UNIVERSITY OF CALIFORNIA

Los Angeles

Performance Debugging Frameworks for FPGA High-Level Synthesis

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Young-kyu Choi

2019

ABSTRACT OF THE DISSERTATION

Performance Debugging Frameworks for
FPGA High-Level Synthesis

by

Young-kyu Choi
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2019
Professor Jingsheng Jason Cong, Chair

Using high-level synthesis (HLS) tools for field-programmable gate array (FPGA) design is becoming an increasingly popular choice because HLS tools can generate a high-quality design in a short development time. However, current HLS tools still cannot adequately support users in understanding and fixing the performance issues of the current design. That is, current HLS tools lack in performance debugging capability. Previous work on performance debugging automates the process of inserting hardware monitors in low-level register-transfer level (RTL) languages which limits the comprehensibility of the obtained result. Instead, our HLS-based flows offer analysis on a function or loop level and provide more intuitive feedback that can be used to pinpoint the performance bottleneck of a design. In this dissertation, we present a collection of HLS-based debugging frameworks for various purposes and characteristics of the design. First, we address the problem in the HLS synthesis step, where an inaccurate cycle estimation is provided if the program has input-dependent behavior. We propose a new performance estimator that automatically instruments code that models the hardware execution behavior and interprets the information from the HLS software simulation. However, the performance estimation result of this flow may not be accurate for a type of designs that cannot be simulated correctly by existing HLS software simulators. To handle such cases, we propose a new software simulator that provides cycle-accurate result based on the HLS scheduling information. If the input dataset is not available for software simula-

tion or high-level models do not exist for all components of the FPGA design, we also present an on-board monitoring flow for automated cycle extraction and stall analysis. Finally, we address the needs of HLS programmers to automatically find the best set of directives for FPGA designs. We propose a design space exploration (DSE) framework to optimize applications with variable loop bounds in Polybench benchmark. A quantitative comparison among the proposed frameworks is shown using the sparse matrix-vector multiplication benchmark.

The dissertation of Young-kyu Choi is approved.

William Hsu

Miryung Kim

Anthony John Nowatzki

Jingsheng Jason Cong, Committee Chair

University of California, Los Angeles

2019

For my wife, Hyejin, my two daughters, Jiho and Jimin, my parents, and my in-law parents

기나긴 박사과정 중에 항상 저를 사랑해준 아내와 딸들, 그리고 항상 격려하고 기도해주신 부모님과 장인 장모님께 감사 드립니다.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank, first and foremost, my advisor Professor Jason Cong for guiding me through the long journey of PhD degree. Although I have made many mistakes during the process, Professor Cong have always shown patience to lead me to the right path. Professor Cong took time to teach many things to me from topic selection, experimentation, paper writing, and presentation. It has been an unforgettable experience, and I will always try to remember his valuable lessons throughout my career.

I gratefully acknowledge the funding received from Intel and NSF (CCF-0903541, CCF-0926127, CCF-1436827, CCF-1723773, #20134321). I also thank the CDSC partners (especially Baidu, Fujitsu, Google, NEC, and VMWare) and the CAPA collaborators (from Intel and Cornell University) for the financial support and the feedback during the regular meetings. I also greatly appreciate the FPGA boards and the software donation from Xilinx.

I would like to thank the defense committee members, Professor William Hsu, Professor Miryung Kim, and Professor Tony Nowatzki, for their encouragement and the research guidance. I thank Professor Hsu for his advice on the 3D CT reconstruction acceleration work in my early PhD days. I thank Professor Kim for the insights on the concept of performance debugging in this dissertation. I thank Professor Nowatzki for his feedback on differentiating HLScope from other works.

I am grateful to Falcon Computing for providing me with a core infrastructure that I used throughout the works in my dissertation. Without the support of Falcon members, I would not have been able to implement my frameworks. I am especially grateful to Dr. Peng Zhang (formerly a post-doctoral researcher in UCLA) for teaching me how to use the ROSE and Merlin Compiler APIs. Dr. Zhang was one of my role model as a researcher and an engineer, and I was lucky to have the opportunity to learn from his eagerness and efficiency in doing research. I also thank Dr. Peichen Pan and Dr. Cody Hao Yu (also a former labmate) for the discussions and the lessons on the FPGA tools.

I would also like to thank the former and current lab members. I thank Yuze Chi for always helping me with tools, servers, and board installation. I thank Dr. Di Wu for teaching me the steps that is needed to go through the PhD course. I thank Jie Wang, Dr. Peng Wei, and Dr. Peipei Zhou for the frequent discussions and sharing tips for the research. I thank Dr. Peng Li for teaching me many functionalities of Vivado HLS and ROSE. I thank Dr. Yu-ting Chen, Dr. Zhe Chen, Professor Zhenman Fang, Nazanin Farahpour, Licheng Guo, Dr. Karthik Gururaj, Dr. Yuchen Hao, Dr. Hui Huang, Dr. Muhuan Huang, Farnoosh Javadi, Hassan Kianinejad, Jason Lau, Professor Jie Lei, Sen Li, Karl Marrett, Hyunseok Park, Weikang Qiao, Zhenyuan Ruan, Atefeh Sohrabizade, Dr. Bingjun Xiao, Xinfeng Xie, Mo Xu, and Bo Yuan for their feedback and discussion. The PhD life would have been much more difficult if it were not for your help and kindness.

I am grateful to Janice Wheeler for carefully editing all of my conference and journal submissions. I also thank Alexandra Luong for her administrative support.

I would also like to express my gratitude to my former advisors, Professor Wonyong Sung and Professor In Kyu Park, and my former mentor, Dr. Kisun You, for their lessons and guidance. They have given me the opportunity to take down this fascinating journey as a researcher.

VITA

2008 Master of Science in Electrical Engineering and Computer Science, Seoul National University, Korea.

2006 Bachelor of Science in Electrical Engineering, Seoul National University, Korea.

PUBLICATIONS

Young-kyu Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei, "In-depth analysis on microarchitectures of modern heterogeneous CPU-FPGA platforms," ACM Trans. Reconf. Tech. and Syst. (TRETS), vol. 12, no. 1, Feb. 2019.

Y. Chi, Young-kyu Choi, J. Cong, and J. Wang, "Rapid cycle-accurate simulator for high-level synthesis," ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA), Seaside, pp. 178-183, Feb. 2019.

Young-kyu Choi and J. Cong, "HLS-based optimization and design space exploration for applications with variable loop bounds," in Proc. IEEE/ACM Int. Conf. Computer Aided Design (ICCAD), San Diego, Nov. 2018.

Young-kyu Choi, P. Zhang, P. Li, and J. Cong, "HLScope+: Fast and accurate performance estimation for FPGA HLS," in Proc. IEEE/ACM Int. Conf. Computer Aided Design (ICCAD), Irvine, pp. 691-698, Nov. 2017.

Young-kyu Choi and J. Cong, "HLScope: High-level performance debugging for FPGA designs," in Proc. IEEE Int. Symp. Field-Programmable Custom Computing Machines (FCCM), Napa, pp. 125-128, May 2017.

Young-kyu Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei, "A quantitative analysis on microarchitectures of modern CPU-FPGA platforms," in Proc. Design Automation Conference (DAC), Austin, pp. 109-114, Jun. 2016.

Young-kyu Choi and J. Cong, "Acceleration of EM-based 3D CT reconstruction using FPGA," IEEE Trans. Biomedical Circuits and Systems (TBCAS), vol. 10, no. 3, pp. 754-767, Jun. 2016.

Young-kyu Choi, J. Cong, and D. Wu, "FPGA implementation of EM algorithm for 3D CT reconstruction," in Proc. IEEE Int. Symp. Field-Programmable Custom Computing Machines (FCCM), Boston, pp. 157-160, May 2014.

CHAPTER 1

Introduction

With multicore scaling coming to an end [47], customization is often considered to be a promising solution that delivers high performance with low power consumption [30]. The efficiency of customized architectures has been demonstrated in various applications including convolution [103], medical imaging [107], neural network [135], and speech processing [56]. However, it is not practical to provide a customized solution for every application with an application-specific integrated circuit (ASIC), because ASIC manufacturing incurs high non-recurring engineering (NRE) cost. As an alternative solution, field-programmable gate array (FPGA) has gained much interest as a choice of customized acceleration platform because it provides near-ASIC performance and energy efficiency while offering reconfigurability. FPGA allows engineers to easily update and improve their designs even after initial deployment. In datacenters, FPGA can be used to hardware accelerate different needs of customers with homogeneous platforms [102]. Due to its popularity, FPGA is now being offered in various cloud computing services such as Amazon Web Service [5] and Microsoft Azure [88].

However, the low-level programming environment of FPGA often created barrier for those who did not have previous hardware design experience. FPGA designs were typically written in register-transfer level (RTL) languages such as Verilog or VHDL, which require behavior of all signals to be specified for every clock cycle. Design, verification, and optimization of applications written in low-level RTL languages became complex tasks that required expertise of experienced engineers. This naturally lead to increased development cost and prolonged time-to-market.

In order to solve this problem, high-level synthesis (HLS) tools such as Xilinx Vivado HLS [31,

130] and Intel OpenCL HLS [62] were introduced.[1] The HLS tools allow programmers to design FPGA applications with high-level languages such as C and OpenCL and automatically transform them into low-level FPGA designs. This reduces the programmers' burden of determining the micro-architectural details of an FPGA design and increases their productivity. It also becomes easier to change the clock frequency or port an existing design into a new platform. Moreover, the verification is simplified because HLS tools allow C-level simulation of FPGA designs.

However, one of the obstacles that prevented more widespread use of HLS can be found in the lack of performance analysis tools. Even if an initial FPGA accelerator design does not meet the required performance, it is difficult to identify the cause of the problem. In contrast, CPU and GPU programmers may use established tools like VTune [64] and NSight [94]. These tools exploit the built-in hardware performance counters and provide line-by-line profiling result. Also, detailed analysis of the performance bottleneck is provided for the programmers.

The process of identifying the performance bottleneck and finding an optimization to fix the problem is called performance debugging. Previous work on FPGA performance debugging relies on instrumenting hardware monitors into DRAM/inter-module FIFO communication channels [42, 45, 62, 71, 76, 108] or into the finite-state machine of a loop pipeline [42, 108] to measure their active/idle cycle ratios. However, their instrumentation is performed from the viewpoint of an individual module with low-level hardware description language (HDL). Such limited scope makes it difficult to identify the FPGA module that is causing another module to be idle (stalled). Instead, we propose a performance debugging methodology based on HLS. Our high-level analysis allows tracing the cause of stalls on a function or loop level, which provides more intuitive feedback to the programmers by pinpointing the bottleneck of an FPGA design.

However, the difficulty in performing performance debugging on HLS tools arises from the fact that HLS tools abstract away the hardware execution model from the programmers to shorten the learning curve and to allow quick modification of various design parameters. As a result, performance-related details such as the execution cycles of individual processing elements or the list of modules that cause other modules to stall are unavailable to the users. A programmer can

[1] For a comprehensive list of the off-the-shelf HLS tools, the readers are referred to [75, 84, 93].

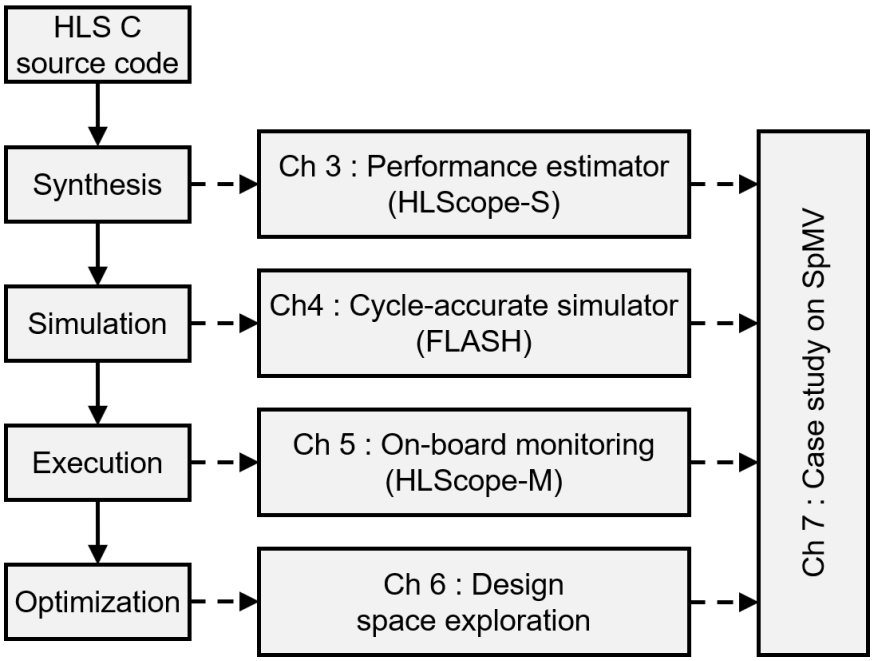


Figure 1.1: HLS development steps and the proposed performance debugging frameworks

only observe a brief synthesis report and machine-generated output code, which is almost impossible to comprehend. If a programmer wanted to analyze the output code for further performance improvement, the person often has to spend many hours to identify the performance bottleneck.

In this dissertation, we propose a collection of HLS-based performance debugging frameworks for FPGA as shown in Fig. 1.1. Each debugging framework is intended to be used at a different place of the design process. The first framework is the performance estimator for HLS. Typically, an HLS user would synthesize a design and obtain the performance estimation in the HLS synthesis report, so that the user can quickly identify the performance bottleneck and restructure the code without the time-consuming bitstream generation process. However, we will show in Chapter 3 that the HLS-reported performance estimate may be inaccurate when the program has input-dependent behavior or external memory access. To solve this problem, we propose a simulation-based modeling (SBM) approach named HLScope-S. HLScope-S automatically instruments code that models the hardware-execution behavior. The model includes the external memory access behavior (e.g., DRAM bandwidth) and the loop execution behavior (e.g., initiation interval and iteration latency). The instrumentation is performed in the granularity of loops and functions to minimize the esti-

mation overhead. Next, we run the HLS software simulation on the instrumented code to reflect the input-dependent behavior on the performance estimation. This performance estimation flow is presented in Chapter 3. We also provide a detailed description of the performance debugging process in this chapter.

The performance estimation flow described in Chapter 3 depends on the software simulator of HLS to reflect the input-dependent behavior of the design. However, we found that the current FPGA HLS commercial software simulators sometimes produce incorrect results (details provided in Chapter 4). This results in HLScope-S to provide an inaccurate performance estimate. To solve this problem, we propose a new HLS simulation flow named FLASH. The main idea behind FLASH is to extract scheduling information from the HLS tool and automatically construct an equivalent cycle-accurate simulation model while preserving C semantics. We show that correctness of the simulation and accurateness of the performance estimation are both achieved by the proposed process. Unlike HLScope-S which inserts performance estimation codes in loops or functions, FLASH simulates in a finer granularity of C statements. In order to accelerate the simulation speed compared to the RTL simulation, we abstract away the allocation, binding, and library information, which were found to be unnecessary for solving the incorrect result problem. The experimental result shows that FLASH runs three orders of magnitude faster than the RTL simulation. The details of the proposed framework is explained in Chapter 4.

The simulation-based flows in Chapters 3 and 4 provide reliable performance estimation when representative input dataset exists. However, in the case where the input may change at the time of FPGA deployment, on-board monitoring becomes necessary. Another problematic case is where high-level models do not exist for all components of the FPGA design. For example, a design may have an RTL sub-component or DRAM with unknown characteristics. In order to provide performance debugging capability for these cases, we propose a HLS-based on-board performance monitoring framework named HLScope-M in Chapter 5. The instrumented monitoring code is described in pure HLS C without involving RTL code, so that the integration process is simplified. In addition to the cycle extraction, we also propose a stall analysis network (SAN) that enables each module to trace the root cause of stall in on-board execution and find the performance bottleneck.

Table 1.1: Comparison of the proposed performance debugging frameworks

	HLScope-S (Chapter 3)	FLASH (Chapter 4)	HLScope-M (Chapter 5)	DSE (Chapter 6)
Purpose	Performance estimation	Simulation	On-board monitoring	Design space exploration
Accuracy	Approximate (~5% error)	Cycle-accurate	Cycle-accurate	Approximate (~5% error)
Time	1-10 mins	1-10 mins	1-10 hours	0.1-1 secs (per design)
Coverage	FPGA+DRAM	FPGA only	FPGA+DRAM	FPGA only
Publication	[24]	[18]	[23]	[25]

Based on the information collected from performance debugging, an FPGA designer would perform optimization. However, many different combinations of HLS directives exist, and it is difficult for designers to manually find the best configuration. Several design space exploration (DSE) frameworks for HLS tools have been recently proposed [37, 72, 100, 111, 137, 138] to solve this problem. However, one of the common limitations found in these tools is that they cannot find a design point with large speedup for applications with variable loop bounds. The reason is that loops with variable loop bounds cannot be efficiently parallelized or pipelined with simple insertion of HLS directives. Also, making highly accurate prediction of cycles and resource consumption on the entire design space becomes a challenging task because of the inaccuracy of the HLS tool cycle prediction and the wide design space. To address these challenges, we propose code transformations that increase the utilization of the compute resources for variable loops, including completely parallel computational pattern and computational patterns with loop-carried dependency (floating-point reduction and prefix sum patterns). In order to rapidly perform DSE with high accuracy, we describe a model that predicts the resource usage from the information obtained from a small number of actual HLS synthesis runs. The cycle estimation model has been derived from HLScope-S. Experiments on applications with variable loop bounds in Polybench benchmarks show that our framework outperforms current state-of-the-art DSE frameworks. This is presented in Chapter 6.

The comparison among the proposed debugging frameworks are provided in Table 1.1. As shown in the table, FLASH and HLScope-M provide cycle-accurate result. In contrast, HLScope-

S and the DSE framework are based on approximated prediction of the execution time so that the cycle estimation process is accelerated. For the time taken to apply these frameworks, HLScope-M takes most amount of time because it involves the bitstream generation process. The rest of the frameworks take similar time in the order of minutes, because they involve code transformation using ROSE infrastructure [106] and a software simulation run. Note that since the DSE process reuses the software simulation for cycle estimation of multiple design points, the time taken per design point is in the order of seconds. In terms of coverage, HLScope-S models the behavior of FPGA modules, DRAM controllers, and DRAM. Similarly, HLScope-M can monitor any type of modules. FLASH and the DSE framework are currently limited to modules that only perform computation and intra-FPGA communication, and it remains as a future work to generalize these frameworks that also model the DRAM access latency.

In order to perform quantitative comparison among the proposed frameworks, we provide a case study on sparse matrix-vector multiplication (SpMV) benchmark. This will be presented in Chapter 7. Also, concluding remarks and future work are provided in this chapter.

CHAPTER 2

Related Works

2.1 Performance Analysis Tools in CPU and GPU

Performance analysis tools have been widely used for CPUs and GPUs to help programmers with analyzing the performance bottlenecks of their programs. For example, Intel's VTune [64] is a performance analysis tool for x86 CPUs. VTune collects the profiling result on each line of code, so that the users can identify and optimize the most frequently used portion of the code. VTune also provides utilization analysis of CPU microarchitecture—such as the that of the instruction fetch and the execution units of CPU. The experience is enhanced with a graphical timeline view of the utilization. The users can also identify memory-related issues with parameters such as the cache miss rate and the effective DRAM bandwidth.

NVIDIA's NSight [94] provides a profiling result for GPU programs with details similar to that of VTune. NSight provides utilization analysis on components specific to GPUs—such as the number of warps [95] per multiprocessor or the effective shared memory bandwidth. If the performance is lower than expected, the users can refer to NSight's detailed analysis on the reason for stall—such as the memory dependency or the synchronization problems. NSight also provides a general advice on how to remove each type of stall and improve the performance.

2.2 Related Works on Performance Estimation

FPGA performance estimation models are used in design space exploration (DSE) tools such as Lin-analyzer [138] and COMBA [137] to find the best design point among possible candidates. Their models predict the performance for common optimizations such as unrolling, pipelining,

and array partitioning. Also, DSE flows such as Aladdin [111] can provide cycle estimation for programs with dynamic behavior using the instruction trace from simulation. However, we will explain in Section 3.2.7 why their models have limitations in accurately predicting the performance of designs actually generated by HLS tools. We will also demonstrate that HLScope-S provides the estimation result faster than the trace-based flows.

For external memory modeling, Aladdin [111] links a cycle-accurate trace-based DRAM simulator to a performance estimator, but cycle-accurate simulation takes a long time compared to the high-level modeling we propose in Section 3.2.4. Work in [9, 13] describes approximately timed transactional-level SystemC simulation with instruction set simulator (ISS), but we would like to raise the level of abstraction to a level of function call or loops, which speeds up the simulation process compared to modeling individual external memory requests. A simple high-level model for a single external memory accessor has been provided in [97], but FPGA typically has multiple processing elements competing for the shared external memory resource. Our model for multiple PE contention will be explained in Section 3.2.4.2. The work in [72] uses a statistical model to estimate the external memory latency of fetching a block, but it is uncertain that such predefined template models for block access can be generalized for arbitrary access patterns.

2.3 Related Work on Simulation

As will be explained in Chapter 4, FLASH is a software-based HLS simulator that provides cycle-accurate result. Previous work on software-based HLS simulation includes LegUp HLS [14], which provides a speedup prediction based on the profiling result of the source code and the execution cycle from its synthesis result. FlexCL [80] takes OpenCL code as input and performs dynamic profiling with a control data flow graph (CDFG) simulation for performance and power estimation. These works, however, do not guarantee cycle-accuracy like FLASH.

There are several SystemC simulators (e.g., [28, 92, 109]) that can achieve cycle-accuracy for the source code that has explicit scheduling information specified by the programmer. However, such code transformation may be too difficult for non-experts. Our flow, on the other hand,

achieves cycle-accuracy for an HLS C source code without requiring user-defined scheduling information.

FLASH has a similar high-level optimization approach with transaction-level modeling (TLM) works [9, 13, 54] in a sense that both FLASH and TLM works accelerate the simulation speed by abstracting unnecessary implementation details. However, even if one tries to apply a similar strategy to optimize the HLS simulation, several issues remain. One of the issue is that, among various type of information created by HLS (e.g., allocation, binding, and scheduling information), it is unclear which one should be kept or abstracted. Another issue is how to automate the abstraction process so that even non-expert users can benefit from this approach. We address both issues in Chapter 4.

There is a class of work that accelerates the simulation of an HLS tool's output RTL code by converting the RTL code into a cycle-accurate C model [83, 113]. Mahapatra, et al. [83] report a speedup of 5X after removing the core computation and only maintaining the IO timing, but such approach cannot be used for data-dependent benchmarks. Verilator [113], on the other hand, can be used to provide a functionally correct and cycle-accurate HLS simulation as our work. Some of the techniques Verilator uses are removal of time delays, randomized unknown value, and creation of table lookups. However, the speedup in Verilator is limited because it is very difficult to *completely* remove allocation and binding information from the RTL code—whereas our approach is free from this overhead since it was never added in the first place. A quantitative comparison is presented in Section 4.8.

2.4 Related Works on FPGA Hardware Debugging

Commercial FPGA vendors provide logic analyzers that can be synthesized into the FPGA to extract logic values. Examples include Xilinx's Chipscope [126] and Altera's SignalTap [4]. These tools typically require users to specify the monitored signals and the trigger conditions from the RTL signal list. Trace buffers are then attached to the monitored signals. Next, the original FPGA design and the monitoring logic are synthesized into a bitstream. The bitstream is executed until

the triggering condition is met, and the content of the trace buffer is offloaded for analysis. One of the problems of using these synthesizable logic analyzers is that the manual selection from the low-level netlist may be too difficult for novice FPGA programmers.

Whereas the synthesizable logic analyzers are typically used for RTL designs, there are FPGA debugging works that target HLS designs [51–53, 89, 90]. These works maintain mapping between the variables in the C source code and the hardware registers so that the programmers can select the variables to be debugged in high-level. Similar to Chipscope and SignalTap, they store the content of the variables in trace buffers (called Event Observability Buffer in [89, 90])—however, they perform further optimizations such as storing only the data value that has changed [51] and reconstructing the data value from other variables offline [53]. After offloading the recorded data from the trace buffer, the program may be replayed as the user steps through the source code [51].

Chipscope, SignalTap, and works in [51–53, 89, 90] aim to provide correctness debugging capability. However, these work do not provide insight on solving the performance problems of an FPGA design. To address this problem, some performance debugging tools have been proposed in literature. Work in [71] profiles heterogeneous system consisting of CPUs and FPGAs to identify the bottleneck nodes with high usage rate. Intel's OpenCL HLS tool [62] profiles the communication channel between computation modules as well as the external memory ports. HwPMI [108] provides comprehensive infrastructure of performance monitors for FIFOs, bus, BRAMs, and finite-state machines (FSMs). Work in [42] targets the FSMs derived from if, while, and for statements (written in C), and measurement modules are inserted in VHDL to collect the statistics of the FSMs. Work in [120] is based on OpenCL and stores events' timestamp and their sequence numbers which can be used to measure the stall latency and monitor the memory access patterns. We will explain the difference of HLScope-M compared to these previous works in Section 5.5.

2.5 Related Works on HLS Design Space Exploration

The automated DSE framework for HLS is described in several published works. The work in [72] and [100] take high-level parallel patterns such as *map* and *reduce* and generate an FPGA

design based on the predefined templates and the statistical performance model. Aladdin [111] omits synthesis and RTL generation and reuses optimization across a large design space for fast exploration among ASIC accelerators. Lin-analyzer [138] takes a similar approach and further considers the FPGA-specific resources (*e.g.,* DSP, BRAM) during its scheduling. MPSeeker [139] uses Gradient Boosted Machine (GBM) technique for resource modeling and explores trade-off between fine-grain and coarse-grain parallelism. Most recently, COMBA [137] and AutoAccel [37] have been proposed. COMBA explores a comprehensive set of HLS optimization directives and finds the best configuration based on their metric-guided search. AutoAccel presents a push-button flow based on their composable, parallel, and pipeline micro-architecture. These works, however, do not guarantee finding an efficient design for applications with variable loop bounds. A quantitative comparison will be provided in Section 6.5.

CHAPTER 3

Performance Debugging with Fast and Accurate HLS Performance Estimator

In this chapter, we will first present the scope and the overall structure of our performance debugging tool, HLScope. We will describe how HLScope performs analysis in high-level semantics such as functions of loops, which is more intuitive than providing feedback from DRAM/inter-module FIFO communication channels [42, 45, 62, 71, 76, 108] or finite-state machine of a loop pipeline [42, 108]. We will next explain the performance debugging parameters that is provided by HLScope. Based on the collected parameters, HLScope evaluates the level of application optimization to identify potential performance bottleneck. This is followed by three sample debugging sessions for various versions of quicksort.

HLScope requires cycle count of each module for its analysis. It is possible to obtain this information from an on-board performance monitoring flow (HLScope-M). However, a drawback of such approach is that the FPGA bitstream generation typically takes many hours. An alternative approach is to use the performance estimate provided in the HLS tool's synthesis report. However, we will show that the synthesis report may become incomprehensible when the program has input-dependent behavior. To solve this problem, we will describe a fast and accurate HLS-based cycle estimation flow (HLScope-S) in this chapter. A high-level modeling of the hardware behavior is inserted into the C source code, and the cycle information is extracted by running a software simulation. We will describe an automated code transformation to enable this process.

3.1 Performance Debugging Framework: HLScope

3.1.1 Scope and Overall Tool Structure

Performance debugging is a process of identifying the performance bottleneck and finding an optimization to fix the problem. We illustrate our performance debugging framework with Vivado HLS [130] because of its widespread use in FPGA designs [20, 21, 23, 32, 34, 78, 79, 79, 105, 135, 138]. Similar to the level of information given in Vivado HLS report, we expect the HLS tool to provide iteration latency (IL), initiation interval (II), and trip count (TC) (will be defined in Section 3.2.2) of loops and the latency of functions. An example of the expected cycle information for the *qsort_compute()* function of the quicksort (Fig. 3.1 (c)) is shown in Table 3.1. We do not expect the HLS tool to provide cycle information about every line of code; for example, in the graph representation of the quicksort code in Fig. 3.2, the cycle information for the else statement in line 40 is not given in Table 3.1. Instrumentation methods to recover the missing cycle information ('?' in Table 3.1) will be extensively discussed in Section 3.2.3.

Table 3.1: Vivado HLS report for the computation part of quicksort code given in Fig. 3.1. IL, II, and TC are explained in Section 3.2.2.

Name	IL	II	TC
qsort_comp	?	-	-
Loop 1	?	-	?
Loop 1.1	?	-	?
Loop 1.1.1	4	4	?
Loop 1.1.2	4	4	?

