$; this is because we need to transfer whole rows, and successive rows are stored consecutively in main memory. Similarly, consider a three-dimensional array a' with shape $Shape(a') = \langle 6, 3, 5\rangle$, i.e., such that the size of the innermost dimension is 5 and the outermost dimension 6. Assume $Shape(\hat{\mathcal{R}}_{a'}) = \langle 4, 2, 5\rangle$. Then $\alpha = 3$; the size of each line is $DataLineSize(\hat{\mathcal{R}}_{a'}) = Shape(\hat{\mathcal{R}}_{a'})_2 \cdot Shape(\hat{\mathcal{R}}_{a'})_3 = 2 \cdot 5 = 10$, while the number of lines is $DataLineNum(\hat{\mathcal{R}}_{a'}) = Shape(\hat{\mathcal{R}}_{a'})_1 = 4$.

We denote $T_{DMA}^{overhead}$ to be the DMA overhead time of one data line. Then the total DMA overhead is:

$$T_{DMA}(\hat{\mathcal{R}}_a) = T_{DMA}^{overhead} \cdot DataLine(\hat{\mathcal{R}}_a)$$

To compute total bus transfer time, we assume the bus transfer is performed through multiple burst transfers. Each burst transfer has a fixed size of data, which we call data access granularity. We denote sizeof(G) to be the number of bytes for data access granularity and denote $T_{BUS}^{overhead}$ to be the bus transfer time per burst transfer and $a.e^{type}$ to be the type of one element in the array a. First, we compute the number of burst transfers in one data line to be:

$$BurstTransfer(\hat{\mathcal{R}}_a) = \lceil DataLineSize(\hat{\mathcal{R}}_a) \cdot \mathrm{sizeof}(a.e^{type}) / \mathrm{sizeof}(G) \rceil.$$

Then the total bus transfer time is:

$$T_{BUS}(\hat{\mathcal{R}}_a) = T_{BUS}^{overhead} \cdot BurstTransfer(\hat{\mathcal{R}}_a) \cdot DataLine(\hat{\mathcal{R}}_a)$$

In the end, we compute the length of memory phase as $T_{DMA}(\hat{\mathcal{R}}_a) + T_{BUS}(\hat{\mathcal{R}}_a)$.

4.3 Tiling Component Optimization

We next discuss how to derive an optimized scheduling solution $(l_1.R, ..., l_L.R), (l_1.K, ..., l_L.K)$ for tilable component $\mathcal{L} = (l_1, ..., l_j, ..., l_L)$. Algorithm 1 returns a solution that is close to the solution that has minimal makespan for one tilable component. The most straight forward approach to find the values for the parameters set is to search the whole space of all possible values, compute the makespan for all of them and pick the values that result in the minimum makespan. However, this approach's time complexity is exponential in L which is too large for practical application. In fact, during our evaluation, even for a component that has loop bound less than 100, searching the space of possible values would take unacceptable time, usually more than 20 hours. Thus, we use Algorithm 1 which is a heuristic optimization to shorten the time of finding a good scheduling solution.

On line 3, $\mathcal{R}$ is a set of thread group assignments of the form $(l_1.R, ..., l_L.R)$. We say that assignment $(l'_1.R, ..., l'_L.R)$ dominates another assignment $(l_1.R, ..., l_L.R)$ if $\forall l_j \in \mathcal{L} : l'_j.R \geq l_j.R$. The function generate_nondominated_thread_ groups$(P, \mathcal{L})$ returns all valid assignments on P cores that are not dominated by another valid assignment. The idea is that dominated assignments do not need to be checked since there exists another assignment that has strictly better parallelism.
Example: consider a component (l_1, l_2) on $P = 10$ cores. The valid, non-dominated thread group assignments are $(10, 1), (5, 2), (3, 3)$, $(2, 5)$, and $(1, 10)$. Note that $(3, 3)$ uses less than P cores.

On lines 5-13, the algorithm iterates over the thread group assignments in $\mathcal{R}$. For each assignment, the function select_tile_sizes(l_j) is used to select a set of possible tile sizes $\mathcal{K}_j$ for loop level l_j. The function iterates over $l_j.K$ from 1 to $l_j.N$, and computes $l_j.M$ and $l_j.Z$ at each step. The value of $l_j.K$ at that step is added to $\mathcal{K}_j$ if it causes $l_j.Z$ decreasing compared to the previous step. This means that $\mathcal{K}_j$ comprises the smallest tile size for each possible number $l_j.Z$ of iteration ranges per thread group. Intuitively, such tile sizes are selected because they result in the most load-balanced schedules, where each thread group is assigned (roughly) the same number of tiles and all iteration ranges have (roughly) the same size.
Example: consider a loop level l_j with $l_j.N = 24$ and $l_j.R = 4$. Then $\mathcal{K}_j = \{1, 2, 3, 6\}$. Note that for example, with $l_j.K = 3$ we are assigning $24/3/4 = 2$ iteration ranges per thread group, and each iteration range comprises exactly 3 iterations; for $l_j.K = 6$ we are assigning $24/6/4 = 1$ iteration range per thread group, and each iteration range comprises exactly 6 iterations. Tile size 4 is not selected because it would result in 6 total iteration ranges, which cannot be load-balanced on 4 thread groups. Tile size 7 would result in 4 iteration ranges just like size 6, but the ranges would comprise $7, 7, 7, 3$ iterations

respectively instead of the more balanced 6 each.

On lines 8-11, the algorithm performs an iterative gradient-descent search, starting from a random solution in $\mathcal{K}_1, ..., \mathcal{K}_L$ and iterating over the loop levels. At loop level l_j, we consider the makespan of the schedule as a function of the tile size $l_j.K$ in $\mathcal{K}_j$, assuming that tile sizes for all other loop levels are fixed. Based on our evaluation in Chapter 6, we experimentally verified that such makespan function is always convex. This is because for decreasing tile sizes, there are two opposite effects at play: memory parallelism improves but the tiling overhead increases. Therefore, the function has a unique minimum, which we can find efficiently; the tile size corresponding to such minimum makespan is assigned to $l_j.K$ on line 11. The overall procedure is repeated max_iter times; based on our evaluation, we experimentally set max_iter $= 3$, as we found that increasing the number of iterations does not further improve the makespan. Finally, on lines 12-13, the algorithm updates the res variable to the smallest makespan value of any solution found so far, which is then returned on line 14. Note that if for a given optimization solution, we cannot fit the data in the available SPM space, the makespan function on line 12 simply returns $+\infty$ to denote that the solution is not feasible. Further note that the result of our heuristic is not guaranteed to be optimal due to the discrete nature of the possible tile sizes in $\mathcal{K}_j$ and thread group assignments in $\mathcal{R}$.

4.4 Application Optimization

Finally, Algorithm 2 shows how to compute the makespan for the whole kernel. The algorithm extracts maximum-size tilable components by performing a depth-first search over the tree recursively calling function extract_component($l, \mathcal{L}$), where l is the next loop level to inspect and $\mathcal{L}$ is either a tilable component or the empty set. The function is first invoked on each loop in $root(\mathcal{T})$ to return the makespan for the subtree rooted at that loop (lines 3-6). The function first adds l to the component $\mathcal{L}$ (line 9). If l is a leaf (lines 10-11), it then uses the function optimize($\mathcal{L}, P$) to determine an optimization solution for $\mathcal{L}$ (see Section 4.3) on P cores/threads; optimize returns the makespan for one execution of $\mathcal{L}$, which is then multiplied by the number of times the first loop in $\mathcal{L}$ (and thus $\mathcal{L}$ itself) is executed. If instead l has a single child (lines 12-13), extract_component is recursively called to extend the component. Finally, if l has multiple children (lines 14-18), we consider the best of two solutions: the one where $\mathcal{L}$ is tiled, and the one where the subtrees rooted at each of the children of l are tiled instead.

Example: when invoked on the tree in Figure 3.2, Algorithm 2 first invokes optimize(l_t). It then invokes optimize on components $(l_{s1\_0}, l_p), (l_{s1\_1}, l_{s2}), (l_{b\_0})$ and (l_{b_1}), returning the

Algorithm 1: Optimize Tilable Component Schedule

```
Input: tilable component $\mathcal{L} = (l_1, ..., l_L)$, number of cores $P$
Output: optimized makespan of $\mathcal{L}$
1  $\mathcal{R}$ = generate_nondominated_thread_groups($P, \mathcal{L}$)
2  res = $+\infty$
3  for $(l_1.R, ..., l_L.R) \in \mathcal{R}$ do
4      for $j \in 1...L$ do
5          $\mathcal{K}_j$ = select_tile_sizes($l_j$)
6      end
7      pick random solution $(l_1.K, ..., l_L.K)$ in $\mathcal{K}_1, ..., \mathcal{K}_L$
8      for $i \in 1...$max_iter do
9          for $j \in 1...L$ do
10             $l_j.K$ = find_minimum($\mathcal{K}_\mathrm{j}$)
11         end
12     end
13     if makespan$\big((l_1.R, ..., l_L.R), (l_1.K, ..., l_L.K)\big) <$ res then
14         res = makespan$\big((l_1.R, ..., l_L.R), (l_1.K, ..., l_L.K)\big)$
15     end
16 end
17 return result
18
19 Function select_tile_sizes($l_j$) is
20     $\mathcal{K}_j = \emptyset$, prev_$Z = +\infty$
21     for $l_j.K \in 1...l_j.N$ do
22         $l_j.M = \lceil l_j.N/l_j.K \rceil, l_j.Z = \lceil l_j.M/l_j.R \rceil$
23         if $l_j.Z <$ prev_$Z$ then
24             $\mathcal{K}_j$ = append($\mathcal{K}_j, l_j.K$)
25             prev_$Z = l_j.Z$
26         end
27     end
28     return $\mathcal{K}_j$
```

following makespan: $\min\big(\text{optimize}(l_a), NT \cdot (\text{optimize}(l_{s1\_0}, l_p) + \text{optimize}(l_{b\_0})) + (NT-1) \cdot (\text{optimize}(l_{s1\_1}, l_{s2}) + \text{optimize}(l_{b\_1}))\big)$. Note that, since $l_a.parallel = \mathit{false}$, the first solution cannot use more than one core. The second solution has better scalability but incurs more overhead since the schedule for each component must be repeated multiple times.

Algorithm 2: Compute makespan

```
   Input: loop tree $\mathcal{T}$, number of cores $P$
   Output: makespan
 1 res = 0
 2 for $l \in root(\mathcal{T})$ do
 3     res += extract_component($l, \emptyset$)
 4 end
 5 return res
 6
 7 Function extract_component($l, \mathcal{L}$) is
 8     $\mathcal{L}$ = append($\mathcal{L}, l$)
 9     if $l.\mathcal{C} == \emptyset$ then
10         return optimize($\mathcal{L}, P$) ·first($\mathcal{L}$).$I$
11     end
12     if $l.\mathcal{C} == \{l'\}$ then
13         return extract_component($l', \mathcal{L}$)
14     end
15     parent_makespan = optimize($\mathcal{L}, P$) · first($\mathcal{L}$).$I$
16     children_makespan = 0;
17     for $l' \in l.\mathcal{C}$ do
18         children_makespan += extract_component($l', \emptyset$)
19     end
20     return min(parent_makespan, children_makespan)
```

Chapter 5

PREM Compiler Implementation

In this chapter, we discuss the program transformation methods applied during our compilation flow to enable the parallel PREM schedule. We begin by introducing the design of our toolchain in Section 5.1. Then we introduce the important procedures inside the compiler. Section 5.2 explains the procedure of performing program transformation, including check of transformation legality in Section 5.2.1 and transformation in polyhedral model in Section 5.2.2. We next discuss the details in data transfer statements generation in Section 5.3, including data access information in Section 5.3.1 and API call parameters in Section 5.3.2.

5.1 Compiler Design

Figure 5.1 shows the high-level block diagram of our implemented toolchain, which can be generally divided in two parts: the compiler proper and the component optimizer. The *PREM compiler* is built on top of a series of polyhedral compilation tools. The polyhedral model [13] provides a representation of loop iterations of a program as a polytope over the space of iteration variables, and is commonly used in optimizing compilers like Pluto [8].

