COMSAT: Modified Modulo Scheduling Techniques for Acceleration on Unknown Trip Count and Early Exit Loops

by

Quoc Long Vinh Ta

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

ARIZONA STATE UNIVERSITY

December 2022

ABSTRACT

Coarse-grain reconfigurable architectures (CGRAs) have shown significant improvements as hardware accelerator whilst demanding low power. Such acceleration inherits from the nature of instruction-level parallelism and exploited by many techniques. Modulo scheduling is a popular approach to software pipelining techniques that provides an efficient heuristic to accelerations on loops, repetitive regions of an application. Existing scheduling algorithms for modulo scheduling heuristic persist on loop exiting problems that limit CGRA acceleration to only loops with known trip count and no exit statements. Another notable limitation is the early exit problem, where loops can only terminate after certain iterations as CGRA moves to kernel stage. In attempts to circumvent such obstacles, COMSAT introduces a modified modulo scheduling technique that acts as an external module and can be applied to any existing scheduling/mapping algorithms with minimal hardware changes. Experiments from MiBench and Rodinia benchmark suites have shown that COMSAT achieved an average speedup of 3x in overall benchmarks and 10x speedup in kernel regions. Without COMSAT techniques, only 25% of said loops would have been able to accelerate, reducing benchmark and kernel speedups to 1.25x and 3.63x respectively.

DEDICATION

Sincere thanks to my family for always being by my side and providing uninterrupted supports.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

In the rapid advancement of technologies, especially in machine learning applications that require immense amount of computations, general purpose CPUs cannot keep up with the demand, thus leaving the workloads to hardware accelerators. Whilst GPUs are the most common accelerator, they suffer from energy efficiency due to numerous low-efficiency cores (Mittal and Vetter (2014)). FPGAs help in their programmability for general purpose accelerators, but they are still power demanding. ASICs, in contrast to GPUs and FPGAs, show their efficiency in computation but exchange for programmability and hence production cost (Nurvitadhi *et al.* (2016)). This leaves CGRA a promising candidate thanks to its high programmability and low energy consumption. Whilst CGRA execution focuses on repetitive regions of a program, many GPU applications can also accelerate on CGRAs (Bouwens *et al.* (2008a)). Among different CGRA models including ADRES (Bouwens *et al.* (2008b)), REMARC (Miyamori and Olukotun (1998)), Morphosys (Singh *et al.* (2000)) and more shown in (Hartenstein (2001)), ADRES shows a promising model for general purpose applications with high energy efficiency (60 GOps/W). The ADRES hardware model consists of a 2D rectangular PE array, each houses an ALU/FU to perform arithmetic, logic and memory operations, a register file and 2 input muxes to choose from its 4 neighbors (overlaps at edges), as shown in Figure 1.1. Since each row of PE shares the same memory buffer, only 1 memory operation can be executed per row per cycle. The configuration memory stores all instructions for every PE at every cycle. Note that above architecture does not include a predicate register file, which is used for fully-predicated issuance scheme (Hamzeh *et al.* (2014)) and is not required

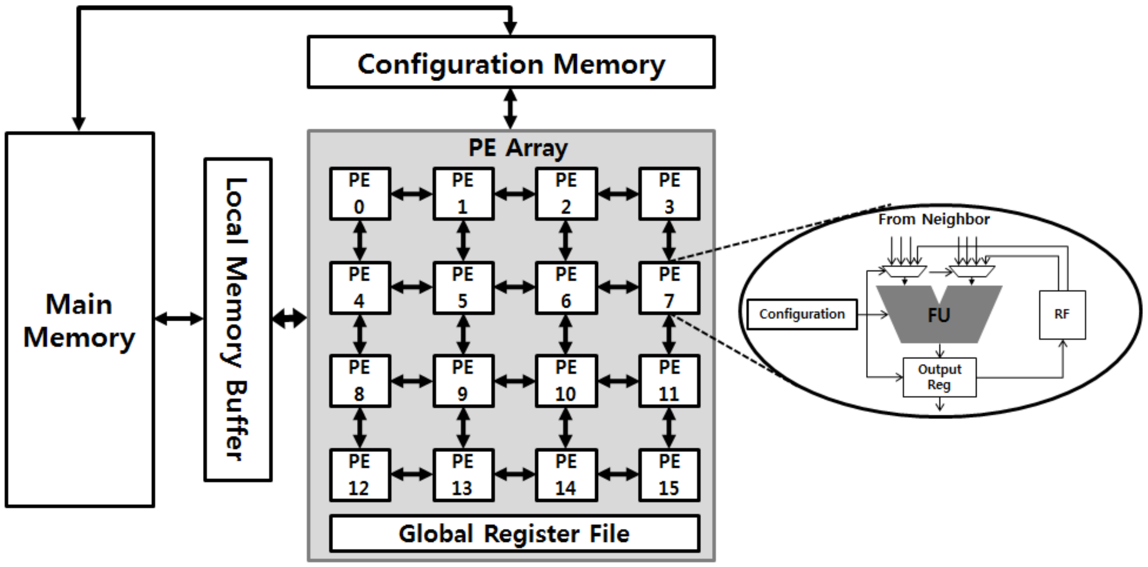


Figure 1.1: CGRA Comprising of Array of PEs Each With a Register File, Configuration Memory Containing Instructions and a Local Memory Buffer in Each Row For Load/Store Operations.

by COMSAT.

Due to the nature of software pipelining, there can be more than 1 iteration being concurrently executed in a cycle and as for the scheduling objective, minimizing initiation interval (II) implies increasing the number of concurrent iterations. Problems arise when a loop tries to terminate at an iteration that by the time its conditions are evaluated, part of the succeeding iterations would have been executed and as CGRA exits the loop, final result would reflect those exceeding trip counts. This is known as the trip count problem and it would have been trivial if trip count was known at compile time so that CGRAs acknowledged the exit even before evaluating loop conditions. Unfortunately, trip count information is rarely given and accurate accelerations on such loops become impossible without human intervention to adjust exits, defeating purpose of compilers. In another observation, modulo scheduled loops require some minimum numbers of iterations to accurately execute them, referred to as the early exit problem. This is because modulo scheduled loops include 3 stages,