As an input, HLScope takes a C code that may have Vivado HLS directives. Our tool will analyze all modules in the top function as a default – or user may specify a particular submodule of interest. Using the APIs in the ROSE compiler infrastructure [106], HLScope first transforms the input code into a tree of nested code blocks. A code block consists of multiple operations with a single program execution path. For the quicksort example in Fig. 3.1, the resulting graph is shown in Fig. 3.2. Vivado HLS will schedule functions without true data dependency to be executed in parallel– thus, HLScope analyzes the variables used in each function for data dependency detection to match Vivado HLS's schedule.

```
1  void qsort_top(float* dram, int batch_num, int n_per_batch ){
2    float lmem[LMEM_MAX];
3    for( int i = 0 ; i < batch_num ; i++ ){
4      load(dram, lmem, n_per_batch, i);
5      qsort_comp(lmem, n_per_batch);
6      store(dram, lmem, n_per_batch, i);
7  } }
```

(a) Top C function (batch_num=128, n_per_batch=1024, LMEM_MAX=1024, `dram': external DRAM port, `lmem': local BRAM)

```
8  void load(float* dram, float lmem[LMEM_MAX],
9            int n_per_batch, int i ){
10   memcpy(lmem,dram+i*n_per_batch, n_per_batch*sizeof(float));
11 }
12 void store(float* dram, float lmem[LMEM_MAX],
13           int n_per_batch, int i ){
14   memcpy(dram+i*n_per_batch,lmem, n_per_batch*sizeof(float));
15 }
```

(b) Load and store function

```
16 void qsort_comp( float lmem[LMEM_MAX], int n_per_batch ){
17   int beg[M], end[M], i=0, L, R; float piv, swap;
18   beg[0]=0; end[0]=n_per_batch;
19   while (i>=0) { // Loop 1            //one round of reordering
20     L=beg[i]; R=end[i]-1;                    // init L & R ptr
21     if (L<R) {
22       piv=lmem[L];
23       while (L<R) { // Loop 1.1       // swap until L & R meets
24         while (lmem[R]>=piv && L<R){ // Loop 1.1.1
25           #pragma HLS pipeline  // decrement R until an element
26           R--;                  // less than pivot is found
27         }
28         if (L<R) { lmem[L++]= lmem[R]; }        // copy it to L
29         while (lmem[L]<=piv && L<R){ // Loop 1.1.2
30           #pragma HLS pipeline  // increment L until an element
31           L++;                  // less than pivot is found
32         }
33         if (L<R) { lmem[R--]= lmem[L]; }        // copy it to R
34       }
35       lmem[L]=piv;beg[i+1]=L+1;end[i+1]=end[i];end[i++]=L;
36       if (end[i]-beg[i]>end[i-1]-beg[i-1]) { //swap qsort order
37         swap=beg[i]; beg[i]=beg[i-1]; beg[i-1]=swap;
38         swap=end[i]; end[i]=end[i-1]; end[i-1]=swap;
39     }}
40     else {
41       i--;
42 } } }
```

(c) Computation function

Figure 3.1: Initial unoptimized C code of non-recursive quicksort (modified from [49])

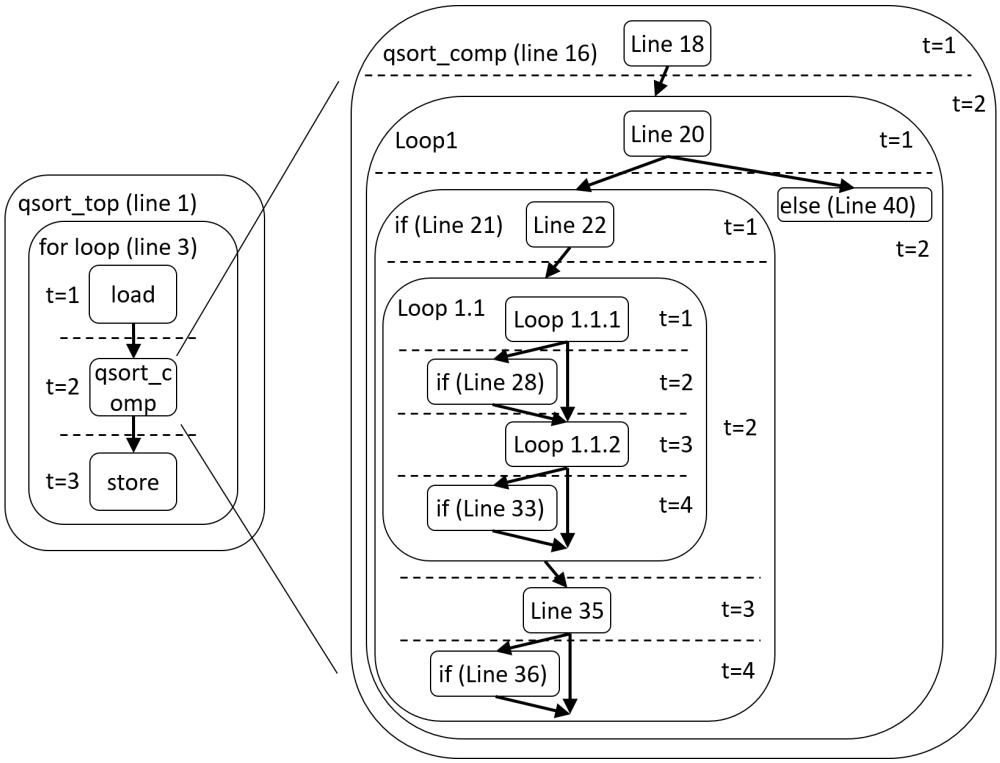


Figure 3.2: Dependency graph for $qsort\_comp()$ given in Fig. 3.1 (c). The dotted line denotes synchronization barriers, and t denotes time frames divided by synchronization barriers.

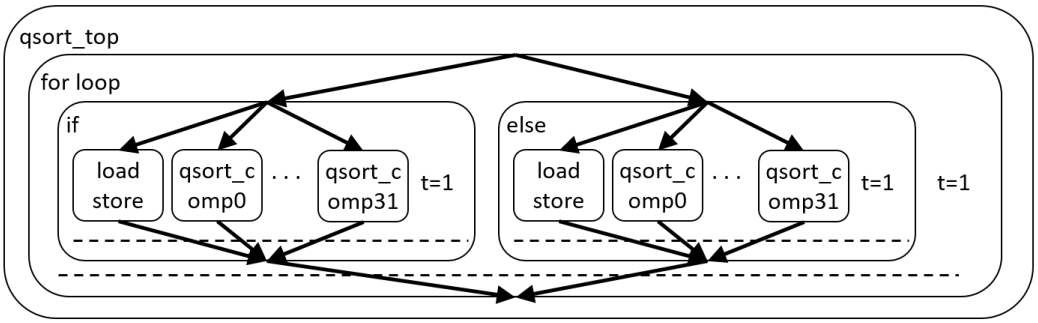
For simplified execution control, Vivado HLS will schedule all submodules to wait for the completion of other submodules executed in parallel. In the example dependency graph for a modified version of the quicksort in Fig. 3.3 (optimization will be explained in Section 3.1.3), $loadstore()$ function (combines *load()* and *store()* of Fig. 3.1) and unrolled functions of $qsort\_comp()$ execute in parallel. After finishing execution, they will wait for all other functions to terminate as well. This is similar to the barrier synchronization in GPU [95], and from now on, we will simply refer to it as *synchronization*. Synchronization is expressed in the dependency graph with the dotted lines as in Fig. 3.2 and Fig. 3.3. To keep track of the execution order, we also assigned time frames t to the groups of blocks that execute in parallel.

In order to avoid the overhead of waiting for other modules to complete (*synchronization overhead*), the user may use the dataflow pragma (*#pragma HLS dataflow* [130]) and FIFO communication channel (*hls::stream<variable type>* [130]). This will allow modules with data dependency

```
void qsort_top(float* dram, int batch_num, int n_per_batch ){
  float lmem0[UNROLL_FACTOR][LMEM_MAX];        //partitioned BRAM for parallel
  float lmem1[UNROLL_FACTOR][LMEM_MAX];        //access, UNROLL_FACTOR=32
  #pragma HLS ARRAY_PARTITION variable="lmem0" complete dim='1'
  #pragma HLS ARRAY_PARTITION variable="lmem1" complete dim='1'

  for( int i = 0 ; i < batch_num ; i++ ){
    if( i%2 == 0 ){ //in even i, lmem0 for memory access and lmem1 for compute
      loadstore(dram, lmem0, n_per_batch, i);
      for( int j = 0 ; j < UNROLL_FACTOR ; j++ ){
        #pragma HLS unroll // 32 duplicated compute PEs
        qsort_comp(lmem1[j], n_per_batch);
    } }
    else{           // in odd i, lmem1 for memory access and lmem0 for compute
      loadstore(dram, lmem1, n_per_batch, i);
      for( int j = 0 ; j < UNROLL_FACTOR ; j++ ){
        #pragma HLS unroll // 32 duplicated compute PEs
        qsort_comp(lmem0[j], n_per_batch);
} } } }
```

(a) Code



(b) Dependency graph

Figure 3.3: Quicksort code and corresponding dependency graph after applying double buffering and unrolling. $loadstore()$ function combines the $load()$ and $store()$ function given in Fig. 3.1, and $qsort\_comp()$ function is the same as that of Fig. 3.1.

to continue with the execution as soon as the input data is available and the output buffer is not full. From now on, we will refer to this module dependency as *parallel by dataflow*. In the dependence graph, we place no barrier between the modules and assume they are executed in the same time frame.

After converting the program into a graph structure, HLScope-S (Section 3.2) measures the number of cycles for each block. HLScope-M (to be presented in Chapter 5), on the other hand, measures the number of cycles only at a function level because of the monitoring logic overhead (Table 5.4). Note that the measurement can be performed at a finer granularity in HLScope-S, because the simulation time overhead is small (~4%, as shown in Table 3.10). In addition to the cycle measurement, HLScope also record the number of DRAM transactions in bytes using the Vivado HLS software simulation with a real input testbench. Based on these measurements, HLScope computes several performance debugging parameters that will be explained in the next subsection.

3.1.2 Performance Debugging Parameters

HLScope provides three groups of performance debugging parameters as shown in Table 3.2. The first group indicates a module's importance for performance debugging.

Table 3.2: List of performance debugging parameters

Probe name	Description
cycle	Execution cycle of each module
PCP	Is this module on performance critical path?
LUT/DSP/BRAM	LookUp Table / DSP48 / Block RAM
stall rate	1 - usage rate of computational resource
DBW	DRAM bandwidth
ADBW	Aggregate DBW among all modules executed in parallel
Reason for stall	Stall type & the name of module causing stall

Parameter *cycle* is the total execution cycle of each module. Cycle information is obtained from the in-FPGA flow (Section 5.2 and Section 5.3) or the simulation flow (Section 3.2.3 and Section 3.2.4). We exclude the period when the module has been stalled due to other modules. Assuming a hierarchical structure of code blocks such as Fig. 3.2 and Fig. 3.3, HLScope will

add the cycle estimate for a child block to its parent block depending on the static dependency analysis of the code. There are three possibilities: serial, parallel, and parallel by dataflow. If true dependency exists between blocks, they can be executed in a serial manner and will be assigned to a different time frame. The execution time of the child blocks c (t_c) will be added to its parent p (t_p) as:

$$t_p = \sum_c t_c \tag{3.1}$$

If there is no dependency, the blocks can execute in parallel and will be assigned to the same time frame. The execution time is the maximum of the parallel blocks:

$$t_p = \max_c (t_c) \tag{3.2}$$

For the modules running in parallel by dataflow (explained in Section 3.1.1), the execution time is also bounded by the slowest block (Eq. 3.2). The difference is that they do have dependency between them, so it takes $\sum_c IL_c$ for data to traverse from beginning to end:

$$t_p = \max_c (t_c) + \sum_c IL_c \tag{3.3}$$

Also, the modules will be assigned to the same time frame.

Using the module dependency and the cycle information, we mark each module as to whether it has an effect on the overall performance or not. We refer to this parameter as *performance critical path (PCP)*. The serially-executing modules are classified 'yes' in PCP. If there are several parallel-executing modules in a time frame, a module with the longest execution time will be classified 'yes' in PCP. Also, going top-down hierarchically, submodules of a module marked 'yes' will be further analyzed in a similar way for the critical path analysis; submodules in all other modules are marked 'no'. For the modules running in parallel by dataflow, we only mark the module with the longest execution time in a time frame as 'yes' and the rest as 'no'.

Next, HLScope report shows compute resource (LUT/BRAM/DSP) for each function. This is obtained from the HLS synthesis report. A function with large LUT/BRAM/DSP consumption and

a large stall rate can be identified as wasting the compute resource. This could be problematic if the application is compute-intensive.

The next group of parameters provided by HLScope indicates the usage rate of resources. The first parameter is *stall rate*. The stall rate $stall_i$ of each module is computed by:

$$stall_i = 1 - t_i/t_{tot} \tag{3.4}$$

where t_i is each module i's execution time and t_{tot} is the total execution time of all modules. This rate shows the usage rate of FPGA computational resources.

Next, we determine DRAM bandwidth (DBW). It is computed by dividing the amount of data written to or read from the external memory. The number of transferred bytes is obtained from the software simulation that will be explained in Section 3.2.4. We also provide the aggregated DRAM bandwidth (ADBW) which computes the combined DRAM bandwidth among all modules executed in parallel. Note that the DRAM access is only measured in the software simulation flow, HLScope-S. Thus, the performance debugging based on the on-board flow, HLScope-M, does not provide DBW and ADBW parameter values.

Finally, we provide the reason for the stall and the name of the module that is causing the stall. If a module is waiting for data from another module, it is classified as a dependency stall. It includes stalls due to other modules executed in serial or stalls in inter-module communication among parallel-executing modules by dataflow. If a module is waiting for other parallel-executing modules to finish, the stall is classified as a synchronization stall. A module could have multiple stall reasons depending on its place in HLS module hierarchy– an example will be shown in Fig. 3.5, where *qsort_comp* PE0 has 2.2% synchronization stall waiting for *qsort_comp* PE27 to finish. It also has 25.2% dependency stall for both *load* and *store* because *qsort_comp* PE 0-31, *load*, and *store* are executed in serial.

Based on the reason for stall provided by HLScope, the programmer can decide which module to focus his/her attention on for further optimization. An example performance debugging session will be presented in the next section.

3.1.3 Performance Debugging for Quicksort

In this section, we will demonstrate performance optimization steps for the quicksort example based on HLScope. We assume that we have 128 sets of 1024 single-precision floating-point numbers to be sorted.

Fig. 3.4 lists the performance debugging parameters collected from initial unoptimized quicksort (code: Fig. 3.1a, dependency graph: Fig. 3.2). The cycle information has been collected from the software simulation flow, HLScope-S. The most obvious performance problem that we can identify from the report is that *qsort_comp()* takes most (96.5%) of the execution time, and probably should be the target for optimization. Note that the stall rate is very high (98.2%/98.3%) for load() and store(), but does not cause too much computation resource to remain idle since the LUT and DSP usage is small (481/445 and 4/4).

```
for( int i = 0 ; i < batch_num ; i++ ){      ||Cycle|PCP |LUT |BRAM|DSP|Stall|DBW |ADBW|Reason for stall             ||
  load(dram,lmem,n_per_batch,i);             ||145k |Yes |481 |0   |4  |98.2 |724M|724M|comp(96.5,dep),store(1.7,dep)||
  qsort_comp(lmem, n_per_batch);             ||7.75M|Yes |2538|3   |0  |3.5  |0   |0   |load(1.8,dep),store(1.7,dep) ||
  store(dram,lmem,n_per_batch,i);            ||140k |Yes |445 |0   |4  |98.3 |748M|748M|comp(96.5,dep),load(1.8,dep) ||
}
```

Figure 3.4: Performance debugging parameters collected from the initial unoptimized version of quicksort. Parameter derived from HLScope-S result. (batch_num=128, n_per_batch=1024, 'dram': external DRAM port, 'lmem': local BRAM).

Based on the analysis from the initial version, we apply unrolling on the compute PEs. The result is shown in Fig. 3.5. We can confirm that the *qsort_comp()* function indeed takes a considerably shorter time—from 7.75M cycles to 231–258k cycles. For the nodes on the time-critical path (*load()*, *qsort_comp()* PE 27, *store()*), however, the analysis shows that the proportion of *load()* and *store()* increased to 25.2%, respectively. This suggests that memory access has now become a major stall reason. A hint for solution can be found in the aggregate DRAM BW. During *qsort_comp()*, the ADBW is 0, which means that DRAM is not being utilized at all. This suggests that modules that do use the DRAM, *load()* and *store()*, can probably be overlapped in *qsort_comp()*.

For memory and compute overlapping, we perform double buffering optimization. The code after optimization was shown in Fig. 3.3, and the corresponding report is presented in Fig. 3.6.

```
float local_data[UNROLL_FACTOR]; //partitioned BRAM for parallel access, UNROLL_FACTOR=32
#pragma HLS ARRAY_PARTITION variable="lmem" complete
                                ||Cycle|PCP |LUT |BRAM|DSP|Stall|DBW |ADBW|Reason for stall                          ||
for( int i = 0 ; i < batch_num ; i++ ){      |    |    |    |   |     |    |    |                                         ||
  load( dram, lmem, ..);            ||132k |Yes |3017|0   |4  |74.8 |797M|797M|comp27(49.5,dep),store(25.2,dep)         ||
  for( int j = 0 ; j < UNROLL_FACTOR ; j++ ){     |    |    |   |     |    |    |                                         ||
#pragma HLS unroll // 32 duplicated compute PEs   |    |    |   |     |    |    |                                         ||
    qsort_comp(lmem[j]..); //PE0      ||247k |No  |2538|3   |0  |52.6 |0   |0   |ld(25.2,dep),comp27(2.2,sync),st(25.2,dep)||
    . . .                 //PE7      ||231k |No  |2538|3   |0  |55.7 |0   |0   |ld(25.2,dep),comp27(5.3,sync),st(25.2,dep)||
    . . .                 //PE27     ||258k |Yes |2538|3   |0  |50.5 |0   |0   |ld(25.2,dep),st(25.2,dep)                 ||
  } . . . . . .                      ||     |    |    |   |     |    |    |                                         ||
  store( dram, lmem, ..);           ||131k |Yes |5809|0   |4  |74.8 |798M|798M|comp27(49.5,dep),load(25.2,dep)          ||
}
```

Figure 3.5: Performance debugging parameters after unrolling qsort_comp() function 32 times. Some PEs not shown for brevity.

For simplicity of design, we combined the *load()* and *store()* modules into one. Also, this report contains cycle information from in-FPGA flow, because HLScope recommends taking a more accurate approach when it detects several PEs running in parallel with similar execution time so that it can provide correct PCP and stall reason analysis.

The report shows that there is some DRAM BW transaction while the *qsort_comp()* module is being executed (ADBW=701M). Also, there are no more dependency stalls as reasons for a stall. These two factors suggest that parallelization is properly taking place. Also, the fact that the stall rate has decreased drastically from the initial version (98.2/3.5/98.3 $\rightarrow$ 0.0/11.0/16.4/7.4) suggests that overall efficiency of the design has improved significantly.

```
for( int i = 0 ; i < batch_num ; i++ ){
  if( i%2 == 0 ){ //in even i, lmem0 for memory access and lmem1 for compute
                                          ||Cycle|PCP |LUT |BRAM|DSP|Stall|DBW |ADBW|Reason for stall    ||
   loadstore(dram,lmem0,..);              ||299k |Yes |9145|0   |8  |0.0  |701M|701M|                   ||
   for( int j = 0 ; j < UNROLL_FACTOR ; j++ ){   ||     |    |    |   |     |    |    |                   ||
#pragma HLS unroll // 32 duplicated compute PEs   ||     |    |    |   |     |    |    |                   ||
     qsort_comp(lmem1[j],..); //PE0               ||266k |No  |2538|3   |0  |11.0 |0   |701M|loadstore(11.0,sync)||
   . . .                    //PE7               ||250k |No  |2538|3   |0  |16.4 |0   |701M|loadstore(16.4,sync)||
   . . .                    //PE27              ||277k |No  |2538|3   |0  |7.4  |0   |701M|loadstore(7.4,sync) ||
  else{ . . . }  //in odd i, lmem1 for memory access and lmem0 for compute    . . . . . .
} }
```

Figure 3.6: Performance debugging parameters after applying double buffering optimization.

The report in Fig. 3.6 indicates that the bottleneck is now *loadstore()*, since all other modules point to this module as their reason for stall, and only *loadstore()*'s PCP is 'yes.' Since the DRAM BW reported that (701MB/s) is far less than ideal BW (9.05GB/s), we may perform some DRAM access optimization for further performance improvement.

3.2 Performance Estimator for HLS

3.2.1 Introduction

The performance debugging process illustrated in Section 3.1 requires the cycle count of each modules. One possible way of obtaining this information is to use the on-board performance monitoring flow that will be explained in Chapter 5. However, a drawback of such approach is that the FPGA bitstream generation typically takes many hours.

An alternative approach is to use the performance estimate provided in the HLS tool's synthesis report. However, the problem of using HLS synthesis report is that the report may become incomprehensible when the program has dynamic execution paths and loop bounds that depend on the input data. A motivating example for a well-known quicksort was presented in Fig. 3.1c) [49] and is shown again in Fig. 3.7. Depending on the value of pivot, the number of iterations for Loop 1.1.1 and Loop 1.1.2 can vary from 1 to N. Also, it may or may not execute some of the conditional statement (*e.g.,* code A, B, and C). The synthesis report by Vivado HLS for this program is shown in Table 3.3; it has not been successful in providing any estimate for the total execution time.

Table 3.3: Vivado HLS report for the quicksort code given in Fig. 3.7. IL, II, and TC are explained in Section 3.2.2.

Name	IL	II	TC
qsort_comp	?	-	-
Loop 1	?	-	?
Loop 1.1	?	-	?
Loop 1.1.1	4	4	?
Loop 1.1.2	4	4	?

There are several reasons why Vivado HLS was unable to provide a cycle estimate. One of the reason is the variable trip count (TC) in the while loops (Loops 1, 1.1, 1.1.1, and 1.1.2). TC is required in the loop cycle estimation (Eq. 3.5), but in many cases TC is provided by user or is input-dependent. Also, quicksort contains dynamic execution path (conditional statements in code A, B, and C) that is unavailable in the report.

Another aspect which Vivado HLS does provide an estimate but is incorrect by a wide margin is the external memory modeling. Vivado HLS does allow setting a latency term for each port, but

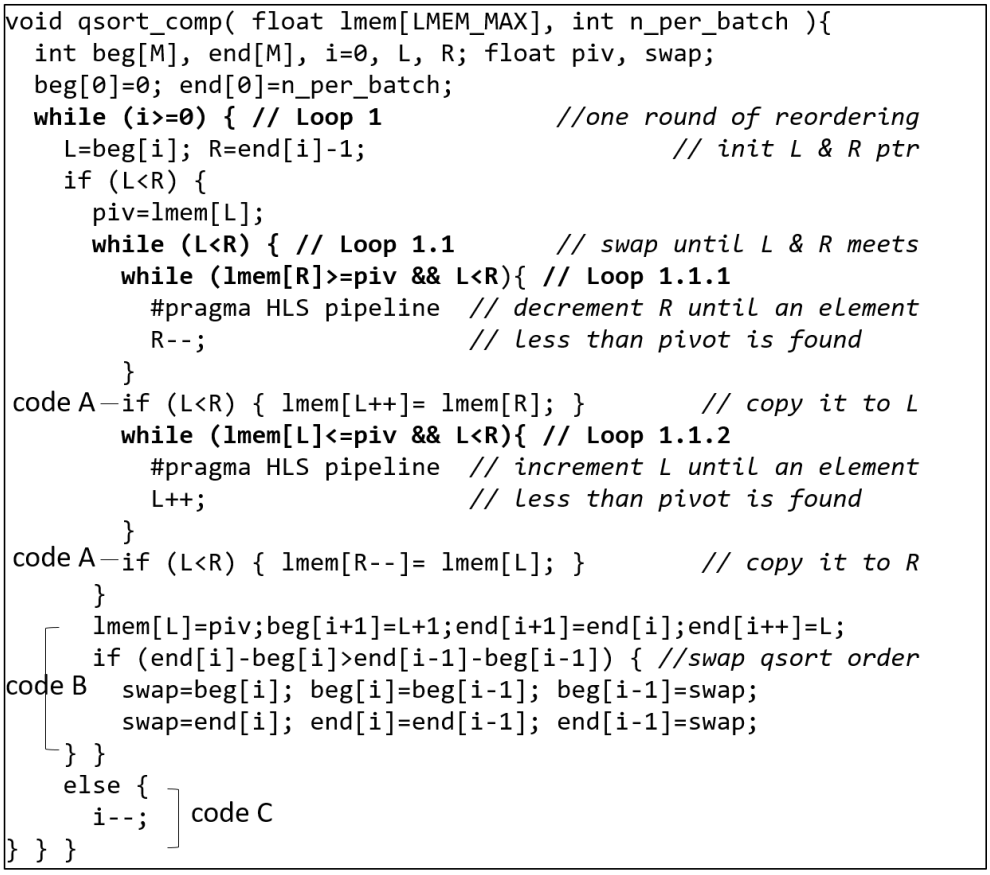


Figure 3.7: C code of non-recursive quicksort [49]

this is a too-optimistic prediction that does not consider the effective DRAM bandwidth and the memory contention. For an example design that has one read and one write port of 512b external memory with very long burst access, Vivado HLS will provide a cycle estimate with the assumption that the kernel may have a DRAM bandwidth of 25.6 GB/s (=2*200MHz*512b). However, [22] reports that the effective bandwidth would be from 4.9 GB/s to 9.5 GB/s depending on the platform and the configuration.

Our work addresses these shortcomings by providing a fast and accurate HLS-based cycle estimate of the FPGA execution. In order to compensate the inaccuracy introduced by the imperfect loops and the dynamic execution path, we provide a HLS-specific code instrumentation technique to extract the best estimate of these inaccuracy sources from the HLS synthesis report. After extracting this hidden performance information, we construct a new simulation file that models the execution behavior of hardware. As we run the software simulation with an input dataset, the

instrumented simulation file provides performance estimate that reflects the dynamic behavior of the design. We refer to this approach as simulation-based modeling (SBM). This will be explained in Section 3.2.3.

We also provide a high-level external memory model for a typical FPGA architecture, where multiple processing elements (PEs) are connected through a bus to the shared external memory. We assume the PEs issue memory requests of various data bitwidth and burst length. Again, to reduce the estimation time, the memory model is incorporated into the HLS software simulation. The main challenge in this part is how to abstract the individual memory access into a high-level model for fast estimation. Another challenge is that outstanding memory access from multiple PEs occurs in parallel, whereas the software simulation is performed in serial. The solution will be explained in Section 3.2.4.

3.2.2 Background: Cycle Estimation of Loops

The parameters used to estimate the execution cycle of a pipelined loop can be explained with Fig. 3.8. The number of cycles to complete one iteration of a loop is called the iteration latency (IL). The execution of each iteration is pipelined, and the input rate of the pipeline is called the initiation interval (II). The number of iterations is referred to as trip count (TC). As can be deduced from Fig. 3.8, the execution cycle for a pipelined loop is [79]:

$$t = II * (TC - 1) + IL \tag{3.5}$$

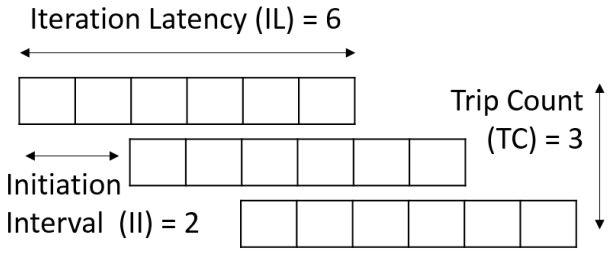


Figure 3.8: Loop pipelining parameters

3.2.3 Improving Cycle Estimation Accuracy for Loops and Conditional Statements

3.2.3.1 Estimation for Loops

However, for loops with input-dependent loop bound (see quicksort example in Fig. 3.7), Vivado HLS cannot provide the cycle estimate because TC is unknown in Eq. 3.5. The tool will only provide the II and IL. We can obtain the actual TC by inserting a simple counter statement, as shown in bold statements in Fig. 3.9. Regardless of the existence of input-dependent loop bound or conditional break statement, TC can be correctly estimated. Then the loop cycle estimation code is inserted after the loop based on the run-time acquired TC. Finally, the cycle estimate for Loop 1.1.1 is hierarchically added into its parent loop (Loop 1.1) cycle estimate.

```
int loop_11_cycle = 0;
while (L<R) {  // Loop 1.1  // Swap until L ptr & R ptr meets
  ......
  int loop_111_cycle = 0;
  int loop_111_TC=0;
  while (arr[R]>=piv && L<R){ // Loop 1.1.1 //Decrement R ptr
    #pragma HLS pipeline      //until an element less than
    R--;                      //pivot is found or L & R meets
    loop_111_TC++;
  }
  loop_111_cycle = (loop_111_TC-1)*loop_111_II + loop_111_IL;
  loop_11_cycle += loop_111_cycle;
  ......                      // Swap arr[L] and arr[R]
}
```

Figure 3.9: Code instrumentation to find dynamic loop bound for loop 1.1.1 in Fig. 3.7. Instrumented code is in bold.

This instrumentation allows the cycle estimate of an unbounded loop to be found, but an accurate estimate cannot be obtained if loops are not perfectly nested. An example can be found in Loop 1.1. Even if the cycles for Loop 1.1.1 and 1.1.2 are found by the TC instrumentation, there is no information about code A. The reason is that Vivado HLS did not give the IL of Loop 1.1 ('?' for Loop 1.1's IL in Table 3.1) because of the existence of unbounded Loops 1.1.1 and 1.1.2. The same problem exists for Loop 1.