First, *pet* [43], the polyhedral extraction tool, is adopted to extract the polyhedral schedule tree from source code. The schedule tree [44] is a data structure containing the polyhedral representation along with data access information of analyzed loop iterations. Then, reusing the dependence analysis [48] implemented by PPCG [47], we retrieve information about data dependencies. After the loop tree $\mathcal{T}$ is generated from the schedule tree, we determine whether each loop level is tilable and parallelizable by checking the

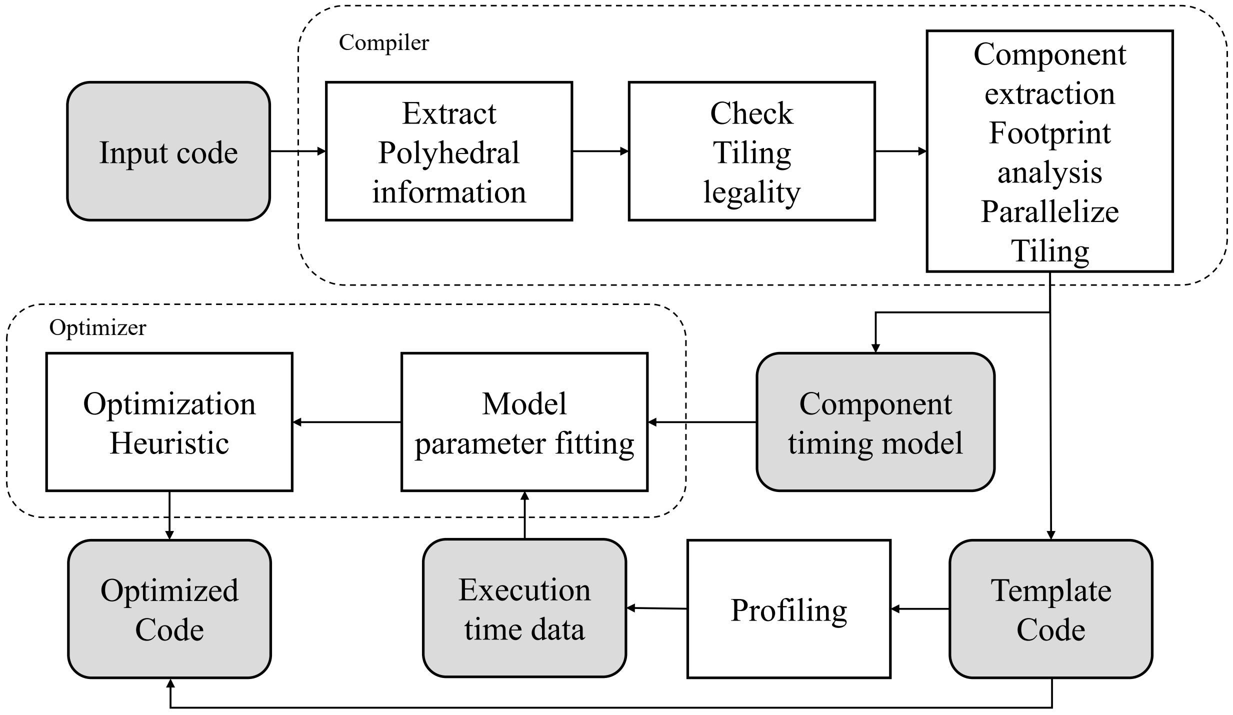


Figure 5.1: Toolchain block diagram. Gray rounded boxes represent data or intermediate representation produced by the previous step, white rectangles are procedures.

legality of transformation with the information of data dependencies. This legality check is performed so that we can generate the loop tree and parallelization information in the application model of Section 3.3. This process is discussed in detail in Section 5.2.1. In this legality check, we also check the canonical data element ranges of segments. The canonical data element ranges of different segments cannot overlap when written access dependencies exist. The details are discussed in Section 5.3.1. After that, we perform tiling transformation on the extracted tilable component, this part is discussed in Section 5.2.2.

Finally, we perform code generation for each tilable component over the loop tree (see Algorithm 1), inserting corresponding memory transfer API calls for each data structure used by the component, and output template code over the tile sizes to be searched.

5.2 Program transformations and Validity

In this section, we first explain how to verify the legality of the transformations. Then we discuss how we perform loop transformations of tiling and parallelization.

5.2.1 Legality of Transformation

One major challenge we encounter during segmenting the loop nest is ensuring the legality of our transformation. Even for a loop nest with fixed-stride and fixed iteration range, it is possible that after tiling and parallelization, the program execution produces wrong results because the program schedule has been changed. For this reason, we have to check the legality of both tiling and parallelization transformations for every loop. As mentioned in Section 3.3, if the legality check fails for tiling a loop level, that loop level is removed from the loop tree by folding it into its parent. If the legality check fails for parallelizing a loop level but succeeds for tiling it, we set the parallelization attribute for this loop to be false.

Once we pick an optimization solution using Algorithm 1 in Section 4.3, we transform the program by tiling or parallelizing it on some of its loop levels. Since the data dependency constraints are extracted from the original program, the original program must have satisfied all the data dependency constraints. There is a possibility that some of the data dependency constraints no longer hold true for the transformed program. In Section 2.2.2, we introduced the data dependence we get from dependence analysis: it is a set that contains all the dependent pairs of instances of statements. We determine if the program still

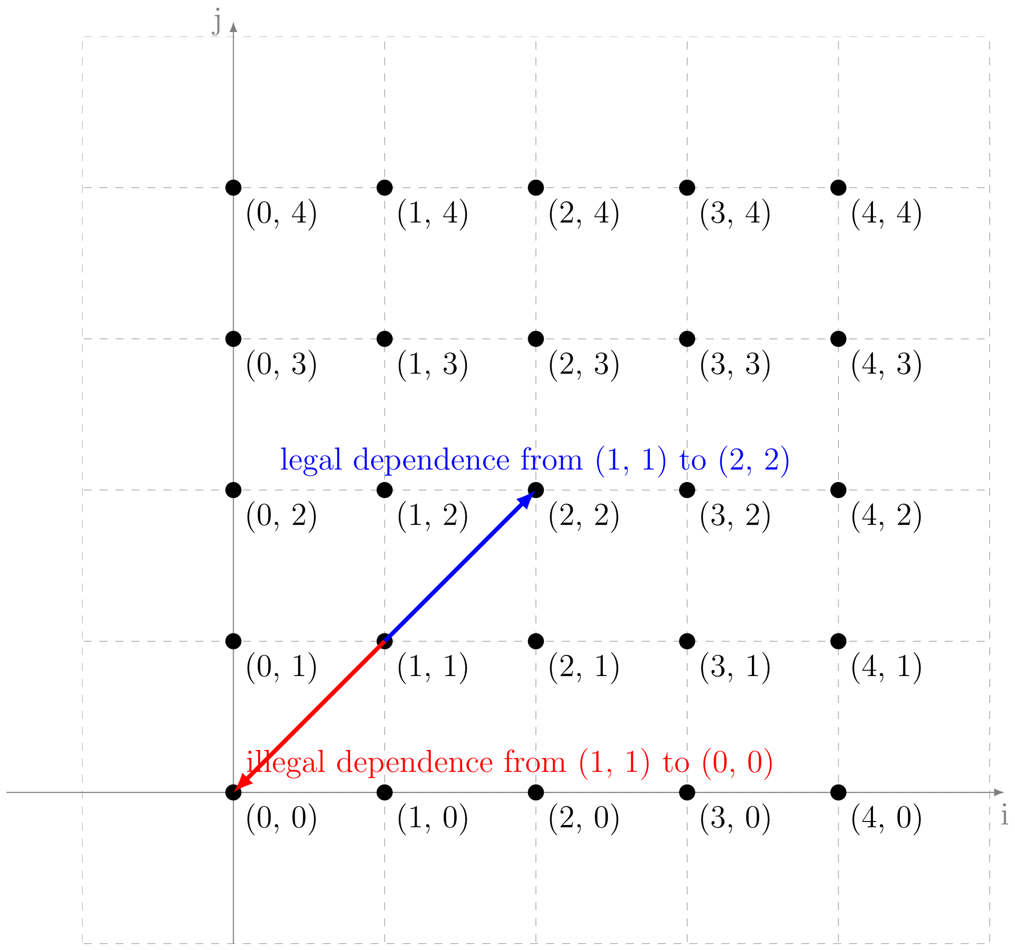


Figure 5.2: dependent pairs of statement instances that is legal or illegal

satisfies these constraints by iterating over each dependent pair in this set and check if the following equation still holds:

$$\Phi_{Stmt_j}(\vec{t}) - \Phi_{Stmt_i}(\vec{s}) \geq \vec{0} \tag{5.1}$$

In Equation 5.1, $\vec{t}$ is the iteration vector of sink instance of $Stmt_j$, $\vec{s}$ is the iteration vector of source instance of $Stmt_i$. This equation assures that the sink instance always executes after the source instance in the given execution schedule. We call $\Phi_{Stmt_j}(\vec{t}) - \Phi_{Stmt_i}(\vec{s})$ the dependency distance between $Stmt_i$ and $Stmt_j$. Note that the $\geq$ operator in Equation 5.1 compares schedules based on lexicographic order.

In Figure 5.2, the lattice points represent the instances of a statement $Stmt_x[i,j]$ in a nested loop where i is the outside loop level and j is the innermost loop level. The schedule of this statement is $\Phi_{Stmt_x}([i,j]) = (i,j)$. Then, if we have a data dependency $Dep_1 = \{Stmt_x[1,1] \to Stmt_x[2,2]\}$, this data dependency is legal because the schedule of source and sink instances satisfies Equation 5.1. On the other hand, if we have another data dependency $Dep_2 = \{Stmt_x[1,1] \to Stmt_x[0,0]\}$, this data dependency is illegal because the distance vector is $(-1,-1)$ which is less than $\vec{0}$ and does not satisfy Equation 5.1. Notice if we have the data dependency $Dep_3 = \{Stmt_x[1,2] \to Stmt_x[2,1]\}$, it is still legal because the distance vector is $(1,-1)$ is larger than $\vec{0}$ since the dependency distance vector is in lexicographic order.

```
for (int t = 0; t < NT; t++)
{
    /* Component 1 start */
    for (int s1 = 0; s1 < NS; s1++)
    {
      for (int p = 0; p < NP; p++){
        if(p==0){
          i[s1] = 0.0; //Stmt1
        }
        i[s1] += U_i[s1][p] * inp_F[t][p]; //Stmt2
      }
    }
    /* Component 1 end */
    ...  ...
    do_something(i); //Stmt3
}
```

Listing 5.1: code example of component

Example: In Listing 5.1, the schedule of Stmt1 is $\{Stmt1[t,s_1,p] \to (t,s_1,p,0)\}$, the schedule of Stmt2 is $\{Stmt2[t,s_1,p] \to (t,s_1,p,1)\}$. The domain of Stmt1 is $\mathcal{D}_{Stmt_1} = \{(t,s_1,p \mid 0 \le t < NT \wedge 0 \le s_1 < NS \wedge p = 0\}$, the domain of Stmt2 is $\mathcal{D}_{Stmt_2} = \{(t,s_1,p \mid 0 \le t < NT \wedge 0 \le s_1 < NS \wedge 0 \le p < NP\}$. Because of this difference of $\mathcal{D}_{Stmt_1}$ and $\mathcal{D}_{Stmt_2}$, Stmt1 is placed inside the `if` statement. The dependency analysis result of this program in Listing 5.1 is a union of map relations between statement instances. The first part of data dependency is between Stmt1 and Stmt2, we can see that in first iteration of loop p, Stmt1 must be executed before Stmt2. Thus, we have the dependency $Dep_1 = \{Stmt1[t,s_1,p] \to Stmt2[t' = t, s_1' = s_1, p' = p] \mid 0 \le t < NT \wedge 0 \le s_1 < NS \wedge p = 0\}$. This means Stmt2 must execute after Stmt1 when p is 0.