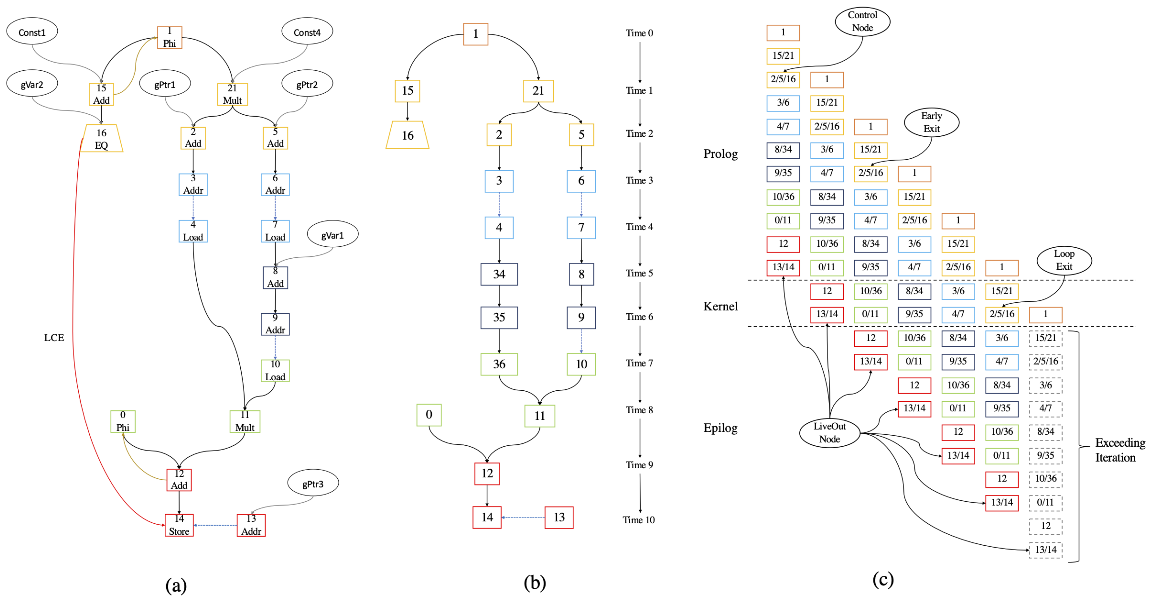


Figure 1.2: (a) DFG of GEMM Benchmark. (b) Iteration Schedule of the DFG. (c) Modulo Schedule.

prolog, kernel and epilog, adding to some iterations that become a minimum trip count requirement. As CGRAs must execute through all these stages before exiting, loops with trip counts fewer than their schedules' minimums are unreliable for execution. Another prominent issue comes from exit statements in some loops since abrupt exits defy trip counts even when they are static, and challenge CGRA's branching ability required to exit. These 3 problems are ubiquitous and tremendously limit the potential acceleration on many applications under modulo scheduling heuristic.

Consider an example of a DFG generated from the innermost kernel loop of a general matrix multiplication (GEMM) application and its iteration schedule generated using CRIMSON (Balasubramanian and Shrivastava (2020)), as shown respectively in Figure 1.2(a) and (b). Figure 1.2(c) illustrates the modulo schedule with dashed lines separating the kernel region from prolog and epilog stages. Each rectangular box in (c) shows a set of nodes from the same iteration that are executed in the same cycle. Each column resembles an entire iteration schedule and a row corresponds to an executing cycle containing operations from across iterations. Note that the edge going from node 16 to 14, denoted as LCE in (a), marks the dependency between control node 16 and live out node 14. The modulo schedule in (c) also points out the boxes containing control node and live out node to highlight their executing cycles. With the focus on control node as it exits the loop in kernel stage, we can see epilog carries an additional iteration, shown as the dashed column in epilog schedule. This is because a new iteration is initiated in the same cycle as control node in kernel, causing epilog to finish off this redundant iteration. As trip count is provided before execution, exit signals from the control node are ignored and CGRAs would switch to epilog 1 iteration early to account for the exceeding iteration. However, loops with unknown trip count are unable to do so since they can only exit upon the signal. Furthermore, we can see this schedule contains 7 columns, indicating that CGRAs have to execute this loop for at least 7 iterations. If the loop was to exit after the third trip count (as indicated by the early exit box in (c)), CGRAs would have exceeded 4 iterations and caused memory fault since it may have read/written to invalid addresses. This is known as the early exit loop problem and it happens when loops try to exit in the prolog stage. Although above example does not have exit statements, for the sake of problem illustration, imagine node 8 was a hypothetical node evaluating conditions for the exit. As node 8 resets and issues exit signal, CGRA finishes off the kernel trip count and branch to epilog schedule. Similar to the unknown trip count problem, we can observe that epilog would carry 3 excessive iterations if the loop was to exit after node 8. With COMSAT techniques applied to the schedule in (c), operations in the exceeding iterations are excluded in epilog stage and different prolog schedules are generated to support early exit loops. As a result, CGRAs running on schedules refined by COMSAT do not require trip count nor predication on any operations whilst still guaranteeing correctness.

Chapter 2

RELATED WORKS

Modulo scheduling technique to software pipelining is a great way to accelerate repetitive regions of an application, but it is not flawless since loops with multiple exits or branches and loops without counters pose challenges (Lavery (1997)). Branching is problematic because nodes in the DFG are scheduled to repeat themselves and there are concurrent iterations being executed in any cycle, as branching happens, certain nodes should not be repeated and data dependencies would be violated. Loops with unknown trip count and exit statements face the same problem since exits are unpredictable and they require branches to terminate loops.

The unknown trip count problem was partially addressed in (Tirumalai *et al.* (1990)), by proposing a technique using rotating predicate file to support full predication issuance scheme. Specifically, a special node called wtop and its rotating predicate file are used to determine whether nodes from succeeding iterations should be executed or not. Operations in the exceeding iterations were NO-OP'ed out thanks to the fully-predicating scheme that predicates every loop operation at runtime. However, this technique is limited to only certain loop conditions and its predicating register size. Each element in the predicate register file corresponds to an iteration's predicate, a value of 1 for the element hints the CGRA to execute operations from that trip count. Since the predicate for each iteration depends on loop conditions, some conditions such as EQUAL or NOT EQUAL may only reset for 1 iteration that signals the CGRA to go to epilog, iterations beyond that would still assert for execution, making exceeding iterations in the epilog stage executed. Furthermore, as every iteration depends on a predicate register for execution, stage count, i.e. number of concurrent iterations in a pipeline, is limited to the register size and hence severely affects schedule quality in bigger loops. LASER (Balasubramanian *et al.* (2018)), though did not explicitly address the problem, proposed an approach to issuing predicated operations by fusing instructions. Analogous to branches in imbalanced loops, LASER can be used to fuse live out nodes with NOOP instructions and take control node's result for predication. However, similar to (Tirumalai *et al.* (1990)), proposed LASER architecture stores every instruction's predication and still relies on a predicate buffer which has a limited size, thus also limiting schedule quality. As the predicate buffer also takes evaluations from loop conditions, LASER may only work on LESS THAN or GREATER THAN conditions. (Aiken *et al.* (1995)) showed an alternative approach to software pipelining by dynamically schedule nodes in different branches at run-time depending on the predicate results. Hardware models running under this scheduling heuristic are immune to the loop branching and exit problems. However, this is not a modulo scheduling technique and still inherits its own challenges. COMSAT, on the other hand, does not depend on runtime predication nor require a register file, can work on any condition of a loop whilst not being limited to register size, enabling many more loop accelerations on CGRA and improving scalability of modulo scheduling technique.

Many modulo scheduling and mapping algorithms (Dave *et al.* (2018a), Balasubramanian and Shrivastava (2020), Hamzeh *et al.* (2012), Dave *et al.* (2018b), Kou *et al.* (2020)) suffer from the loop exiting problems since they work under an assumption that trip count is given at compile time and loops do not contain exit statements. Therefore, when executing loops with unknown trip counts, CGRAs rely on loop conditions to exit and not the conditions for exit statements, leading to false terminations. Even for loops with known trip counts, CGRAs would require a lower bound on their schedules, limiting the usability of accelerators. COMSAT proposes an ideal solution to this problem by providing simple modifications to the generated schedules whilst minimally affects scheduling quality. With COMSAT techniques applied, CGRAs are able to accelerate on many more loops that previously required manual adjustments for correctness.