This problem can be formalized as follows. Suppose that a parent loop p is composed of

multiple child blocks c (loop, function call, or conditional statements), and only one of these child blocks, $c1$, is a loop. p is not a perfectly nested loop due to the existence of $c2, c3$.... Assuming true dependency exists between c, the iteration latency of the parent block (IL_p) is the sum of child block execution time (t_c):

$$IL_p = \sum_c t_c = (\sum_{c \neq c1} t_c) + II_{c1} * (TC_{c1} - 1) + IL_{c1}. \tag{3.6}$$

If dependency does not exist between c, the summation in Eq. 3.6 can be changed to max(), similar to Eq. 3.2.

If the trip count of $c1$ (TC_{c1}) is not known at compile time, the HLS compiler will be unable to provide IL_p even if the rest of child execution time ($\sum_{c \neq c1} t_c$) is known. The variable trip count instrumentation (Fig. 3.9 [23]) on TC_{c1} allows estimation of $II_{c1} * (TC_{c1} - 1) + IL_{c1}$ in Eq. 3.6, but the estimate is not accurate since p's non-perfectly nested region ($\sum_{c \neq c1} t_c$) has not been considered.

To solve this problem, we can force the HLS compiler to provide the missing information $\sum_{c \neq c1} t_c$ by assigning an arbitrary TC_{c1}. In Vivado HLS, this can be achieved with the pragma "*#pragma HLS loop_tripcount min=100 avg=100 max=100*" where 100 is the arbitrary trip count. With this instrumentation, HLS will provide IL_p, II_{c1}, TC_{c1}, and IL_{c1}. Then the imperfect loop part is estimated by subtracting ($II_{c1} * (TC_{c1} - 1) + IL_{c1}$) from IL_p. Whether the arbitrary TC_{c1} matches the actual trip count in execution is irrelevant, since IL_p also increases at the exact same rate of II_{c1}, as shown in Eq. 3.6. If there are multiple child loops (c1,..,cN) in p, the cycle estimation for the non-perfectly nested region of p can be generalized as

$$\sum_{c \neq c1,..,cN} t_c = IL_p - \sum_{c=c1,..,cN} (II_c * (TC_c - 1) + IL_c). \tag{3.7}$$

An example instrumentation for the quicksort is shown in Fig. 3.11. After pragma insertion in Fig. 3.10, Vivado HLS will provide a report on the assumption that child loop bounds are fixed (Table 3.4). An estimate for the non-perfectly nested region of Loop 1.1 can then be obtained by

```
……
while (i>=0) {
  #pragma HLS loop_tripcount min=100 avg=100 max=100
  ……
    while (L<R) {
      #pragma HLS loop_tripcount min=100 avg=100 max=100
      while (arr[R]>=piv && L<R){
        #pragma HLS loop_tripcount min=100 avg=100 max=100
        ……
      }
      ……
      while (arr[L]<=piv && L<R){
        #pragma HLS loop_tripcount min=100 avg=100 max=100
        ……
      }
      ……
    }
    ……
}
```

Figure 3.10: Quicksort code after pragma insertion

subtracting the cycle estimate of Loops 1.1.1 and 1.1.2 from Loop 1.1 (811-400-400 =11). The estimate for Loop 1 can be obtained in a similar way (81114-811*100 =14). The estimates for Loop 1.1 and Loop 1 will be automatically inserted into the simulation code (Fig. 3.11). In the software simulation process, the HLS estimate from the arbitrary loop bounds will be ignored and will instead be estimated as Eq. 3.5 with the instrumented TC.

For the quicksort example, HLScope-S first automatically inserts the pragma on every loop to make Vivado HLS assume that the loop bounds are fixed (Fig. 3.10). This allows Vivado HLS to generate a report in Table 3.4. An estimate for the non-perfectly nested region of Loop 1.1 can then be obtained by subtracting the cycle estimate of Loops 1.1.1 and 1.1.2 from Loop 1.1 (811-400-400 =11). The estimate for Loop 1 can be obtained in a similar way (81114-811*100 =14). Both cycle estimates will be automatically inserted into the simulation code as shown in Fig. 3.11. In the software simulation process, the HLS estimate from the arbitrary loop bounds will be ignored and will instead be estimated as Eq. 3.5 with the instrumented TC.

```
int qsort_cycle = 0; // global variable
qsort_cycle += (81114-811*100);
……
int loop_1_cycle = 0;
while (i>=0) {  // loop 1
  ……
  if (L<R) {
    loop1_cycle += (81114-811*100);
    piv=arr[L];
    int loop_1_1_cycle = 0;
    while (L<R) { // loop 1.1
      loop_11_cycle += (811-400-400);
      int loop_111_tc=0;
      while (arr[R]>=piv && L<R){ // loop 1.1.1
        ……
        loop_111_tc++;
      }
      loop_11_cycle += (loop_111_tc-1)*4+4;
      ……
          <loop 1.1.2 is similar to loop 1.1.1>
      ……
    }
    loop_1_cycle += loop_11_cycle;
    ……
    if (end[i]-beg[i]>end[i-1]-beg[i-1]) {
      ……
  } }
  else {
    loop1_cycle += (4-0);
    ……
} }
qsort_cycle += loop1_cycle;
```

Figure 3.11: Quicksort code after inserting cycle estimation code

Table 3.4: HLS report for quicksort after code modification

Name	IL	II	TC
qsort_comp	401~8111401	-	-
Loop 1	4~81114	-	100
Loop 1.1	811	-	100
Loop 1.1.1	4	4	100
Loop 1.1.2	4	4	100

3.2.3.2 Estimation for Conditional Statements

The cycle estimate for most conditional statements will be automatically reflected because the cycle addition routine will only be processed when certain paths have been executed. However, some conditionals that exist between loops or functions will not be properly processed. For example, Vivado HLS does not provide a separate estimate for the if and the else part of the conditional statements in code A, B, and C.

Let us assume that a parent block p has multiple child blocks c, and HLS provides a range of cycle estimates for p: t_{pmin}~t_{pmax}. If some of c are conditional statements, multiple execution paths l will exist in p. For the quicksort example in Fig. 3.11, the dependency graph is again shown in Fig. 3.12. In Loop 1, two execution path can be found: l_1 along the if statement in line 22 and l_2 along the else statement in line 40. The execution time for every execution path l is guaranteed to have the minimum latency (t_{lmin}) of its parent's minimum latency (t_{pmin}):

$$t_{lmin} = t_{pmin} \tag{3.8}$$

If Eq. 3.8 was not true for an execution path l' (i.e., $t_{l'min} < t_{pmin}$), p can be executed faster by following path l'. This violates the assumption that t_{pmin} is the minimum latency of p. Thus, Eq. 3.8 is true.

Suppose that there are d ($d \in c$) child blocks along one of the execution paths l. Among d blocks, e have a known latency and f do not ($e, f \in d, e \cap f = \varnothing$). For example in Fig. 3.12, the blocks with a known latency in Table 3.4 belong to e (*e.g.,* Loop 1.1.1), and the rest of the blocks belong to f (*e.g.,* else part in line 40).

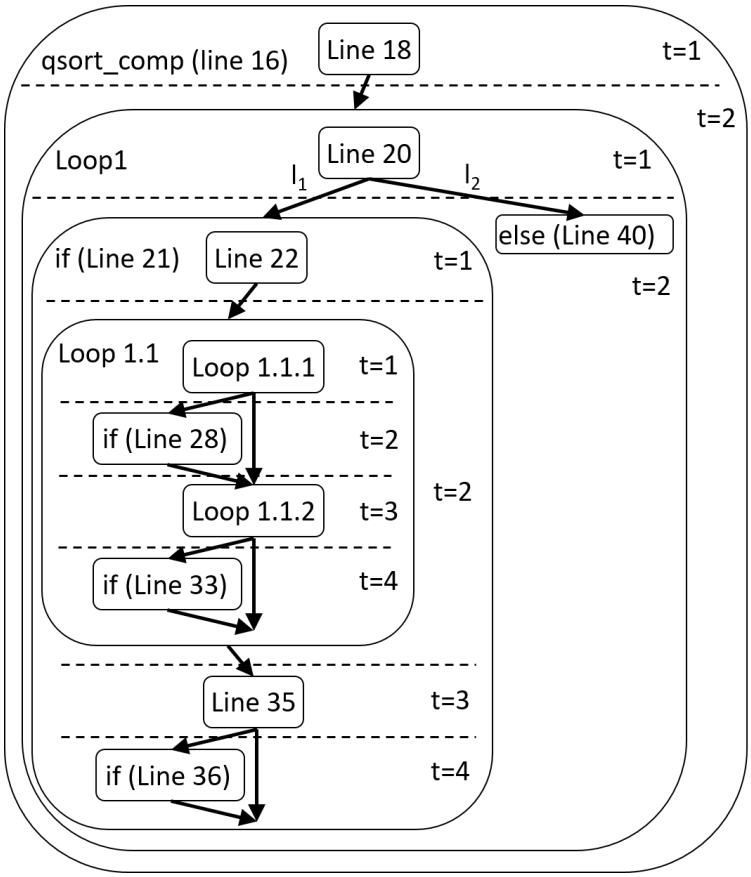


Figure 3.12: Dependency graph for $qsort\_comp()$ given in Fig. 3.11. The dotted line denotes synchronization barriers, and t denotes time frames divided by synchronization barriers. Reproduced from Fig. 3.2.

Then the minimum summed latency of unknown blocks f are:

$$(\sum_f t_f)_{min} = t_{lmin} - \sum_e t_{emax} = t_{pmin} - \sum_e t_{emax} \tag{3.9}$$

If an execution path contains a cycle estimation less than $(\sum_f t_f)_{min}$, the cycle difference is added to the simulated code. Although exact cycle is still unknown, the cycle for unknown blocks are at least partially compensated.

In the quicksort example, Loop 1 has a minimum latency of 4 ($t_{pmin} = 4$, Table 3.4). The execution path along the else part in line 40 has no known latency ($\sum_e t_{emax} = 0$). Thus, according to Eq. 3.9, the cycle estimate for this execution path ($(\sum_f t_f)_{min}$) is compensated as (4-0) cycle.

The instrumented code is shown in Fig. 3.11.

Since this is a minimum analysis, the cycle estimate may have some error. In practice, however, most of time-consuming blocks are in the form of loops, and Vivado HLS will report II of those loops. Thus, we can still obtain an accurate estimate with the proposed techniques, as will be evaluated in Section 3.2.5.2.

3.2.3.3 Estimation Result for Quicksort

After applying all the techniques explained in this section, our tool automatically generates the code presented in Fig. 3.11. Running software simulation on this instrumented code with N=131072 elements provides a prediction of 14.6M cycles, which is a 2.0% difference from the cycle-accurate RTL co-simulation (Table 3.5). The estimation time is four orders of magnitude faster. Compared to the baseline software simulation, the instrumented code incurred a small 2.5% time overhead. Also, the final instrumented version provides a 48% more accurate cycle estimate than the simple TC counting version.

Table 3.5: Cycle estimation and simulation time for quicksort

	Baseline	Simple TC counting[23]	+ imperfect loop & cond stmt est	RTL co-sim
Cycle Est	N/A	7.42M (-50%)	14.6M (-2.0%)	14.9M (0%)
Sim Time	0.0394s (1X)	0.0395s (1.002X)	0.0404s (1.025X)	817 (20736X)

3.2.4 External Memory Access Model for HLS Software Simulation

For our external memory access model, we assume an environment where there are several PEs on an FPGA, and they are connected to a single external memory through a bus and a DRAM controller. We do not model cache and instead assume that FPGA programmers explicitly control the internal BRAM as a scratchpad memory for higher performance. We also assume that the bus can accept several outstanding requests from PEs, as is the case with the AXI4 bus standard [125].

In Vivado HLS, the simulator will assume that data can be fetched from the external mem-

ory every cycle. As a result, the estimated DRAM BW will be ($\#ext\_port * bitw * f_{bus}$), where $\#ext\_port$ is the number of external memory port, $bitw$ is the data bitwidth, and f_{bus} by the frequency of the bus. As mentioned in the introduction, it will assume an ideal DRAM bandwidth of 25.6 GB/s for a design with $\#ext\_port$=2, $bitw$=512b, and f_{bus}=200MHz, but only 4.9GB/s~9.5GB/s is achieved during on-board testing [22].

A simple estimation model for a single DRAM transaction is [97]:

$$t_{MEM} = DSIZE/DBW + DLAT \tag{3.10}$$

where the $DSIZE$ is the length of each transaction, DBW is the DRAM bandwidth, and $DLAT$ is the latency. $DSIZE$ is obtained from the size field of the memcpy function. If the access is in the form of global array reference, it is found from the length of the consecutive array index from the loop bound of the loop iterator. Vivado HLS also informs programmers if it has flattened several loops to extend the consecutive access length. Similar to the cycle estimate routine, the length is used as a variable that is determined in simulation in run time. Note that if $DSIZE$ is larger than the maximum bus burst length (1KB for AXI), Vivado HLS will automatically divide it into several outstanding memory requests of maximum bus burst length.

We test two boards: ADM-PCIE-7V3 [2] and ADM-PCIE-KU3 [3]. The external bandwidth and the latency ($DLAT_R$ and $DLAT_W$) for both boards are shown in Table 3.6. These figures were measured using the methodology in [22]; but the listed bandwidth cannot be achieved when the access is not consecutive or if the bus bitwidth varies. The refined model will be explained in the following subsection.

Table 3.6: Read and write bandwidth and latency measured from ADM-PCIE-7V3 and ADM-PCIE-KU3 using the methodology in [22]

	Bandwidth		Latency	
	Read	Write	Read	Write
ADM-PCIE-7V3	9.5 GB/s	8.9 GB/s	542ns	356ns
ADM-PCIE-KU3	10.3 GB/s	9.6 GB/s	434ns	325ns

3.2.4.1 Bandwidth Model Refinement

We refine the DRAM bandwidth (DBW) model by assuming it is bounded by all components in the memory access pipeline: physical DRAM module bandwidth (DBW_P), DRAM controller bandwidth (DBW_C), and the bus data bandwidth (DBW_B). The effective bandwidth is computed as:

$$DBW = min(DBW_P, DBW_C, DBW_B) \tag{3.11}$$

- Physical DRAM Bandwidth

In ADM-PCIE-7V3, we found that when short (4B) but many (>1MB) discrete memory data are accessed, the effective external memory bandwidth is reduced to about 7% (0.053GB/s) of the bandwidth achieved with the consecutive memory access of the same length (0.75GB/s). Such a large reduction cannot be explained with Eq. 3.10 since the memory access latency would have been amortized due to several outstanding requests (data size > 1MB) made on the bus.

The reason can be found in the bandwidth constraint of the physical DRAM module. For illustration, let us consider outstanding requests to random memory location that is expressed in Fig. 3.13.

```
for( int i = 0 ; i < N ; i++ ){
#pragma HLS pipeline II=1
  bram[i] = dram[addr[i]];
}
```

Figure 3.13: Random memory access example in HLS. Variable *bram* is a *float*-type local memory and variable *dram* is an external port. Local memory *addr* has been pre-initialized with random memory location.

Assuming a closed row policy [77], the minimum time between access in a different address in a DRAM module is $t_{RC} = t_{RAS} + t_{RP}$, where t_{RC} is the row cycle time, t_{RAS} is the row address select (RAS) time, and t_{RP} is the RAS precharge time [58]. For the ADM-PCIE-7V3 and ADM-PCIE-KU3, the DRAM specification [70] states that $t_{RAS} = 36ns$, $t_{RP} = 13.5ns$, and $t_{RC} = 49.5ns$. In practical implementation, this theoretical latency is often exceeded, and extra

overhead is added on the controller side (t_{CO})—that is, the latency becomes t_{RC} + t_{CO}. From the random access experiment in Fig. 3.13, we found that the t_{RC} + t_{CO} is 76ns, which suggests that the controller overhead (t_{CO}) is (76ns-49.5ns)=26.5ns in ADM-PCIE-7V3. In ADM-PCIE-KU3, t_{CO} is calculated as (62ns-49.5ns)=12.5ns. Note that the average latency of 76ns for 4B of data is 0.053GB/s (=4B/76ns), which is the bandwidth obtained in the random access experiment.

We found that discrete memory access of stride 2, 4, 8, and 16 achieve similar external memory bandwidth as the random access described in Fig. 3.13. This suggests that the Xilinx controller does not concatenate outstanding requests of nearby memory addresses into a same burst DRAM access. Thus, we model strided access in the same way as the random access.

If requested data length is larger than the number of DRAM data pins $\#dq = 128b$, each $\#dq$ bits of data will be sent every $f_{ddr} = 1.33GHz$ cycle. In addition to t_{RC} + t_{CO}, the initial overhead includes RAS to column address select (CAS) time (t_{RCD}) added to CAS time (t_{CAS}), which is 13.5ns. Since the CAS signal is given in parallel to RAS signal, the transaction time for burst length len (unit of $\#dq = 64b$) is $t_{RCD} + t_{CAS} + len/f_{ddr} + t_{RP} + t_{CO}$.

In summary, the physical DRAM module access time (t_{D_P}) and DBW_P can be approximated as:

$$t_{D_P} = max(t_{RAS}, t_{RCD} + t_{CAS} + len/f_{ddr}) + t_{RP} + t_{CO} \tag{3.12}$$

$$DBW_P = len * \#dq/t_{D_P} \tag{3.13}$$

- Bus Data Bandwidth

If the kernel's external port data bitwidth ($bitw$) is less than the maximum bus data bitwidth supported (512b for SDAccel), the overall DRAM bandwidth might be limited by the bus data bandwidth (DBW_B). DBW_B is computed by multiplying $bitw$ (*e.g.,* float: 32b, uint512: 512b) by the frequency of the bus(f_{bus}):

$$DBW_B = f_{bus} \times bitw; \tag{3.14}$$

- DRAM Controller Bandwidth

Since we do not have knowledge of the inner workings of Xilinx's DRAM controller propriety, it is difficult to construct a good model for the DRAM controller bandwidth (DBW_C). Thus, we indirectly measure it by putting many outstanding long consecutive access requests (>512MB) with maximum bus data size (512b).[1] Since DBW_P and DBW_B achieve their peak values in this test method, the measured bandwidth, if lower than DBW_P and DBW_B, can be assumed to be DBW_C. As mentioned previously, the bandwidth for ADM-PCIE-7V3 is measured as 9.5GB/s for read and 8.9GB/s for write, which is smaller than $DBW_{Pmax} = 21GB/s$ and $DBW_{Bmax} = 12GB/s$. Thus, DBW_C is set to 9.5GB/s for read and 8.9GB/s for write.

The external bandwidth modeling for ADM-PCIE-7V3 and ADM-PCIE-KU3 described in this section is summarized in Table 3.7. Compared to the model in Eq. 3.10 [97] and the corresponding measurement result in Table 3.6, the proposed model more accurately predicts the effective bandwidth for non-consecutive external memory access or access with limited bus bitwidth. The quantitative evaluation of the proposed model will be shown in Section 3.2.5.

Table 3.7: External bandwidth modeling for ADM-PCIE-7V3 and ADM-PCIE-KU3

	ADM-PCIE-7V3	ADM-PCIE-KU3
DBW	$min(DBW_P, DBW_B, DBW_C)$	
DBW_P	$len * \#dq/t_{D_P}$ (see Eq. 3.13)	
DBW_B	$200MHz \times bitw$	$250MHz \times bitw$
DBW_C	9.5 GB/s(RD) 8.9 GB/s(WR)	10.3 GB/s(RD) 9.6 GB/s(WR)

3.2.4.2 Modeling Multiple PE Contention

In this section we explain the cycle estimation process when multiple PEs try to access the memory at the same time. Rather than using a time-consuming cycle-accurate transaction model that accounts for individual DRAM access, we propose a high-level estimation method for our fast software simulation-based framework. As an example, we consider the for loop (p) in Fig. 3.14 which contains three PEs ($c = c1, c2, c3$): *load()*, *qsort_comp()*, *store()*.

Since we assume a single external memory controller, contention among PEs may incur addi-

[1]Note that this measurement method itself coincides with the method used in [22], but [22] used this method to obtain DRAM bandwidth (DBW), whereas we use it to obtain one constraint (DBW_C).

```
for( int i = 0 ; i < N ; i++ ){ //p
  load(dram_portA, bram_load[0], ...); //c1
  qsort_comp(bram_load[1], bram_store[1], ...); //c2
  store(dram_portB, bram_store[0], ...); //c3
}
```

Figure 3.14: Code example for external memory access from multiple PEs, where *load()*, *qstore_comp()*, *store()* have no dependency with double buffering

tional delay. Suppose that a block p contains $c = c1, c2, ...$ PEs, and PE $c1$ (*e.g.*, *load()*) has m_{c1} external memory transfers. According to Eq. 3.10, the execution time of $c1$ (t_{c1}) would consist of memory transfer time ($t_{MEM_{c1}} = \sum_{m_{c1}}(DSIZE_{m_{c1}}/DBW_{m_{c1}} + DLAT)$) and computation time ($t_{COMP_{c1}}$). However, t_{c1} may become larger than ($t_{COMP_{c1}} + t_{MEM_{c1}}$) if the data transfer time of other PEs c (*e.g.*, *store()*) executing in parallel cannot be completely overlapped with the non-data-transfer time of PE $c1$, that is, ($t_{COMP_{c1}} + \sum_{m_{c1}} DLAT$). This is expressed in Eq. 3.15:

$$t_{c1} = \sum_{m_{c1}} \frac{DSIZE_{m_{c1}}}{DBW_{m_{c1}}} + \max(t_{COMP_{c1}} + \sum_{m_{c1}} DLAT, \sum_{c \neq c1} \sum_{m_c} \frac{DSIZE_{m_c}}{DBW_{m_c}}) \tag{3.15}$$

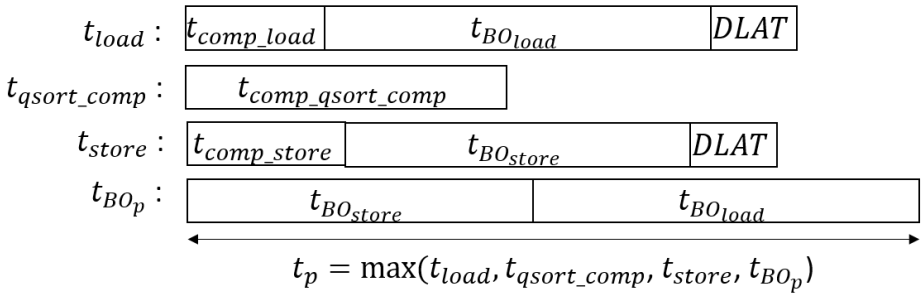


Figure 3.15: Computing cycle estimate (Eq. 3.18 and Eq. 3.19) for the example given in Fig. 3.14, in the for loop that contains $load()$, $qsort\_comp()$, $store()$ PEs

However, a major challenge in implementing Eq. 3.15 is that the software simulation process is usually sequential, and we cannot model the concurrent execution of memory access. For the example in Fig. 3.14, *load()*, *qsort_comp()*, and *store()* will execute in parallel in hardware, but the software simulation process will compute them serially. Then, the estimation routine in *load()* does not have the knowledge of $DSIZE_{store}$ or DBW_{store}, since *store()* has not been executed yet.

Even if the HLS software simulation process does allow multi-threaded execution in the future so that *store()* and *load()* can communicate in flight, having a mutex or semaphore for hundreds of PEs will significantly slow down the simulation and diminish the advantage of having a high-level prediction.

We solve this problem by carrying additional estimation results without $DLAT$ and resolving the summation part of Eq. 3.15 hierarchically. For each PE $c1$, we accumulate only the data term ($DSIZE/DBW$) of t_{MEM}, excluding the latency term ($DLAT$):

$$t_{BO_{c1}} = \sum_{m_{c1}} DSIZE_{m_{c1}}/DBW_{m_{c1}} \tag{3.16}$$

where $t_{BO_{c1}}$ is the minimum bus occupation cycle of PE $c1$. The insight behind this term is that the computation part and the DRAM latency might be overlapped in multiple outstanding requests, but at least $t_{BO_{c1}}$ is needed for each data in PE $c1$ to be transferred through the bus.

We also keep track of t_{c1}, as if there was only one PE:

$$t_{c1} = t_{COMP_{c1}} + t_{MEM_{c1}} \tag{3.17}$$

Next, we can hierarchically compute the cycle estimate for t_{BO_p} and t_p at p (the for loop) of the PEs:

$$t_{BO_p} = \sum_{c} t_{BO_c} \tag{3.18}$$

$$t_p = \max(\max_{c}(t_c), t_{BO_p}) \tag{3.19}$$

Eq. 3.18 represents that the bus occupation cycle of p is the sum of c's bus occupation cycles t_{BO_c}. This implies that the access to external memory is serialized. Finally, Eq. 3.19 [2] represents that the overall execution time is the maximum of c's individual latencies and p's minimum bus occupancy cycle. Unlike Eq. 3.2 which simply takes the maximum of nested PEs' cycles, the

[2]It is also possible to verify that inserting Eqs. 3.16, 3.17, and 3.18 into Eq. 3.19 will give the same equation if Eq. 3.15 is inserted into Eq. 3.2. Also, note that the max function in Eq. 3.19 can be replaced with an add function if the submodules are executed in serial rather than in parallel.

proposed equation can account for the case where *combined* DRAM access cycles of multiple PEs dominate the compute time. This can be more easily understood graphically in Fig. 3.15, where t_{BO_p} is the largest term in Eq. 3.19 because the system is memory bound.

Since the proposed estimation method does not account for the exact ordering of memory access, the estimate of the individual parallel-running module's execution time is not guaranteed to be exact. For example, in Fig. 3.15 it is uncertain when load() or store() submodule will exactly terminate. However, what this model does predict is the *combined* execution time of parallel-running modules. This is more important since the performance critical path and the stall reason is evaluated based on the most time-consuming module (Section 3.1.2). The accuracy evaluation will be presented in Section 3.2.5.2.

3.2.5 Experimental Results

3.2.5.1 Experimental Setup

For our evaluation platform, we use Alpha Data's ADM-PCIE-7V3 board [2] that has Xilinx's Virtex 7 690T FPGA and two Kingston's DDR3L-1333 SDRAMs [70]. For the FPGA synthesis, we use Xilinx's SDAccel 2016.2 [128] and Vivado HLS 2016.3 [130] software tools. Also, we have tested on Alpha Data's ADM-PCIE-KU3 board [3] that has Xilinx's Ultrascale KU060 FPGA and two of Kingston's DDR3L-1333 SDRAM. For this board, we used Xilinx's SDAccel 2016.4 synthesis flow.

The benchmark we used includes quicksort [49], Cholesky [99], convolutional neural network [135], matrix multiplication [68], logistic regression [6], decompression [78], and compression [34]. We also use four applications from MachSuite [105], which is a collection of common applications for accelerator environments. We exclude some applications that were similar to other benchmarks or had some functional correctness problem. The original code has been optimized with pipelining, double buffering, data reuse, longer DRAM access burst, and duplication.

3.2.5.2 Performance Estimation Accuracy

To evaluate the accuracy of our software simulation-based estimation model, we use the in-FPGA cycle extraction flow (Chapter 5) so that the exact cycle count of individual submodules can be obtained. For FIFO-based dataflow modules, we compare the number of active cycles obtained from SAN monitors (Section 5.3.3) with the estimated result. The applications are classified as having blocks with explicit synchronization (modules are only in serial or parallel) or parallel by dataflow (Section 3.1.1). Also, we classified the submodules inside each application as being mainly compute-bound or DRAM-bound.

The estimation error is obtained by averaging the absolute difference between on-board testing and the simulated results. It is shown in Table 3.8. The compute-bound modules are on average 1.4% different, and the DRAM-bound modules are on average 13.6% different. On ADM-PCIE-KU3, the averaged error rate is 2.5% for compute-bound and 22% for DRAM-bound on the same benchmarks.

Table 3.8: Average cycle estimation error (absolute difference between on-board cycle measurement and HLScope-S cycle estimation) for ADM-PCIE-7V3 and ADM-PCIE-KU3 boards.

Inter-	Appl	Compute-bound			DRAM-bound		
Comm	Name	#subm	7V3	KU3	#subm	7V3	KU3
Explct Synch	Qsort[49]	33	4.0%	5.1%	5	8.5%	8.4%
	Cholesky[99]	1	0.56%	1.7%	2	0.57%	20%
	ConvNN[135]	1	0.05%	0.53%	3	1.5%	10%
	Mat mul[68]	1	0.04%	0.04%	3	37%	39%
	Log reg[6]	3	1.4%	4.3%	1	0.76%	18%
	AES[105]	1	3.2%	3.2%	2	34%	32%
	KMP[105]	1	0.36%	3.4%	1	0.90%	2.2%
	NW[105]	1	0.03%	0.04%	2	43%	79%
	SpMV[105]	1	3.24%	11%	2	12%	10%
Paral by data flow	Vecadd	4	0.0%	0.0%	3	2.2%	7.5%
	Mat mul[68]	6	0.05%	0.0%	6	30%	34%
	Decomp[78]	3	1.3%	0.91%	2	0.17%	5.2%
	Compr[34]	3	0.23%	1.2%	3	4.9%	22%
AVG		-	1.1%	2.5%	-	13.6%	22%

Even with the proposed method, some inaccuracy still exists. The reason for inaccuracy in the compute part is the missing cycle information for some of the dynamic execution paths (Section 3.2.3.2). The inaccuracy in DRAM part is due to constructing a high-level behavioral model

(Section 3.2.4), including the multiple PE contention model, rather than simulating each memory access. In fact, the error rate is relatively higher for designs with more DRAM-bound modules executed in parallel. However, as mentioned in Section 3.2.4.2, our model is more focused on making accurate predictions for modules on the performance critical path, so that the performance debugging framework can correctly analyze the stall reason. The error rate of the submodule with the longest execution time among parallel-executing submodules is shown in Table 3.9. As expected, the error rate has decreased from 13.6% to 5.0% for ADM-PCIE-7V3. For ADM-PCIE-KU3, it decreases from 22% to 9.4%. This suggests that the proposed modeling is reliable for performance debugging.