The second data dependency is between Stmt2's instances because each of Stmt2's instance must read array i, which is the output from its last iteration. The dependency is $Dep_2 = \{Stmt2[t, s_1, p] \rightarrow Stmt2[t' = t, s_1' = s_1, p' = p + 1] \mid 0 \leq t < NT \wedge 0 \leq s_1 < NS \wedge 0 \leq p < NP - 1\}$ The complete dependency is the union of these two map relations.

The schedule of Stmt1 is $\{Stmt1[t, s_1, p] \rightarrow (t, s_1, p, 0)\}$, the schedule of Stmt2 is $\{Stmt2[t, s_1, p] \rightarrow (t, s_1, p, 1)\}$. We apply the schedule to the dependency relation to get the dependence distance $\{(0, 0, 0, 1)\}$ for Dep_1 and $\{(0, 0, 1, 0)\}$ for Dep_2. For each loop level in component, we check its corresponding index in related dependence distances, if all of them are 0, then this loop level can be parallelized. We can see clearly that *s1* is parallelizable, but p is not because it does not meet the requirement in Dep_2.

To check if it is legal to tile both loop levels *s1* and p, we tile *s1* with $l_{s1}.K = 3$ and p with $l_p.K = 4$. The process will be discussed in details in next Section 5.2.2. The resulting schedule of Stmt1 would be $(t, s1, p, 0) \rightarrow (t, floor(s1/3), floor(p/4), s1 \bmod 3, p \bmod 4, 0)$ and the resulting schedule of Stmt2 would be $(t, s1, p, 0) \rightarrow (t, floor(s1/3), floor(p/4), s1 \bmod 3, p \bmod 4, 1)$. If we again apply the two schedules to the dependency relation Dep_1, for which it must hold $p = 0$, we obtain $\Phi_{Stmt_1} = (t, floor(s1/3), 0, s1 \bmod 3, 0, 0)$ and $\Phi_{Stmt_2} = (t, floor(s1/3), 0, s1 \bmod 3, 0, 1)$, their dependency distance is $\{0, 0, 0, 0, 0, 1\}$ which is larger than $\vec{0}$. Hence, Dep_1 is still satisfied. Similarly, if we apply the schedules to Dep_2, for which $p \geq 0$, we have $\Phi_{Stmt_1} = (t, floor(s1/3), floor(p/4), s1 \bmod 3, p \bmod 4, 0)$ and $\Phi_{Stmt_2} = (t, floor(s1/3), floor((p{+}1)/4), s1 \bmod 3, (p{+}1) \bmod 4, 1)$, their dependency distance is $\{0, 0, floor((p + 1)/4) - floor(p/4), 0, ((p + 1) \bmod 4) - (p \bmod 4), 1\}$. Though $((p + 1) \bmod 4) - (p \bmod 4)$ could be less than 0, $floor((p + 1)/4) - floor(p/4)$ is always larger than 0 for $p \geq 0$. Hence, their dependency distance is larger than $\vec{0}$ and Dep_2 is still satisfied. Since all the dependencies are satisfied, it is legal to tile loop levels *s1* and p.

5.2.2 Tiling Transformation

As discussed in Section 4.3, we search the loop tree of tilable components to find a set of $l.R$ and $l.K$ parameters, which we call optimization solution. It is used to transform the original loop program into a tiled loop program. Next, we discuss how we perform the transformation given the optimization solution.

In Section 2.3, we decided to use a loop transformation template modeled by the polyhedral model to tile the nested loop program. We model the SCoP of a tilable component as a set of statements. From the definition of tilable component in Section 3.4, we know that the number of the loop indexes that is shared by all the statements are the loop number of the tilable component. Since the loops are perfectly nested, they must be shared by all the statements inside them. Thus, for a tilable component $\mathcal{L} = (l_1, ..., l_j, ..., l_L)$ which has L perfectly nested loops, the statements inside $\mathcal{L}$ share the first L indexes as common indexes.

In the SCoP of the tilable component $\mathcal{L}.SCoP = \{Stmt_1, Stmt_2, ..., Stmt_n\}$, for each statement, the innermost leaf loop level could be different sub-loops inside tilable component $\mathcal{L}$, we denote the sub-loop levels below shared loop levels until leaf loop level of statement $Stmt_i$ to be $l_{L+1}^{Stmt_i}.idx, ..., l_{leaf}^{Stmt_i}.idx$ if the leaf loop level is not l_L. Also, we denote all the loop levels from root loop level to the starting loop level of tilable component $\mathcal{L}$ to be $l_{root}, ..., l_1$. Thus, any instance of statement $Stmt_i$ is:

$$Stmt_i[l_{root}.idx, ..., l_1.idx, ..., l_L.idx, l_{L+1}^{Stmt_i}.idx, ..., l_{leaf}^{Stmt_i}.idx].$$

To apply the tiling transformation to each statement, we construct a transformation function that maps the original schedule of each statement to the tiled schedule. If multiple statements are contained in the same innermost leaf loop, we use $Stmt_i.order$ to denote the relative execution order of $Stmt_i$, $Stmt_i.order = 0$ if $Stmt_i$ is the first statement to be executed in this innermost leaf loop. The original schedule of any statement is denoted to be:

$$\Phi(Stmt_i[l_{root}.idx, ..., l_1.idx, ..., l_L.idx, ...l_{leaf}^{Stmt_i}.idx]) = \\ (l_{root}.idx, ..., l_1.idx, ..., l_L.idx, ...l_{leaf}^{Stmt_i}.idx, Stmt_i.order),$$

where $l.begin \leq l.idx < l.begin + l.N \cdot l.S$ holds true for any loop l that contains this statement. The transformed schedule for this statement is:

$$\Phi(Stmt_i[l_{root}.idx, ..., l_1.idx, ..., l_L.idx]) = (l_{root}.idx, ..., \lfloor l_1.idx/l_1.K \rfloor, ... \lfloor l_L.idx/l_L.K \rfloor, \\ l_1.idx \bmod l_1.K, ..., l_L.idx \bmod l_L.K, l_{L+1}^{Stmt_i}.idx, ..., l_{leaf}^{Stmt_i}.idx, Stmt_i.order).$$

This transformation is for the condition of single core execution. For multicore execution, we simply replace the index of tiled loops as discussed in Section 3.4.

Example: as an example, we apply the transformation to component $\mathcal{L}_1 = (l_{s1}, l_p)$ in Listing 5.1. In this component, there are two statements. The schedule of $Stmt_1$ is

$$\Phi(Stmt_1[t, s1, p]) = (t, s1, p, 0)$$

and schedule of $Stmt_2$ is

$$\Phi(Stmt_2[t, s1, p]) = (t, s1, p, 1)$$

When $l_{s1}.R = 1$, $l_p.R = 1$, meaning single core execution and $l_{s1}.K = 4$, $l_p.K = 3$, we construct the transformation for $Stmt_1$ which is

$$\mathcal{T}_1(\Phi(Stmt_1)) = (t, s1, p, 0) \rightarrow (t, floor(s1/3), floor(p/4), s1 \bmod 3, p \bmod 4, 0)$$

and transformation for $Stmt_2$ which is

$$\mathcal{T}_2(\Phi(Stmt_2)) = (t, s1, p, 1) \rightarrow (t, floor(s1/3), floor(p/4), s1 \bmod 3, p \bmod 4, 1)$$

Given the transformation, *isl* library would transform the corresponding schedule of each instance of the statements and perform code generation. The resulting code is shown in Listing 5.2.

```
for (int t = 0; t < NT; t++)
{
    /* Tiled component 1 start */
    for (int s1_0_t = 0; s1_0_t < floor(NS/3); s1_0_t++)
    {
      for (int p_t = 0; p_t < floor(NP/4); p_t++)
      {
        for (int s1_0 = s1_0_t * 3; s1_0 < MIN(NS, s1_0_t*3+3); s1_0++)
        {
          for (int p = p_t * 4; p < MIN(NP, p_t*4+4); p++)
          {
            if(p==0){
              i[s1] = 0.0; //Stmt1
            }
            i[s1] += U_i[s1][p] * inp_F[t][p]; //Stmt2
          }
        }
      }
    }
    /* Tiled component 1 end */
    ... ...
    do_something(i); //Stmt3
}
```

Listing 5.2: code example of tiled component

We then add extra nodes in the schedule for the three macros for data transfer statements: `BUFFER_ALLOC_APIS`, `DATA_SWAP_APIS` and `BUFFER_DEALLOC_APIS`, as detailed in Section 3.5. They are inserted before the tilable component, before execution of each tile and right after the tilable component. To add extra schedule dimensions for these data transfer statements, we perform another transformation on the statements, which is

$$\mathcal{T}_{macro}(\Phi(Stmt)) = (t, floor(s1/3), floor(p/4), s1 \bmod 3, p \bmod 4, ...) \rightarrow \\ (t, 1, floor(s1/3), floor(p/4), 1, s1 \bmod 3, p \bmod 4, ...)$$

This transformation adds a 1 before the schedule of element loop part and another 1 before the schedule of tiled loop part, then we insert the three macro statements, whose schedules are $\Phi(Stmt_{Alloc}) = ((t, 0))$, $\Phi(Stmt_{Swap}) = ((t, 1, floor(s1/3), floor(p/4), 0, 0, 0, 0))$ and $\Phi(Stmt_{Dealloc}) = ((t, 2))$. The index of schedule for `BUFFER_ALLOC_APIS` and `BUFFER_DEALLOC_APIS` is 0 and 2, respectively, this makes them execute before/after the tilable component. The index of schedule for `DATA_SWAP_APIS` is 0, this makes it execute before the other statements in each tile. The result code is shown in Listing 5.3.

```
for (int t = 0; t < NT; t++)
{
    /* Tiled component 1 start */
    BUFFER_ALLOC_APIS //Macro Stmt
    for (int s1_0_t = 0; s1_0_t < floor(NS/3); s1_0_t++)
    {
      for (int p_t = 0; p_t < floor(NP/4); p_t++)
      {
        DATA_SWAP_APIS //Macro Stmt
        for (int s1_0 = s1_0_t * 3; s1_0 < MIN(NS, s1_0_t*3+3); s1++)
        {
          for (int p = p_t * 4; p < MIN(NP, p_t*4+4); p++)
          {
            if(p==0){
              i[s1_0] = 0.0; //Stmt1
            }
            i[s1_0] += U_i[s1_0][p] * inp_F[t][p]; //Stmt2
          }
        }
      }
    }
    BUFFER_DEALLOC_APIS //Macro Stmt
    /* Tiled component 1 end */
    ... ...
    do_something(i); //Stmt3
}
```

Listing 5.3: code example of tiled component with macro statement inserted

5.3 Data Transfer Statements Generation

To explicitly generate the data transfer statements, we need memory access analysis of the accessed arrays. In Section 2.2.1, we discussed how data access is modeled in polyhedral compilation model and how to calculate the elements visited by a certain data access under a specified domain. The data access information of each statement is extracted by *pet*, including if this data access is read or write. In Section 3.5, we discussed the generation of the API calls, yet we have not talked about the parameters required by such calls. In this section, we further explain how to compute them.

5.3.1 Memory Access Analysis

Based on Section 3.5 and Section 4.2, to generate the data transfer statements for each tilable component and to compute the length of memory phases, for each array a we need to compute: (1) its canonical data element range $\hat{\mathcal{R}}_a$ for each tile/segment in the component; (2) its bounding box, which is the maximum size of the canonical range over all tiles/segments. In details:

1. The canonical data element range determines the size parameter for all `swap` calls.
2. The canonical data element range is also used to determine the number of data lines and the size of each line, which is needed to compute the memory phase length.
3. The bounding box determines how much memory area to be allocated on SPM for each array buffer allocated in the `BUFFER_ALLOC_APIS` macro.
4. The bounding box provides the `dpitch` parameter for `swap2d_buffer` and `swapnd_buffer` calls.