Chapter 3

BACKGROUND AND DEFINITION

At the first stage of compilation, targeted loop is analyzed to search for data dependencies and realize data flow, which are represented in a data-flow graph (DFG). Operations that hold or store the loop outcomes are called live out nodes and operations that evaluate loop conditions and emit exit signals are called (loop) control nodes. Similarly, exit statements accompanied by their conditions are also recognized as control nodes since they can terminate loops. All operations in the DFG establish an entire iteration and an iteration schedule holds all these nodes with their execution cycles respective to the start of an iteration (first scheduled operation). This iteration schedule is duplicated and placed II cycles apart to realize a modulo schedule.

Modulo scheduling is a scheme to scheduling operations of an iteration such that when the iteration is repeated every II cycles, no data dependency are violated and no resource usage are conflicted (Rau (1996)). At a glance, once an operation is mapped to a PE, it will be repeatedly executed in that same PE every II cycles until the loop terminates. Modulo schedules consist of 3 stages, prolog, kernel and epilog. Kernel is the repetitive stage of a schedule where every node of an iteration is executed. Prolog is analogous to the filling-pipeline stage starting from the root nodes of the DFG down to where kernel can be realized. Epilog is analogous to the draining stage of a pipeline where no new iterations are initiated and nodes are repeated until the final iteration is reached. Due to resource constraints from hardware models and recurrent constraints from data dependencies, minimum II (MII) is determined and acts as an upper bound on schedule quality. MII is calculated by $Max\{ResMII, RecMII\}$, where ResMII is MII due to resource constraint and RecMII is MII due to recurrent constraint. (Balasubramanian and Shrivastava (2020)) shows how to estimate these 2 quantities. A loop stage is a set of nodes that reconstruct the same iteration in a kernel trip count. In II cycles of kernel, all stages of an iteration schedule are executed. Depending on the kernel start cycle, the maximum number of stages (stage count) is as follow:

$$SC = \left\lceil \frac{schedule\ cycles}{II} \right\rceil + 1$$

Loops are categorized into 2 types: pre-conditioned and post-conditioned loops. Pre-conditioned loop is the same as while loop where its conditions has to satisfy before going to the first iteration. Do-while loop is an example of post-conditioned loop where its conditions apply for the succeeding iteration. In a perspective of modulo scheduling, loop control node's outcome applies to its belonging iteration in pre-conditioned loops and succeeding iteration for post-conditioned loops. Techniques proposed in this paper assume that target loops are post-conditioned. This assumption is insignificant because transformation from pre-conditioned to post-conditioned loops can be done by evaluating their conditions before starting acceleration on the loop body.

Exit signals in post-conditioned loops imply that the iteration triggering such signal should be the final one. Since CGRAs usually take some cycles before switching to epilog, they accidentally initiates further iterations. With trip count being known before acceleration, CGRAs would switch to epilog a certain number of iterations before even realizing the exit signal. However, very few loops can ignore evaluating their conditions at run-time. Even when trip count is given, manual adjustments are made to account for the extra iterations that CGRAs take for switching to epilog. Loops containing exit statements experience the same problem since trip count is never given and the exceeded iterations are also reflected in epilog. Moreover, control nodes scheduled in prolog stage cannot terminate the loop since accelerators have to execute through kernel and epilog stages, thus modulo schedules require a lower bound on the number of iterations, regardless of trip count status. These factors extremely limit the number of loops that can be accelerated and diminish automation of compilers since human intervention is involved.

Chapter 4

COMSAT

In the absence of trip count and with the assumption that live out nodes are scheduled after control nodes, the exceeded live out nodes are always reflected in epilog schedule. Thus it is only necessary to revise epilog stage to account for the exceeding iterations. The first part of COMSAT proposes an algorithm to find and delete nodes from the exceeding iterations, solving the unknown trip count and exit statements problems. In the second part, COMSAT shows how early exit loops should be executed and proposes an algorithm to generate different schedules for prolog stage. When scheduling with above assumption, circular dependencies in a DFG may occur and no valid schedule would be found in such case, COMSAT shows a simple solution to alleviating this problem in the following sections. Lastly, we show 2 examples of how the scheduling assumption is implemented in 2 popular modulo scheduling algorithms, IMS (Rau (1996)) and CRIMSON (Balasubramanian and Shrivastava (2020)). The COMSAT technique suite is outlined as follow:

1. Epilog adjustment shows how loop stages are constructed to find the number of exceeding iterations and from that, Algorithm 1 shows how exceeding iterations are excluded from the epilog stage.

2. Prolog schedule generation solves the early exit problem by showing what nodes should be collected and how schedule versions should look like (Algorithm 2).

3. Circular dependencies disable scheduling algorithms to find a valid schedule. This section shows a simple solution to breaking circular graphs in DFGs.

Algorithm 1: Revise_Epilog_Schedule(*epilog_schedule*, *loop_stage_map*, *control_op*)

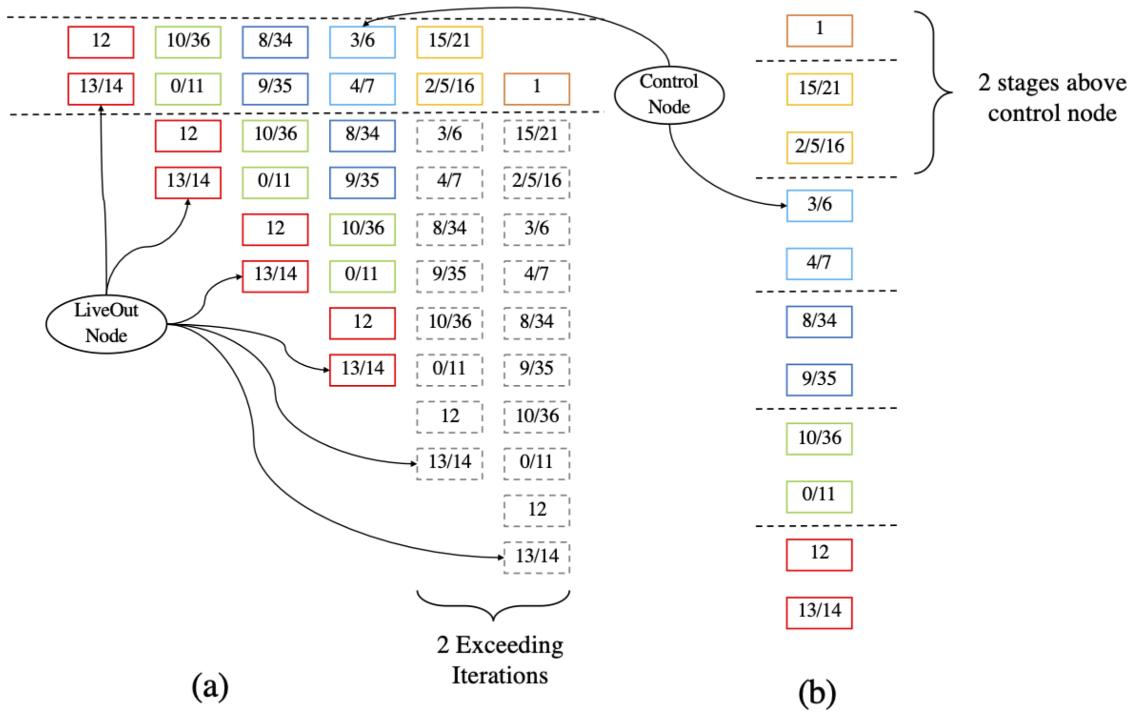
```
1 $N\_exceeding\_iteration \leftarrow$
    $get\_exceeding\_iteration(control\_op, loop\_stage\_map)$;
2 for $N = 1 \rightarrow N\_exceeding\_iteration$ do
3     for $cycle = |epilog\_schedule| \rightarrow 0$ do
4         $scheduled\_ops \leftarrow epilog\_schedule[cycle]$;
5         for each $Operation \in scheduled\_ops$ do
6             if is_at_highest_stage(Operation, schedule_ops, loop_stage_map)
               then
7                 $remove\_operation(epilog\_schedule, Operation)$;
```

4. Constraint implementation will show how scheduling algorithms can be modified to adhere to the scheduling assumption (Algorithm 4.4 and 4.4).