Table 3.9: Cycle estimation error rate of the most time-consuming among parallel DRAM-bound submodules for ADM-PCIE-7V3 and ADM-PCIE-KU3 boards.

Inter-module Comm Type	Application Name	AVG($\|Dif\|$) 7V3	KU3
Explct Synch	Qsort[49]	8.5%	8.4%
	Cholesky[99]	0.17%	9.0%
	ConvNN[135]	0.31%	1.8%
	Mat mul[68]	12%	15%
	Log reg[6]	0.76%	18%
	AES[105]	8.0%	4.0%
	KMP[105]	0.90%	2.2%
	NW[105]	14%	14%
	SpMV[105]	12%	10%
Paral by data flow	Vecadd	2.7%	10%
	Mat mul[68]	2.0%	2.9%
	Decomp[78]	0.32%	0.32%
	Compr[34]	3.6%	26%
AVG		5.0%	9.4%

3.2.6 Software Simulation Flow Overhead

The software estimation overhead consists of two parts: first, the code instrumentation for hardware cycle estimation and second, overhead in the software simulation process. The code instrumentation takes 5-98 seconds. The software simulation overhead depends on the computational complexity of the original code compared to the inserted code. The comparison between the original software simulation time and the instrumented code software simulation time is shown in

Table 3.10. It shows that overhead is 4% on average. The code instrumentation and simulation overhead is approximately two orders of magnitude faster than the FPGA bitstream generation. This shows that the proposed flow is suitable for rapid analysis.

Table 3.10: Time overhead of SW simulation flow. Consists of code instrumentation and additional SW simulation time (unit:s).

Appl Name	Instr Time	SW Sim Unmodif	Instr SW Sim Est	Bitstr Gen
Qsort[49]	27	0.026	0.029 (1.12X)	1h27m
Cholesky[99]	5	0.083	0.089 (1.07X)	36m
ConvNN[135]	64	60	64 (1.07X)	1h47m
Mat mul[68]	43	62	65 (1.05X)	2h23m
Log reg[6]	36	563	564 (1.0X)	1h34m
AES[105]	51	62	65 (1.05X)	3h29m
KMP[105]	9	128	129 (1.01X)	1h21m
NW[105]	8	56	59 (1.05X)	1h38m
SpMV[105]	12	7.3	7.4 (1.01X)	2h5m
Vecadd	37	0.20	0.21 (1.05X)	1h30m
Mat mul[68]	76	125	125 (1.0X)	4h12m
Decomp[78]	98	0.80	0.80 (1.0X)	1h28m
Compr[34]	91	19	20 (1.05X)	5h35m
AVG		(1.0X)	(1.04X)	

3.2.7 Comparison with Related Work

Although design space exploration (DSE) tools such as Aladdin [111], Lin-analyzer [138], COMBA [137] consider various factors such as resource contention (e.g., number of BRAM accesses per cycle) and data dependence, it is not guaranteed that HLS tools will generate a design that matches their prediction. The HLS tools are known to update its scheduling, binding, and allocation algorithms—this may result in loop's II and IL to be changed in future HLS versions. Even for a same HLS version, HLS tools may use a hardware IP (e.g., DSP) of different latency depending on the FPGA platform or clock frequency. HLScope-S, on the other hand, builds the performance estimation model based on the HLS synthesis report. This ensures that the performance model reflects the actual architecture implemented by the HLS tool and allows the model to easily adapt to any algorithmic changes in the HLS kernel.

Similar to HLScope-S, Aladdin [111] and Lin-analyzer [138] can be used to provide cycle

estimate for programs with dynamic behavior since they utilize the instruction trace generated in C simulation. However, collecting instruction trace takes a relatively long time compared to our high-level cycle estimator based on native C software simulation. For example, instruction trace generation for sorting 131072 elements in Aladdin took 188 seconds, whereas HLScope-S took 0.0404 seconds (Table 3.5).

LegUp HLS [14] provides a flow to obtain hardware cycle estimation by profiling the software for the number of times each basic block was executed. Then it multiplies the obtained execution number by the basic block cycle given by Legup HLS. Our work, on the other hand, is more focused on how to instrument the code to extract unknown basic block cycles that are apparently hidden by HLS. Thus, HLScope-S only requires high-level synthesis reports like Table 3.4 and does not need to extract HLS LLVM compiler's internal data, which could be proprietary information.

3.2.8 Conclusion

In this chapter, we have described a high-level cycle estimation methodology for input-dependent FPGA designs using the HLS software simulation process. A source-to-source code instrumentation technique was used to automate the cycle extraction process. Also, we provided a high-level estimation model for DRAM access for a typical bus-based architecture with outstanding requests and multiple PEs. Experiments showed that our estimation has an error rate of 1.1% in compute-bound modules and 5.0% in performance-critical DRAM-bound modules, with a modest 4% time overhead in software simulation.

CHAPTER 4

Cycle-Accurate Software Simulator for HLS

Although the performance estimation flow (HLScope-S) in Chapter 3 accurately reflects the input-dependent behavior of an HLS design, it has some limitations. The issue is that its estimation routine depends on the HLS software simulation. As will be explained in Section 4.2, HLS software simulators sometimes produce incorrect result in the cases where data ordering is not maintained, module latency is ignored, or feedback path exists. Incorrect simulation often leads to incorrect performance estimation—for example, the performance estimation for a matrix multiplication benchmark with data ordering and feedback problems has 60% error rate (Table 4.8) compared to a RTL simulation result.

In this chapter, we present a new HLS simulation flow named FLASH. The main idea of FLASH is to extract scheduling information from the HLS tool and automatically construct an equivalent cycle-accurate simulation model. We show that the correctness of the simulation and the accurateness of the performance estimation are both achieved by the proposed process. FLASH runs slightly slower than HLScope-S because the code transformation and the simulation are performed at the granularity of individual C statements—however, this can be justified by the fact that FLASH achieves cycle-accuracy while running three orders of magnitude faster than the RTL simulation. The formal problem statement and the details of the automated code transformation process are described in the remainder of this chapter.

4.1 Introduction

Although the field-programmable gate array (FPGA) has many promising features that include power-efficiency and reconfigurability, the low-level programming environment makes it difficult

for programmers to use the platform. In order to solve this problem, many high-level synthesis (HLS) tools such as Xilinx Vivado HLS [31, 130] and Intel OpenCL HLS [62] have been released. These tools allow programmers to design FPGA applications with high-level languages such as C or OpenCL. This trend is reinforced by recent efforts on FPGA programming with languages of higher abstraction—such as Spark or Halide [110, 115, 134].

Even though such progress has been made on the design automation side, a large semantic gap still exists on the simulation side. Programmers often need to use low-level register-transfer level (RTL) simulators (e.g., ModelSim [86], NCSim [11], or VCS [117]) or on-board emulators (e.g., Palladium [12], Veloce [87], or Zebu [118]) and try to map the result back to HLS. The result is often incomprehensible to those who are not FPGA experts. Moreover, low-level RTL simulation takes a very long time. Some work has been done to automate hardware probe insertion from the HLS source file [23, 51–53, 89, 90, 120]; however, this work requires regeneration of the FPGA bitstream if there is a change in the debugging point, and the turnaround time is often in hours. On-board emulators also have a similar problem and take a long time for the bitstream generation.

These problems can be partially solved by the software-based simulators provided by HLS tools. The HLS software simulators compile the C or OpenCL source code for native execution on the host machine. It takes little time to reconfigure the debugging points, and no semantic gap exists between the simulation and the design. However, a well-known shortcoming of these simulators is that most of them do not provide performance estimation. In addition, we found a critical deficiency—they sometimes provide *incorrect* results.

An example can be found in the molecular dynamics simulation [33] (Fig. 4.1). Multiple distance processing elements (Dist PEs) filter out faraway molecules above threshold and send them to Force PE. The pruned molecules will create a bubble (empty data) in the FIFO, and Force PE will process only the valid data (after non-blocking read) in the order they are received from any of the FIFOs. However, if the modules are instantiated in the order of (Dist PE1, PE2, . . . Force PE) in the source file, Vivado HLS software simulator will finish the simulation of Dist PE1 first, followed by Dist PE2, and so on. As a result, by the time the Force PE is simulated, the bubbles in the FIFOs are completely removed, and the Force PE output ordering can be entirely

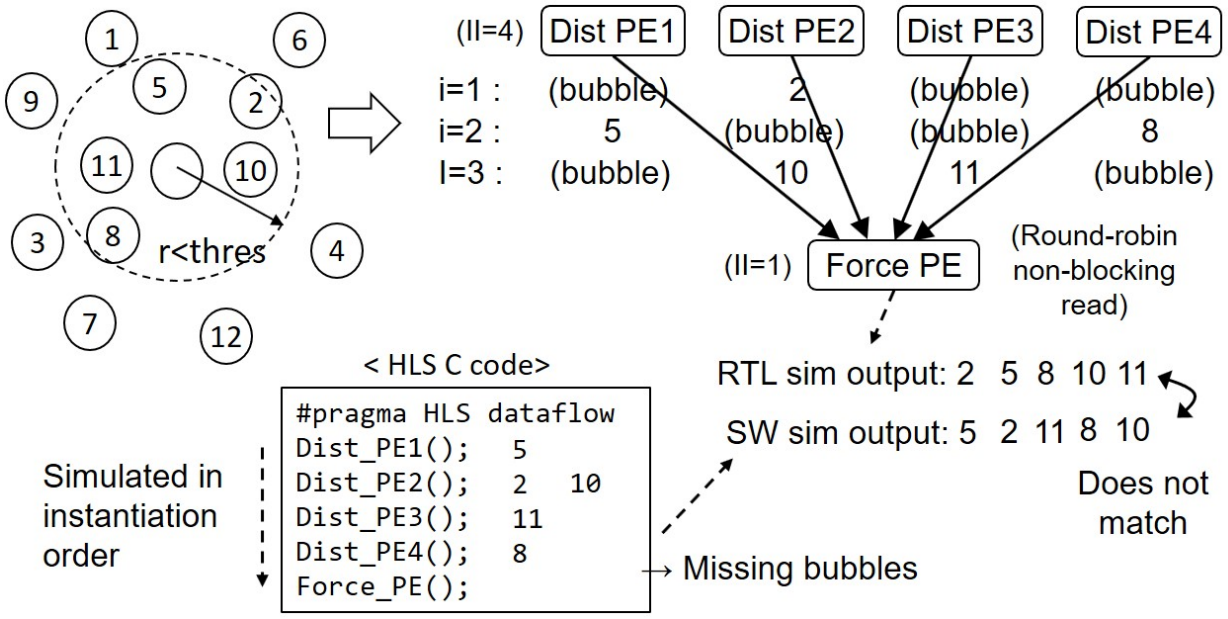


Figure 4.1: Molecular dynamics simulation [33]

different from the RTL simulation. If one was trying to quickly trace the source of a problem that was observed in the output of a RTL simulation, the person would not be able to reproduce the problematic state in the software simulation.

Another problematic example can be found in the artificial deadlock situation [43], which occurs when the depth of the FIFO is smaller than the latency difference among modules (details in Section 4.2.2). The first issue is that the HLS software simulator cannot detect the deadlock situation and proceeds as if there is no problem with the design. The second issue is that after we apply a transformation to remove the deadlock, the HLS tool also cannot simulate the amount of performance degradation (Section 4.8.3). We also found a problem in the simulation of feedback loops where the feedback data is ignored by the HLS tool (Section 4.2.3).

The primary reason for the incorrect simulation result is that HLS software simulators do not guarantee cycle accuracy. The comparison between the software simulator of the two most popular ([75]) commercial FPGA HLS tools, Xilinx Vivado HLS [130] and Intel OpenCL HLS [62], is presented in Table 4.1. Vivado HLS assumes unlimited FIFO depth which makes it difficult to accurately model FIFO fullness/emptiness. Also, the sequential simulation execution model prevents correctly simulating designs with feedback loops (Section 4.2.3). Intel OpenCL HLS

Table 4.1: Comparison of the software simulation of Xilinx Vivado HLS [130] and Intel OpenCL HLS [62]. Undesirable characteristics are in bold.

	Xilinx Viv HLS C Sim	Intel OpenCL HLS Sim
FIFO depth	**Unlimited**	Exact
Exec model	**Sequential**	Concurrent
Feedback	**Not supported**	Supported
Sim speed	∼5 Mcycle/s	**∼1 Mcycle/s**
Sim order	Deterministic	**Non-deterministic**
Max # mods	No limit	**256**
Cycle-acc	**Not cycle-accurate**	**Not cycle-accurate**

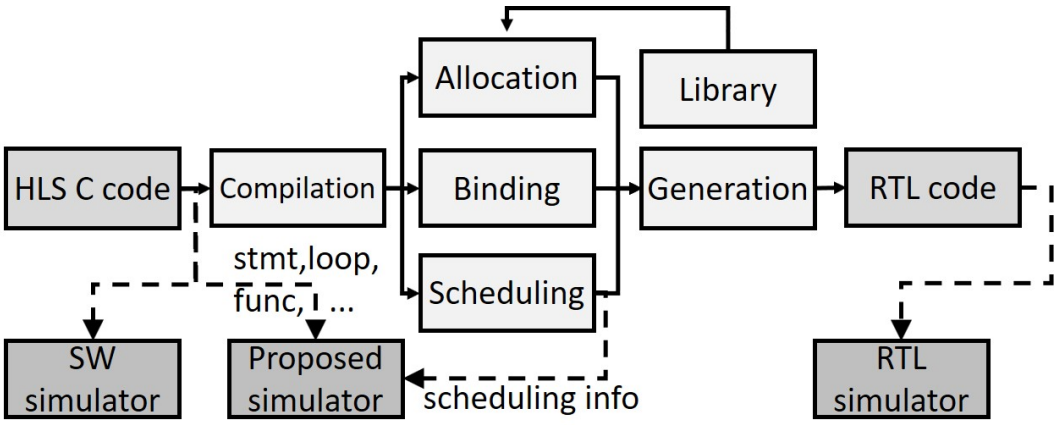


Figure 4.2: HLS design steps [40] and simulation flows

simulates about 5X slower than Vivado HLS, but it correctly simulates the FIFO depth. The tool assigns a thread to each module for concurrent simulation; however, the execution order of the threads is not deterministic and may produce different results in different simulation runs for cases in Section 4.2.

HLS design steps and conventional simulation flows are shown in Fig. 4.2. A software simulator runs fast but provides no cycle estimation and may have the correctness problem. An RTL simulator is accurate but runs slow since it incorporates low-level implementation details. Our solution to these problems is based on the idea that it may be possible to tackle both problems by simulating based on the scheduling information. It would be faster than the RTL simulation without the allocation / binding information and the component libraries; and it would solve the correctness problem of the software simulation and provide accurate performance estimation with its cycle accuracy.

Although simulating solely based on the scheduler output (LLVM IR + scheduling information) is a possible option, we have instead decided to simulate in C syntax and augment it with scheduling information. The reason is that we wanted to raise the simulation abstraction level to further accelerate the simulation process and also make it easier for programmers to understand what is being simulated. To our knowledge, this is the first HLS-based software simulation flow that takes such an approach.

By taking such an approach, however, several challenges were encountered (elaborated on in Section 4.3). One problem is how to guarantee cycle-accuracy of untimed C statements. Moreover, correctly simulating the task-level and pipelined parallelism that is inherent in hardware (and the corresponding RTL simulation) in sequential C semantics is a significant challenge.

In this chapter we propose FLASH—an HLS software simulation (HSS) flow that addresses these challenges. We describe transformations that allow cycle-accurate simulation of FIFO communication (defined in Section 4.3). Also, a method will be presented to simulate task-level and pipelined parallelism with C semantics. These steps will be described in Section 4.4.

In order to simulate pipelined parallelism, variables needs to duplicated to match the depth of a loop pipeline (explained in Section 4.4.2.1). However, this results in a redundant data copy which slows down the simulation. We propose optimization techniques to reduce this overhead in Section 4.5.

We obtain the scheduling information from the HLS synthesis report and automatically generate a new simulation code based on the information. The new simulation code was made compatible with the conventional HLS software simulator for easy integration with the existing tool. The overall flow is described in Section 4.6.

FLASH also provide correctness and performance debugging support for programmers. In order to reduce the debugging effort of detecting deadlocks or stalls, we provide a set of source-level directives. Also, the debugging time is shortened by allowing variables to be added to the capture list in the middle of simulation. This will be explained in Section 4.7.

The contribution of this chapter can be summarized as follows:

- We show that simulating based on the scheduling information can help solve the correctness issue of HLS software simulators and rapidly provide accurate performance estimation.
- We develop a framework that allows fast cycle-accurate simulation of an HLS design. Several code transformation techniques have been presented to enable this process. Moreover, optimizations were proposed to accelerate the simulation speed.
- We propose unique debugging features for HSS.

Our current initial version is based on the Vivado HLS tool, but we hope to extend our work to the Intel HLS tool if it provides detailed internal scheduling information in the future.

4.2 Problem Description and Motivating Examples

In this section we describe three classes of problems that cause current HLS tools to produce incorrect software simulation results. The problems are demonstrated with relevant examples in the literature.

4.2.1 Data Ordering Problem

The problem of incorrect output ordering in the HSS for molecular dynamics simulation was presented in our introduction. In this section we discuss the cause of this problem in more detail. Fig. 4.3 shows the timing diagram of the FIFO transactions among Dist PE1, Dist PE2, and Force PE in Fig. 4.1. Dist PE1 and Dist PE2 communicate with Force PE through FIFO F1 and F2 respectively. Consider a case where data (2) is written to F2 before data read from F1 and F2, and F1 is written (data 5) afterwards (illustrated in the RTL simulation part of Fig. 4.3). At the time of the first F1 non-blocking read attempt (t_{F1_RD1}), the first F1 write (t_{F1_WR1}) has not yet occurred ($t_{F1\_RD1} < t_{F1\_WR1}$), and the successful F2 read precedes the successful F1 read ($t_{F2\_RD1} < t_{F1\_RD2}$).

In the Vivado HLS software simulation however, data (5) is available in the first read attempt to F1, because Dist PE1 is evaluated entirely before Dist PE2 and Force PE. That is, unlike the

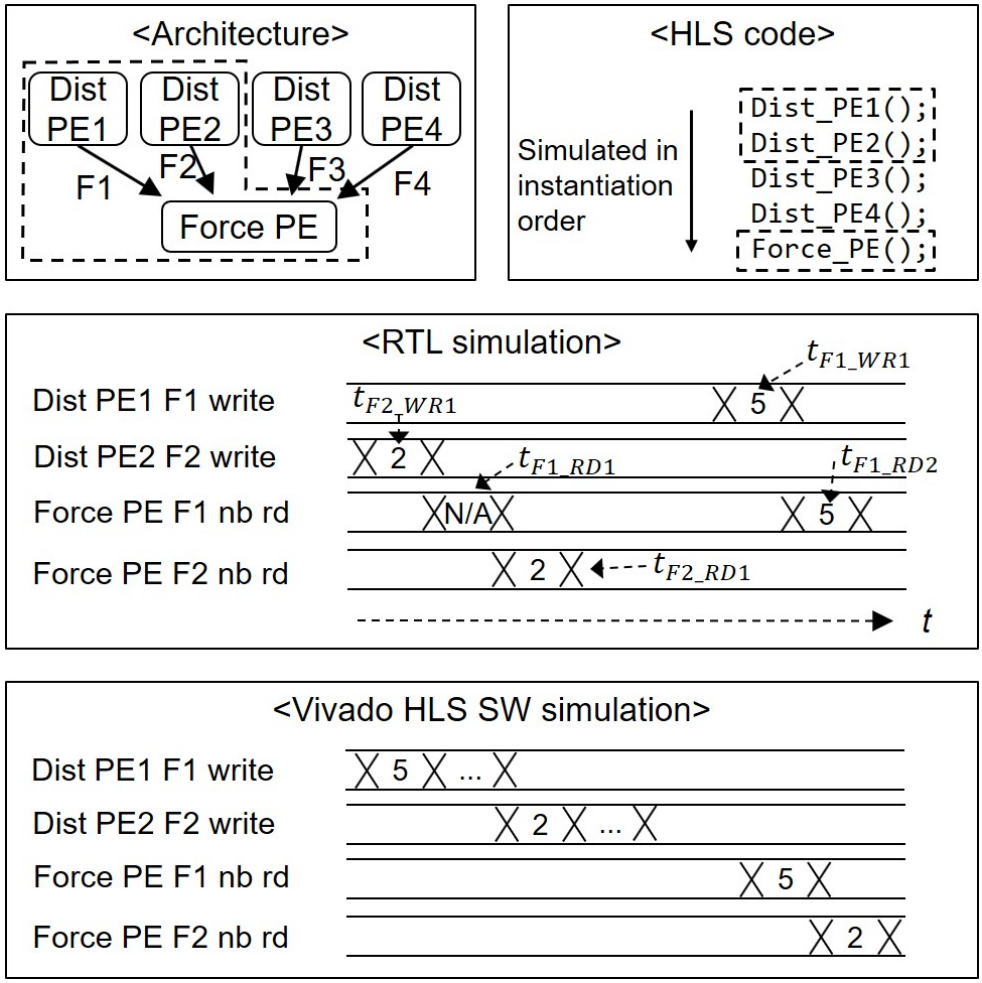


Figure 4.3: Timing diagram of the molecular dynamics simulation in Fig. 4.1 (FIFO transactions among only Dist PE1, Dist PE2, and Force PE are shown)

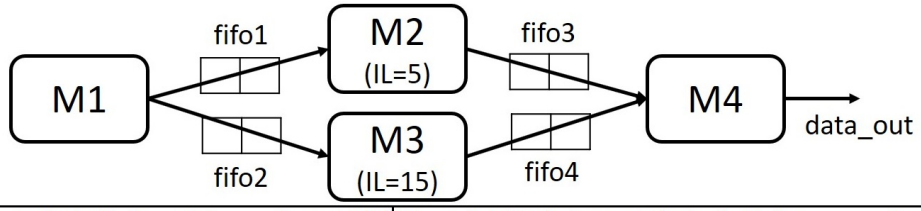
RTL simulation, the first F1 write has already occurred before the first F1 read attempt ($t_{F1\_RD1} > t_{F1\_WR1}$). As a result, the successful F2 read happens after the successful F1 read ($t_{F2\_RD1} > t_{F1\_RD2}$). The ordering of the data processed at Force PE is not maintained. If the HSS has evaluated the FIFO write correctly before each FIFO read attempts (i.e., $t_{F1\_RD1} < t_{F1\_WR1} < t_{F1\_RD2}$ and $t_{F2\_WR1} < t_{F2\_RD1}$), this problem would not have occurred. In the Intel HLS, the simulation order of the producer modules is undetermined, and a similar data ordering problem occurs.

As demonstrated in the example, the HLS software simulator should evaluate the FIFO writes before each non-blocking read attempt in the same order as in the RTL simulation. If not, data ordering problem may occur. The *data ordering problem* is defined as a case where a consumer module M_C is reading data in a non-blocking fashion from multiple producer modules M_P through FIFOs, and the order of data processed at M_C in the RTL simulation is not maintained in the HSS.

4.2.2 Artificial Deadlock and Stall

Consider an example in Fig. 4.4 where the module `M2` has a latency of 5 and `M3` has a latency of 15. All FIFOs have a depth of 2. After `M2` has produced two output elements, `M4` cannot consume any of them because `fifo4` is still empty due to the long latency of `M3`. Because of back pressure from `M2` and `fifo3`, `fifo1` becomes full. Then `M1` will stop producing output to `fifo2` because `fifo1` and `fifo2` have to be written in the same cycle. `fifo2` will eventually become empty, which blocks the pipeline of `M3`. Even though `M3` has consumed some remaining data in `fifo2`, `fifo4` is still empty because of `M3`'s long latency. Then none of the modules can do any further useful work, and the circuit deadlocks. This is called an artificial deadlock. The artificial deadlock is caused by the mismatching latency of multiple datapaths and inadequate FIFO depth to balance the latency difference [43]. This can be observed in real applications, such as the dataflow-based architecture for stencil computations in [17] that contains various modules and FIFOs with different latencies and depths.

In order to reproduce the deadlock situation, an HLS software simulator should create the output data after reading input with a delay that reflects the module latency. However, existing




```
01 void M1(stream<int>& f_out1,.. f_out2){
02   for (int i = 0; i < N/16; k++) {
03     for (int j = 0; j < 16; k++) {
04 #pragma HLS pipeline II=1
05       f_out1.write(16*i+j);
06       f_out2.write(16*i+j+10);
07 } } }
08
09 void M2(stream<int> & f_in, ... f_out){
10   for(int i = 0; i < N; i++){
11 #pragma HLS pipeline II=1
12     int data = f_in.read();
13     int temp = data*711;
14     f_out.write(temp);
15 } }
16
```

```
17 void M3(stream<int>& f_in, ... f_out){
18  for(int i = 0; i < N; i++){
19 #pragma HLS pipeline II=1
20     int temp = f_in.read();
21     f_out.write((int)((float)temp*3));
22 } }
23 // M4 is omitted
24
25 void toy_mpath(int data_out[N]){
26   stream<int> fifo1,fifo2,fifo3,fifo4;
27 #pragma HLS stream var=fifo1,.. depth=2
28 #pragma HLS dataflow
29   M1(fifo1, fifo2);
30   M2(fifo1, fifo3);
31   M3(fifo2, fifo4);
32   M4(fifo3, fifo4, data_out);
33 }
```

Figure 4.4: Structure and code for motivating example `toy_mpath`

HLS software simulators evaluate each iteration of a loop as if the data is instantaneously passed from input to output. Thus, the latency among different datapaths is not simulated, and the artificial deadlock does not occur. As a result, even after running a HSS, the user is unaware of a potential problem that might occur during actual on-board execution.

We will refer to this problem as the *module latency problem*. Suppose that a module has a sequence of C FIFO read and write statements $cstmt_1$, ... $cstmt_c$, ... $cstmt_C$. If $cstmt_c$ is a blocking read/write, multiple read/write attempts may be performed before the read/write is completed. If non-blocking, read/write is always completed on the first attempt. We assume that the HLS tool has scheduled a delay of $delay_c$ cycles between the completion of $cstmt_c$ and the first attempt of $cstmt_{c+1}$. This delay reflects the computation latency. The module latency problem is defined as a case where HSS fails to simulate $delay_c$ between the first attempt of $cstmt_{c+1}$ and the completion of $cstmt_c$ for some of $c = 1 ... C - 1$.

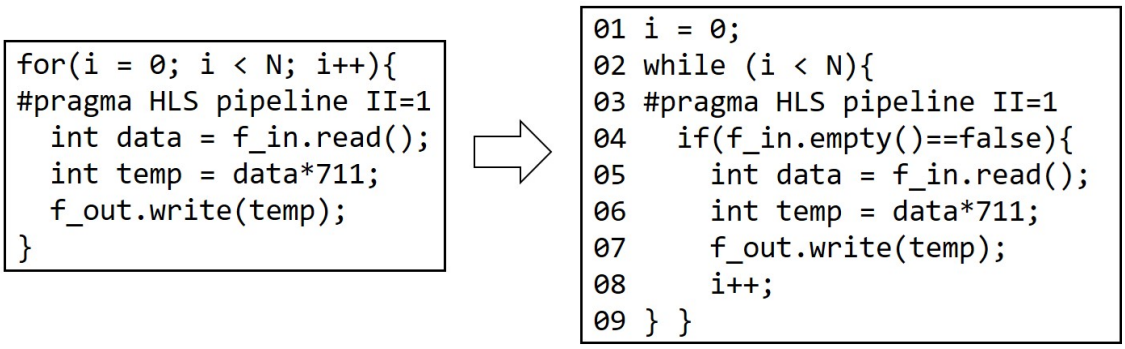


Figure 4.5: Modified code of `M2` in Fig. 4.4 to avoid artificial deadlock

The second problem was found after we applied code transformation[1] to avoid the deadlock. Fig. 4.5 shows the transformation for `M2` in Fig. 4.4. If the input FIFO is empty, a bubble is inserted into the pipeline (line 4)—this allows the pipeline to keep processing the already-read data even if there is no additional input. A similar transformation is applied on `M3`. The deadlock situation is removed since `M4` can now receive the output from `M3`.