In Section 5.2.2, we applied tiling transformation to divide the instances of a statement into tiles. For convenience, we denote the domain of a statement tile as $\mathcal{D}^{tile}_{Stmt}[P_1, ..., P_x]$. $P_1, ..., P_x$ are the values of outer loop indexes and tiled loop indexes of a particular tile of *Stmt*. $\mathbf{U}_{\mathcal{D}^{tile}_{Stmt}}$ is defined as the set that contains all the tile domain set of *Stmt*.

Example: In Listing 5.1, in tilable component $\mathcal{L}_1 = (l_{s1}, l_p)$, the accessed arrays are `i`, `U_i` and `inp_F`. Among them, array `i` is both read and written to by $Stmt_1$ and $Stmt_2$. For simplicity, we show the access relations of the original program because tiling would not change the access relations of statement instances. Its access relations are:

$$\mathcal{A}_i^{Stmt_1} = \{Stmt_1(t, s1, p) \rightarrow i(s1)\}$$

$$\mathcal{A}_i^{Stmt_2} = \{Stmt_2(t, s1, p) \rightarrow i(s1)\}$$

Access relation of array `U_i` is:

$$\mathcal{A}_{U\_i}^{Stmt_2} = \{Stmt_2(t, s1, p) \rightarrow U\_i(s1, p)\}$$

And access relation of array `inp_F` is:

$$\mathcal{A}_{inp\_F}^{Stmt_2} = \{Stmt_2(t, s1, p) \to inp\_F(t, p)\}$$

If we put all the instances of $Stmt_1$ which are executed in a tile in a domain set, it is denoted as:

$$\begin{aligned}&\mathcal{D}_{Stmt_1}^{tile}[T, S1, P] = \{(t, s1, p) \mid t = T \wedge\\ &l_{s1}.begin + S1 \cdot l_{s1}.K \cdot l_{s1}.S \leq s1 < min(l_{s1}.begin + (S1+1) \cdot l_{s1}.K \cdot l_{s1}.S, l_{s1}.begin + l_{s1}.N \cdot l_{s1}.S) \wedge\\ &l_p.begin + P \cdot l_p.K \cdot l_p.S \leq p < min(l_p.begin + (P+1) \cdot l_p.K \cdot l_p.S, l_p.begin + l_p.N \cdot l_p.S)\}\end{aligned}$$

In this domain set of a tile $\mathcal{D}_{Stmt_1}^{tile}[T, S1, P]$, T is the value of loop level t, $S1$ is the index of iteration ranges of loop level s1, ranging from 0 to $l_{s1}.M - 1$, P is the tile index of iteration ranges of loop level p, ranging from 0 to $l_p.M - 1$. Given $\mathcal{D}_{Stmt_1}^{tiled}$ and the value of $T, S1, P$, we can calculate the set of data elements that are read/write in the given tile. Here we have $l_{s1}.begin = 0, l_p.begin = 0, l_{s1}.S = 1, l_p = 1$, when $T = 0, S1 = 0, P = 0$ and $l_{s1}.R = 1, l_{s1}.K = 3, l_p.R = 1, l_p.K = 4$, the domain set of this tile is $\mathcal{D}_{Stmt_1}^{tiled} = \{(t, s1, p) \mid t = 0 \wedge 0 \leq s1 < 3 \wedge 0 \leq p < 4\}$. After we apply $\mathcal{D}_{Stmt_1}^{tiled}$ to access relation map $\mathcal{A}_i^{Stmt_1}$, the resulting data element range is:

$$\mathcal{R}_i^{Stmt_1} = \mathcal{D}_{Stmt_1}^{tiled} \times \mathcal{A}_i^{Stmt_1} = \{i(s1) \mid 0 \leq s1 < 3\}$$

This means in the tile where $T = 0, S1 = 0, P = 0$, the elements i[0], i[1], i[2] of array i are written to since $\mathcal{A}_i^{Stmt_1}$ is a write access. In SPM, what we allocate is an area that has fixed width and height. However, the data element range of an array is a set of accessed elements, which implies the data elements could be sparse and not in fixed range. However, in execution, we load a range of data. To ensure the range of data elements can be fit into the allocated area on SPM, given data element range $\mathcal{R}_a^{Stmt}$, we calculate the canonical data element range and use it for data transfer. The canonical data element range is calculated by finding the minimum and maximum index among all the data element accesses. It uses a rectangular shaped range instead of sparse data elements for the accessed data element range.

Assume the length of dimensions of the array a is $n = length(Shape(a))$, then an element in the array a is denoted as $a(idx_1, ..., idx_n)$. We define the minimum index on loop level l of the data element accesses in $\mathcal{R}_a^{Stmt}$ to be:

$$\begin{aligned}idx_l^{\min}(\mathcal{R}_a^{Stmt}) = \min\{idx_l \mid \exists idx_1, ..., idx_{l-1}, idx_l, idx_{l+1}, ..., idx_n :\\ a(..., idx_{l-1}, idx_l, idx_{l+1}, ...) \in \mathcal{R}_a^{Stmt}\}\end{aligned}$$

and maximum index to be

$$idx_l^{\max}(\mathcal{R}_a^{Stmt}) = \max\{idx_l \mid \exists idx_1, ..., idx_{l-1}, idx_l, idx_{l+1}, ..., idx_n : \\ a(..., idx_{l-1}, idx_l, idx_{l+1}, ...) \in \mathcal{R}_a^{Stmt}\}$$

Then the canonical data element range is:

$$\hat{\mathcal{R}}_a^{Stmt} = \{a(idx_1, ..., idx_n) \mid \forall idx_l \in \{idx_1, ..., idx_n\} : idx_l^{\min}(\mathcal{R}_a^{Stmt}) \leq idx_l \leq idx_l^{\max}(\mathcal{R}_a^{Stmt})\}$$

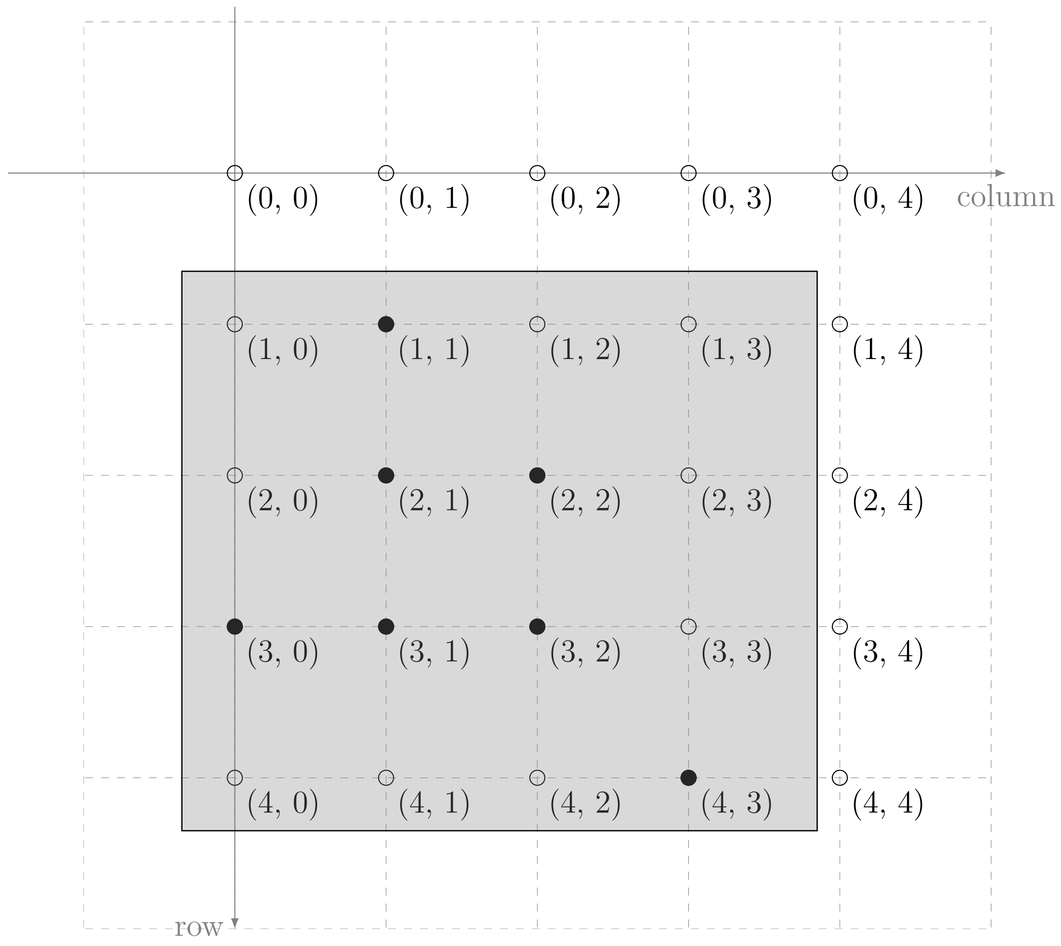


Figure 5.3: canonical data element range of accessed data elements in arr[5][5]

Example: In Figure 5.3, we show the dots as data elements in an array arr[5][5]. We assume that during the execution of a tile, arr[1][1], arr[2][1], arr[2][2], arr[3][0], arr[3][1], arr[3][2] and arr[4][3] are accessed. Then, the canonical data element range of them is $\hat{\mathcal{R}}_{arr} = \{arr[i][j] \mid 1 \leq i \leq 4 \wedge 0 \leq j \leq 3\}$, which is marked in gray box.

Once we have the canonical data element range $\hat{\mathcal{R}}_a^{Stmt}$ of statement $Stmt$ accessing array a, we can compute the canonical data element range of all statements in tilable component $\hat{\mathcal{R}}_a$. We define the minimum index of loop level l in $\hat{\mathcal{R}}_a$ to be

$$idx_l^{\min}(\hat{\mathcal{R}}_a) = \min\{idx_l^{\min}(\hat{\mathcal{R}}_a^{Stmt}) \mid \forall Stmt \in \mathcal{L}.SCoP\}$$

and the maximum index to be

$$idx_l^{\max}(\hat{\mathcal{R}}_a) = \max\{idx_l^{\max}(\hat{\mathcal{R}}_a^{Stmt}) \mid \forall Stmt \in \mathcal{L}.SCoP\}$$

Then $\hat{\mathcal{R}}_a$ is defined as

$$\hat{\mathcal{R}}_a = \{a(idx_1, ..., idx_n) \mid \forall idx_l \in \{idx_1, ..., idx_n\} : idx_l^{\min}(\hat{\mathcal{R}}_a) \leq idx_l \leq idx_l^{\max}(\hat{\mathcal{R}}_a)\}$$

Given the canonical data element range $\hat{\mathcal{R}}_a$, we denote $Shape(\hat{\mathcal{R}}_a)$ as its shape. It contains dimension sizes that is computed as $|idx_l^{\max} - idx_l^{\min} + 1|$, which means the distance between maximum index and minimum index on loop level l. Hence, $Shape(\hat{\mathcal{R}}_a)$ contains all the distances between maximum index and minimum indexes on each loop level.

$$Shape(\hat{\mathcal{R}}_a) = \langle |idx_l^{\max}(\hat{\mathcal{R}}_a) - idx_l^{\min}(\hat{\mathcal{R}}_a) + 1| : l \in \{1, ..., n\}\rangle$$

$Shape(\hat{\mathcal{R}}_a)$ is a tuple, the lth element of $Shape(\hat{\mathcal{R}}_a)$ is $Shape(\hat{\mathcal{R}}_a)_l$, starting from 1. $\mathbf{U}_{\hat{\mathcal{R}}_a}$ is defined as the set that contains all the canonical data element ranges for each tile of the array a.

Example: In Figure 5.3, the shape of canonical data element range of arr is $Shape(\hat{\mathcal{R}}_{arr}) = \{4, 4\}$ and $Shape(\hat{\mathcal{R}}_{arr})_1 = 4$.