4.1 Epilog Adjustment

The epilog schedule adjustment works under a scheduling assumption that control nodes are scheduled before live out nodes to ensure that live out nodes in the exceeding iterations are always in epilog stage. This is because each control node only appears once in a kernel trip count, and as it issues an exit signal, the next redundant control node will be scheduled in epilog stage. Since live out nodes are scheduled after the redundant control node, they are also redundant and guaranteed to be in epilog. Note that this also applies to exit statements, as when their conditions satisfy, CGRA would switch to epilog where exceeded live out nodes are removed from the schedule.

Figure 4.1(b) illustrates loop stages separated by dashed lines, constructed from



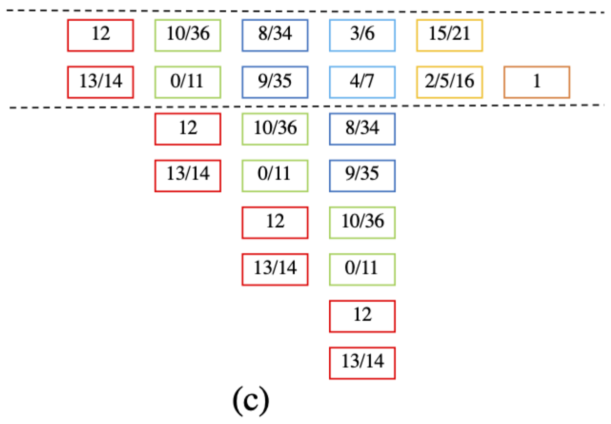
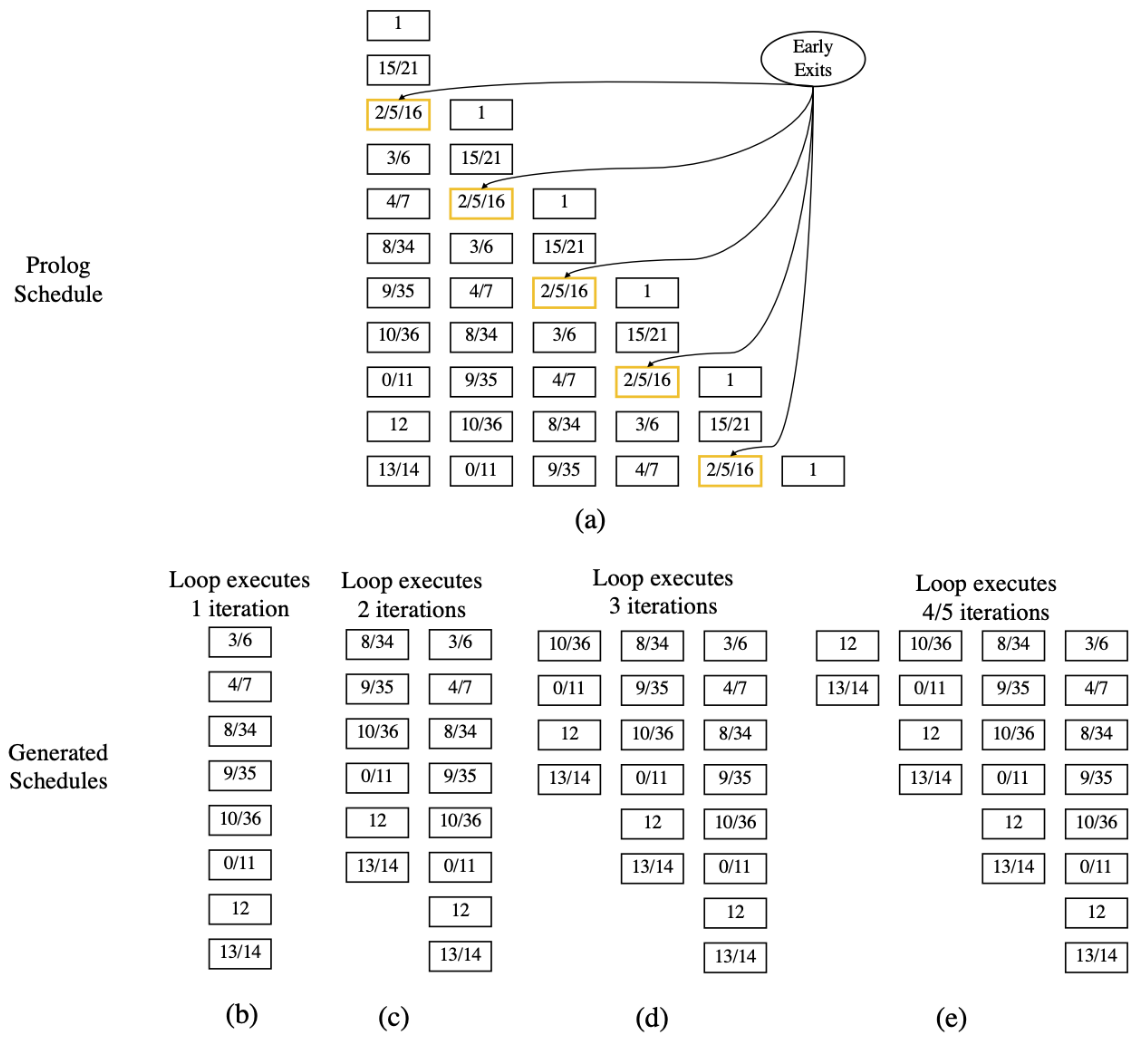


Figure 4.1: (a) Kernel and Epilog Schedules From GEMM With Control Node Moved Down 1 Stage. (b) Loop Stages Constructed From (a). (c) Adjusted Epilog Schedule.

the kernel schedule in (a). Note that this schedule in Figure 4.1(a) is taken partially from Figure 1.2(c) with the control node hypothetically moved to either node 3 or 6 (indicated by control node's box), we can see that control node is moved down 1 stage, leaving 2 stages above itself. Accordingly, there are now 2 iterations exceeded in the epilog schedule (Figure 4.1(a)). Thus, the number of stages scheduled above control nodes is the number of exceeding iterations as CGRA switches to epilog. The redundant nodes, including live out nodes, in the exceeding iterations are then eliminated in the epilog schedule, as shown in (c). Algorithm 1 shows how these nodes are collected and removed. The 3-level nested loop looks at every cycle (row) in epilog schedule from bottom up (line 3) and removes nodes that are scheduled at the highest stage (as done in line 6 and 7), this process repeats for all exceeding iterations (line 2). Removing nodes at the highest stage excludes redundant operations because nodes in the top stage always belong to the latest iteration at every cycle. Although Figure 4.1(c) still shows some redundant operations in kernel, these are not live out nodes and would not affect loop correctness.