However, even though the deadlock was avoided, it still takes several cycles for the module to produce an output after reading the input data. This causes a delay that we call *artificial stall*. Since HLS tools do not consider the delay due to the latency of a module, such performance degradation cannot be simulated.

4.2.3 Missing Data from Feedback Path

As mentioned previously, the Vivado HLS software simulator evaluates functions in the order in which they are instantiated in the source code. This causes a problem if a feedback path exists that passes data from later instantiated functions to earlier ones. At the time earlier functions are simulated, the data would not be available. As a result, Vivado HLS simulates the program as if the feedback FIFOs are always empty. We will refer to this problem as the *feedback problem*. The

[1] Alternative solutions are presented in [43], but they require modification to the HLS scheduling, allocation, and binding kernels. It is also possible to adequately increase the buffer size that can prevent the deadlock situation [74] (the Intel HLS compiler has this functionality). However, since the efficiency of the solution for avoiding artificial deadlock is not the focus of this work, we apply a solution that only requires simple source-to-source transformation of the loops without an elaborate analysis of the whole circuit. This method cannot be used to resolve all deadlocks—such as the one that is caused by circular wait.

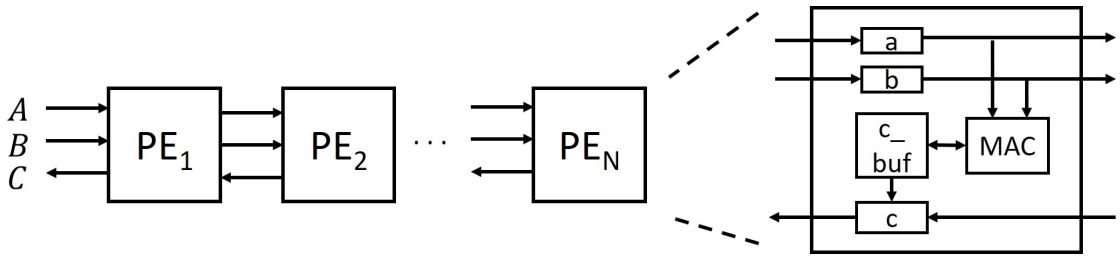


Figure 4.6: Matrix multiplication with linear systolic array architecture

feedback problem refers to the case where at the time of a read from a FIFO in a feedback path, the data in the FIFO buffer in the HSS does not match the data in the RTL simulation. The Intel HLS simulator can simulate the feedback data from blocking read correctly because a thread simulating each module can wait for others to pass the data. However, it is not guaranteed that the feedback data from non-blocking read will arrive at the right timing.

We demonstrate this problem with matrix multiplication example ($C = A \times B$) in linear systolic array architecture [36, 68]. As shown in Fig. 4.6, each PE computes one column of the matrix C ($C_{ij} \mathrel{+}= A_{ik} * B_{kj}$). Data from the matrix A and B are fed into the array in the forward direction, while the results of matrix C are collected in the backward direction. If the modules are instantiated in the order of PE_1, PE_2, ..., and PE_N, Vivado HLS will simulate PE_1 assuming the FIFO for C is always empty, and this will cause the tool to produce incorrect results.

4.3 Problem Statement and Challenges

Before we provide the problem statement, we will define the concept of *FIFO communication cycle-accurate* (FCCA) simulation. The FIFO communication refers to the FIFO-accessing expressions in the source code (listed in the second column of Table 4.2). A FIFO communication statement refers to a statement with a FIFO-accessing expression. Let us assume that a FIFO communication has been evaluated in HLS software simulation (HSS) at cycle t. We declare that the FIFO communication is simulated *cycle-accurately* if the FIFO input value and the FIFO output value of the FIFO APIs (FAPIs) in the HSS match the FIFO input ports (`din`, `rd_en`, `wr_en`) and the FIFO output ports (`dout`, `empty`, `full`) [127] in RTL simulation at the same cycle t.

Table 4.2: The FIFO communication in the C source code, the corresponding FIFO IP RTL ports and C variables, and the corresponding FLASH simulation code (Vivado HLS FIFO APIs [130] and FIFO IP RTL ports [127] are in `monospace font`)

Description	FIFO comm in source code	RTL ports & C variables	FLASH simulation code
Blocking read	rdata = fifo.`read`()	1 == !`empty`, `rd_en` = 1, rdata = `dout`	(stall cond: fifo_rnum==0), test = (..rnum!=0), rdata = fifo_arr[fifo_rptr++]; fifo_rnum−−;
Non-blk read	test = fifo.`read_nb`(rdata)	test = !`rd_en` = !`empty`, rdata = `dout`	
Blocking write	fifo.`write`(wdata)	1 == !`full`, `wr_en` = 1, `din` = wdata	(stall cond: fifo_wnum==0), test = (..wnum!=0), fifo_arr[fifo_wptr++] = wdata; fifo_wnum−−;
Non-blk write	test = fifo.`write_nb`(wdata)	test = !`wr_en` = !`full`, `din` = wdata	
Empty	test = fifo.`empty`()	test = `empty`	test = (fifo_rnum == 0)
Full	test = fifo.`full`()	test = `full`	test = (fifo_wnum == 0)

That is, the C variables and the RTL signals have the same value as described in the third column of Table 4.2 at the same cycle t. The FIFO input value of the FAPIs refers to the value of "wdata", and the FIFO output value of the FAPIs refers to the value of "test" and "rdata" in Table 4.2. If all FIFO communication in a C source code is simulated cycle-accurately in HSS, the simulation will be FCCA.

For example, the non-blocking read expression, "test = fifo.`read_nb`(rdata)", is cycle-accurately simulated if the value of "rdata" matches `dout` and "test" has the toggled value of `empty` at the same cycle as the RTL simulation.

We assume that we simulate an HLS design that is composed of multiple finite-state machine (FSM) modules (inferred from C functions) that use streaming FIFOs to communicate between modules. The modules execute concurrently (with directive `#pragma HLS dataflow`).

Our main goal is to construct an HLS software simulator that is FCCA. The input and output of the simulator is defined as follows:

Input: (1) An HLS C source code (2) Scheduling information (3) Input data of the design

Output: Output data of the design

The scheduling information is defined as the information on the FSM state transition and the assigned FSM state of FIFO communication, conditional statements, and loop statements.

FCCA simulator does not have the data ordering, module latency, and feedback problems that were described in Section 4.2.

We will prove that an FCCA simulator does not have the data ordering problem. Recall that the data ordering problem occurs when a consumer module M_C is reading data in a non-blocking

fashion from multiple producer modules M_P through FIFOs, and the order of data processed at M_C is not maintained in the HSS. Since FIFO communication is cycle-accurate, the relative ordering of FIFO reads and writes matches that of the RTL simulation. That is, the number of FIFO reads and writes and the data before each FIFO read match that of the RTL simulation. At the first FIFO read attempt of M_C, the first data processed in M_C matches the RTL simulation, because M_P's FIFO writes before the first FIFO read attempt matches the RTL simulation. The data ordering problem has not occurred in the first FIFO read attempt (*base case*). Let us assume that the data ordering problem has not occurred up to the k'th FIFO read attempt. Among the k attempts, we assume there were s successful reads. If a FIFO was empty after k'th FIFO read attempts and no FIFO write has happened until $(k+1)$'th FIFO read attempts, data ordering problem has not occurred in the $(k+1)$'th read attempt, because no new data was processed between k'th and $(k+1)$'th read. If FIFO was not empty at $(k+1)$'th read, the data must be from the $(s+1)$'th FIFO write. This is because FIFO provides the data to the reader in the order data was written. The data in $(s+1)$'th FIFO write in HLS simulation matches that of the RTL simulation, because the relative ordering of FIFO write is maintained in the FIFO communication cycle-accurate simulator. Thus, the data ordering problem has not occurred in the $(k+1)$'th read attempt (*induction step*).

We have explained that the feedback problem occurs because the relative ordering was not kept among the writes and the reads to the FIFO in the feedback path. Since the ordering is maintained in an FCCA simulator, the feedback problem also does not occur.

Since all FIFO transactions occur at the same cycle as in the RTL simulation, the delay between any consecutive pair of FIFO reads and writes of a module matches that of the RTL simulation. Thus, the FCCA simulator does not have the module latency problem.

In addition to the main goal of achieving cycle-accurate FIFO communication, the simulator should be able to provide the execution cycles of each module to help programmers apply performance optimization. Also, if the modules deadlock, the simulator should provide the content of the registers (e.g., the state of a module or the number of empty FIFO buffers) for debugging purposes. Moreover, the simulation code should be semantically similar to the source code as much as possible (as opposed to being a low-level code such as RTL), so that users can easily understand

what is being simulated.

With such complicated requirements, several challenges arise:

- **Challenge 1 : FCCA simulation**

  It is difficult to discover the exact cycle when statements are executed since the information given by the HLS tool is very limited. For example, the Intel OpenCL HLS tool only provides loop initiation intervals (II). The Vivado HLS tool provides slightly more information—it provides a list of LLVM IR and the corresponding state of a FSM. However, mapping such low-level representation (e.g., lines 27–31 of Fig. 4.7) back to the original C code is a difficult task.

  Also, even if the schedule of all operations is known, the FCCA simulator has to *selectively* execute statements that correspond to a particular FSM state at each cycle. Another issue is that the simulator needs to model the FSM stalls due to FIFO being empty or full. Furthermore, the content of the variables in the previous state has to be available, and the updated variables have to be stored for the next state simulation.

- **Challenge 2 : Simulation of parallelism**

  HLS designs have multiple levels of parallelism including task-level parallelism and pipelined parallelism. Cycle-accurately simulating parallelism in a C syntax becomes a difficult task because the value of variables and the simulation order of the statements become different from that of the source code. For example, if the statement in line 21 of Fig. 4.4 is executed 14 cycles after the statement in line 20, we would need to simulate line 21 with a “temp" value that corresponds to iteration i and line 20 with that of iteration $i + 14$ in a single cycle.

- **Challenge 3 : Loop and function simulation**

  We would need to construct an equivalent model of high-level C semantic such as loops and functions.

4.4 Automated Code Generation for Rapid Cycle-Accurate Simulation

In this section we provide a solution to each challenge in Section 4.3 and describe our proposed automated simulation code generation flow. For illustration, we will use the `toy_mpath` example (Fig. 4.4) after modifying the source code to avoid the deadlock as shown in Fig. 4.5.

4.4.1 FIFO Communication Cycle-Accurate Simulation

We will describe the properties of FLASH and the corresponding code transformation. Based on these properties, we will explain how FLASH achieves FIFO communication cycle-accurate (FCCA) simulation.

We make the following assumptions:

(**A1**) HLS tool schedules a FIFO communication of a C source code to be executed at a particular FSM state of a module.

(**A2**) HLS tool provides the information on the assigned FSM state of its FIFO communication, conditional statements, and loop statements.

(**A3**) HLS tool provides the information on a module's FSM state transition.

(**A4**) An FIFO communication statement is composed of constants, operators, variables, and a single FIFO API.

(**A5**) The value of all constants and the behavior of all operators in the HSS match that of the RTL simulator.

(**A6**) A FIFO is a deterministic system which is simulated cycle-accurately by a HSS at t if the FIFO input values of the FIFO APIs and the FIFO IP behavior at 1, 2, ..., $t-1$ are simulated cycle-accurately.

(**A7**) All FIFOs are initially (t=1) empty.

```
01 ================================================================
02 + Verbose Summary: Schedule
03 ================================================================
04 * Number of FSM states : 7
05 * Pipeline : 1
06   Pipeline-0 : II = 1, D = 5, States = { 2 3 4 5 6 }
07 * Dataflow Pipeline: 0
08
09 * FSM state transitions:
10 1 -->
11         2  / true
12 2 -->
13         7  / (!tmp)
14         3  / (tmp)
15 3 -->
16         4  / true
17 4 -->
18         5  / true
19 5 -->
20         6  / true
21 6 -->
22         2  / true
23 7 -->
24
25 * FSM state operations:
26
27 State 6 <SV = 5> <Delay = 1.75>
28 ... Operation 27 ... ---> "call void (...)* @_ssdm_op_Spec
29 ... Operation 28 ... --->   "call void @_ssdm_op_Write
.ap_fifo.volatile.i32P(i32* %f_out_V, i32 %temp)"
30 ... Operation 29 ... --->   "br label %._crit_edge ...
31 ... Operation 30 ... --->   "%empty_40 = call i32 ...
```

Figure 4.7: Vivado HLS scheduling report for `M2` of Fig. 4.5

Table 4.3: FIFO IP behavior (assumes all FIFO APIs are evaluated at t)

Vivado HLS FIFO API	FIFO IP Behavior
`read()`	while(1) {If `empty` is false at t, `dout`=(first element in the FIFO buffer) at t and dequeue the element at $t+1$ and exit loop.}
`read_nb()`	If `empty` is false at t, `dout`=(first element in the FIFO buffer) at t and dequeue at $t+1$. Else, do nothing.
`write()`	while(1) {If `full` is false at t, enqueue the value of `din` at t to the FIFO buffer at $t+1$ and exit loop.}
`write_nb()`	If `full` is false at t, enqueue at $t+1$ the value of `din` at t to the FIFO buffer. Else, do nothing.
`empty()`	If there are no data in the FIFO buffer, `empty`=1 at t. Else, `empty`=0 at t.
`full()`	If the number of data in FIFO matches the FIFO buffer size, `full`=1 at t. Else, `full`=0 at t.

```
01 void M2_SIM(){                          //simulation function for M2
02   static int M2_state = 1;//use “static” var for the next cycle
03   ...
04   if(M2_state == 1){      //state conditional block for state 1
05     ...          //computation stmt & communication for state 1
06     M2_state = 2;                //state transition for state 1
07   }
08   else if(M2_state == 2){ //state conditional block for state 2
09     ...          //computation stmt & communication for state 2
10     M2_state = 7;                //state transition for state 2
11   }
12 }               //exit sim function after simulating one cycle
```

Figure 4.8: Simulation function structure for selective simulation of an FSM state (M2_SIM is simulated at line 9 of Fig. 4.10)

The *FIFO IP behavior* in (**A6)** refers to the behavior of FIFO IPs in response to the Vivado HLS FIFO APIs (FAPIs) as specified in Table 4.3.

4.4.1.1 Matching Simulated State of Statements

As mentioned in (**A1**), let us assume that HLS tool schedules a FIFO communication of a C source code to be executed at a particular FSM state (st) of a module. FLASH simulates the FIFO communication at the same st scheduled by the HLS tool. In order to achieve this, we first need to obtain the HLS scheduling information of the FIFO communication (**A2**). This is found from parsing FIFO-related keywords in the scheduling report. For example, the state when FIFO “f_out" performs the write operation (line 7 of Fig. 4.5) is found to be 6, because `op_Write.ap_fifo` and “f_out" keywords are detected in line 29 of Fig. 4.7. Similarly, the FIFO read statement (line 5

of Fig. 4.5) is assigned to state 2 from the scheduling report (not shown in the figure).

Next, we need to ensure that only the FIFO communications statements assigned to each FSM state are selectively simulated at every cycle. We declare an FSM state variable (line 2 of Fig. 4.8) for each module and copy statements to the conditional block that correspond to its simulated state (*state conditional block*). An example can be found for the `M2` module in lines 4–7 ($st = 1$) and lines 8–11 ($st = 2$) of Fig. 4.8. After the simulation function of a module has been called, only the statements for a single FSM state are simulated, and then the function exits. That is, a single clock event is simulated by a function entrance and exit.

Since FLASH aims for cycle-accuracy of FIFO communication, the computation statements do not need to be evaluated cycle-accurately. For computation statements, we can assign an arbitrary state as long as it does not violate the timing causality with the cycle-known FIFO communication that has dependency with the computation statement. We group the computation statements to a few FSM states as much as possible; if the statements are spread among multiple FSM states, the variables shared across the states need to be declared local static variables (residing in .data or .bss sections) which may be stored in DRAM. This is inefficient if the variable has a short life and could have been optimized to a CPU register.

For example, the computation statement in line 6 of Fig. 4.5 has a dependency with both the FIFO read and the FIFO write statements. It may be assigned to any state between 2 and 6 without violating the time causality, but to reduce the number of FSM states with statements, it should be assigned to either 2 or 6. We choose to assign it to state 2, following the as-soon-as-possible scheduling policy, as it tends to reduce the number of variables being passed between the states.

In some cases we might need to change the evaluation order of some computation statements even though it causes a dependency problem. For example, suppose that we add a statement between line 7 and line 8 of Fig. 4.5 that is dependent on line 7 and references i. Since the loop index update statement in line 8 is scheduled to state 2 (Section 4.4.3) and line 7 is scheduled to state 6 from the scheduling report, the new statement between lines 7 and 8 will incorrectly reference i which has already been updated to the next iteration. This is solved by copying i to a temporary variable before evaluating the loop index update statement and renaming any reference

of i that has the dependency problem to this temporary variable.

4.4.1.2 Cycle-Accurate FSM State

FLASH cycle-accurately simulates the FSM state of a module at t (st_t). By induction, st_t is cycle-accurate if the initial state at $t = 1$ is known ($st_{t=1} = 1$) and the state transition Δ_t matches the RTL simulation at 1, 2, ..., t-1. Δ_t matches the RTL simulation, if the state transition information can be obtained from the HLS tool report and a state transition statement that reflects this information is evaluated at t. Also, Δ_t should be stalled if `empty` or `full` signals have been asserted when blocking reads or writes have been evaluated.

As mentioned in (**A3**), Vivado HLS provides the state transition information in its scheduling report. For example, the loop in module `M2` in Fig. 4.4 is evaluated in states 2 to 6, as shown in line 6 of Fig. 4.7. The state transition of the loop is composed of intra-loop state transition (e.g., state 2 to 3, as shown in line 14 of Fig. 4.7) and loop exit (e.g., state 2 to 7, as shown in line 13). FLASH obtains this information and inserts the state transition statement into the simulation code. For example, line 10 of Fig. 4.8 reflects the loop exit state transition from state 2 to 7. The method used by FLASH to correctly simulate the state transition stalls will be discussed in Section 4.4.1.4.

Vivado HLS schedules a module to finish its execution at a particular FSM state. Since FLASH cycle-accurately simulates the FSM state of a module, the estimation of a module's execution time is cycle-accurate.

4.4.1.3 FIFO Behavior Modeling

FLASH simulates the FIFO IP behavior that has been specified in the Table 4.3. In the FLASH simulation code, the FIFO is implemented as a circular buffer with read/write pointers (fifo_rptr and fifo_wptr) and an array (fifo_arr). In order to simulate the behavior of `full` FAPI (Table 4.3) correctly, the array length is set to FIFO buffer size (FIFO_SIZE) plus one, because one buffer space is kept empty in circular buffer implementation [39]. Also, we declare fifo_rnum and fifo_wnum variables to denote the number of data and buffer spaces available in the FIFO. FAPIs in the source

code are transformed based on the fourth column of Table 4.2. An example is shown in Fig. 4.9, which is the transformed simulation code from `M2` in Fig. 4.5. Line 7 of Fig. 4.5 is transformed to: "fifo3_arr[fifo3_wptr++] = temp_st6; fifo3_wnum−−;" (lines 12-13 of Fig. 4.9). The comparison of blocking FAPIs (`write` and `read`) with non-blocking FAPIs (`write_nb` and `read_nb`) in Table 4.3 shows that the FIFO transaction should not occur if the FIFO full or FIFO empty condition is satisfied. This behavior in reflected in the stall condition code in the fourth column of Table 4.2. The stall condition will be further explained in Section 4.4.1.4.

In addition to decreasing the number of buffer spaces (fifo3_wnum−−) for FIFO write, we would need to increase the number of available data (fifo3_rnum++). However, this process is delayed until all other statements in the current cycle have been simulated. The reason is to match the Xilinx FIFO IP behavior (Table 4.3) of allowing a data written to an FIFO to be available for read one cycle after it has been written. The implementation details of this delayed processing will be provided in Section 4.4.2.2.

4.4.1.4 FSM Stall Modeling

FLASH cycle accurately models the FSM stalls due to FIFO being full or empty. If a stall condition is met, none of the statements of current FSM state should be simulated, and the simulation function should exit. To achieve this, the stall condition is placed at the beginning of a state conditional block. The simulation code for the stall condition is "fifo_rnum==0" for FIFO empty and "fifo_wnum==0" for FIFO full (Table 4.2). These codes are also used for stall conditions of blocking reads and writes. For example, the stall condition that corresponds to the FIFO blocking write in line 7 of Fig. 4.5 is : "if(p1_en_st6 && fifo3_wnum==0)". This condition has been added to line 5 of Fig. 4.9. Also, the simulation exit statement has been added to line 7 of Fig. 4.9. Note that we add a enable signal "p1_en_st6" to the stall condition of a pipelined loop, because the FIFO write occurs at FSM state 6 (more details in Section 4.4.2.1).

FLASH can detect a deadlock by checking if state transition did not occur (stalled) in all modules. It is enabled by the source-level trigger directive that will be explained in Section 4.7.2.

Also, it is worth noting that applying the classic event-driven simulation approach (e.g., [10,

114]) made little difference in the simulation speed of FLASH. The reason is that the stall condition is placed at the beginning of a state conditional block and prevents most of the statements from being evaluated when a module is stalled. That is, there is little overhead in processing a module without an event that requires simulation, and this diminishes the benefit of applying the event-driven approach.

4.4.1.5 Correctness of the Variable Reference

As explained in Section 4.4.1.1, all statements that have dependency with $cstmt$ have been evaluated before the simulation of $cstmt$. The value of variables written by statements with the same st as $cstmt$ is correctly supplied to $cstmt$, because the they are simulated in the same state conditional block. A problem occurs when reading variables written by statements with FSM states other than st, because the simulation function exits after each cycle. This problem is solved using the `static` keyword in a variable declaration (e.g., line 2 of Fig. 4.8 and line 2 of Fig. 4.9). By using this technique, the contents of the variables are restored and saved regardless of the simulation function entrance or exit.

4.4.1.6 Proof of FCCA Simulation

We will prove the FLASH is an FCCA simulator.

Let us first focus our attention on finding the condition needed to obtain the correct FIFO input value of the FAPIs. As explained in Section 4.4.1.2, the FSM state of a module matches the RTL simulation at t in FLASH if FIFO `empty` and `full` signals (which cause state transition stalls) of all FIFOs connected to the module match the RTL simulation at 1, 2, ..., $t-1$. $cstmt$ of a module is simulated at the same FSM state as the HLS tool's scheduling using the method explained in Section 4.4.1.1. Thus, $cstmt$ is simulated at the same cycle t as the RTL simulation if FIFO `empty` and `full` (FIFO output signals) of all FIFOs connected to the module signals match the RTL simulation at 1, 2, ..., $t-1$.

Among the four components of $cstmt$ (**A4)**, we have assumed that constants and operators of

the HSS match that of the RTL simulation (**A5**). For variables (the third component) of $cstmt$, FLASH evaluates all statements that has dependency with $cstmt$ and passes the value of variables from these statements to be read in $cstmt$ (Section 4.4.1.5). If these statements with dependency were also communication statements, it will produce correct value of variable if its FIFO output value of the FAPIs match the RTL simulation. Such FAPIs may have been evaluated at any cycle 1, 2, ..., t. The modeling of FAPIs (the fourth component of $cstmt$) has been explained in Section 4.4.1.3. The union of the discussed conditions needed to obtain the correct FIFO input value of the FAPIs at t is that the FIFO output values of the FAPIs of all connected FIFOs at 1, 2, ..., t needs to match the RTL simulation. This is summarized in Lemma 4.4.1.

Lemma 4.4.1. *If the FIFO output values of the FAPIs of all FIFOs connected to a module at 1, 2, ..., t match the RTL simulation, the FIFO input values of the FAPIs of all FIFOs connected to a module at t match the RTL simulation.*

By induction, we can also show that the FIFO input values of the FAPIs of all FIFOs connected to a module at 1, 2, ..., $t-1$ match the RTL simulation. If we enlarge the scope of Lemma 4.4.1 to all FIFOs $f_{p=0...F}$ and all modules $m_{x=0...M}$ of a design, we can conclude that:

Corollary 4.4.1.1. *If the FIFO output values of the FAPIs in $m_{x=0...M}$ at 1, 2, ..., t match the RTL simulation, the FIFO input values of the FAPIs in $m_{x=0...M}$ at 1, 2, ..., t match the RTL simulation.*

Next, we move on to finding the condition needed to obtain the correct FIFO output value of the FAPIs. As stated in (**A6**), we assume that a FIFO is a deterministic system which is simulated correctly by a HSS at t if the FIFO input values of the FAPIs and the FIFO IP behavior at 1, 2, ..., $t-1$ are simulated correctly. The FIFO IP behavior is simulated correctly using the method in Section 4.4.1.3. Then Lemma 4.4.2 is derived from (**A6**):

Lemma 4.4.2. *If the FIFO input values of the FAPIs at 1, 2, ..., $t-1$ match the RTL simulation, the FIFO output values of the FAPIs at t match the RTL simulation.*

Similar to the argument made in deriving Corollary 4.4.1.1, the FIFO output values of the FAPIs at 1, 2, ..., $t-1$ match the RTL simulation as well. Also, we enlarge the scope of Lemma 4.4.2 to all FIFOs and all modules of a design:

Corollary 4.4.2.1. *If the FIFO input values of the FAPIs of $m_{x=0...M}$ at 1, 2, ..., $t-1$ match the RTL simulation, the FIFO output values of the FAPIs of $m_{x=0...M}$ at 1, 2, ..., t match the RTL simulation.*

From Corollary 4.4.1.1 and Corollary 4.4.2.1, we can reach the following conclusion:

Theorem 4.4.3. *The FIFO input and output values of the FAPIs of in $m_{x=0...M}$ at 1, 2, ..., t match the RTL simulation.*

Proof: Since we assume that all FIFOs are initially empty (**A7**), all FIFO output values of the FAPIs are known at $t = 1$. Then by Corollary 4.4.1.1, the FIFO input values of all FAPIs at $t = 1$ match the RTL simulation. The input and output values of all FIFO APIs match the RTL simulation at $t = 1$ (*base case*). Let us assume that Theorem 4.4.3 is true at $t = k$. Since the input values of all FIFO APIs at $t = k$ match the RTL simulation, the output values of all FIFO APIs at $t = k + 1$ match the RTL simulation (Corollary 4.4.2.1). This implies that the input values of all FIFO APIs at $t = k + 1$ match the RTL simulation (Corollary 4.4.1.1). Thus, the input and output values of all FIFO APIs match the RTL simulation at $t = k + 1$ (*induction step*). Thus, Theorem 4.4.3 is true by induction.

Since the FIFO input and output values of the FAPIs of $m_{x=0...M}$ at all cycles 1, 2, ..., t match the RTL simulation, FLASH is a FIFO communication cycle-accurate simulator.

4.4.2 Simulation of Parallelism

4.4.2.1 Pipelined Parallelism

At each cycle, all statements in a pipelined loop are executed in parallel in a pipelined fashion. The number of FSM states to be simulated corresponds to the loop iteration latency (IL, also called pipeline depth). If we simulate only a particular FSM state conditional block of a pipelined loop, it would not be possible simulate this parallelism.

To solve this problem, we would need to simulate all FSM states of a pipelined loop. It is possible to make an exception to the simulation structure by traversing through multiple state con-

```
01 static bool p1_en_st3, ... p1_en_st6  = false; //enable signals
02 static int temp_st3, ... temp_st6; //6    //pipelined variables
03 ...
04 else if(M2_state == 2){ //starting state for the pipelined loop
05   if(p1_en_st6 && fifo3_wnum==0){//if stalled due to FIFO3 full
06     debug_stall_mod_cnt++; //cnts stalled modules(see Sect 8.B)
07     return;         //exit without any changes (see Sect 5.A.4)
08   }
09   ...
10   if( p1_en_st6 == true ){   //enabled 4 cycles after FIFO read
11     p1_en_st6 = false;       //disables enable signal after use
12     fifo3_arr[fifo3_wptr++] = temp_st6; //7   //FIFO data write
13     fifo3_wnum--;                              //(see Sect 5.A.3)
14   }
15   ...
16   if( p1_en_st3 == true ){    //enabled 1 cycle after FIFO read
17     p1_en_st3 = false;       //disables enable signal after use
18     p1_en_st4 = true;                //enable signal propagation
19     temp_st4 = temp_st3;  //copies variable for next pipe stage
20   }
21   if( i_st2 < N ){ //2     //loop exit condition (see Sect 5.C)
22     if( fifo1_rnum != 0 ){ //4              //if FIFO not empty
23       data_st2 = fifo1_arr[fifo1_rptr++]; //5  //FIFO data read
24       fifo1_rnum--;                            //(see Sect 5.A.3)
25       temp_st2 = data_st2 * 711; //6 // comp stmt mapped to st2
26       i_st2++; //8       // loop iterator update (see Sect 5.C)
27       p1_en_st3 = true; //enables if path for later pipe stages
28       temp_st3 = temp_st2;//copies variable for next pipe stage
29 ... } } }
```

Figure 4.9: Simulation code that models pipelined loop parallelism for `M2` of Fig. 4.5 (provides details for line 9 of Fig. 4.8)

ditional blocks in a single cycle for pipelined loops; but this would over-complicate the simulation structure. For a simpler solution, we choose to move all of the pipelined loop's state conditional blocks into the conditional block of a single state. The reallocated conditional blocks are referred to as *pipeline stage conditional blocks*. As shown in Fig. 4.9, the contents of FSM states 2, 3, and 6 have been moved to pipeline stage conditional blocks in lines 21-29, lines 16-20, and lines 10-14. Note that if a pipelined loop's II is larger than 1, FLASH makes II-1 state conditional blocks for this loop, and the pipeline stage conditional blocks at state st are placed at the state conditional block of $st\%II$.