After we compute the shape of canonical data element range of every tile, we find the minimum area on SPM that each canonical data element range can be fitted in, which is named bounding box. We use $BoundingBox(a)$ to denote the bounding box of all the accesses of array a in the whole tile.

$$BoundingBox(a) = \langle \max\{Shape(\hat{\mathcal{R}}_a)_l \mid \forall \hat{\mathcal{R}}_a \in \mathbf{U}_{\hat{\mathcal{R}}_a}\} \mid l \in \{1, ..., length(Shape(\hat{\mathcal{R}}_a))\}\rangle$$

The bounding box is a sequence where the lth element is noted as $BoundingBox(a)_l$ where $l \in \{1, ..., length(BoundingBox(Stmt, a))\}$. $BoundingBox(Stmt, a)_1$ refers to the size of the outermost dimension that should be allocated for an array a on SPM.

Finally, after we compute the information of the canonical data element ranges of all the tiles/segments, we need to check these canonical data element ranges to verify if the transformation is legal. On the same core, if there exist a RAW or WAW dependency between two consecutive segments of the same array when their canonical data element ranges are overlapped to each other and are not the same, then this transformation is not legal. This is because we are using the double buffer streaming strategy, after the execution of the first segment, the written-out data of this segment is transferred from one buffer on SPM to main memory. At the same time, the second segment is reading/writing data on another buffer on SPM. If these two data areas are overlapped to each other, then the second segment would read the invalid input data or its written-out data could overwrite part of the written-out data of the last segment. Similarly, if there is a RAW or WAW dependency between two segments of the same array on different cores when their canonical data element ranges are overlapped to each other, we also mark this transformation illegal because each core has a separate SPM and this could also cause invalid data being read or written. In both cases, the loops involved in the transformation would be marked as non-tilable / non-parallelizable when constructing the loop tree.

5.3.2 API Call Parameters

Next, we discuss how API calls are generated. This includes the `allocate_buffer` call and the `swap` call.

The `allocate_buffer` call is used to allocate a buffer with `R/W` attributes using the `attr` parameter. For an array a, if it is read by any statement $Stmt$ in $\mathcal{L}.SCoP$ and never written to in the tilable component $\mathcal{L}$, it is allocated as `RO`. If the array a is only written in the tilable component $\mathcal{L}$, or all elements of the array a are read and written, and there is no read access to one element happen before the written access to the same element of the array a, the array a is allocated as `WO`. If the array a is both read and written in the tilable component $\mathcal{L}$ and there is any read access to one element happen before the written access to the same element of the array a, the array a is allocated as `RW`.

The `swap` call is used to transfer arrays with any number of dimensions. The algorithm of generating API calls for buffer swap is shown in Algorithm 3. Given the shape of the array a that is $Shape(a)$, the canonical data element range of array a for a given tile/segment $\hat{\mathcal{R}}_a$ and its bounding box $BoundingBox(a)$, this algorithm generates the `swap_buffer` / `swap2d_buffer` / `swapnd_buffer` calls required by the segment and array. In this algorithm, the function `generate` is used for generating a given API call in macro block. The `generate` function takes two inputs: first is the name of the function that would be generated, second is a list of values of parameters for the function to be generated. Note that for simplicity of exposition, in Algorithm 3 we explain how to generate the API calls required by each segment. Each such API call is inserted in an early segment, so that data can be loaded before the segment starts, as explained in Section 3.5. Furthermore, for arrays with constant change stride, as also explained in Section 3.5, in reality we only generate four `swap` calls: two for the first and second segment in $SegmentToSwap_a(i)$, and two more for the remaining segments; one for the 3rd, 5th... segments in $SegmentToSwap_a(i)$ and another one for the 4th, 6th... segments. In addition, the starting addresses and sizes computed in Algorithm 3 are put in an array `swap_params` that is then accessed by the `swap` calls.

In each `swap_buffer` / `swap2d_buffer`, a `id` parameter is used to specify the information of buffer on SPM, this `id` is one of the two buffer pointers `buf1` / `buf2` on SPM. One `id` is bonded to one buffer on SPM with its shape. For an array a with one or two dimensions, we allocate buffer that binds to `id` named `a_id` on SPM. The process of selecting `buf1` or `buf2` in each segment is discussed in Section 3.5. For an array a of n dimensions, the buffer bonded to `a_id` is an N-dimensional array with size of $\prod_{i=1}^{n} BoundingBox(a)_i$.

`src` is the address of the data to be transferred in main memory, which is the address of $\hat{\mathcal{R}}_a$. Given $\hat{\mathcal{R}}_a$, we know the index of its first element is $a[idx_1^{\min}(\hat{\mathcal{R}}_a)][idx_2^{\min}(\hat{\mathcal{R}}_a)]...[idx_n^{\min}(\hat{\mathcal{R}}_a)]$. Assume the element type of the array a is e^{type}, then we have the starting address to be $(e^{type}*)a$ which is a pointer that points to an element of type e^{type}. The relative offset *AddressOffset* between the starting address of the array a and the starting address of $\hat{\mathcal{R}}_a$ is the sum of starting index of each dimension multiplied the production of length of the following dimensions. The starting index of dimension i is $idx_i^{\min}(\hat{\mathcal{R}}_a)$, the rest of the dimensions are from $i+1$ to n , thus the length of production of rest dimensions is $\prod_{j=i+1}^{n} Shape(a)_j$. Then the relative offset can be computed as the sum of this term in each dimension, which is $\sum_{i=1}^{n-1}(idx_i^{\min}(\hat{\mathcal{R}}_a) \cdot \prod_{j=i+1}^{n} Shape(a)_j) + idx_n^{\min}(\hat{\mathcal{R}}_a)$. And `src` is $(e^{type}*)a$ + AddressOffset cast to type `uint64_t*`.

The rest parameters are data sizes `size`, array sizes in main memory `spitch` and buffer sizes on SPM `dpitch`. They are computed as $Shape(\hat{\mathcal{R}}_a)$, $Shape(a)$ and $BoundingBox(a)$ multiply the size of element type sizeof(e^{type}). Notice $Shape(a)$ and $BoundingBox(a)$ are only used from dimension 2 to n since `spitch` and `dpitch` do not require the outermost dimension.

Example: In Figure 5.4, we have an array d in main memory, its data type is double meaning e^{type} is double and sizeof$(e^{type}) = 8$. We call its innermost dimension dim3, outermost dimension dim1. It has shape of $Shape(d) = \langle 6, 5, 4\rangle$ which means dim1=6, dim2=5, dim3=4. Its canonical data element range is a cube with shape of $Shape(\hat{\mathcal{R}}_d) = \langle 4, 3, 2\rangle$ starting at index of $\langle 2, 0, 2\rangle$. The elements in this canonical data element range are:

$$\begin{aligned}\hat{\mathcal{R}}_d = \{&d(2,0,2), d(2,0,3), d(2,1,2), d(2,1,3), d(2,2,2), d(2,2,3),\\ &d(3,0,2), d(3,0,3), d(3,1,2), d(3,1,3), d(3,2,2), d(3,2,3),\\ &d(4,0,2), d(4,0,3), d(4,1,2), d(4,1,3), d(4,2,2), d(4,2,3),\\ &d(5,0,2), d(5,0,3), d(5,1,2), d(5,1,3), d(5,2,2), d(5,2,3)\}\end{aligned}$$

Its bounding box on SPM is $BoundingBox(d) = \langle 5, 4, 3\rangle$. In this case, the dimension of array d is 3 which is more than two, so we would use `swapnd_buffer` for data swap. The SPM buffer has size $5 \times 4 \times 3$ as the bounding box of the array d is $\langle 5, 4, 3\rangle$.

The relative offset in main memory is computed as $idx_1^{\min}(\hat{\mathcal{R}}_d) \cdot (Shape(d)_2 \cdot Shape(d)_3) + idx_2^{\min}(\hat{\mathcal{R}}_d) \cdot Shape(d)_3 + idx_3^{\min}(\hat{\mathcal{R}}_d) = 2 \cdot (5 \cdot 4) + 0 \cdot 4 + 2 = 42$, then `src` is $(double*)a + 42$. `size` is $\{Shape(\hat{\mathcal{R}}_d)_1, Shape(\hat{\mathcal{R}}_d)_2, Shape(\hat{\mathcal{R}}_d)_3 \cdot \text{sizeof}(e^{type})\} = \{4, 3, 16\}$, `spitch` is $\{Shape(d)_2, Shape(d)_3 \cdot \text{sizeof}(e^{type})\} = \{5, 32\}$ and `dpitch` is $\{BoundingBox(d)_2, BoundingBox(d)_3 \cdot \text{sizeof}(e^{type})\} = \{4, 24\}$. Hence, the generated API is:

```
swapnd_buffer(d_id, (uint64_t*)((double*)a+42), 3, (int[]){4, 3, 16},
(int[]){5, 32}, (int[]){4, 24});
```

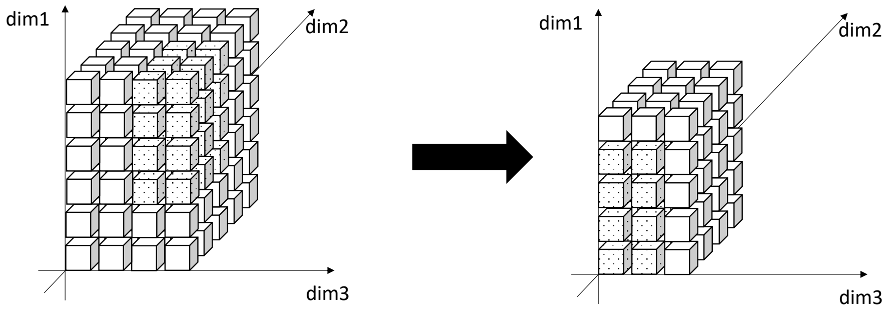


Figure 5.4: 3 dimension data transfer example

Algorithm 3: Generate API for N-dimension buffer swap

```
Data: shape of array $a$: $Shape(a)$
      canonical data element range of array $a$: $\hat{\mathcal{R}}_a$
      bounding box of array $a$: $BoundingBox(a)$
      type of one data element in array $a$: $e^{type}$
1  switch $n = length(Shape(\hat{\mathcal{R}}_a))$ do
2      case 1 do
3          generate( swap_buffer , {a_id, (uint64_t*)(($e^{type}$*)a+$idx_1^{\min}(\hat{\mathcal{R}}_a)$),
            $Shape(\hat{\mathcal{R}}_a)_1 \cdot$ sizeof($e^{type}$)})
4      end
5      case 2 do
6          AddressOffset=$Shape(a)_2 \cdot idx_1^{\min}(\hat{\mathcal{R}}_a) + idx_2^{\min}(\hat{\mathcal{R}}_a)$
7          generate( swap2d_buffer , {a_id, (uint64_t*)(($e^{type}$*)a+AddressOffset),
            $Shape(\hat{\mathcal{R}}_a)_2 \cdot$ sizeof($e^{type}$), $Shape(\hat{\mathcal{R}}_a)_1$, $Shape(a)_2 \cdot$ sizeof($e^{type}$),
            $BoundingBox(a)_2 \cdot$ sizeof($e^{type}$)})
8      end
9      otherwise do
10         AddressOffset=$\sum_{i=1}^{n-1}(idx_i^{\min}(\hat{\mathcal{R}}_a) \cdot \prod_{j=i+1}^{n} Shape(a)_j) + idx_n^{\min}(\hat{\mathcal{R}}_a)$
11         generate( swapnd_buffer ,{a_id, (uint64_t*)(($e^{type}$*)a+AddressOffset), n,
            (int[]){$Shape(\hat{\mathcal{R}}_a)_1, ..., Shape(\hat{\mathcal{R}}_a)_{n-1}, Shape(\hat{\mathcal{R}}_a)_n \cdot$ sizeof($e^{type}$)},
            (int[]){$Shape(a)_2, ..., Shape(a)_{n-1}, Shape(a)_n \cdot$ sizeof($e^{type}$),
            (int[]){$BoundingBox(a)_2, ..., BoundingBox(a)_{n-1}, BoundingBox(a)_n \cdot$
            sizeof($e^{type}$)}})
12     end
13 end
```

Chapter 6

Evaluation

In this chapter, we evaluate the effectiveness of our optimizer on a variety of kernels. Specifically, we employ kernels from PolyBench-NN[42], which includes CNN, LSTM, Maxpool, Sumpool and RNN kernel with pre-configured loop bound size. We begin by detailing the platform configuration in Section 6.1. We then present results for all listed kernels in Section 6.2. Finally, we provide a more in-depth evaluation of the CNN kernel based on loop bound sizes selected from GoogLeNet[39] in Section 6.3 to highlight how the solution picked by our optimizer changes depending on the loop bounds and shape of arrays.