4.2 Prolog Schedule Generation

CGRAs following modulo schedules, which are composed of several iterations, limit themselves to loops with trip counts higher than their minimums since they cannot exit after finishing prolog. To enable execution on early exit loops, CGRAs should remain in prolog stage upon exit signals, and not initiate/execute further iterations. COMSAT solves this problem by generating schedules modified from prolog's, and since there can be more than 1 exiting iteration, COMSAT has to generate as many schedules as the number of control nodes in the prolog stage to accommodate for every exit. Each generated schedule corresponds to an iteration where exit signal is issued, and it should complete previous iterations whilst excluding succeeding ones.



Prolog Schedule
Early Exits
1
15/21
2/5/16
3/6
4/7
8/34
9/35
10/36
0/11
12
13/14
(a)
Generated Schedules
Loop executes 1 iteration
Loop executes 2 iterations
Loop executes 3 iterations
Loop executes 4/5 iterations
(b)
(c)
(d)
(e)

Figure 4.2: (a) Prolog Schedule From Fig. 1.2(c). (b)(c)(d)(e) Respective Prolog Versions Starting From Control Node's Cycle.

Since there are multiple schedules added, accelerators need additional information on which to branch to. Thus, instruction format for control nodes is modified to include this information and architectures are slightly tweaked to support branching in prolog stage when loop conditions fail and CGRAs need to branch to the corresponding schedule.

Figure 4.2(a) takes the prolog schedule from Figure 1.2(c) and shows how the generated schedules look like. Each generated schedule in (b), (c), (d) and (e) illustrates

Algorithm 2: Generate_Prolog_Schedule(*prolog_schedule*, *iteration_schedule*, *control_op*)

```
1  $N\_versions \leftarrow count\_operation(prolog\_schedule, control\_op)$;
2  $start\_cycle \leftarrow get\_cycle(iteration\_schedule, control\_op)$;
3  for $N = 0 \rightarrow N\_versions - 1$ do
4      $start\_cycle \leftarrow start\_cycle + (N * II)$;
5      $adding\_schedule \leftarrow \varnothing$;
6      for $iter = 0 \rightarrow N$ do
7          $start\_cycle \leftarrow start\_cycle - (iter * II)$;
8          $appending\_op \leftarrow get\_subset(iteration\_schedule, start\_cycle)$;
9          $append\_set(adding\_schedule, appending\_op, 0)$;
10     $append\_set(prolog\_schedule, adding\_schedule, final\_prolog\_cycle)$;
```

the modified schedules that would be branched to if loop exits after 1, 2, 3, 4 or 5 iterations, respectively. Note that each schedule includes nodes in the previous iterations and excludes nodes in the succeeding ones. Algorithm 2 shows how different schedule versions are generated given prolog and iteration schedules. At every version being generated from line 3, *start_cycle* is moved down and *adding_schedule* set is initiated. Operations are then added to this schedule set for every iteration (line 6). The *get_subset* function in line 8 extracts partial iteration schedule starting from the *start_cycle* to the end. This partial schedule is then appended to the *adding_schedule* at the cycle 0 (line 9). After all intended iterations are added, the *adding_schedule* is appended to the end of prolog schedule in line 10 and *final_prolog_cycle* is updated. As all possible schedules are generated, addresses pointing to each version are calculated by their relative offsets (not part of the algorithm), which are then updated to

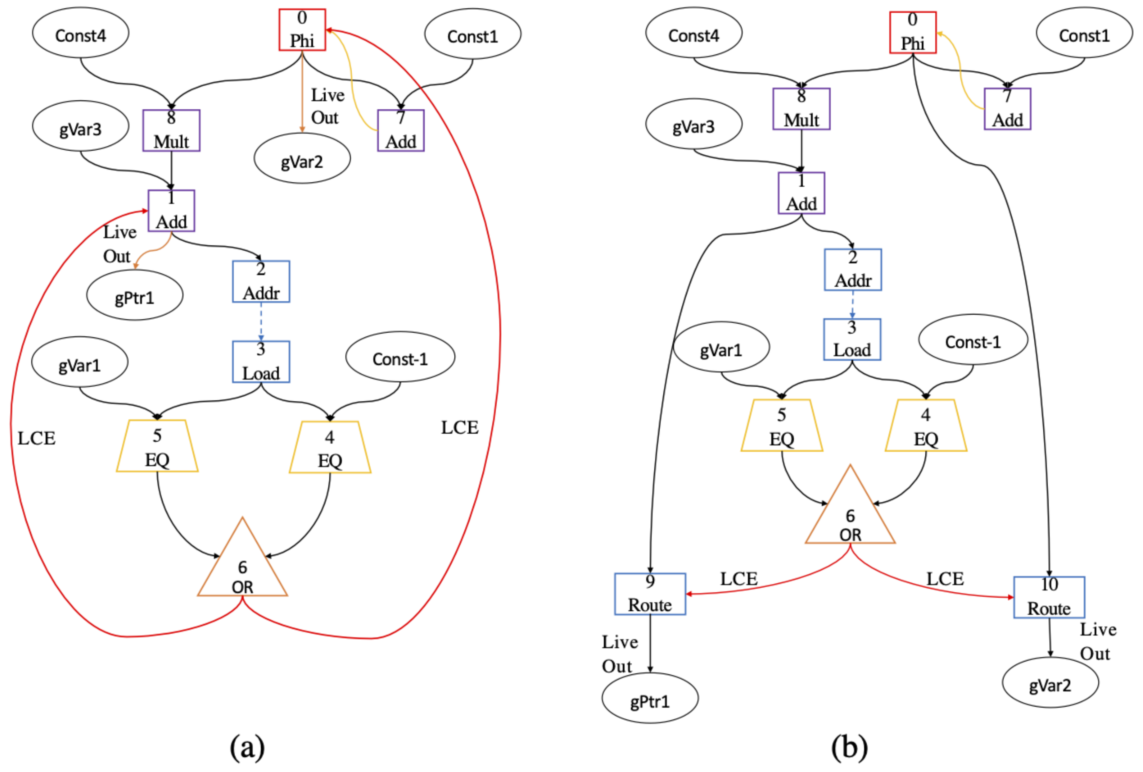


Figure 4.3: (a) DFG Containing Circular Dependency Created by Loop Control Edge (LCE). (b) Route Nodes are Added To De-Circular Dependency.

the control node's instructions.

4.3 Circular Dependency

As algorithms schedule operations and try to meet the assumption, circular dependency would happen when there is a data flow from control nodes to live out nodes. This would fail algorithms to find a valid schedule regardless of II, yet the solution to this problem is as simple as adding a route node to every live out that causes circular dependency. This route node then becomes a live out itself, replaces the role of its route source. Consider a different example leading to this problem with a DFG shown in Figure 4.3(a). The 2 live out nodes (0 and 1) have 2 data flows both going to the control node 6, and create 2 circular graphs (note that the edge from node 7 to 0 does not pose circular dependency because it is an inter-iteration dependency). To break these circular graphs, 2 route nodes, 1 for each live out, are added and replace their sources' roles so that control node 6 now points to these new nodes, shown in Figure 4.3(b) and the resultant DFG shows no more circular dependency. Although adding route nodes might affect mapping qualities, they only have effects under limited resources as there is only 1 dependency edge for each route node.