The contents of each pipeline stage conditional block may be computed if the value of enable signals is one (lines 10, 16 of Fig. 4.9). For example, the FIFO write at lines 12-13 is evaluated if the enable signal "p1_en_st6" is one. One of the purposes of using an enable signal is to selectively simulate statements in the pipeline loop prologue and epilogue. Another reason is that the enable signal can invalidate statements in a pipeline bubble (from the artificial deadlock avoidance transformation in Section 4.2.2). The value of enable signals are propagated through the pipeline stages as shown in line 18.

It is important to note that the order of each pipeline stage conditional block has been *reversed* (st6, ... st3, st2). This limits the value of enable signals to be only copied to the immediate next pipeline stage in simulation of a single cycle.

Even if a same variable is used in different statements of the original source code, we cannot assume that they have the same value if they have been assigned to different pipeline stage conditional blocks in simulation. For example, suppose that line 6 of Fig. 4.5 is performed at FSM state 2, and line 7 is performed at state 6. In a single cycle of the pipelined loop simulation, "temp" of line 7 would correspond to loop iteration *i*, whereas "temp" of line 6 would correspond to loop iteration *i+4*. Thus, they would have different values.

For correct simulation, we keep multiple copies of the same variable for each pipelined stage of a loop. The variables are copied through the pipeline like shift registers. For example, the "temp" variable is copied from loop pipeline stage 3 to stage 4 at line 19 of Fig. 4.9. Variables "data" and "i" are not copied to the next pipelined stage after performing cycle-based variable

```
01 void (*MList[M])();                    //module func ptr list
02 void (*FList[F])();                     //FIFO func ptr list
03 Mlist[0] = M1_SIM;          .... Mlist[3] = M4_SIM;   //init
04 Flist[0] = F1_SIM;          .... Flist[3] = F4_SIM;
05
06 while(1){                                      //scheduler loop
07   ...   // loop until until deadlock or all modules finish
08   for(x = 0; x < M; x++)              //simulate all modules
09     Mlist[x]();
10   for(p = 0; p < F; p++)                //simulate all FIFOs
11     Flist[p]();
12   ...
13   cycle++;
14 }
```

Figure 4.10: Module/FIFO simulation scheduler to model task-level parallelism

liveness analysis that will be explained in Section 4.5.1. Similar to the enable signals, the content of pipelined variables is only copied to the immediate next state in a single cycle, since the order of the pipeline stage conditional block has been reversed. Optimization of the pipelined variables is discussed in Section 4.5.

Because of the duplicated pipelined variables, the readability of the simulation code could be reduced. In order to diminish this side effect, FLASH places the line number of the original variable declaration in the source code as a comment of the duplicated pipelined variable declaration in the simulation code. For example, the line number 6 of the original variable declaration of “temp" in Fig. 4.5 is written as a comment of the duplicated pipelined variable declaration in line 2 of Fig. 4.9. Also, the original line numbers of the computation and communication statements are placed at the comments of the simulation statements (e.g., lines 21-26 of Fig. 4.9).

4.4.2.2 Task-Level Parallelism

As discussed in Section 4.4.1.1, the statements in an FSM state are simulated by calling the simulation function of a module. Thus, the task-level parallelism can be simulated by calling all simulation functions in a round-robin fashion. This is processed in the *module simulation loop*

```
01 void F3_SIM(){                          //simulation function for fifo3
02   fifo3_wnum = (fifo3_rptr > fifo3_wptr) ?    //update fifo3_wnum
        fifo3_rptr-fifo3_wptr-1 : fifo3_rptr-fifo3_wptr+FIFO3_SIZE;
03   fifo3_rnum = FIFO3_SIZE - fifo3_wnum;       //update fifo3_rnum
04 }
```

Figure 4.11: FIFO simulation code for `fifo3` (F3_SIM is simulated at line 11 of Fig. 4.10)

shown in lines 8-9 of Fig. 4.10.

As mentioned in Section 4.4.1.3, the update of the buffer spaces and the number of available data is delayed until all modules in the current cycle have been simulated. The variables update code, shown in Fig. 4.11, is performed in the *FIFO simulation loop* (lines 10-11 of Fig. 4.10).

The module simulation loop and the FIFO simulation loop form the *scheduler loop* as shown in lines 6-14 of Fig. 4.10.

4.4.3 Loop and Function Simulation

Simulation of statements inside a pipelined loop has been discussed in Section 4.4.2.1. The loop initialization statement is simulated upon initial entrance to the first FSM state (st_L) of a loop L. The loop update is simulated at the end of each loop. If L is a pipelined loop with II of II_L, the loop update occurs at the end of $st_L + II_L - 1$, because FLASH uses $II_L - 1$ state conditional blocks (Section 4.4.2.1). This causes loop iterator update statements (e.g., line 8 of Fig. 4.5) to be evaluated to the end of $st_L + II_L - 1$. If the loop condition is met after the update, state transition for loop exit occurs. For a flattened loop (e.g., `M1` in Fig. 4.4), the update and the loop condition check is performed starting from the innermost nested loop, as illustrated in Fig. 4.12.

A function call is simulated by sending a module enable signal to the scheduler loop (Fig. 4.10). Next, the function argument values are copied into the newly called module.

```
j++;                        //inner loop update
if( !(j<16) ){               //inner loop cond
  i++;                      //outer loop update
  j=0;                       //inner loop init
  if( !(i<N/16) ){          //outer loop cond
    M1_state = 4; //st trans for loop exit
} }
```

Figure 4.12: Loop condition and update for flattened loop in `M1` of Fig. 4.4

4.5 Optimization of Pipelined Loops Simulation

Pipelined loops typically account for most of the execution time of many FPGA designs. To simulate pipelined parallelism, we need copies of variables for each iteration latency of a pipelined loop (Section 4.4.2.1). However, a naive implementation could lead to making redundant copies of the variables. This section discusses how to optimize this routine. The effect of optimizations presented in this section will be evaluated in Section 4.8.2.

4.5.1 Cycle-Based Variable Liveness Analysis

The pipelined variables are only needed in the pipeline stages where the variables are being accessed. To ensure this, we first perform variable liveness analysis [1] to find the range of statements where each pipelined variable is alive. Next, the FSM state of communication statements and the computation statements are obtained from the scheduling report and the dependency analysis. From the FSM state information of statements, the statement liveness range of each variable is translated into a cycle liveness range. Based on this cycle information, we place a limit on the pipeline stages where each pipelined variable is copied.

For the example in `M2` of `toy_mpath` (Fig. 4.5), we first perform liveness analysis on each variable to find that variable “data" is live in lines 5–6, variable “i" in line 8, and variable “temp" in lines 6–7. Then we assign the states for communication and computation statements in `M2` of `toy_mpath` as was shown in Section 4.4.1.1. That is, statements in lines 5, 6, and 8 of Fig. 4.5 are assigned state 2, and the statement in line 7 is assigned state 6. Based on this information,

```
01 static bool p1_en[5];                          //enable signal array
02 static int temp[5]; //6                    //pipelined variable array
03 static int ptr_st2 = 4, ptr_st6 = 0;   //pipe variable pointers
04 ...
05 else if(M2_state == 2){
06   ...
07   if( p1_en[ptr_st6] == true ){
08     p1_en[ptr_st6] = false;  //disables enable signal after use
09     fifo3_arr[fifo3_wptr++] = temp[ptr_st6]; //7    //read from
10                                          //pipelined variable array
11     fifo3_wnum--;
12   }
13   //conditional blocks for pipeline stages 3, 4, 5 are removed
14
15   if( i_st2 < N ){ //2
16     if( fifo1_rnum != 0 ){ //4
17       p1_en[ptr_st2] = true;    //enables later pipeline stages
18       data_st2 = fifo1_arr[fifo1_rptr++]; //5
19       fifo1_rnum--;
20       temp[ptr_st2] = data_st2*711; //6            //written to
21                                          //pipelined variable array
22       i_st2++; //8
23   } }
24   ptr_st2 = (ptr_st2 + 1) % 5;            // pipelined variable
25   ptr_st6 = (ptr_st6 + 1) % 5;               // pointers update
26 } }
```

Figure 4.13: The code after applying pointer-based variable access optimization to the initial code provided in Fig. 4.9

the statement liveness range is converted into a cycle liveness range—variables “data" and “i" are live at cycle 2 and variable “temp" from cycles 2 to 6. As a result, only variable “temp" is copied through the pipeline stages.

4.5.2 Pointer-Based Variable Access

One of the problems of declaring a pipelined variable for each pipeline stage (as in Fig. 4.9) is that the same value is copied repeatedly. Assuming a pipelined loop has `I` iterations, `V` variables, and `IL` iteration latency, the complexity of copying pipelined variables is `O(I×V×IL)`.

We propose an alternative method of copying the value of a pipelined variable only once and changing the pointer to the pipelined variable. The modification to the initial code provided in Fig. 4.9 is shown in Fig. 4.13. We first exploit the fact that the value of the pipelined variable is used in the immediate next pipeline stage—thus, the pipeline variable pointer for stage *st* (ptr_{st}) update can be simplified into $(ptr_{st} + 1)\%IL$ (lines 24–25). Next, the pipeline variable pointer is shared among all variables and enable signals in the same pipeline stage since all variables and enable signals are copied together to the next pipeline stage if the loop pipeline has not been stalled. An example is shown for “temp" variable (line 20) and “p1_en" enable signal (line 17). Note that this optimization has not been applied to variables “i" and “data", because “i" and “data" are only used in pipeline stage 2 (Section 4.5.1). Finally, we remove the pipeline stage conditional blocks that do not evaluate any statement (line 13—pipeline stages 3, 4, and 5 are removed), because variables and enable signals no longer need to be copied.

Since the data is copied only once, the complexity of pipelined variable copy is $O(I \times V)$. The pipelined variable pointers are shared among all variables in the same pipeline stage—thus, the complexity of the pointer update appears to be $O(I \times IL)$. However, as mentioned in Section 4.4.1.1, we group statements to only a few FSM states. The pointer update is not performed on pipeline stages that do not evaluate any statement (e.g., lines 24-25). Assuming there are C stages with communication statements ($C \ll IL$), the pointer update complexity is $O(I \times C)$. Thus, the overall pipeline variable complexity of the proposed method is $O(I \times (V+C))$, which is a large improvement over $O(I \times V \times IL)$.

4.6 Overall Flow

The overall simulation framework of FLASH is shown in Fig. 4.14. Given an input Vivado HLS C design source code, users specify optional debugging directives such as module execution cycle measurement or deadlock triggering (to be explained in Section 4.7). Then FLASH performs a preprocessing step of adding labels to the source code so that loops and functions can be easily identified. The transformation step uses the APIs in the ROSE [106] and the Merlin [48] compilers.

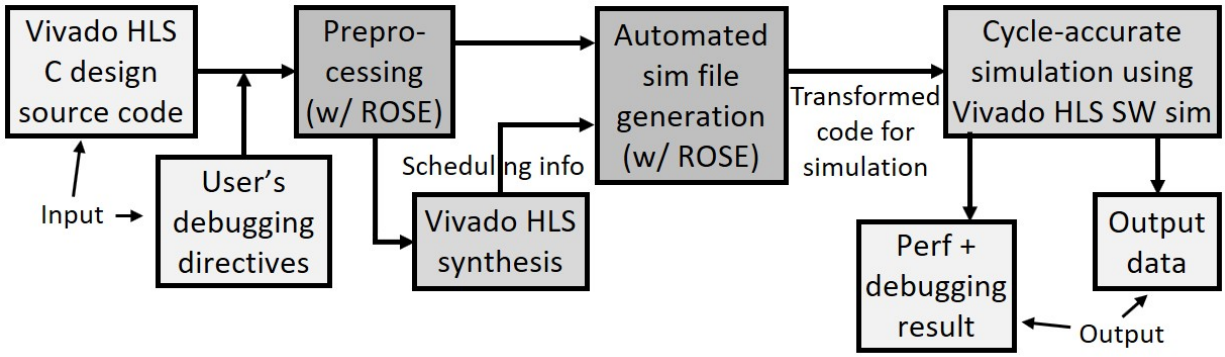


Figure 4.14: Overall simulation framework of FLASH

Table 4.4: Debug directives for FLASH

	Syntax	Description	Target
Trigger-related	DEADLOCK	Triggered at deadlock	Module with dataflow pragma
	STALL	Triggered when module or loop has been stalled	Any module or loop
	MODULE_DONE	Triggered when a module completes its execution	Any module
	FIFO_FULL FIFO=<name>	Triggered at FIFO full condition	Any FIFO
	FIFO_EMPTY FIFO=<name>	Triggered at FIFO empty condition	Any FIFO
	EQUAL VAR=<name> VAL=<val>	Triggered when variable equals to value provided	Any stmt with a variable reference
	GREATER VAR=<name> VAL=<..>	Triggered when variable is greater than value provided	Any stmt with a variable reference
Data-related	DUMP VAR=<name> FILE=<name>	Dumps variable data into the specified file	Any stmt with a variable reference
	COMP VAR=<name> FILE=<name>	Triggered when variable differs from golden data in file	Any stmt with a variable reference
Perf-related	TRIP_COUNT	Measures the loop trip count (e.g. for data-dependent loop)	Any loop
	EXEC_CYCLE	Measures the number of execution cycles for module or loop	Any module or loop
	STALL_CYCLE	Measures the number of stalled cycles for module or loop	Any module or loop
	FULL_CYCLE	Measures the number of cycles when FIFO was full	Any FIFO
	EMPTY_CYCLE	Measures the number of cycles when FIFO was empty	Any FIFO
	AVG_BUFFER	Measures average number of buffer space available in FIFO	Any FIFO

The transformed code is fed into the Vivado HLS for synthesis. Based on the scheduling report given by the HLS tool, the input code is automatically transformed for rapid FCCA simulation (Section 4.4 and Section 4.5). The simulation code has been made compatible with the Vivado HLS software simulator for easy integration with the existing tool. As a final output, FLASH provides the total execution cycles and other user-specified debugging results in addition to the output data that the design is expected to produce.

4.7 Source-Level Correctness Debugging and Performance Debugging

FLASH provides an option of enabling various source-level correctness and performance debugging features that will be explained in this section.

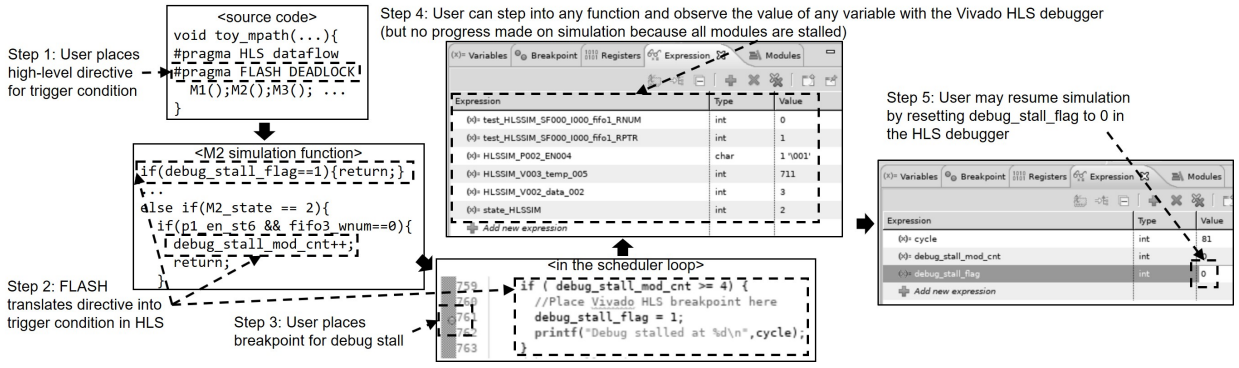


Figure 4.15: An example debugging session for deadlock detection using FLASH

4.7.1 Live Capture

FPGA tools such as Xilinx's ChipScope [126] or Intel's SignalTap [63] capture the data in the FPGA and display it to users for debugging. One of the problems with these configurable logic analyzers is that additional signals often need to be inserted into the capture list to continue tracing the source of a bug after the initial analysis. This requires iterative adjustment of the signal capture list until the bug has been isolated, and the bitstream generation for each analysis often takes hours to finish. Many of the hardware-based HLS debuggers described in Section 2.3 also require a long turnaround time due to similar reasons.

Software debuggers, on the other hand, do not require signals to be listed in advance. However, due to the lack of cycle accuracy, putting a trigger (breakpoint) on a C source code does not allow the users to observe the signals at a particular cycle of interest. Another problem is that users have limited visibility. For example, local variables in a function different from the trigger cannot be observed unless the user progresses to that function—by which time the content of many variables would have been changed.

To solve these problems, we exploit the fact that FLASH stores intermediate variables (Section 4.4.1) and the fact that FLASH runs on top of an established commercial tool, the Vivado HLS, that provides software debugging features. Upon detection of a trigger condition (details in Section 4.7.2), FLASH sets a debug stall flag. All modules are stalled upon the detection of this flag (implementation is similar to the pipelined loop stall modeling in Section 4.4.1.4). When the simulation has been stalled for debugging, users can step into any function and observe any

local variables of interest by adding a new variable into the HLS debugger variable watch list. The variables to be captured no longer need to be predetermined.

In the debugger variable watch list, the users can expect variables in the FIFO communication and the FSM state variables to match the RTL simulation at all cycles (Section 4.4.1). However, the timing when the variables in the computation statements match the RTL simulation may not be accurate.

In order to pause at the trigger point, FLASH guides the users to place a breakpoint on the source code line where the debug stall flag is set. The breakpoint is detected by the Vivado HLS debugger. To resume the simulation after observation, the user can modify the value of the debug stall flag to 0 using the Vivado HLS debugger.

4.7.2 Source-Level Event Trigger and Performance Measurement

In the Xilinx Chipscope [126], users are required to specify signal names and their value for the tool to start capturing the data (trigger condition). Since the HLS tool applies several transformations in generating RTL file from a C source code, manually identifying the correct trigger condition from a RTL file may be error-prone for novice users. To ease this process, many hardware-based HLS debuggers [51–53, 89, 90, 120] allow users to specify variables to be traced or put breakpoints on the source code; however, none of these abstract the trigger condition of events such as deadlock or module/FIFO stall.

FLASH provides a set of source-level directives which can be specified by users to halt computation upon an event of interest. The list is given in the trigger-related row of Table 4.4. The directive is always preceded by: `#pragma FLASH <syntax>`. For example, the deadlock detection directive is `#pragma FLASH DEADLOCK`. FLASH automatically converts the directives into a stall condition that increments a debug variable that counts the number of stalled modules. For the case of `M2` in Fig. 4.4, it will be stalled if the FIFO is full (line 14). If the directive for deadlock detection is found in the source code, FLASH inserts a code that increments the debug variable “debug_stall_mod_cnt" upon stall (line 6 of Fig. 4.9). After simulating each cycle, FLASH checks to see if “debug_stall_mod_cnt" matches the number of all modules in the design.

If so, FLASH sets the debug stall flag that would pause the simulation (Section 4.7.1).

An example debugging session for deadlock detection is shown in Fig. 4.15. After the user places a directive in the source code for deadlock triggering, FLASH translates the directive into a code that stalls the simulation of all modules. Then the user can step into any function and observe the value of any variable with the Vivado HLS debugger. The user may choose to continue the simulation by resetting the debug stall flag to 0.

FLASH also supports directive-based performance measurement. The list is given in the performance-related row of Table 4.4. The functionality includes module execution and stall cycle measurement, as well as FIFO full and empty cycle measurement.

4.7.3 Large Data Debugging

Hardware-based HLS debuggers such as [51–53, 89, 90, 120] optimize the storage and transfer of variable data in an FPGA to be analyzed for correctness. However, the amount of traced variable is still limited by the BRAM size and the DRAM bandwidth. Being a software-based debugger, FLASH is not limited by the FPGA hardware resource restriction when performing such data-driven debugging—even for multiple variables. Examples include large data dump and large golden reference comparison—the user directives for these functions are shown in the data-related row of Table 4.4.

4.8 Experimental Results

4.8.1 Experimental Setup

For HLS synthesis, we use the Vivado HLS 2018.2 [130]. For FPGA, we target Xilinx's Ultrascale KU060 [129]. The target clock frequency is 250MHz. The simulation is conducted with a server node that has an Intel Xeon Processor E5-2680v4 [66] and 64GB of DRAM. The simulation files were compiled with –O3 flag.

The experiment is performed on `toy_mpath` (Fig. 4.4) and several dataflow benchmarks:

Table 4.5: Simulation preparation time breakdown (preprocessing, HLS synthesis, and simulation file generation: Fig. 4.14)

Benchmark	Preproc	HLS Synth	SimFile Gen	Total
Toy_mpath	7.6s	25s	7.9s	40s
Stencil	19s	68s	30s	117s
MD_sim	9.2s	38s	8.8s	56s
Mat_mul	8.7s	36s	12s	57s
Cholesky	15s	98s	37s	150s
NW	18s	99s	26s	143s
LUD	7.1s	21s	8.9s	37s
SpMV	13s	78s	25s	116s

stencil [17], molecular dynamics simulation [33] (Fig. 4.1), matrix multiplication [36], Cholesky decomposition [82], Needle-Wunsch [105], LU decomposition [99], and sparse matrix-vector multiplication [136]. The benchmarks ([99, 105]) that were not originally designed to execute modules in parallel with FIFO communication were modified to incorporate this optimization.

The FLASH simulation result is compared to that of Vivado HLS C and RTL simulation (Verilog), and Verilator 4.012 simulation [113]. Since Vivado HLS RTL files contain core library calls that cannot be processed by Verilator, we have manually replaced them with a behavioral Verilog model.

4.8.2 Execution Time

As mentioned in Section 4.6, preprocessing, HLS synthesis, and simulation file generation steps are needed to prepare the files for the proposed simulation. The time breakdown of the steps is presented in Table 4.5.

The effect of optimizations in Section 4.5 is shown in Table 4.6. The baseline version uses the techniques introduced in our earlier publication [18] and does not have cycle-based variable liveness analysis (Section 4.5.1) and pointer-based variable access (Section 4.5.2) optimizations. The table shows that the proposed optimizations result in 1.55X speedup on average. The speedup is greater for benchmarks that have a large (>12) averaged pipeline depth among all variables. The average speedup for Stencil, MD_sim, LUD is 2.28X, and the averaged speedup for the

Table 4.6: Speedup after applying optimizations (cumulative spedup shown)

Benchmark	Avg pipe var depth	Baseline	Var liveness (Section 4.5.1)	Ptr var acc (Section 4.5.2)
Toy_mpath	5.4	0.522s (1.00X)	0.483s (1.08X)	0.441s (1.18X)
Stencil	13	2.43s (1.00X)	1.16s (2.09X)	1.12s (2.17X)
MD_sim	34	0.184s (1.00X)	0.0728s (2.53X)	0.0565s (3.26X)
Mat_mul	7.8	0.0802s (1.00X)	0.0722s (1.11X)	0.0716s (1.12X)
Cholesky	8.7	0.0617s (1.00X)	0.0600s (1.03X)	0.0530s (1.16X)
NW	3.2	0.236s (1.00X)	0.224s (1.05X)	0.224s (1.05X)
LUD	19	0.0345s (1.00X)	0.0266s (1.30X)	0.0242s (1.43X)
SpMV	10	0.0853s (1.00X)	0.0808s (1.06X)	0.0803s (1.06X)
AVG	-	(1.00X)	(1.41X)	(1.55X)

rest of the benchmarks is 1.12X. This is because the proposed optimizations reduce copies of the variables in loop pipelines.

As explained in Section 4.4.1, FLASH use the FSM state assignment information and the FSM state transition information. The resource allocation / binding information, and the component library that exist in RTL code have been abstracted in FLASH, and the computation statements are instead simulated natively on the host machine. The result of this abstraction can be seen in Table 4.7. FLASH is about 1,630X (=2,800/1.72) faster than the RTL simulation. This confirms our initial speculation that simulating based on the scheduling information will result in much faster speed while solving the correctness problems.

Since our flow reflects the scheduling information, we can expect some slowdown compared to the Vivado HLS C simulation. One source of overhead is the frequent FIFO stall which lengthens the simulation process. Another source of overhead is copying pipeline variables and enable signals (this overhead is reduced by the optimizations in Section 4.5 as was shown in Table 4.6). However, it is interesting to note in that for some benchmarks such as `Toy_mpath` and `Stencil`

Table 4.7: Simulation time comparison among Vivado HLS C simulation, Vivado HLS RTL simulation, Verilator, and FLASH simulation

Benchmark	Viv HLS C Sim	Viv HLS RTL Sim	Verilator	FLASH
Toy_mpath	0.765s (1.00X)	519s (678X)	120s (157X)	0.441s (0.576X)
Stencil	1.92s (1.00X)	101s (52.6X)	119s (62.0X)	1.12s (0.583X)
MD_sim	0.0652s (1.00X)	89s (1,370X)	7.3s (112X)	0.0565s (0.867X)
Mat_mul	0.0680s (1.00X)	180s (2,650X)	29.2s (429X)	0.0716s (1.05X)
Cholesky	0.0124s (1.00X)	90s (7,260X)	27.7s (2,230X)	0.0530s (4.27X)
NW	0.136s (1.00X)	68s (500X)	27.1s (199X)	0.224s (1.65X)
LUD	0.0319s (1.00X)	129s (4,040X)	16.4s (514X)	0.0242s (0.759X)
SpMV	0.0200s (1.00X)	117s (5,850X)	55.5s (2,780X)	0.0803s (4.0X)
AVG	(1.00X)	(2,800X)	(810X)	(1.72X)

in Table 4.7, FLASH was even faster than the Vivado HLS C simulation. This suggests that there was an unexpected factor which has negated the simulation speed overhead of the proposed flow. We found that this is largely attributed to the fact that the Vivado HLS tool can allocate an unlimited FIFO buffer for C simulation (Table 4.1). To model FIFO, the Vivado HLS C simulator uses the C++ Standard Template Library (queue.h), which incurs the overhead of dynamically allocating buffer and copying its content. For example, the C simulation time of `Toy_mpath` reduces from 0.765s to 0.128s if we replace FIFO library calls with fixed-size arrays (array size is set to the number of total FIFO elements written). FLASH simulation flow does not have this problem, because the FIFO library calls have been replaced with array-based communication (Section 4.4.1.3). The average slowdown of FLASH compared to the Vivado HLS C simulation is 1.72X.

Compared to the RTL simulation, Verilator increases the simulation speed by 3.45X (=2,800X/810X). However, as mentioned in Section 2.3, the speedup is limited because it is difficult to completely remove resource allocation and binding information from the RTL file after they have been added. FLASH does not have this overhead, and as a result, FLASH outperforms Verilator by two orders

of magnitude while also achieving the cycle accuracy.

Please note that in our initial research stage, we also evaluated a similar code transformation flow that produces a SystemC simulation file. However, the overhead in the SystemC simulation environment caused a 2-3X slowdown compared to the proposed C-based flow, which motivated us to follow the current approach. Despite the slowdown, SystemC-based approach could be more useful to some tool developers if compatibility with existing SystemC simulation frameworks has a higher priority.

4.8.3 Accuracy

As explained in Section 4.3, the correctness problem is solved by simulating FIFO communication in a cycle-accurate manner. The data value and the data ordering has been verified by comparing the output of the FLASH simulator with that of the Vivado HLS RTL simulator.

In Table 4.8 we compare the cycle estimation accuracy with the Vivado HLS synthesis report after we specify the maximum loop bound for each loop. The estimation error rate is small for `Stencil`, because [17] has a built-in mechanism to allocate adequate buffers. For the rest of the benchmarks, we have applied a small (1–2) FIFO depth (an example was shown in Fig. 4.4). This causes the FIFO buffer to be frequently full and empty and leads to worse performance. Thus, the HLS synthesis reports' estimate is smaller than the RTL simulation result. For `LUD` and `SpMV`, on the other hand, the Vivado HLS tool provides a very large overestimate of the execution cycles. The reason is that these applications have variable loop bounds, and Vivado HLS generates the cycle estimate based on the maximum possible loop bounds [24]. FLASH simulates FIFO stalls and loops with variable bounds in a cycle-accurate fashion, and the estimated execution time accurately matches that of RTL simulation.

4.9 Concluding Remarks

With a new HLS software simulation flow based on the scheduling information, we were able to solve the correctness issue and also provide accurate performance estimation. A cycle-accurate

Table 4.8: Total execution cycle estimated by Vivado HLS synthesis report and FLASH, and its error rate compared to the RTL-simulated result

Benchmark	RTL sim	Viv HLS syn rpt	FLASH
Toy_mpath	4,500,010	4,000,019	4,500,010
	-	(-11%)	(0%)
Stencil	524,309	524,299	524,309
	-	(~0%)	(0%)
MD_sim	12,089	10,524	12,089
	-	(-13%)	(0%)
Mat_mul	330,006	131,075	330,006
	-	(-60%)	(0%)
Cholesky	40,741	34,996	40,741
	-	(-14%)	(0%)
NW	245,725	131,112	245,725
	-	(-47%)	(0%)
LUD	201,260	561,153	201,260
	-	(180%)	(0%)
SpMV	163,859	395M	163,859
	-	(240K%)	(0%)

simulation result was obtained three orders of magnitude faster than from RTL simulation, because the new simulation flow is not slowed by allocation / binding information and component library. We have described an automated code generation flow that enables this new simulation flow.