6.1 Platform Configuration

We use the following default system configuration throughout this section. The system is configured with 8 cores running at 1 GHz, each of them is connected with an independent 128 KByte SPM. These SPMs are connected and controlled by a central DMA controller. The main memory is configured with a data access granularity of 64 bytes and a line overhead of $T_{DMA}^{overhead} = 40$ ns, which are representative of a 64 bits width DRAM device. The default bus speed between main memory and SPM is 16 GByte/s, resulting in $T_{BUS}^{overhead} = 0.0625$ ns/byte.

We compile the code to ARM64. For execution time estimation, we run the kernel on the gem5 architectural simulator [7] using the AtomicSimpleCPU model for ARM architecture. After each `end_segment` call, a pseudo instruction `dumpstats` is inserted. When this instruction get executed, the simulator records the statistics during the execution of this tile.

Note that the simulated code does not include the time of API calls; therefore, we decided to add the API overhead to each execution phase length based on the worst-case time measured in [36] on a similar hardware platform with core frequency normalized by 1 Ghz. The data of API overhead is shown in Table 6.1. We verified that for all tested scenarios, the maximum difference between the actual makespan of a kernel and the one predicted through our timing model introduced in Section 4.2 is within 5% by running the final compiled kernel in gem5.

PREM API	WCET normalized
`allocate_buffer`	1139 ns
`dispatch`	861 ns
`DMA_int_handler`	1187 ns
`allocate`	1503 ns
`end_segment`	1878 ns
`deallocate`	861 ns
`allocate2d`	1103 ns
`deallocate_buffer`	776 ns
`swap_buffer`	1914 ns
`swap2d_buffer`	1248 ns

Table 6.1: Normalized worst-case execution time of PREM APIs from [36]

Note that since we did not modify the API implemented in [36], we do not have overhead results for the `threadID` and `swapnd_buffer` calls. We decided to assume that `threadID` has no additional time cost as the value of it is stored in a specific register on core, while `swapnd_buffer` has a similar worst case execution time as `swapnd_buffer` call as they share structural similarity in implementation. Note that similarly to [36], this assumes the availability of a programmable DMA component that can efficiently load/unload the multidimensional array (including the required strides in SPM and memory) in hardware; practically speaking, this could only be implemented for a limited number of dimensions. This said, we notice that in the kernels that are used in this evaluation, the maximum number of dimensions of any employed array is 4. We argue that on these corpus of kernels, the actual programs are unlikely to use arrays with more than 4 dimensions. We further argue that since CNN kernel used in the evaluation is widely used in image processing, it is also representive in the area of computer vision.

6.2 Polybench Kernels

We apply our optimizer on kernels in the PolyBench-NN benchmark suite, which was proposed to test polyhedral optimizations on machine learning kernels. Specifically, we use all 5 forward passes (for inference), representing 5 DNN layers. We use the `LARGE` problem size since it uses approximately 25 MB in each kernel and thus cannot fit entirely in the local memory of a typical MPSoC system.

When compiling the kernels, we enable PolyBench's `POLYBENCH_USE_SCALAR_LB` option. This option ensures that scalar loop bounds, rather than parametric ones, are passed to *pet*. Under this option, our toolchain can compute the tiling size and parallelization selections during compile time.

We compare our optimized results with the following approaches: **(1)** An ideal single-core case where we assume that the SPM size is unlimited, and data transfers take 0 time; hence, there is no need to tile the kernel. **(2)** The greedy approach from related work [29]. In this case, we find the outermost loop level that can be tiled while guaranteeing that the resulting segments fit into the SPM, and tile only at that loop level with the maximum allowed tile size. Whether possible, iterations of outer loops are executed in parallel.

Figure 6.2: Running time of generating Figure 6.1 with Optimization Heuristic

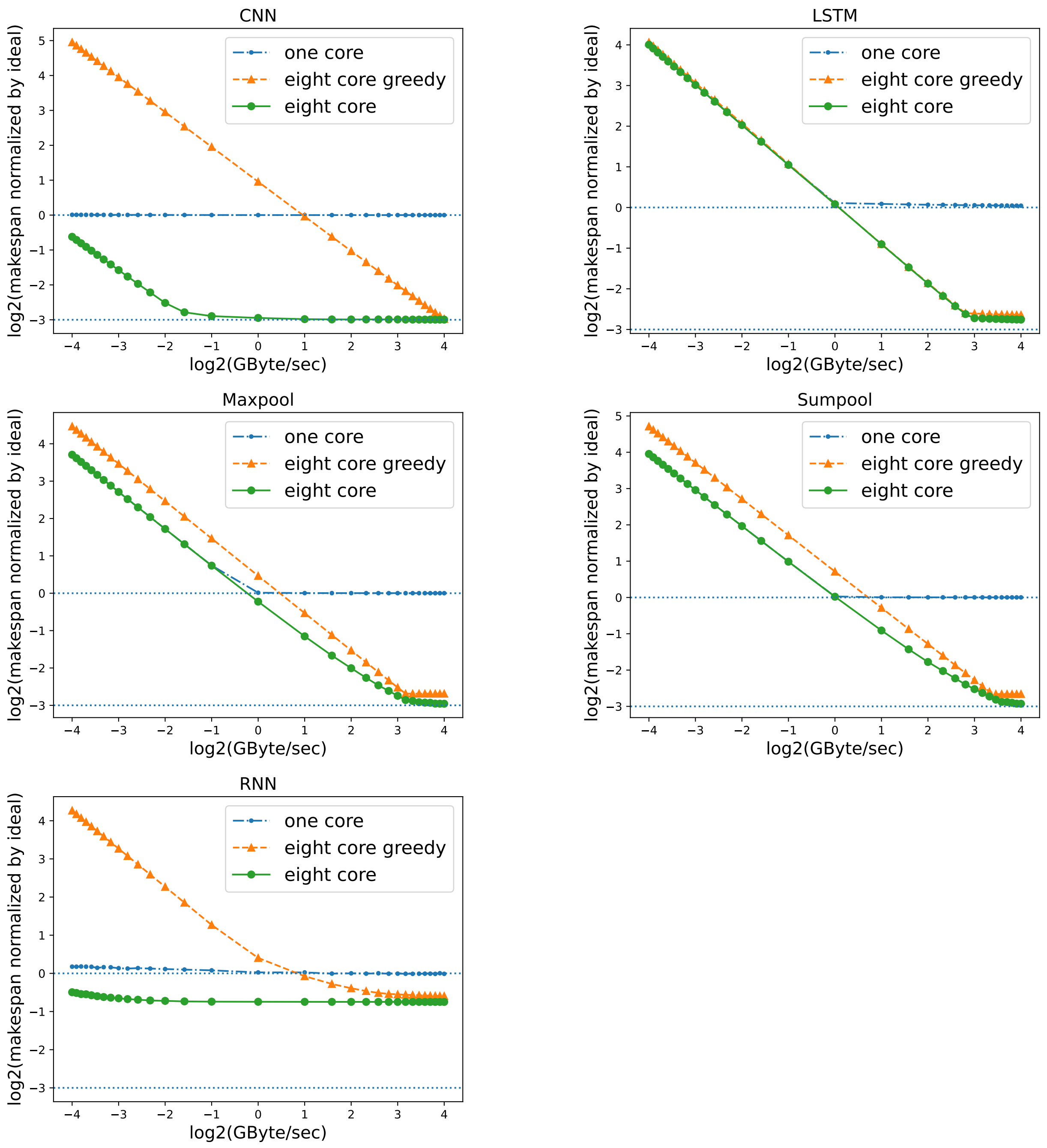
	cnn	lstm	maxpool	sumpool	rnn
min	154.66 sec	1.77 sec	58.02 sec	28.57 sec	56.92 sec
max	407.03 sec	30.71 sec	277.98 sec	296.53 sec	571.11 sec
average	154.66 sec	10.93 sec	138.79 sec	118.03 sec	251.21 sec

Figure 6.3: Running time of generating Figure 6.1 with Greedy Approach

	cnn	lstm	maxpool	sumpool	rnn
min	0.52 sec	0.0013 sec	0.10 sec	0.10 sec	0.27 sec
max	0.55 sec	0.0016 sec	0.11 sec	0.12 sec	0.31 sec
average	0.52 sec	0.0014 sec	0.10 sec	0.10 sec	0.27 sec

Figure 6.1 shows results for our approach on either 1 or 8 cores, as well as the greedy case on 8 cores, normalized by the ideal single-core case, where we vary the memory bandwidth from 1/16 to 16 Gbytes/s. Note that due to the logarithmic scale on the y-axis, a value of 0 corresponds to the makespan of the ideal case, while a value of -3

Figure 6.1: Makespan of forward passes in PolyBench-NN, normalized by the ideal single-core case



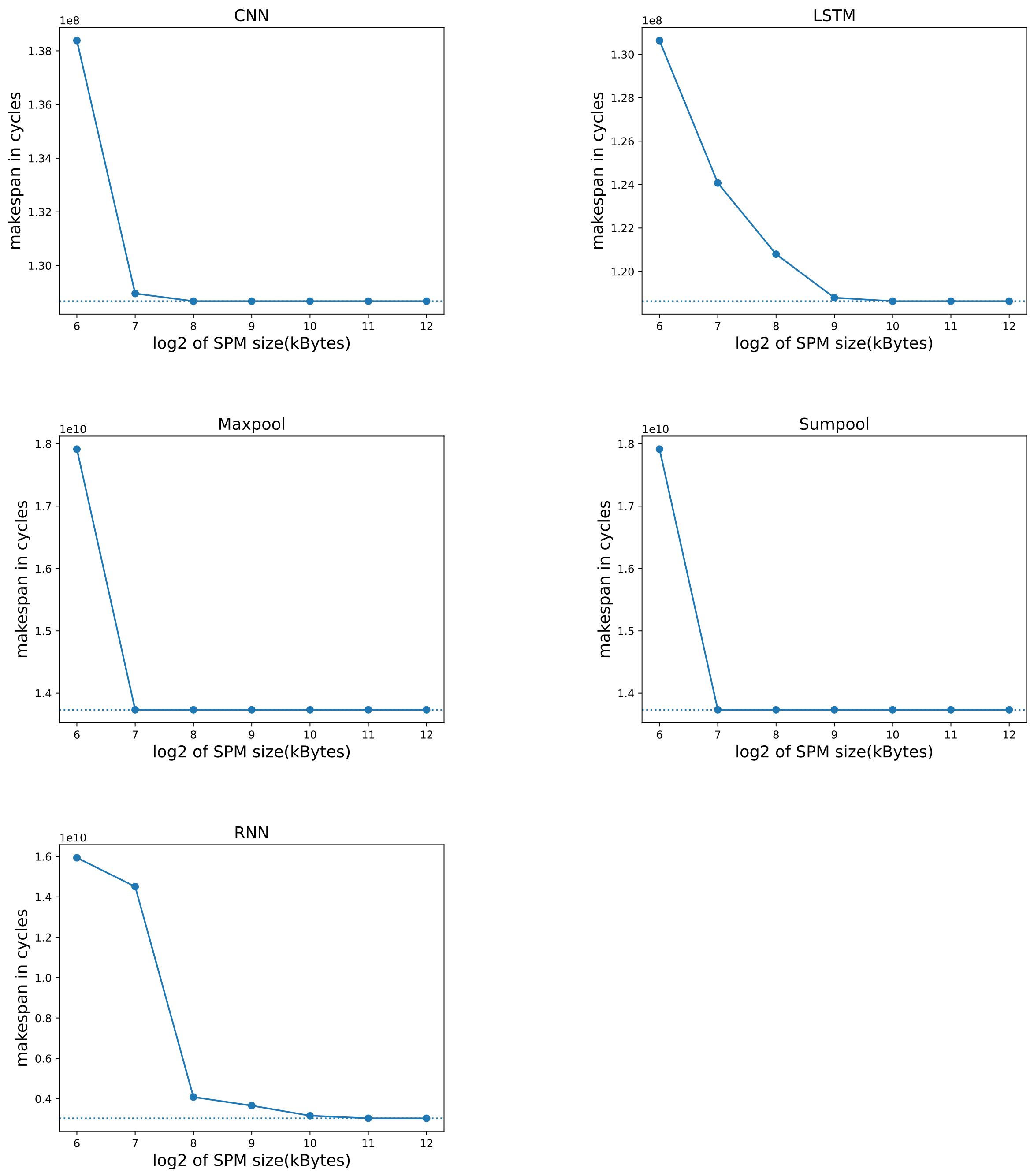
represents perfect scalability on 8 cores. All graphs reach a plateau for different values of memory bandwidth, indicating that the schedule becomes computation-bound. Given sufficient memory bandwidth, for the 1 core case, our approach comes very close to the ideal case, while for the 8 core case, 4 out of the 4 benchmarks show excellent scalability. RNN performs worse because one major component inside this kernel is not parallelizable. Except for lstm, our approach can better utilize memory bandwidth compared to greedy. Specifically for the CNN case in Figure 6.1, greedy performs poorly because the single loop level it tiles on results in large data arrays being loaded every tile.