4.4 Constraint Implementation

Although epilog's and prolog's techniques work as external modules coupled to scheduling algorithms, the algorithms themselves have to be modified accordingly to meet the scheduling requirement, yet the changes are simple. Algorithm 3 shows a modified iterative modulo schedule algorithm from (Rau (1996)). Compare to the original IMS algorithm, variables $ControlNodeScheduled$ and $ControlNodeTimeSlot$ are added and updated in lines 21, 22, 23 to track schedule time of control nodes. Lines 12-17 adjusts minimum time of the scheduling live out node to after variable $ControlNodeTimeSlot$ and if control nodes are not yet scheduled, the live outs are pushed back for later. Algorithm 4 shows a modified routine used by CRIMSON (Balasubramanian and Shrivastava (2020)) to find a scheduling time slot. Lines 4-11 are added to adjust ASAP times for live outs and ALAP times for control nodes. This actively restraints the available time slots of control nodes to strictly less than available time slots of the earliest scheduled live out.

Algorithm 3: Iterative_Schedule(II, Budget)

```
1  $Operation, Estart, MinTime, MaxTime, TimeSlot, ControlNodeTimeSlot$
     : integer
2  $NeverSchedule, UnscheduledOperations$ : list
3  $ControlNodeScheduled \leftarrow false$;
4  for each Operation do
5      $NeverSchedule[Operation] \leftarrow true$;
6      $UnscheduledOperations \leftarrow Operation$;
7  $Budget \leftarrow Budget - 1$;
8  while $UnscheduledOperations \neq \varnothing$ & $Budget \geq 1$ do
9      $Operation \leftarrow HighestPriorityOperation()$;
10     $Estart \leftarrow CalculateEarlyStart()$;
11     $MinTime \leftarrow Estart$;
12     if $Operation == liveout$ then
13         if $ControlNodeScheduled == true$ then
14             if $MinTime \leq ControlNodeTimeSlot$ then
15                 $MinTime = ControlNodeTimeSlot + 1$;
16         else
17             continue;
18     $MaxTime \leftarrow MinTime + II - 1$;
19     $TimeSlot \leftarrow FindTimeSlot(Operation, MinTime, MaxTime)$;
20     $Schedule(Operation, TimeSlot)$;
21     if $Operation == control$ then
22         $ControlNodeScheduled \leftarrow true$;
23         $ControlNodeTimeSlot \leftarrow TimeSlot$;
24     $Budget \leftarrow Budget - 1$;
25 return ($UnscheduledOperations == empty$);
```

Algorithm 4: Find_Random_ModuloTime(Op, $CGRA$)

```
1  $Op\_ASAP \leftarrow get\_RC\_ASAP(Op)$;
2  $Op\_ALAP \leftarrow get\_RC\_ALAP(Op)$;
3  $sched\_slot \leftarrow \varnothing$;
4  $control\_Op \leftarrow get\_control\_node()$;
5  $liveout\_Op \leftarrow get\_earliest\_liveout\_node()$;
6  if $Op == liveout\_Op$ & $Scheduled[control\_Op]$ then
7      if $Prev\_Sched\_Time[control\_Op] > Op\_ASAP$ then
8          $Op\_ASAP \leftarrow Prev\_Sched\_Time[control\_Op] + 1$;
9  if $Op == control\_Op$ & $Scheduled[liveout\_Op]$ then
10     if $Prev\_Sched\_Time[liveout\_Op] > Op\_ALAP$ then
11         $Op\_ALAP \leftarrow Prev\_Sched\_Time[liveout\_Op] + 1$;
12 $time\_slots \leftarrow get\_all\_timeslots(Op\_ASAP, Op\_ALAP)$;
13 $Randomize(time\_slots)$;
14 while $sched\_slot == \varnothing$ & $|time\_slots| > 0$ do
15     $current\_time \leftarrow time\_slots[0]$;
16     if $ResourceConflict(Op, current\_time, CGRA)$ then
17         $time\_slots \leftarrow Subtract(current\_time, time\_slots)$;
18         continue;
19     else
20         $sched\_time \leftarrow current\_time$;
21 if $sched\_slot == \varnothing$ then
22     if $!Scheduled[Op] \parallel op\_ASAP > Prev\_Sched\_Time[Op]$ then
23         $sched\_slot \leftarrow op\_ASAP$;
24     else
25         $sched\_slot \leftarrow Prev\_Sched\_Time[Op] + 1$;
26 return $sched\_slot$;
```

Chapter 5

EXPERIMENT

Techniques proposed in this paper are implemented in CGRA Compilation Framework (CCFV2.0) (Dave and Shrivastava (2018)) and as a modified version of an existing scheduling algorithm, CRIMSON (Balasubramanian and Shrivastava (2020)). CCF compiles programs targeting ARMv7a single-core, single-threaded processor along with the chosen kernel loops at O3 optimization level. Loops are compiled by first generating DFGs using LLVM4.0 (Lattner and Adve (2004)) then passed onto scheduling and mapping algorithms, namely CRIMSON (Balasubramanian and Shrivastava (2020)) and PathSeeker (Balasubramanian and Shrivastava (2022)). An instruction generator then takes the mapping output to generate binary files containing all instructions. The binary files are fed into Gem5 (Lowe-Power *et al.* (2020)) embedded with CGRA ADRES model (Bouwens *et al.* (2008b)) for simulation and results. Experiments are run on 2 widely used benchmark suites, MiBench (Guthaus *et al.* (2001)) and Rodinia (Che *et al.* (2009)), with applications taken from automotive, telecommunication, machine learning, network, security and physics/biology simulation fields. The CGRA model is implemented with either 8x8 or 4x4 PE configuration depending loop size, and a register file of 16. Note that a predicate register file is excluded from the hardware model because it is redundant with COMSAT. Since simulation time depends on the host OS and may vary on workloads, it is not a good performance metric, so we use number of cycles emulated by Gem5 instead.

Table 5.1: Benchmark Profile

Suite	Benchmark	Application	COMSAT	HP	LASER	Dynamic
MiBench	Adpcm	Telecom	2	2	2	2
MiBench	Basicmath	Automotive	1	1	1	1
MiBench	Bitcount	Automotive	1	0	0	1
MiBench	Dijkstra	Network	2	1	1	2
MiBench	FFT	Telecom	3	1	1	1
MiBench	GSM	Telecom	6	6	6	0
MiBench	Patricia	Network	6	2	2	6
MiBench	Rijndael	Security	10	4	4	9
MiBench	SHA	Security	3	3	3	2
MiBench	StringSearch	Office	3	2	2	2
MiBench	Susan	Automotive	12	11	11	12
Rodinia	Backprop	ML	5	5	5	4
Rodinia	BFS	Network	3	1	1	2
Rodinia	B+Tree	Network	18	6	6	16
Rodinia	Hotspot3D	Simulation	3	3	3	3
Rodinia	Kmeans	ML	7	2	2	7
Rodinia	LUD	Mathematic	2	0	0	2
Rodinia	MRI-gridding	Simulation	4	4	4	4
Rodinia	Myocyte	Simulation	9	9	9	0
Rodinia	NN	Network	2	0	0	2
Rodinia	NW	Simulation	9	5	5	3
Rodinia	Srad	Imaging	5	1	1	5