We hope that the promising results presented in this work will motivate the HLS commercial tool industry to provide additional routines that simulate based on the scheduling information only. This will substantially decrease the validation time of the customers who wish to rapidly estimate cycle-accurate performance, obtain correct output data, or detect possible deadlock situations.

CHAPTER 5

On-Board Monitoring for Performance Debugging

5.1 Introduction

In Chapters 3 and 4, simulation-based performance debugging flows were proposed. These flows have high accuracy in predicting the performance of HLS designs when representative input datasets exist for simulation purpose. However, if the input may vary at the time of FPGA deployment, on-board monitoring becomes necessary. Another problematic case is where high-level models do not exist for all components of the FPGA design. For example, a design may have an RTL sub-component or DRAM with unknown characteristics.

To address these challenges, on-board monitoring becomes necessary. Although there are some works that provide on-board correctness debugging capability for HLS designs [51–53, 89, 90], these works do not provide insight on solving the performance problems. Instead, we propose an HLS-based in-FPGA performance monitoring flow called HLScope-M. Using non-blocking FIFO access and pipelining, we propose a source-to-source (S2S) transformation method that can be expressed in HLS without the need for mixed HLS-HDL flow. The code instrumentation for parameter extraction is automated, as we will be demonstrating with the quicksort example. Finally, we will propose Stall Analysis Network (SAN) that analyzes the stall reason even for designs without explicit synchronization.

5.2 Cycle Extraction Based on In-FPGA Monitoring

An in-FPGA monitoring flow can be used to extract cycle information more accurately than the simulation-based flow, at the cost of spending time on generating the FPGA bitstream. For easy in-

tegration with HLS-based synthesis flow [128], we would like the monitoring logic to be expressed in pure HLS code. Then the biggest challenge is to extract cycle information, which is hidden from programmers in HLS. To solve this problem, we propose a technique of using non-blocking FIFO read and pipelining.

HLScope-M instruments two types of probes: 'pX' probes (*e.g.,* p0, p1, and p2) to signal start and end of module execution ('1' for start and '0' for end) and 'p_endmtr' to signal monitor termination. It also inserts 'dbg_memX' debug registers (declared as global variable) to record accumulated execution cycle of each module. Finally, a monitor logic is added for overall processing.

A sample code instrumentation is given in Fig. 5.1 for quicksort. HLScope-M inserts a monitoring logic *monitor_3p()* to run in parallel with *qsort_top()* which contains three submodules under analysis - *load(), qsort_comp(), and store()* (Fig. 5.1a). For each submodule, 'pX' probe sends 1/0 to signal module start/end (Fig. 5.1b). These 'pX' probes are connected to the monitor through FIFO (Fig. 5.1a). For monitor termination, 'p_endmtr' sends a value of 0 at the end of *qsort_top()* module (Fig. 5.1a). Note that a monitor start signal is unnecessary, because the monitor begins its operation as soon as the whole design has been reset.

In the monitoring logic, variable 'cycle' corresponds to the actual hardware cycle. This is possible since the loop is pipelined to II=1, and all FIFO reads are declared non-blocking (Fig. 5.1c). Then the loop can run continuously regardless of the input. Note that 'cycle' is incremented by one for each loop iteration.

Based on 'cycle' variable, we subtract the cycle obtained at the start of submodule 0 execution from the cycle obtained at the end, and accumulate to 'dbg_mem0' (Fig. 5.1d). The same is done for submodule 1 and 2. After the 'p_endmtr' has been processed for program termination (Fig. 5.1c), 'dbg_memX' content is written to DRAM for offline analysis (Fig. 5.1e).

```
void qsort_top(..., stream<bool> & p0, p1, p2, p_endmtr ){
  float lmem[LMEM_MAX];
  for( int i = 0 ; i < batch_num ; i++ ){
    load(dram, lmem, n_per_batch, i, p0);
    qsort_comp(lmem, n_per_batch, p1);
    store(dram, lmem, n_per_batch, i, p2);
  }
  p_endmtr.write(0); // signals monitor termination
}
void qsort_top_new(...){
#pragma HLS dataflow //monitor and modules run in parallel
  stream<bool> p0, p1, p2, p_endmtr;  // FIFO probes
  qsort_top(..., p0, p1, p2, p_endmtr);
  monitor_3p(p0, p1, p2, p_endmtr);
}
```

(a) Connection from modules under analysis to the monitor logic

```
void qsort_compute( ..., stream<bool>&p1 ){
  p1.write(1); // module start signal through probe
  ... ...
  p1.write(0); // module end signal
}
```

(b) Signals monitor for module start/end

```
void monitor_3p(stream<bool>&p0, p1, p2, p_endmtr){
  bool endmtr=0; int p0_start=0; ...; int cycle=0;
  while(endmtr == 0){
#pragma HLS PIPELINE II=1 //loops once per HW cycle
    . . .
    // start/end probe processing (please see (d))
    . . .
    bool endmtr_data = 1;     //monitor termination
    if ((p_endmtr.read_nb(endmtr_data)) == 1) {
      if (endmtr_data == 0) { endmtr = 1; }
    }
    cycle += 1;   //"cycle" becomes actual HW cycle
} }
```

(c) Monitor logic structure & termination probe processing

```
  bool p1_data = 1;
  if ((p1.read_nb(p1_data)) == 1){ //non-blk read
    if (p1_data == 0) {            //success
      dbg_mem1 += cycle – p1_start;
    }                        // record module end
    else {
      p1_start = cycle;     // record module start
  } }
```

(d) Start/end probe processing in monitor logic

```
void dbg_top( . . ., int* dbg_dram ){
  qsort_top_new( . . . );
  . . . dbg_dram[1] = dbg_mem1; . . . //dbg_memX written to
}                                     //DRAM for analysis
```

(e) Writing debug registers to external memory for offline analysis

Figure 5.1: Code instrumentation for in-FPGA monitoring (instrumented code in bold)

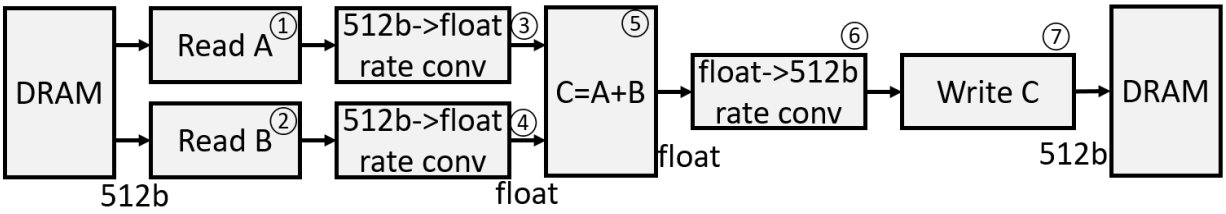


Figure 5.2: Dataflow vector add connected through FIFO. Circled number is the module ID.

5.3 In-FPGA Stall Analyzer for FIFO-based Dataflow Application

The in-FPGA monitor insertion proposed in the previous section can provide a stall reason analysis only when modules have explicit synchronization among them. The analysis will not be accurate if modules have no explicit synchronization and utilize FIFOs as means of communication. For a simple example of vector add in Fig. 5.2, all modules will execute in parallel and wait for data to be produced or consumed by other modules. Thus, module 5 (C=A+B) will stall even if only one of the modules in the A/B read pipeline (FIFO being empty) or C write pipeline (FIFO being full) stalls. Then the challenge becomes tracing the exact source of stalls in run-time.

We solve this problem by instrumenting a series of probes into modules under analysis (MUA) and FIFOs. Also, we instantiate a network of the probe processing logic (*monitors*). The details will be presented in the following subsections.

5.3.1 Code Instrumentation for Module Under Analysis

We start by regaining the visibility of the module status (in execution or stalled) that is hidden due to HLS abstraction. Fig. 5.3 describes the process for module 1 (*Read A*), and Table 5.1 lists the series of probes used. Probe *p_status* is inserted to indicate whether the module is in inter-module communication mode or not. *p_status.write(1)* is inserted before start of the pipeline that contains the inter-module communication (*fifo_512.write()*), and *p_status.write(0)* is inserted to signal the end of pipeline. [2]

Even if the module is executing the inter-module communication pipeline, it may stall if the

[2]It is also possible to analyze the stall reason due to DRAM access with two-bit (*e.g.,* 0:in computation, 1:in inter-module communication pipeline, 2:accessing DRAM) status probe, rather than one-bit.

```
void read_A(..., hls::stream<uint512> &fifo_512_r,
           hls::stream<bool> &p_status, &p_active, ... )
{
  uint512 bram[1024];
  for (int i = 0; i < vec_size ; i++) {
    memcpy( bram, global + i * 1024, 1024*64 );
    p_status.write(1); //start of inter-mod comm pipeline
    for (int j = 0; j < 1024; j++) {
#pragma HLS PIPELINE II=1
      p_active.write(1); //pipeline active
      fifo_512_r.write(bram[j]);
    }
    p_status.write(0); //end of inter-mod comm pipeline
  }
  p_exit_mtr.write(0); p_exit_fifo.write(0); //module end
}
```

Figure 5.3: Code instrumentation for *Read A* module. Instrumented code in bold.

Table 5.1: List of probes for module under analysis

Probe name	Description
p_status	Is in inter-module communication mode?
p_active	Is the communication pipeline not stalled?
p_exit_mtr	Terminate signal for stall analysis monitor.
p_exit_fifo	Terminate signal for instrumented FIFO.

FIFO is full or empty. To observe this, we instrument *p_active* that writes '1' if the communication pipeline is active. If stalled, it will not write anything.

Finally, we also instrument logic termination signal (value of 0) for the monitor logic (*p_exit_mtr*) and instrumented FIFOs (*p_exit_fifo*) that will be explained in the following subsections.

5.3.2 Code Instrumentation for FIFOs

The activeness of the module under analysis can be observed using the previous techniques, but if multiple FIFOs are connected to a module, it is difficult to find which FIFO is causing the stall. For this analysis, we also instrument measurement logics into the inter-module FIFOs. An example is given for the FIFO between module 1 and module 3 in Fig. 5.2. Two measurement logics are inserted between the write and the read module: *full_mtr* and *empty_mtr* that generate the fullness

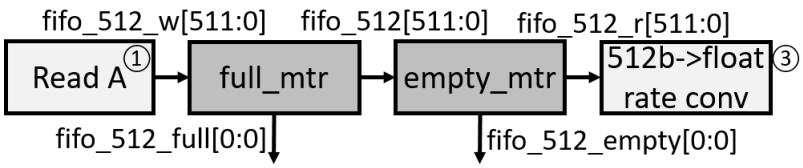


Figure 5.4: Instrumentation of *full_mtr* and *empty_mtr* for FIFO

(*fifo_512_full*) and emptiness (*fifo_512_empty*) signal of the FIFO (Fig. 5.4).

The detailed code of the *full_mtr* logic is shown in Fig. 5.5. If fifo write to *fifo_512* is blocked, it will send '1' via wire *fifo_A_full* to the monitor logic (explained in the next subsection) to signal fullness of the FIFO. Also, all data reads and writes are performed in non-blocking mode, and the pipeline is set to II of 1. As a result, the throughput of the original FIFO is maintained, and the instrumented logic does not cause additional stall to the original logic. In addition, a temporary data buffer is used to store a read data that was not written due to output being stalled. Though omitted, similar code is used for *empty_mtr* logic that monitors the emptiness of the FIFOs.

5.3.3 Monitor for Stall Analysis Network

In order to analyze the stall reason for module 5, for example, we would have to observe the status of all read pipelines (modules 1, 2, 3, 4) and write pipelines (modules 6, 7). However, this is not scalable since an interconnect of quadratic complexity would be needed between all MUAs and all monitors.

Instead, we propose a distributed stall analysis network (SAN). Part of the proposed logic is shown in Fig. 5.6. One monitor is instantiated per module under analysis. Each monitor classifies its host MUA as being active, or being stalled due to its neighbor. As described in Table 5.2, module being "active" is the period when the module is not in inter-communication mode (*p_status=0*) or when it is in communication and the pipeline is active (*p_status=1* and *p_active=1*). The module being stalled is when the module is in communication and the pipeline is not active (*p_status=1* and *p_active=N/A*). The reason for stall can be found by observing the empty or full signal sent from the monitored FIFO logic. If multiple stall reasons are asserted, we designed monitor to

```
void full_mtr(hls::stream<uint512> & fifo_512_w, fifo_512,
           hls::stream<bool> & fifo_512_full, p_exit_fifo)
{
  bool loop_exit = 0;
  float data_buffer; // temporary data buffer
  bool data_in_buffer = 0; //data exists in data_buffer?

  while( loop_exit == 0 || data_in_buffer == 1 ){
#pragma HLS pipeline II=1
    if( data_in_buffer == 0 ){ // data_buffer empty
      if( fifo_512_w.read_nb(data_buffer) == 1 ){
        if( fifo_512.write_nb(data_buffer) == 0 ){
          fifo_512_full.write(1); // FIFO is full
          data_in_buffer = 1; //data stored to data_buffer
    } } }
    else{ // data_buffer not empty
      if( fifo_512.write_nb(data_buffer) == 1 ){
        if( fifo_512_w.read_nb(data_buffer) == 0 ){
          data_in_buffer = 0; // data_buffer empty
      } }
      else{
        fifo_512_full.write(1); // FIFO is full
    } }
    bool exit_data;
    if( p_exit_fifo.read_nb(exit_data) == 1 ){
      if( exit_data == 0 ){
        loop_exit = 1; // terminate monitoring
    } }
} }
```

Figure 5.5: Code for *full_mtr* logic

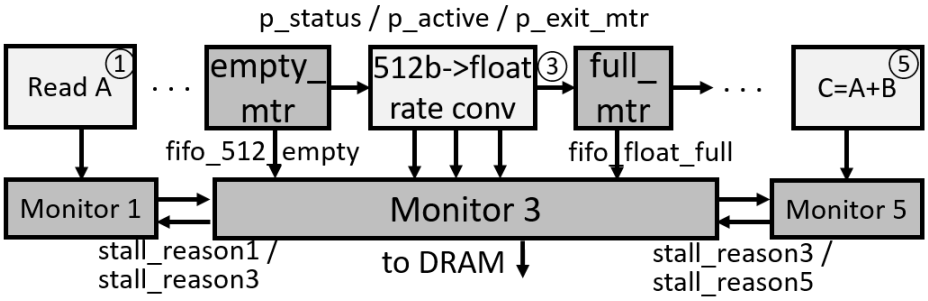


Figure 5.6: Distributed stall analysis network

arbitrarily select one of the reasons for the simplicity of the implementation.

Table 5.2: List of states in the SAN monitor 3

Monitor state		Active		Stalled due to 1	Stalled due to 5
Input	p_status	0	1	1	1
	p_active	-	1	N/A	N/A
	f_512_empty	-	-	1	0
	f_float_full	-	-	0	1
Output	st_reas3	3		st_reas1	st_reas5
Register	act_cyc	+=1		-	-
	st_cyc[st_reas1]	-		+=1	-
	st_cyc[st_reas5]	-		-	+=1

If the FIFO port that causes the stall is identified, each monitor will record the stall reason sent from a neighbor monitor as being the root cause of the stall. This reason is generated at each monitor. If the MUA is active (*p_status=0 || (p_status=1 && p_active=1)*), the monitor will broadcast its own ID to its neighbor monitors. If its MUA is inactive (*p_status=1 and p_active=N/A*), it will pass on the stall reason (*st_reasX*) of the FIFOs that have been stalled.

We record the amount of cycles for each monitor state (active or stalled due to particular module) in registers (*act_cyc* or *st_cyc[X]*). When monitoring has finished (*p_exit_mtr=0*), each monitor will write its register contents to the DRAM for offline analysis.

Since each module passes the stall reason from its neighbors, it can identify the stall module even if it is several hops away. Also, SAN is scalable because the connection of each monitor is limited to its MUA, instrumented FIFOs, and the neighbor monitors.

5.4 Experimental Results

The in-FPGA flow is cycle accurate by its nature and does not need accuracy testing. In this section, we provide the time and logic overhead.

5.4.1 Experimental Setup

For our evaluation platform, we use the Alpha Data ADM-PCIE-7V3 board [2] that has Xilinx's Virtex 7 690T FPGA. For the FPGA synthesis, we use the Xilinx SDAccel 2016.2 [128] and Vivado HLS 2016.3 [130] software tools.

5.4.1.1 Time Overhead

The time taken to perform code instrumentation, module dependency analysis, critical path analysis, and stall reason analysis of various versions of quicksort (Section 3.1.3) is 13–33 seconds (shown in Table 5.3). The synthesis takes takes several hours.

Table 5.3: Overhead of in-FPGA debugging flow for various versions of quicksort.

	Dbg Time	Bitstr Gen
unopt	13s	1h27m
unrolled	22s	2h4m
doublebuf	33s	1h55m

5.4.2 Logic Overhead

The in-FPGA flow has logic overhead for the performance monitor insertion (Table 5.4). The logic consumption increases with the number of modules to be monitored at the rate of approximately 170 LUTs per probe. Also, one BRAM per probe is needed since it is FIFO-based.

Table 5.4: Logic overhead of monitors for in-FPGA flow.

# of probes	LUT	DSP	BRAM
4	654	0	4
16	2802	0	16
35	6072	0	35

SAN requires extra logic for on-board implementation. Each probes (*e.g., p_active* and *p_status*) in a MUA consumes about 20 LUTs. The instrumented FIFO consumes LUT that approximately increases linearly with the data bitwidth, as shown in Table 5.5. The monitor resource usage slightly increases with the number of neighboring monitors, as shown in Table 5.6. None of them

consume any DSP or BRAM.

Table 5.5: Logic overhead of instrumented FIFO

	empty_mtr		full_mtr	
bitw	float	uint512	float	uint512
LUT	46	526	49	529

Table 5.6: Logic overhead of SAN monitor

# of neighbors	1	2	3
LUT	903	930	957

5.5 Comparison with Related On-board Debugging Work

As explained in Section 2.4, on-board debugging flows such as Chipscope, SignalTap, and works in [51–53, 89, 90] provide functional correctness debugging capability. HLScope-M, on the other hand, can be used to provide insight on solving the performance problems of an FPGA design (please refer to Section 3.1 for detailed performance debugging process).

Compared to other on-board performance debugging works [42, 62, 71, 108, 120] described in Section 2.4, HLScope-M has two main differences. The first difference is that HLScope-M analyzes the stall reason of individual modules and points to the root cause of the stall in high level. Such analysis assists programmers to quickly identify the performance bottleneck. The second difference is that HLScope-M's monitor is based on pure HLS code for easy integration with the existing HLS synthesis flows. Note that mixed HDL-HLS flow complicates the integration process in synthesizing FPGA program—for example, Intel's OpenCL HLS tool [62] does not allow mixed HLS-HDL flow, and Xilinx's SDAccel [128] (based on Vivado HLS [130]) allowed it only in their recent versions.

5.6 Concluding Remarks

In this chapter, we have described on-board cycle measurement framework for HLS-based FPGA performance debugging. Also, SAN has been proposed to analyze stall reason of modules without

explicit synchronization. The code instrumentation and measurement has been automated in our tool. The flow is cycle-accurate, and the monitor consumes few hundreds of LUTs per monitored module.

CHAPTER 6

Fast Design Space Exploration for Applications with Dynamic Behavior

6.1 Introduction

The performance debugging flows introduced in previous chapters help FPGA designers collect information such as the number of cycles, the stall rate, and the DRAM access time. Based on this information, FPGA designers would perform optimization. However, there are many different combinations of HLS directives that can be applied for each design, and it takes several minutes for the HLS tool to transform a C-level design into a RTL code. This makes it difficult for a HLS designer to find the set of parameters that will maximize the performance. This calls for automated design space exploration (DSE) for HLS. A considerable amount of literature can be found on this topic (Section 2.5). However, the previous work has difficulties finding an efficient design for applications with variable loop bounds in the innermost loops. For example, the average speedups for Polybench [99] benchmarks with variable loop bounds [99] are 2.3X and 1.0X using AutoAccel [37] and COMBA [137], while our proposed approach achieved a 75X speedup.

The main reason for such small speedup in conventional DSE works is that commonly used HLS directives $unroll$ and $pipeline$ may not be suitable for this type of application. As will be explained in Section 6.2, these common directives would generate processing elements (PE) with low utilization (<0.2) unless the code is properly transformed. For this reason, existing DSE tools that simply insert optimization directives will not be able to fully optimize the application.

Another reason for the poor optimization is due to the difficulty in performing cycle analysis. Vivado HLS will provide a very large range of cycles for variable loops, and the exact performance

after each optimization becomes difficult to predict. It is possible to estimate the performance of applications with variable bounds based on the software simulation flow as proposed in HLScope [23, 24]. However, HLScope requires HLS synthesis of every design point to extract the loop cycle parameters—which is often infeasible. Works such as [111, 137, 138] use their own schedulers without the actual Vivado HLS synthesis; however, the accuracy might not be very satisfactory for loops with variable bound as will be shown in the experimental result.

To solve these problems, we present our initial work on an HLS-based optimization and DSE framework that improves the performance—even in the presence of variable loop bounds. First, we will demonstrate the deficiency in commonly used HLS pragma $unroll$ and $pipeline$ if used on variable loops. As a solution, we will propose source-to-source HLS code transformations that increase the utilization of the compute resources for variable loops with partial unrolling and pipelining in Section 6.2.2.

Furthermore, we identify the efficiency challenge that originates from the loop-carried dependency and the variable loop bounds. In particular, we analyze floating-point variable-loop reduction and prefix sum patterns that frequently appear in Polybench [99]. We will present code transformations to optimize these computation patterns in Sections 6.2.3 and 6.2.4.

Next, we present a cycle and resource estimation model for the variable loops in Section 6.3. In order to make an accurate resource estimation for large design spaces in a short time, our model interpolates based on a small number of actual HLS synthesis runs and considers the sharing of operands and arrays for various unrolling and array partitioning factors. The cycle estimation model has its basis on the flow in Chapter 3 (HLScope-S), because the performance estimation can be obtained most quickly with the high-level approach of HLScope-S (quantitative comparison will be provided in Section 7.1.3). The difference is that whereas HLScope-S provides cycle estimate for a single design, the work in this chapter uses the data collected from the software simulation to predict the execution time for multiple designs. The details will be explained in Section 6.3.

The overall framework and the DSE flow are explained in Section 6.4. Finally, the experimental result and the comparison with other works on the Polybench benchmark are presented in Section 6.5.

```
for(int k = 0; k < N; k++){        //loop1
  for (int j = k + 1; j < N; j++){ //loop2
    A[k][j] = A[k][j] / A[k][k];
  }
  ...
}
```

Figure 6.1: Variable loop bound example in LU benchmark

6.2 HLS Code Transformation for Variable-Bound Loops

In this section we will first identify the limitation of applying directives for pipelining and unrolling based on the maximum bound. Next, we will demonstrate the effectiveness of applying source-to-source transformation. In Polybench [99] the applications with variable loop bounds can be classified into three patterns: completely parallel, reduction, and prefix sum. We will discuss the transformation for each pattern in the following subsections.

6.2.1 Loop Pipelining and Loop Unrolling Based on the Maximum Loop Bound

Since HLS tools cannot unroll the loops with variable bounds, a common optimization strategy is to pipeline the loop. For illustration, we optimize loop 2 of the LU baseline code (Fig. 6.1), which has matrix size N=512. After pipelining, the loop can be executed in 130,831 cycles (measured using technique described in [24]). However, pipelining cannot exploit the data-level parallelism that exists in the loop.

Another intuitive optimization strategy is to unroll based on the maximum loop bound measured from testbench profiling. The loop bound is fixed to a constant value as shown in Fig. 6.2. Condition ($if(j >= k + 1 \&\& j < N)$) is added to invalidate the execution of iterations where the loop index is not between the upper bound and the lower bound of the loop.

However, the resulting unrolled architecture suffers from a severe PE efficiency problem. Even if the dividers were all instantiated, profiling shows that the architecture can process only 130,816 divisions in 4,440,576 cycles—resulting in no speedup. On average, a divider PE is only perform-

```
for (int j = 1; j < 512; j++){
#pragma HLS unroll complete
  if(j >= k + 1 && j < N ){
    A[k][j] = A[k][j] / A[k][k];
} }
```

Figure 6.2: Loop unrolling for loop 2 of baseline LU code (Fig. 6.1) based on the maximum loop bound found in profiling

ing 0.000057 divisions per cycle. There are two reasons for such inefficiency: First, many PEs are left idling when the loop trip count is smaller than the maximum loop bound. Second, the unrolled PEs are not pipelined. Due to such a low PE utilization problem, Vivado HLS only instantiates a single divider and shares it across the loop iterations.

6.2.2 Partial Unrolling with Pipelining

In order to exploit the loop parallelism while solving the PE inefficiency problem described in the previous subsection, we apply code transformations based on partial unrolling and pipelining. The idea is to place fewer PEs but allow them to proceed to other iterations in a pipelined fashion so that the effective PE utilization ratio would be increased.

Vivado HLS-compatible code transformation steps for the LU benchmark (Fig. 6.1) are as follows. As a preprocessing step, a common array reference (*e.g.,* $A[k][k]$) that is invariant to the loop is replaced with a temporary scalar variable ($lc1$) and moved out of the loop, as shown in line 3 and line 10 of Fig. 6.3. The reason is that Vivado HLS will synthesize it to an actual BRAM lookup and unnecessarily consume additional read port per iteration.

Next, we separate the original loop into two loops L2_1 (line 4) and L2_2 (line 7). The inner loop bound L2_2 is fixed to a constant $L2\_UF$ (line 7) so that the HLS tool may fully unroll it after inserting a pipeline directive on the outer loop (line 5). Assuming the original loop has lower bound lb and upper bound ub, the outer loop's lower/upper bound is set to $lb/L2\_UF$ and $(ub - 1)/L2\_UF + 1$ (line 4), so that the boundary iterations will be included as well. Arrays referenced in the loop are partitioned (line 2) to the unrolling factor ($L2\_UF$) in the array dimension (dim=2)

```
01 #define L2_UF 4
02 #pragma HLS ARRAY_PARTITION variable=A factor=4 dim=2

03 float lc = A[k][k];                // loop-invariant code motion
04 for (int j1 = (k + 1)/L2_UF; j1 < (N-1)/L2_UF + 1; j1++){//L2_1
05 #pragma HLS pipeline
06 #pragma HLS DEPENDENCE variable=A inter false
07   for(int j2 = 0; j2 < L2_UF; j2++){ //fully unrolled    //L2_2
08     int j = j1 * L2_UF + j2;
09     if(j >= k + 1 && j < N ){//from orig loop's upper/lower bnd
10       A[k][j] = A[k][j] / lc;
11 } } }
```

Figure 6.3: Code after applying the proposed partial unrolling and pipelining techniques to loop 2 of Fig. 6.1

referenced by the unrolled loop's index (j). If the loop index exists in more than one dimension (*e.g.,* $A[j][j]$), the proposed transformation is not valid. We also place a conditional statement to exclude iterations that were not between lb and ub (line 9).

As a last step, if there was no loop-carried dependency before splitting into two loops, we insert a pragma to declare inter-loop dependency to $false$ (line 6) for better performance [130]. This is legal because j ($= j1 * L2\_UF + j2$) increases monotonically, and thus $A[k][j]$ will never reference the same array address in previous iterations.

The cycle estimation model of the proposed transformation will be presented in Section 6.3.2.1. Comparison of the execution cycles and the PE efficiency (average divisions per cycle per divider PE) is shown in Table 6.1. The proposed transformation exploits the loop parallelism and allows exploring various partial unrolling factors for wider design space exploration.

Table 6.1: Comparison of the execution cycles and the PE efficiency for loop pipelining (Section 6.2.1), loop unrolling based on the maximum bound (Section 6.2.1), and the proposed partial unrolling with pipelining (Section 6.2.2)

	Pipeline	Max Unr	Proposed Transformation		
			$UF_T = 2$	$UF_T = 4$	$UF_T = 8$
Ex cyc	130,831	4,440,576	81,792	49,088	32,736
PE eff	1.0	0.000057	0.80	0.67	0.50

```
for (int j = i + 1; j < N; ++j){ // loop 3
  x = A[i][j];
  for (int k = 0; k < i ; ++k){  // loop 4
    x = x - A[j][k]*A[i][k];
  }
  A[j][i] = x * p[i];
}
```

Figure 6.4: Variable reduction example in Cholesky benchmark

6.2.3 Transformation for Variable Reduction

A reduction pattern is detected when a reduction operator [15] (*e.g.*, addition or multiplication) is applied on multiple array elements over a loop, and the result is reduced to a single variable or an array element. Subtraction can also be computed as a reduction pattern after replacing subtraction with addition, and the sign of the final reduction result is flipped. An example can be found in loop 4 of Polybench's Cholesky benchmark (Fig. 6.4). Note that in a strict sense, a floating-point addition is not a reduction operation (needs to have commutative and associative property), but it can be computed as a reduction pattern if some errors are tolerable [15, 27].