As we can see in Figure 6.1, the overhead of API calls which are inserted is very small, as the result of one core case is very close to ideal one case. In fact, the maximum API overhead suffered in all these experiments is 4.37%, thus the overhead of API calls does not make a big difference. This contributes to the fact that, at fast bus speed, our optimizer produces similar result with greedy approach. The reason for this is at fast bus speed, memory time is practically zero compared to computation time, thus not only we are computation bound, but we also do not care too much about the memory transfer time of first/last segment. Plus, as we have a small API call overhead, it turns out that the number of segments does not matter. Thus, the conclusion is any solution that performs reasonable load-balancing among the cores would have basically the same performance. This also implies that we should potentially focus more on the optimization of balance between memory phase length and computation phase length than the optimization of API call overhead.

We can also see that at slow bus speed, the kernels' total makespans are largely determined by bus speed. This is because at slow bus speed, the total makespan is bounded by memory time as we are using a single DMA controller for all the memory operations on all cores. Hence, in this case, our optimizer attempts to maximize memory reuse and minimize the memory transfers during program's execution. This is also the case where our optimizer outperforms the greedy approach. In Section 6.3, we would show the detailed mechanism of our optimization approach.

Finally, Table 6.2 shows the runtime required to generate the graphs in Figure 6.1 for our approach on 8 cores, where we execute Algorithm 2 with single process python interpreter on an Intel Xeon processor running at 3.5Ghz. There exists difference between min and max time because our search algorithm picks a random tiling and parallelization selection as initial search position. When a small tiling size is picked, it would construct a DAG graph with much more nodes compared with normal tiling size, and it takes longer to compute the longest path length in this DAG. Compared with the greedy approach, our approach takes longer to produce a best selection because instead of simply computing array size to fit in SPM, our approach searches for tile sizes that produce shortest makespan.

Figure 6.4: Makespan vs SPM size of Polybench kernels



To better study the benefit a larger SPM size could bring to these kernels, in Figure 6.4, we show the makespan under different SPM sizes. In this figure, the y-axis is the total makespan in cycles and the x-axis is log2 of the SPM size on each core. The dotted line in each sub-figure shows the makespan under infinite SPM size. We can see that as the SPM size increases, the makespan decreases until it reaches a plateau. Notice that under infinite SPM size, the best selection does not necessarily load everything at the beginning of the first segment. Even at slow bus speed, given large enough SPM size, our approach performs better than greedy approach because with limited SPM size, it is not enough for greedy approach to maximize memory reuse by loading everything, but it is enough for our approach to have max memory reuse pattern. Limited SPM size is not a significant drawback if tiling overhead is small enough because loading an array's different parts over multiple segments requires the same total data transfer size in bytes as loading all of the array once.

6.3 CNN kernels in GoogLeNet

To further study the behavior of our optimization framework, in this section, we use the CNN kernel as an example and study its performance under a wider range of parameters.

```
for (int n = 0; n < NN; n++)
  for (int k = 0; k < NK; k++)
    for (int p = 0; p < NP; p++)
      for (int q = 0; q < NQ; q++)
        for (int c = 0; c < NC; c++)
          for (int r = 0; r < NR; r++)
            for (int s = 0; s < NS; s++)
              out_F[n][k][p][q] += W[k][c][r][s] * inp_F[n][c][p + NR
  - r - 1][q + NS - s - 1];
```

Listing 6.1: source code of CNN kernel used in Polybench

Listing 6.1 is the source code of CNN kernel with filter stride set to 1. Three arrays are involved in computation. out_F is the array of output feature map, W is the array of feature weights and inp_F is the array of input features. CNN is a kernel with multiple loop levels, Table 6.5 shows what each loop bound represents.

Figure 6.5: Loop Bounds in CNN

NN	Number of Input Images in batch
NK	Number of Output feature maps
NP, NQ	Size of output feature map
NC	Number of Input feature maps
NR, NS	Size of filter kernel

In an actual neural network architecture, the size of convolution operator changes in different network layers. Generally, the convolution's map size becomes smaller as the number of features grows. We study the CNN kernels used by GoogLeNet [39] as an example to show the best tiling and parallelization selections can be different under different loop bound sizes.

GoogLeNet uses three different filter sizes, 1x1, 3x3 and 5x5. Here, we use the 3x3 filter with different feature number and feature map size for comparison. Table 6.6 shows the best tiling and parallelization selection for the various kernels. We only test with batch size NN set to 1 and filter stride set to 1, thus the table only reports feature map size and number of input/output features. It also only reports the thread group number and tile size for loop level k, c, p, q because the two filter loops r, s are only 3 iterations, and they are too small to tile efficiently. For the sake of conciseness, for each map size, we use the kernel with only the largest and smallest number of features.

The thread group numbers and tile sizes are the best selections under 1/512 GBytes/s bus speed, which is very slow. We can see these selections are different with the same CNN kernel of different loop bounds, and they are generally difficult to find manually, justifying the need for an automated optimization tool.

Figure 6.6: Best selections for CNN with different loop bounds

loop bound (NK/NP/NQ/NC)	thread group number ($l_k.R/l_p.R/l_q.R$)	tile size ($l_k.K/l_p.K/l_q.K/l_c.K$)
128 / 28 / 28 / 96	4 / 2 / 1	32 / 14 / 28 / 5
192 / 28 / 28 / 128	2 / 3 / 1	48 / 10 / 28 / 3
208 / 14 / 14 / 96	4 / 1 / 1	52 / 14 / 14 / 7
320 / 14 / 14 / 160	1 / 1 / 1	64 / 14 / 14 / 3
320 / 7 / 7 / 160	4 / 1 / 1	80 / 7 / 7 / 16
384 / 7 / 7 / 192	8 / 1 / 1	80 / 7 / 7 / 16

6.3.1 Comparison with greedy approach

Figure 6.1 shows that our approach significantly outperforms the Greedy one at low bus speeds. To understand why, in this section we discuss the solutions found by Greedy and our approach in more details. The CNN kernel we use for comparison is selected from GoogLeNet. It has a loop bound of {'k': 128, 'p': 28, 'q': 28, 'c': 96, 'r': 3, 's': 3}. The greedy selection is R: {'k': 8, 'p': 1, 'q': 1} K: {'k': 1, 'p': 2, 'q': 28, 'c': 96, 'r': 3, 's': 3}, we call it *selection_greedy*. Under 1/32 Gbyte/s, the obtained makespan is 1,460,278,989 cpu cycles. The best selection of this CNN kernel for our approach under 1/32 Gbyte/s is R: {'k': 4, 'p': 2, 'q': 1}, K: {'k': 32, 'p': 7, 'q': 28, 'c': 16, 'r': 3, 's': 3}, we call it *selection_best*. It has a total makespan of 142,497'144 cycles, which is around 10x less than selection_greedy.

The reason why selection_best outperforms selection_greedy is that selection_best transfers a total of 4,579,328 bytes, while selection_greedy transfers 45,628,416 bytes, which is nearly 10x larger. In each segment, we access the following arrays: out_F[n][k][p][q], W[k][c][r][s], inp_F[n][c][p+2-r][q+2-s]. In selection_best, we load the following arrays, out_F, W, inp_F. In array out_F, $n = 1$, $k = 32$, $p = 7$, $q = 28$ and it is an $32 \cdot 196$ array, which is 6272 elements. In array W, $k = 32$, $c = 16$, $r = 3$, $s = 3$ and it is an $32 \cdot 144$ array, which is 4608 elements. In array inp_F, $n = 1$, $c = 16$, $p = 7$, $q = 28$ and it is an $16 \cdot (7 + 3) \cdot (28 + 3)$ array which is 4960 elements. The total SPM occupation is 15840 elements $\cdot$ 4 bytes per element $\cdot$ 2 buffers, which is 126,720 bytes.

In selection_greedy, we load out_F of a $2 \cdot 28$ array which is 56 elements, W of an 864 elements array, inp_F of $96 \cdot (2+3) \cdot (28+3)$ array which is 14880 elements. The total SPM occupation is $15800 \cdot 4 \cdot 2$, which is 126,400 bytes.

We can see that selection_greedy has a similar SPM occupation per segment with selection_best. However, the memory efficiency of selection_greedy is worse because each time a new segment is executed, loop index p would change, thus selection_greedy requires swapping in/out inp_F which is a very large array every segment. This can be solved if a much larger SPM is provided to store the whole inp_F array, then it would not need to swap in/out, thus save a lot of memory loading time in selection_greedy. Yet with current SPM size, it cannot see that this is a bad idea to reload such a large array every time and thus causing a very long total makespan.

On the other hand, selection_best performs better because with similar memory transfer per segment, the segments in selection_best contains more innermost loop iterations. selection_best has 104 segments in total, and selection_greedy has 776 segments in total. For the number of innermost iterations captured in one segment, we find a full selection_best segment contains 903,168 innermost iterations and a full selection_greedy segment contains

Figure 6.7: Best selections for CNN under different bus speed

bus speed (Gbytes/sec)	thread group number ($l_k.R/l_p.R/l_q.R$)	tile size ($l_k.K/l_p.K/l_q.K/l_c.K$)
1/64	4 / 2 / 1	32 / 14 / 28 / 3
1/64 + 0.01	4 / 2 / 1	32 / 14 / 28 / 4
1/64 + 0.02	2 / 4 / 1	32 / 7 / 28 / 12
1/64 + 0.03	2 / 4 / 1	32 / 7 / 28 / 8
1/64 + 0.04	8 / 1 / 1	16 / 14 / 14 / 12
1/64 + 0.05	8 / 1 / 1	16 / 7 / 14 / 16
1/64 + 0.06	2 / 2 / 2	16 / 7 / 14 / 16
1/64 + 0.07	8 / 1 / 1	8 / 4 / 28 / 16
1/64 + 0.08	8 / 1 / 1	8 / 7 / 14 / 24
1/64 + 0.09	2 / 2 / 2	8 / 7 / 14 / 24
1/64 + 0.10	2 / 2 / 2	8 / 7 / 14 / 24

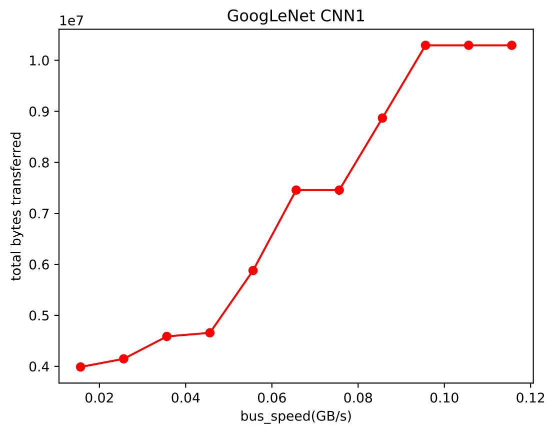
120,960 innermost iterations. This means that the bad tiling shape of selection_greedy would create inefficient memory transfer patterns and increase CNN kernel's makespan.