5.1 COMSAT-Enabled Scalability

Table 5.1 profiles applications from MiBench and Rodinia benchmark suites with a number of kernel loops, that take a good portion of simulated cycles, accelerated by COMSAT techniques and a proportion of such loops that can be accelerated by HP's (Tirumalai *et al.* (1990)) and LASER (Balasubramanian *et al.* (2018)) techniques, as well as the number of unknown-trip-count loops. The table inferred that out of 116 kernel loop accelerations enabled by COMSAT, only 70 loops can be safely applied by HP's and LASER without exceeding intended iterations. In comparisons to existing scheduling/mapping techniques, that require human intervention for executions on the 85 dynamic-trip-count loops, COMSAT, being an external module coupled to the algorithms to modify the schedules, has enabled accelerations on these loops without the required adjustments. Note that above table only profiles loops based on their conditions, which regards to the first limitation of HP's and LASER techniques, whereas the other limitation on predicate register size depends on hardware implementation and schedule quality, and cannot be explicitly illustrated.

5.2 Benchmark Speedup

Figure 5.1 illustrates speedups of kernel computations and entire benchmarks, in log scale, compared to ARMv7a single-threaded processor. The kernel loops are first compiled and mapped using PathSeeker, then applied to COMSAT for dynamic trip count support and finally executed on CGRA, embedded in Gem5, coupled with the ARM processor. With exceptions of SHA and LUD benchmarks, total speedup trends with kernel speedup, showing the dominance of accelerating kernel computations with respect to the entire program. Myocyte and Needleman-Wunsch (NW) benchmarks did not show overall improvements because large amount of computa-

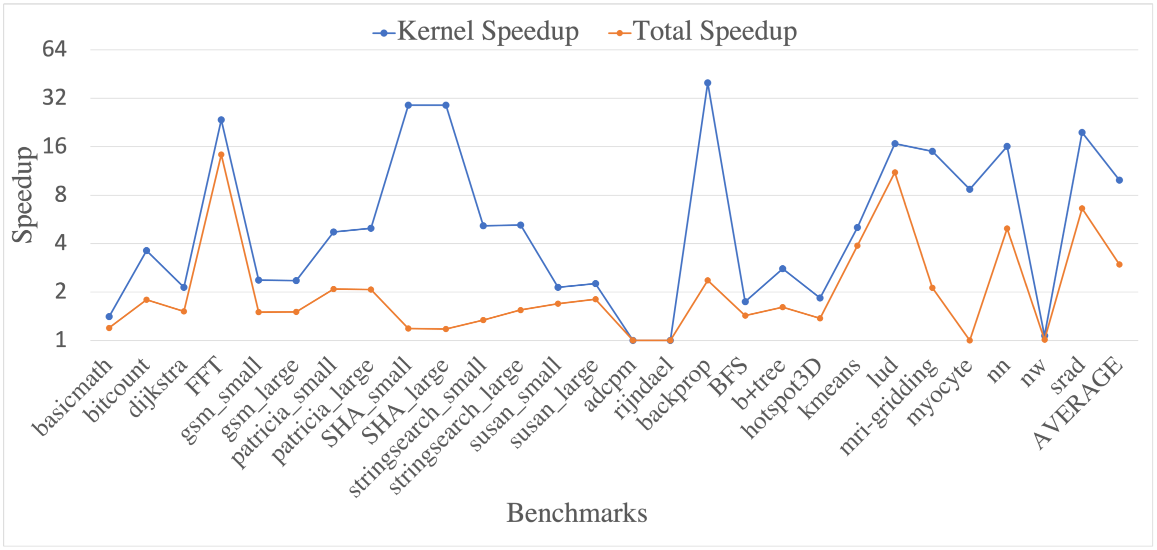


Figure 5.1: Benchmark and Kernel Speedup Achieved by COMSAT.

tions are pre-determined (static trip count) and the kernels were written to support acceleration using multi-threading technique instead of loops, as shown in Table 5.1 where Myocyte and NW only have 0 and 3 dynamic loops, respectively. Figure 5.1 shows that COMSAT has enabled average speedup, across MiBench and Rodinia suites, of 3x with average kernel speedup of 10x. Without our techniques to support dynamic trip count, accelerations would have only been possible for GSM, Myocyte and Needleman-Wunsch benchmarks, resulted in 1.25x and 3.63x for total and kernel speedups, respectively. Adpcm and Rijndael gave 0x acceleration because they failed to give mapping solutions with PathSeeker under constrained time.

To better illustrate CGRA performance, we compare our results with another paper (Bu *et al.* (2018)) that also aims at accelerating MiBench suite. This paper proposes 2 techniques, loop parallelism and procedure level speculation, for acceleration on both kernel loops and serial programs (control-intensive regions). Since we mainly focus on kernel loop acceleration, only results for loop parallelism technique are analyzed. As illustrated in Figure 6 in (Bu *et al.* (2018)), experiments only show 4 benchmarks in common, bitcount, dijkstra, patricia and susan, that can be put in

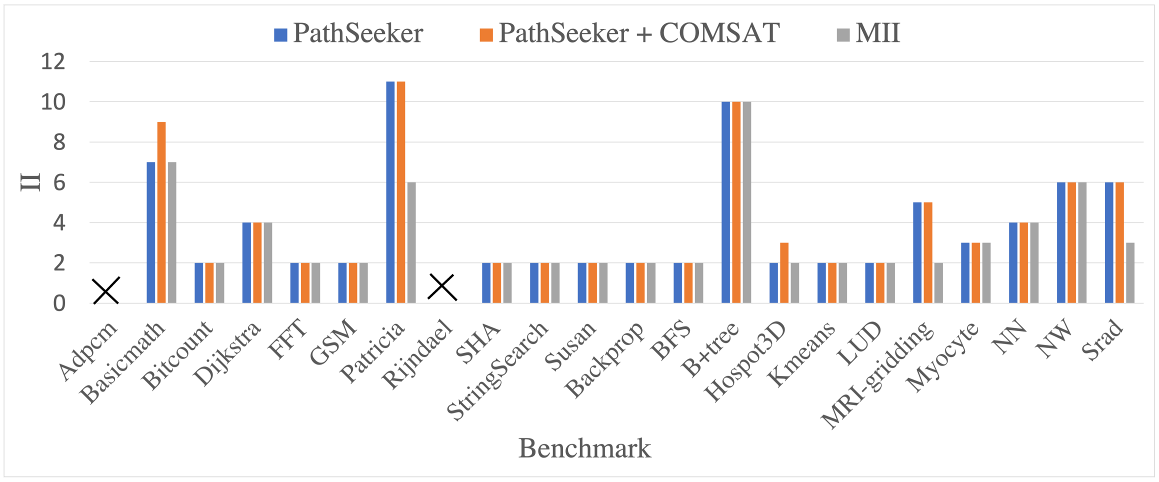


Figure 5.2: Effects of II When Imposing Scheduling Requirement Needed For COMSAT.

comparisons. We can observe that COMSAT has achieved upper bounds in accelerations for 3 out of 4 benchmarks, except Dijkstra. Especially in Susan and Patricia cases, COMSAT's total benchmark speedups are comparable to their kernel speedups. Dijkstra gave an interesting result as COMSAT only showed kernel acceleration of 2x whilst Bu *et al.* (2018) achieved 10x. This is because most loops in the benchmark involves function calls that limits CGRA execution but Bu *et al.* (2018) techniques can take advantage of accelerating those loops.

5.3 Mapping Quality

As COMSAT introduces a scheduling constraint between control and live out nodes, we then analyze how this requirement affects mapping quality. DFGs generated by LLVM are scheduled using CRIMSON (Balasubramanian and Shrivastava (2020)) and mapped by PathSeeker (Balasubramanian and Shrivastava (2022)). When live out nodes are scheduled before control nodes, they have to be routed down the iteration and routing nodes are added, increasing complexity for PathSeeker. However, Figure 5.2 shows that this is not the common case as COMSAT only affected II in 2 out of 20 benchmarks, namely Basicmath and Hotspot3D. The event was occasional for 2 reasons, (1) it only occurs in small loops where computation branch is shorter than loop control branch in a DFG and (2) routing nodes affect II when mapping on smaller CGRAs with limited resources. Note that Adpcm and Rijndael benchmarks failed to find solutions even without COMSAT because kernel loops were too large whilst mapping time was constrained. In other benchmarks that PathSeeker generated valid mappings, COMSAT gave the same II even when PathSeeker could not achieve the minimum, this implies the scheduling constraint does not affect mapping qualities in these loops.

Chapter 6

CONCLUSION

Accelerators operating on modulo schedules have been experiencing loop exit problems and without proper hardware model modifications, loop executions often reflects an incorrect number of iterations and affects its final result, some cases lead to memory issues. This article has shown solutions to such problems, bringing much more potential acceleration on loop whilst simplifying hardware model requirements. Experiments shows that about 75% of loops do not have static trip count, acceleration on these loops would have been much more difficult and even impossible to execute without manual adjustments. Early exit loop problem has not also been mentioned in related topics and many assume that accelerated loops execute much further than their minimum trip counts. While this is true for most of the benchmarks, the limitation prevents accelerators from being more of a general purpose and restricts themselves from many other applications.

REFERENCES

Aiken, A., A. Nicolau and S. Novack, “Resource-constrained software pipelining”, IEEE Transactions on Parallel and Distributed Systems **6**, 12, 1248–1270 (1995).

Balasubramanian, M., S. Dave, A. Shrivastava and R. Jeyapaul, “Laser: A hardware/software approach to accelerate complicated loops on cgras”, in “2018 Design, Automation Test in Europe Conference Exhibition (DATE)”, pp. 1069–1074 (2018).

Balasubramanian, M. and A. Shrivastava, “Crimson: Compute-intensive loop acceleration by randomized iterative modulo scheduling and optimized mapping on cgras”, IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS **39**, 11, 3300–3310 (2020).

Balasubramanian, M. and A. Shrivastava, “Pathseeker: A fast mapping algorithm for cgras”, in “2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)”, pp. 268–273 (2022).

Bouwens, F., M. Berekovic, A. Kanstein and G. Gaydadjiev, *Architectural Exploration of the ADRES Coarse-Grained Reconfigurable Array* (Springer Berlin Heidelberg, 2008a).

Bouwens, F., M. Berekovic, B. D. Sutter and G. Gaydadjiev, *Architecture Enhancements for the ADRES Coarse-Grained Reconfigurable Array* (Springer Berlin Heidelberg, 2008b).

Bu, D., Y. Wang, L. Li, Z. Liu, W. Yu and M. Musariri, “Exploring parallelism in mibench with loop and procedure level speculation”, in “2018 IEEE Intl Conf on Parallel Distributed Processing with Applications, Ubiquitous Computing Communications, Big Data Cloud Computing, Social Computing Networking, Sustainable Computing Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)”, pp. 141–146 (2018).

Che, S., M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing”, in “In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)”, pp. 44–54 (2009).

Dave, S., M. Balasubramanian and A. Shrivastava, “Ramp: Resource-aware mapping for cgras”, in “Proceedings of the 55th Annual Design Automation Conference”, DAC ’18 (Association for Computing Machinery, 2018a).

Dave, S., M. Balasubramanian and A. Shrivastava, “Ureca: Unified register file for cgras”, in “2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)”, pp. 1081–1086 (2018b).

Dave, S. and A. Shrivastava, *CCF: A CGRA Compilation Framework*, Arizona State University, `https://github.com/MPSLab-ASU/CCF-20.04` (2018).

Guthaus, M., J. Ringenberg, D. Ernst, T. Austin, T. Mudge and R. Brown, “Mibench: A free, commercially representative embedded benchmark suite”, in “Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)”, pp. 3–14 (2001).

Hamzeh, M., A. Shrivastava and S. Vrudhula, “Epimap: Using epimorphism to map applications on cgras”, in “DAC Design Automation Conference 2012”, pp. 1280–1287 (2012).

Hamzeh, M., A. Shrivastava and S. Vrudhula, “Branch-aware loop mapping on cgras”, in “2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)”, pp. 1–6 (2014).

Hartenstein, R., “A decade of reconfigurable computing: a visionary retrospective”, Proceedings of the conference on Design, automation and test in Europe pp. 642–649 (2001).

Kou, M., J. Gu, S. Wei, H. Yao and S. Yin, “Taem: Fast transfer-aware effective loop mapping for heterogeneous resources on cgra”, in “2020 57th ACM/IEEE Design Automation Conference (DAC)”, pp. 1–6 (2020).

Lattner, C. and V. Adve, “Llvm: a compilation framework for lifelong program analysis and transformation”, in “International Symposium on Code Generation and Optimization, 2004. CGO 2004.”, pp. 75–86 (2004).

Lavery, D. M., *Modulo Scheduling for Control-Intensive General-Purpose Programs*, Ph.D. thesis (1997).

Lowe-Power, J., A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mck, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon and . F. Zulian, “The gem5 simulator: Version 20.0+”, URL `https://arxiv.org/abs/2007.03152` (2020).

Mittal, S. and J. S. Vetter, “A survey of methods for analyzing and improving gpu energy efficiency”, ACM Comput. Surv. **47**, 2 (2014).

Miyamori, T. and K. Olukotun, “Remarc: Reconfigurable multimedia array coprocessor”, in “IEICE Transactions on Information and Systems E82-D”, pp. 389–397 (1998).

Nurvitadhi, E., J. Sim, D. Sheffield, A. Mishra, S. Krishnan and D. Marr, “Accelerating recurrent neural networks in analytics servers: Comparison of fpga, cpu, gpu, and asic”, in “2016 26th International Conference on Field Programmable Logic and Applications (FPL)”, pp. 1–4 (2016).

Rau, B. R., “Iterative modulo scheduling”, Compiler and Architecture Research , 1 (1996).

Singh, H., M. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh and M. Filho, “Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications”, IEEE Transactions on Computers pp. 465–481 (2000).

Tirumalai, P., M. Lee and M. Schlansker, “Parallelization of loops with exits on pipelined architectures”, in “Proceedings of the 1990 ACM/IEEE Conference on Supercomputing”, Supercomputing ’90, p. 200212 (IEEE Computer Society Press, 1990).