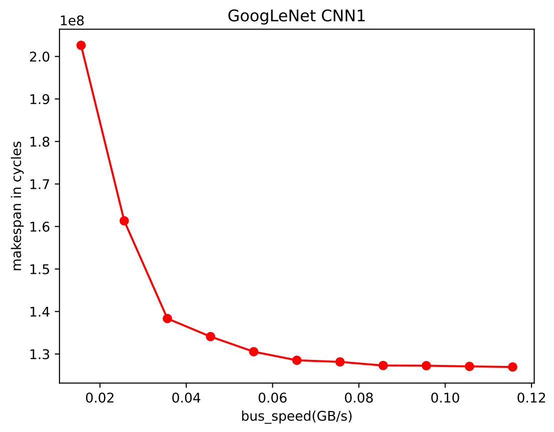
6.3.2 Boundary region

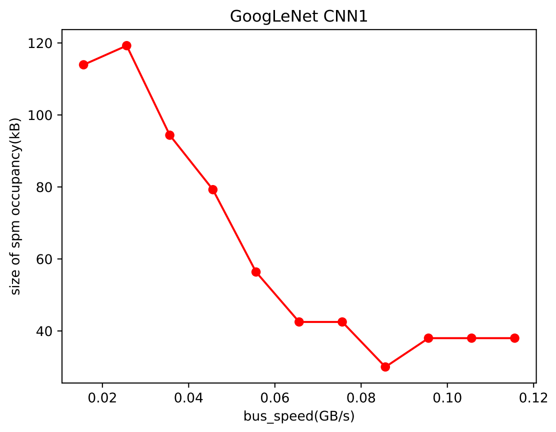
Our evaluation so far provides two key insights: at fast bus speeds, where the execution is computation bounded, the tile sizes, and thread group selections do not significantly affect the resulting makespan - it is simply sufficient to balance execution between the available threads/cores. On the other hand, at slow bus speeds, where the execution becomes memory bounded, our optimizer is driven to find solutions that maximize memory reuse. But what happens at the boundary between these two extreme situations? To study the behavior of our optimizer in the boundary region, we use the same CNN kernel in the last subsection with loop bounds of {'k': 128, 'p': 28, 'q': 28, 'c': 96, 'r': 3, 's': 3}. For this kernel, we experimentally set the boundary region to range from around 1/64 Gbyte/s to 1/8 Gbyte/s. We then increase the bus speed with the step of 0.01 Gbyte/s and Figure 6.8 shows the result and Table 6.7 are the best selections we found for each bus speed value.

From the figure, we can see the program's execution transits from totally memory bounded to computation bounded in the non-linear bus speed region. Under relatively fast bus speed, we can see the best selections tend to take less SPM size. This enables them to have shorter time loading/unloading before/after first/last segment, which is crucial when each selection has a similar total computation time. Meanwhile, the total bytes transferred increase, but this does not become a problem any more under faster bus speed. Thus, the conclusion is that it is acceptable to progressively increase the total bytes transferred as a

Figure 6.8: Makespan, transferred data and SPM utilization for CNN under different bus speed







price paid to reduce the time of loading/unloading before/after first/last segment, as long as the program's execution does not become memory bounded.

Chapter 7

Conclusions and Future Work

In this thesis, we extended PREM to execute nested loop structure programs as a parallel application on an SPM-based system. We presented a workflow that automatically generates PREM-compliant optimized nested loop programs. To achieve this, polyhedral compilation tools are adopted to analyze the original program and generate an optimized program using techniques of tiling and parallelization. We modeled the timing cost for API call overhead and data transfer time based on the optimization solution selection. An optimization solution is then found using a heuristic algorithm to minimize the makespan of the program. We evaluated the proposed approach on kernels from the PolyBench-NN benchmark, using the gem5 architectural simulator to obtain execution time bounds. We exhibit that a complex interplay exists between tile sizes and the quality of the resulting schedule.

This work could be further expanded in multiple directions. First, in this work, SPM is only adopted as a first level SPM which load data directly from main memory. The disadvantage of this approach is the first level SPM is usually small. This disadvantage could be potentially addressed by adopting a multi-level SPM system. Instead of loading required data from main memory to L1 SPM every single segment, the required data of multiple segments can be loaded into L2 SPM at once and later again load into L1 SPM when the data is required in current segment. The challenge of this approach might be grouping multiple segments into a larger “block” and hiding the data transfer time from main memory to L2 SPM every a few segments. This might also bring challenges to the scheduling algorithm on multicore architecture.

Second, this work provides parallel execution functionality, but only for a single application. In real world, multiple applications execute on real-time operating system. This is supported in [38] using multi-segment streaming with 3-phase execution model. This functionality could be better integrated with multi-level SPM described above. By breaking the schedule into blocks that contain multiple segments, a different application can be scheduled after the block of the current application completes. The double buffer streaming strategy can also be applied for application switch, using one partition of the L2 SPM to load/unload the data of one application from/to main memory while the another application is used to run the current application. This idea could be used to achieve faster context switching time between different tasks since the data is already in L2 SPM before the execution of the task.

Third, in this work, we only evaluate our approach by executing the code in a simulator environment. It would be highly desirable to execute the compiled applications on an actual platform. This would require extensions to the OS infrastructure introduced in [36] for the PREM streaming system. In addition, the hardware platform employed in [36] uses only three cores, while our technique is designed to scale to a larger number of cores. Ideally, we would thus prefer to target a larger multicore system. A potential target is the open hardware design introduced in [24]; this platform provides a multicore accelerator that integrates up to 8 RISC-V cores with individual SPMs, plus a larger platform-level L2 SPM.

References

[1] 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.

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

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

[4] Matthias Becker, Dakshina Dasari, Borislav Nicolic, Benny Akesson, Vincent Nelis, and Thomas Nolte. Contention-Free Execution of Automotive Applications on a Clustered Many-Core Platform. In 2016 28th Euromicro Conference on Real-Time Systems (ECRTS). IEEE, 2016.

[5] M. Benabderrahmane, L. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In R. Gupta, editor, Compiler Construction, pages 283–303, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[6] Ian J. Bertolacci, Catherine Olschanowsky, Ben Harshbarger, Bradford L. Chamberlain, David G. Wonnacott, and Michelle Mills Strout. Parameterized diamond tiling for stencil computations with chapel parallel iterators. In Proceedings of the 29th ACM on International Conference on Supercomputing, ICS '15, page 197–206, New York, NY, USA, 2015. Association for Computing Machinery.

[7] N. Binkert, B. Beckmann, G. Black, S. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. Hill,

and D. Wood. The gem5 simulator. SIGARCH Comput. Archit. News, 39(2):1–7, aug 2011.

[8] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In International Conference on Compiler Construction (ETAPS CC), April 2008.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] P. Feautrier and C. Lengauer. The polyhedron model. In David Padua, editor, Encyclopedia of Parallel Computing, pages 1581–1592. Springer, 2011.

[14] 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.

[15] 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.

[16] Bjorn Forsberg, Maxim Mattheeuws, Andreas Kurth, Andre Marongiu, and Luca Benini. A synergistic approach to predictable compilation and scheduling on commodity multi-cores. In Proceedings of Languages, Compilers, Tools and Theory of Embedded Systems, LCTES'20, 06 2020.

[17] Björn Forsberg, Luca Benini, and Andrea Marongiu. Heprem: A predictable execution model for gpu-based heterogeneous socs. IEEE Transactions on Computers, 70(1):17–29, 2021.

[18] Philip Ginsbach, Lewis Crawford, and Michael F. P. O'Boyle. Candl: A domain specific language for compiler analysis. In Proceedings of the 27th International Conference on Compiler Construction, CC 2018, page 151–162, New York, NY, USA, 2018. Association for Computing Machinery.

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

[20] Tobias Grosser, Armin Größlinger, and Christian Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. Parallel Process. Lett., 22, 2012.

[21] Zhao Gu and Rodolfo Pellizzoni. Optimizing parallel PREM compilation over nested loop structures. In Rob Oshana, editor, DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022, pages 1249–1254. ACM, 2022.

[22] M. Hassan and R. Pellizzoni. Bounding dram interference in cots heterogeneous mpsocs for mixed criticality systems. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 37(11):2323–2336, Nov 2018.

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

[24] Andreas Kurth, Pirmin Vogel, Alessandro Capotondi, Andrea Marongiu, and Luca Benini. HERO: heterogeneous embedded research platform for exploring RISC-V manycore accelerators on FPGA. CoRR, abs/1712.06497, 2017.

[25] Leslie Lamport. The parallel execution of do loops. Commun. ACM, 17(2):83–93, feb 1974.

[26] 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.

[27] 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.

[28] Joel Matějka, Björn Forsberg, Michal Sojka, Přemysl Šůcha, Luca Benini, Andrea Marongiu, and Zdeněk Hanzálek. Combining PREM Compilation and Static Scheduling for High-Performance and Predictable MPSoC Execution. Parallel Computing, 12 2018.

[29] J. Matějka, B. Forsberg, M. Sojka, Z. Hanzálek, L. Benini, and A. Marongiu. Combining prem compilation and ilp scheduling for high-performance and predictable mpsoc execution. In Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM'18, pages 11–20, New York, NY, USA, 2018. ACM.

[30] 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.

[31] 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.

[32] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. Graphite: Polyhedral analyses and optimizations for gcc. In proceedings of the 2006 GCC developers summit, volume 6, pages 90–91. Citeseer, 2006.

[33] Juan M. Rivas, Joël Goossens, Xavier Poczekajlo, and Antonio Paolillo. Implementation of Memory Centric Scheduling for COTS Multi-Core Real-Time Systems. In 31st Euromicro Conference on Real-Time Systems (ECRTS 2019), volume 133, 2019.

[34] 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.

[35] Benjamin Rouxel, Stefanos Skalistis, Steven Derrien, and Isabelle Puaut. Hiding communication delays in contention-free execution for spm-based multi-core architectures. In 2019 31th Euromicro Conference on Real-Time Systems (ECRTS), pages 1–23, July 2019.

[36] M. Soliman, G. Gracioli, R. Tabish, R. Pellizzoni, and M. Caccamo. Segment streaming for the three-phase execution model: Design and implementation. In Proceedings of the 40th Real-Time Systems Symposium, RTSS'19, 12 2019.

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

[38] Muhammad Refaat Sedky Soliman. Automated compilation framework for scratchpad-based real-time systems. PhD thesis, University of Waterloo, 2019.

[39] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions, 2014.

[40] 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.

[41] Rohan Tabish, Renato Mancuso, Saud Wasly, Sujit S. Phatak, Rodolfo Pellizzoni, and Marco Caccamo. A Reliable and Predictable Scratchpad-centric OS for Multicore Embedded Systems. In 2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). IEEE, 2017.

[42] H. Vaidya, A. Patwardhan, R. Upadrasta, and A. Badrinaaraayanan. When polyhedral optimizations meet deep learning kernels. 12 24th IEEE International Conference on High Performance Computing, Data, and Analytics, 2017.

[43] S. Verdoolaege and T. Grosser. Polyhedral extraction tool. 01 2012.

[44] S. Verdoolaege, S. Guelton, T. Grosser, and A. Cohen. Schedule trees. 01 2014.

[45] Sven Verdoolaege. isl: An integer set library for the polyhedral model. volume 6327, pages 299–302, 09 2010.

[46] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, Mathematical Software – ICMS 2010, pages 299–302, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[47] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for CUDA. ACM Transactions on Architecture and Code Optimization, 9(4):1–23, 1 2013.

[48] Sven Verdoolaege and Gerda Janssens. Scheduling for PPCG. 06 2017.

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

[50] 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.

[51] 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.

[52] 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.