Many FPGA designs utilize a binary tree structure when implementing a reduction circuit [91, 140]. In Vivado HLS, this structure is inferred by specifying directive: "*#pragma HLS unroll factor=xxx*." However, this implementation style is inefficient for floating-point variable loop reduction. The width of the tree has to be set to half of the maximum of the loop bound (256 in the example), and the depth has to be set to the log_2 of the width (8 in the example). If the loop bound is much smaller than the maximum, many adders will be left idle. To increase the PE efficiency, Vivado HLS will share the adders between different levels in the reduction tree. However, the efficiency is still low, because Vivado HLS does not properly pipeline the PEs when a partial unrolling factor is specified (Section 6.2.1). For example, when loop 4 of Cholesky benchmark is unrolled to the factor of 256 times (could not be fully unrolled to 512 due to the resource limitation), the floating-point adder (FADD) efficiency is 0.008 (= 22M adds / 21M cycles / 256 FADDs).

Inserting pipelining directive is also not very efficient for floating-point reduction. The reason

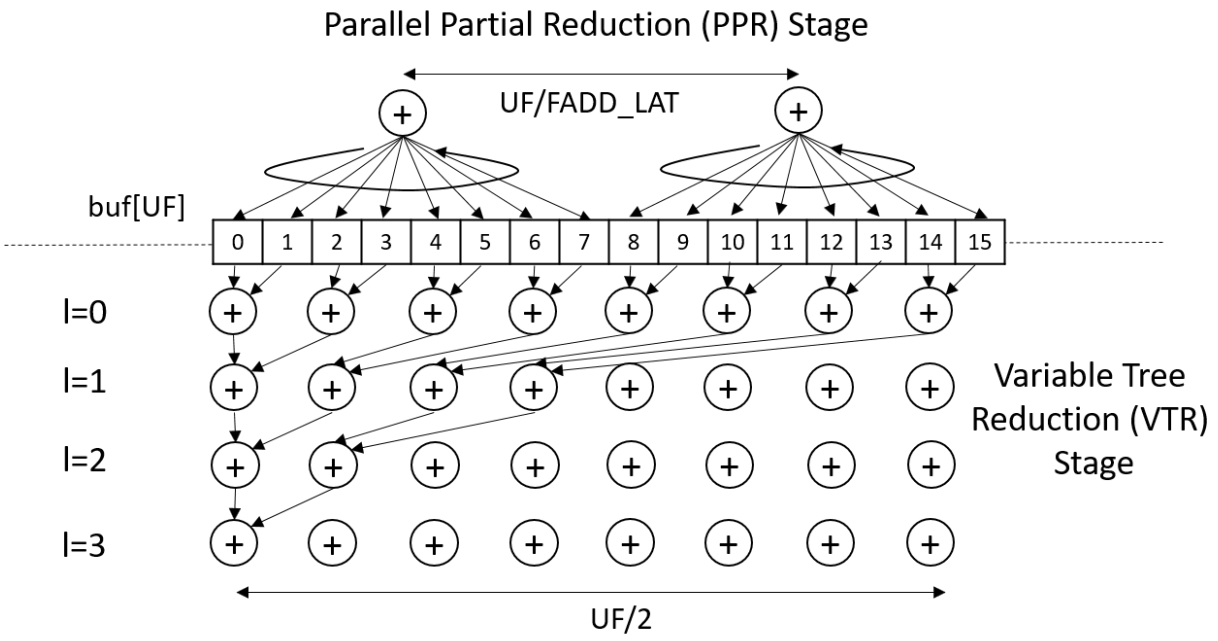


Figure 6.5: The computation pattern of variable reduction

is that there is a true loop-carried dependency, and the result of the previous iteration cannot be immediately produced because of the long latency of the floating-point operations (FADD_LAT). In loop 4 of the Cholesky example, FADD_LAT is 8 cycles. Thus, the average PE efficiency for pipelining is only 0.12, and requires 179M cycles to complete. To solve this problem, the work in [62, 142] proposes using shift registers (of length that matches FADD_LAT) to remove the dependency. The average PE efficiency improves to 0.16, but the parallelism is still limited.

We propose a code transformation to address these problems and to enable design space exploration. The reduction operation is divided into two stages (Fig. 6.5) depending on whether the number of elements to be reduced exceeds UF or not (UF: addition unrolling factor).

The first stage (lines 9 15 in Fig. 6.5) is called parallel partial reduction (PPR) stage. In this stage, a large number of input numbers are reduced to an array of partial sums. In order to increase the efficiency, we adopt the dependence-free pipelining [62, 142] by allowing a single FADD to write intermediate results to $FADD\_LAT$ registers. This is achieved by specifying II to $FADD\_LAT$ (line 10 of Fig. 6.6). This will induce HLS to schedule $FADD\_LAT$ additions in $FADD\_LAT$ cycles—effectively achieving II=1 for each FADD. For higher performance, we in-

```
00 #define UF 16                                                  // unroll factor
01 #pragma HLS ARRAY_PARTITION variable=A cyclic factor=2 dim=2
02 #pragma HLS RESOURCE variable=A core=RAM_2P_BRAM  //we assume A is in BRAM
03 float buf[UF];                      //buf is a completely unrolled register
04 #pragma HLS ARRAY_PARTITION variable=buf complete

05 for( int a = 0 ; a < UF ; a++ ){
06 #pragma HLS unroll complete
07   buf[a] = 0;                                                 //initialize to 0
08 }

09 for( int k1 = (0)/UF; k1<(i-1)/UF+1; k1++ ){                 // L4_1, PPR stage
10 #pragma HLS pipeline II=8
11  for( int k2 = 0 ; k2 < UF ; k2++ ){                          // fully unrolled
12    int k = k1*UF+k2;
13    buf[k2] += ( k >= 0 && k < i ) ? A[j][k]*A[i][k] : 0;
14  }          (from baseline loop's    (from baseline loop's comp
15 }           lower / upper bound)          to be reduced)

16 int limit = log(min(TC,UF));          //sets reduction loop limit for early
17                                         // termination (uses table look up)

18 for ( int l = 0; l < limit; l++ ){                          // L4_2, VTR stage
19 #pragma HLS pipeline II=8
20 #pragma HLS DEPENDENCE variable=buf inter false
21   for( int a = 0 ; a < UF/2 ; a++ ){
22     buf[a] = buf[2*a] + buf[2*a+1];
23 } }

24 x -= buf[0];                              //final result is stored in buf[0]
```

Figure 6.6: HLS code for loop 4 of Fig. 6.4 after transformation

crease the parallelism to $UF/FADD\_LAT$. In order to reduce the number of iterations for small loop bound, early termination within a reduction level is allowed by setting the loop bound of the PPR stage to $(ub-1)/UF+1$ to lb/UF (line 9).

The second stage (lines 18 23 in Fig. 6.5) is called variable tree reduction (VTR) stage. With $UF/2$ FADDs, each level can be computed after FADD_LAT (=8) cycles, because the number of elements to be added per level is equal to or less than $UF/2$. In order to reduce the tree depth for a small loop bound TC, we support early termination by pipelining each level with a variable loop bound (line 18 of Fig. 6.6). VTR stage loop bound ($limit = log_2(min(TC, UF))$) is precomputed based on the table lookup (lines 16–17). Note that Vivado HLS instantiates only $UF/4$ FADDs for VTR stage, which increases the pipeline depth of the loop by 1 (FADD_LAT+1 cycles in total).

The cycle estimation model of the proposed transformation will be presented in Section 6.3.2.2. The performance comparison of the proposed reduction scheme with conventional pipelining, dependence-free pipelining [62, 142], and conventional unrolling is presented in Table 6.2. The DSP efficiency of the proposed scheme is higher than the conventional unrolling scheme with the same unrolling factor by 41X (UF=4) to 29x (UF=16). Similar high efficiency can be observed in FF and LUT as well. The high efficiency diminishes with the larger unrolling factor—but nonetheless, the proposed scheme was able to find a final design point that is 14X, 2.2X and 1.7X faster than the conventional pipelining, dependence-free pipelining, and conventional unrolling. Also, due to the early termination functionality for small variable bound across and within reduction levels, the latency for loop bound of 1 (42 cycles) is smaller than dependence-free pipelining (56 cycles) and conventional unrolling of factor 256 (168 cycles).

6.2.4 Transformation for Variable Prefix Sum

A prefix sum is a computational pattern where the output array y contains a running sum of the input array x ($y_k = \sum_{j=0}^{k} x_j$) [27, 57]. The prefix sum pattern is detected when there exists a loop with an assignment statement written to an array element $y[k]$ with a value that is the sum of the array element assigned in the previous iteration ($y[k-1]$) and an element from the input array ($x[k]$). An example is presented in loop L2_2 of Fig. 6.7 for rotated integral image application [81]

Table 6.2: Comparison of the total execution cycles, resource consumption, and latency of various loop bounds (for cases min=1, max=512) for the proposed variable-bound reduction scheme with conventional pipelining, dependence-free pipelining [62, 142], and conventional unrolling, for loop 4 of the Cholesky benchmark

	Unr Fac	Total Cycles	Resource			Latency	
			DSP	FF	LUT	LB=1	LB=512
Pipe	-	179M	5	653	680	15	4103
DPip	-	29M	11	3131	2904	56	560
Unrolling	4	537M	16	1922	1674	4104	4104
	8	336M	32	3706	3036	2568	2568
	16	202M	64	7234	5691	1544	1544
	256	22M	1K	114K	86K	168	168
Proposed	4	30M	7	2016	2911	42	570
	8	20M	14	3760	5032	42	322
	16	16M	28	7192	9252	42	202
	128	13M	224	56K	67K	42	114

```
L2 : for(int a=0; a<N; a++){ // Compute lower-left triangle
  L2_1 : for(int j=0; j<N-a; j++){ //Aggregate all except intimg[i-1][j-1]
    int i = a+j;
    float arg1 = (j!=N-1 && i!=0) ? intimg[i-1][j+1] : 0;
    float arg2 = (j!=0 && j!=N-1 && i!=0 && i!=1) ? intimg[i-2][j] : 0;
    float arg3 = img[i][j];      float arg4 = (i!=0) ? img[i-1][j] : 0;
    data[j] = arg1 - arg2 + arg3 + arg4;
  }

  psum[0]= data[0]; //init
  L2_2 : for(int k=1; k<N-a; k++){ //compute integral image in diagonal
    psum[k] = psum[k-1] + data[k]; //prefix sum pattern
  }

  L2_3 : for(int k=0; k<N-a; k++){ //copy prefix sum result into intimg
    intimg[a+k][k] = psum[k];
} }
```

Figure 6.7: Baseline code for rotated integral image computation [81] used in face recognition.

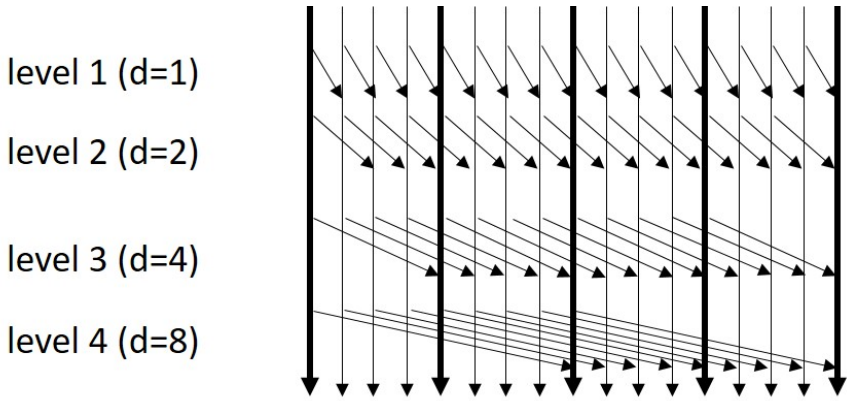


Figure 6.8: Kogge-Stone prefix sum algorithm [73].

in face detection. Similar to the reduction pattern (Section 6.2.3), the true dependency between $psum[k]$ and $psum[k-1]$ prohibits II becoming 1 when the loop is pipelined and $psum$ is a floating-point variable. Applying an unrolling directive as suggested in [69] results in a serialized addition due to the dependency and does not bring any speedup.

In order to increase the performance with parallelization, we use the Kogge–Stone algorithm [73]. Although the algorithm is not work-efficient [27, 57], the consecutive and regular memory access pattern helps simplify the data fetch circuitry between BRAMs and PEs. The algorithm is presented in Fig. 6.8. In each level l, addition is performed with an array element that is d apart:

$$y_k^l = y_k^{l-1} + ((k \geq d)?y_{k-d}^{l-1} : 0). \tag{6.1}$$

Distance d is multiplied by a factor of 2 in each level.

However, direct implementation of Eq. 6.1 [57] results in a performance improvement of only 0.94X, 1.1X, and 1.2X with unrolling factor 2, 4, and 8, compared to the pipelined version. There are two reasons for such a small speedup.

The first reason is that Vivado HLS will assume that the memory access stride would be an arbitrary number when the stride is a variable (*e.g.*, d in Eq. 6.1, since d increases in a power of 2). Thus, Vivado HLS will infer wiring from each adder to all memory partitions (M) that results in a large II. However, the actual wiring required for each adder is only $logM$ (d=1, 2, 4, ..., M/2).

```
01 #define N 512                                          // max prefix sum length
02 #define UF 4                                                  // # of FADD
03 #define LOGUF 2                                                // LOG2(UF)
04 float dpsum[N];
05 #pragma HLS ARRAY_PARTITION variable=(data,psum,dpsum) cyclic factor=4

06 int l_max = ...;    // compute log2 of the loop bound N-a (code omitted)
07 dpsum[0] = psum[0];                                                 //init
                    (from baseline loop's
08 if( l_max >= 1 ){  lower / upper bound)                          //case : d=1
09   for(int k1 = (N-a-1)/UF; k1 >= 1/UF; k1--){   //traverse in decr order
10 #pragma HLS DEPENDENCE variable=psum, dpsum inter false
11 #pragma HLS pipeline II=1
12     for( int k2 = UF-1 ; k2 >= 0 ; k2-- ){
13       int k = k1* UF + k2;
14       int kd = k1* UF + k2 - 1;                                    // d=1
15       if( 0 <= kd && k < N-a ){
16         psum[k] = data[k] + data[kd];                              // add
17         dpsum[k] = psum[k];   //duplicate psum into dpsum for next level
18 } } } }

19 if( l_max >= 2 ){                                              //case : d=2
         ...
20       int kd = k1* UF + k2 - 2;                                    // d=2
           ...
21         psum[k] = psum[k] + dpsum[kd];                             // add
22 } } } }

23 int dUF = 1;                                             // case : d >= UF
24 for( int l = 0 ; l < (l_max-LOGM) ; l++ ){
25 for(int k1 = (N-a-1)/UF; k1 >= 1/UF; k1--){     //traverse in decr order
26 #pragma HLS DEPENDENCE variable=psum, dpsum inter false
27 #pragma HLS pipeline II=1
28     for( int k2 = UF-1 ; k2 >= 0 ; k2-- ){
29       int k = k1* UF + k2;     int kd = (k1-dUF) * UF + k2;
30       if( 0 <= kd && k < N-a ){
31         psum[k] = psum[k] + dpsum[kd];                             // add
32         dpsum[k] = psum[k];   //duplicate psum into dpsum for next level
33   } } }
34   dUF *= 2;                                    //increase dUF (=d*UF) by 2
35 }
```

Figure 6.9: Proposed transformation of variable prefix sum in loop L2_2 of rotated integral image computation (Fig. 6.7)

The second reason is that, as can be seen from Eq. 6.1, two read (y_k^{l-1}, y_{k-d}^{l-1}) and one write (y_k^l) ports are required per iteration. This requirement holds regardless of whether d is a multiple of the memory partition M. Since Vivado HLS schedules up to two read or write ports per memory partition each cycle, II=1 cannot be achieved. Another related problem is that y_k^{l-1} of Eq. 6.1 is later accessed by the term y_{k-d}^{l-1} when $k = d$. This will result in overwriting y_k^{l-1} with y_k^l and cause an access conflict problem. Note that [57] solves the latter problem with a double buffering technique, but this requires additional loops to copy the result back from the ping-pong buffer when the number of levels is odd. Also, HLS cannot achieve II=1 with double-buffering coding style. The reason is that HLS will make a conservative assumption that up to two reads and one write could occur per cycle.

We will present solutions for both problems. The first problem is solved by explicitly enumerating all possible memory access strides. The code transformation for loop L2_2 of rotated integral image example (Fig. 6.7) is shown in Fig. 6.9. We assume the number of FADD (UF) is same as the memory partitioning factor. As can be seen in lines 8-18 (d=1) and lines 19-22 (d=2), all levels with d less than M are explicitly enumerated. When d is a multiple of M, all the operands for Eq. 6.1 can be obtained from the same memory partition of $psum$, and thus can be packed into a single loop, as shown in lines 23-35.

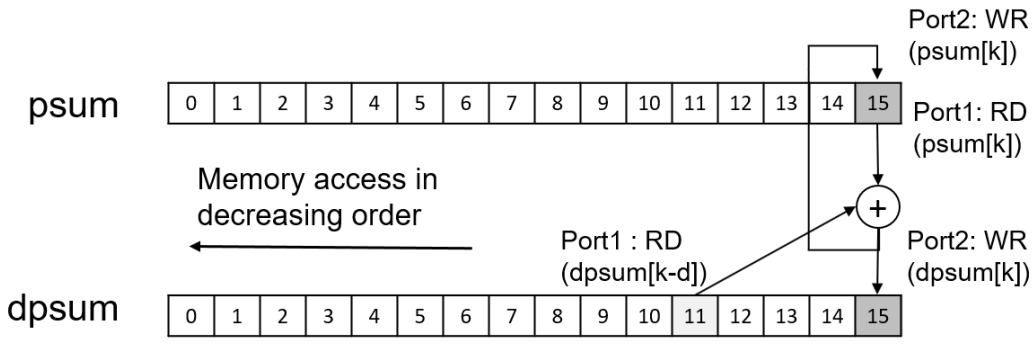


Figure 6.10: Proposed computation pattern for variable prefix sum (UF=1, d=4)

The second problem is solved by making a duplicate copy of y and traversing the array in a decreasing order. The duplicated copy of y is referred to as $dpsum$. The proposed computation pattern is shown in Fig. 6.10. For simplified presentation, the figure shows the case for the unrolling

factor (UF) is 1, and the stride (d) is 4. At each cycle, $psum[k]$ and $dpsum[k-d]$ is read and the added result is written to $psum[k]$ and $dpsum[k]$. That is, one read and one write is performed for both $psum$ and $dpsum$ per cycle. This allows the loop to be executed with II=1. The corresponding code is shown in lines 23-35 of Fig. 6.9. Starting from the upper bound of the loop iterator (lines 25 and 28), we perform the addition in Eq. 6.1 (line 31), with y_{k-d}^{l-1} in array $dpsum$ and y_k^{l-1} in array $psum$. The result y_k^l is updated to both arrays $psum[k]$ and $dpsum[k]$ (line 32). Although y_k^{l-l} is now overwritten, this does not cause a problem since this value is never accessed again with monotonically decreasing loop iterators $k1$ and $k2$.

Unlike the double buffering solution [57], the proposed solution not only solves the access conflict problem but also achieves II=1. It is worth noting that the proposed scheme takes advantage of the fact that the computation and memory access schedule can be controlled by FPGA HLS programmer. It is not applicable to CUDA GPU environment [57], because execution in a monotonically decreasing order cannot be guaranteed among multiple CUDA threads.

The cycle estimation model of the proposed scheme will be presented in Section 6.3.2.3. The comparison of the proposed scheme with the conventional pipelining, unrolling [69], and direct implementation of Kogge-Stone [57] is shown in Table 6.3. Both pipelining and unrolling infers an architecture that lacks parallelism and has low PE efficiency because of the dependency that exists between $y[k]$ and $y[k-1]$. Direct implementation of Kogge-Stone has a limited speedup with increasing parallelism due to the large II. The performance of the proposed scheme is superior, because we were able to achieve II=1 for all loops. Also, support for early termination (lines 6, 8, 19, 24 of Fig. 6.9) reduces the cycle for short loop bounds. As a result, the proposed scheme allows DSE to find a design point that is 9.7X faster than the pipelined version.

6.3 Cycle / Resource Estimation

Vivado HLS provides cycle and resource estimate for a single design in its synthesis report. If the design space is large, generating an HLS report for every possible space is not feasible since it takes several seconds to minutes to generate a single design. Aladdin [111], Lin-analyzer [138],

Table 6.3: Comparison of the total execution cycles, resource consumption, and latency of various loop bounds (for cases min=1, max=512) for the proposed variable-bound prefix sum scheme with conventional pipelining, unrolling [69], and Kogge-Stone algorithm [57] for loop L2_2 of rotated integral image

	Unr	Total	Resource			Latency	
	Fac	Cycles	DSP	FF	LUT	LB=1	LB=512
Pipe	-	917K	0	575	814	9	3586
Unr [69]	4	2.4M	2	877	1372	4637	4637
	8	2.4M	2	1525	3026	4674	4674
K-S [57]	4	974K	6	2876	3911	35	3781
	8	822K	8	5385	7166	35	3141
Proposed	4	331K	8	2679	3820	19	1253
	8	198K	16	5299	7590	22	691
	16	134K	32	11K	16K	25	417
	64	95K	128	43K	71K	31	229

and COMBA [137] solve this problem by estimating cycles based on their own scheduling/binding/allocation algorithms. However, the problem with this approach is that there is no guarantee that the HLS tool's algorithm will match such behavior. Even if their model accurately predicts one version of the HLS tool, it cannot account for the future improvement to the HLS tool. Also, it might not correctly reflect the latency difference from using another platform (with different hardware IPs) and various clock frequency settings. This might increase the chance of performing DSE based on inaccurate cycle / resource estimate.

In our framework, we extract basic cycle and resource information from the HLS tool for few designs. Based on this information, we predict the cycle and resource consumption for the entire design space based on our model. The details of this process will be explained in the following subsections. Compared to making predictions from independent scheduling/binding/allocation algorithms, making prediction from designs that has already been implemented by HLS tool increases the likelihood that the estimation will match that of the actual design. Quantitative evaluation of the proposed approach will be shown in Section 6.5.

The cycle estimation process described in this section is based on the performance estimator HLScope-S in Chapter 3. The reason we have chosen HLScope-S rather than FLASH or HLScope-M is that HLScope-S requires shortest amount of time for cycle estimation (Table 7.2). The similarity of the cycle estimation process between this DSE work and HLScope-S is that we instrument

code to extract the number of trip counts in the software simulation. Also, we obtain latency/II information of loops and functions from the HLS synthesis report. The difference is that we use this information to estimate cycle for multiple designs, rather than a single design. The cycle model reflects the design parameters to quickly estimate the execution time of a large design space. The cycle model will be presented in Section 6.3.2.

6.3.1 Resource Estimation

The resource consumption for pipelined loops is obtained from the synthesis report of the Vivado HLS tool. For loop unrolling and array partitioning, however, synthesizing every possible design with the HLS tool becomes difficult due to the large design space. Assuming there are L innermost loops that may each be unrolled up to U times, and A arrays that may each be partitioned up to P times, a naive approach would be to perform $U^L * P^A$ HLS synthesis.

One alternative approach could be to linearly interpolate from a few unrolling factors and array partitions for every loop, assuming all loops' resource consumption is independent from one another. This assumption is not true, however, because of the resource sharing between the loops. As explained by Li, *et al.,* [79], modern HLS tools, including Vivado HLS, will share operators that exceed certain thresholds across loops for high operator utilization. Floating-point operators exceed this threshold and will be shared across loops. Then a new challenge arises to efficiently predict the resource sharing for many possibilities of unrolling factors and array partitions.

We propose a resource prediction method that reduces the number of HLS synthesis and is still based on the actual HLS synthesis report. The high-level strategy is to separate sharable and non-sharable operators from a loop and linearly interpolate the non-sharable resource. The resource for sharable operators is estimated as a maximum of all loops. Finally, we estimate the mux required for sharing the operators and the arrays.

For each loop l=(0, ..., L) that is to be unrolled (after applying transformations described in Sections 6.2.2, 6.2.3, and 6.2.4), we generate and synthesize a new version of code with R different unrolling factor for each loop. R is the number of data points we use to estimate the resource usage for the rest of the design space. Next, we compute the resource difference between each version to

estimate the increase rate of resource consumption. By referring to the sharable operator (floating-point operators) usage report, the resource increment is separated into sharable resource and non-sharable resource consumption (NS_l). Based on the non-sharable resource consumption for R designs, we use a linear regression technique to find the slope and the intercept of the data points. Then the non-sharable resource of a loop can be estimated for arbitrary unrolling factors.

The estimation error with different R for LU benchmark is shown in Table 6.4. Since large R increases the number of synthesis runs, we have decided to limit R to 3. Our framework generates loops with unrolling factor 4, 8, and 16 (R=3), since non-linear characteristic is sometimes observed for small (1, 2) unrolling factors.

Table 6.4: FF/LUT estimation error rate for various R for LU benchmark

	R=2	R=3	R=4
FF/LUT	1.2% / -6.9%	0.9% / -5.5%	0.5% / -7.2%

To estimate the resource for sharable operators, we find the maximum of each type k of operators (*e.g.,* floating-point adder or multiplier) after loop unrolling each loop (S_{lk}). Next, we estimate the resource consumed by mux used for sharing each type of operator among loops. Likewise for the arrays, we estimate the mux needed for different loops to have access to the same a arrays. Then we estimate LUT consumption for the input operands of each operator and the address/data of arrays based on the bitwidth and the number of sharing ports. The LUT consumption model for mux can be found in [35].

The above resource consumption estimation can be summarized as:

$$\sum_{l} NS_l + \sum_{k} \max_{l}(S_{lk}) + \sum_{k} mux(l,k) + \sum_{a} mux(l,a) \tag{6.2}$$

Since only $R+1$ (=4) synthesis are required for each unrolled loop, the number of HLS synthesis runs to estimate the resource for various unrolling factors is $(R+1) * TL$, where TL is the number of innermost loops. This is a large improvement over the naive approach of performing $U^L * P^A$ synthesis. As a result, the design space exploration time is reduced to a few hundreds of seconds, as will be presented in the experimental section.

Our resource estimation method differs from AutoAccel [37] in that AutoAccel does not model resource sharing among loops. Our work has more similarity with [79] in a sense that we separate sharable and non-sharable resource of a loop. The difference is that [79] does not explore loop unrolling or array partitioning and thus does not perform any linear regression or function separation. Instead, they assume that the non-sharable part stays constant over multiple designs. However, assuming a constant non-sharable resource increases the LUT/FF estimation error rate from 0.9%/-5.5% to 32%/-48% in LU because non-sharable resource can increase very rapidly with loop unrolling.

6.3.2 Cycle Estimation

6.3.2.1 Model for Partial Unrolling with Pipelining

For the LU benchmark in Fig. 6.3, if the transformed loop's ($L2\_1$'s) initiation interval, iteration latency, and unroll factor is II_T, IL_T, UF_T, the number of execution cycle is $II_T * \{(ub-1)/UF_T + 1 - (lb/UF_T) - 1\} + IL_T$ [79]. This is approximated as

$$\simeq II_T * (ub - 1 - lb)/UF_T + IL_T = II_T * (TC - 1)/UF_T + IL_T \tag{6.3}$$

where $TC(= ub - lb)$ is the trip count of the original loop (loop 2 in Fig. 6.1).

6.3.2.2 Model for Variable Reduction

For the Cholesky benchmark in Fig. 6.6, the execution cycles of the PPR stage can be directly derived from Eq. 6.3: $II_{PPR} * (TC - 1)/(UF/FADD\_LAT) + IL_{PPR}$. The loop in the VTR stage will be executed $log_2(min(TC, UF))$ times. Thus the total estimated cycle is

$$II_{PPR} * (TC - 1)/(UF/FADD\_LAT) + IL_{PPR} + II_{VTR} * (log_2(min(TC, UF)) - 1) + IL_{VTR} \tag{6.4}$$

6.3.2.3 Model for Variable Prefix Sum

The code in Fig. 6.9 is a collection of partially unrolled loops modeled in Eq. 6.3. The number of levels is $l_{max} = logTC$. Thus, the total estimated cycle is

$$\sum_{l=0}^{l_{max}} (II_l * (TC - 1)/UF + IL_l). \tag{6.5}$$

6.3.2.4 Total Cycles

If the trip counts of all variable loops were available, the total cycle would simply be an accumulated number of the cycles computed in Eqs. 6.3, 6.4, 6.5. However, since storing all trip counts is an expensive process, we simplify the computation by first computing the average of trip counts (AVG_TC) and the number of loop occurrences (OCC) during the software simulation. Then the trip counts TC in Eqs. 6.3, 6.4, 6.5 are replaced with AVG_TC, and the entire equation is multiplied by OCC. Also, we approximate the II, IL of the unrolled loops based on the value already obtained from the selected (R=3) synthesis. As will be shown in the experimental section, these approximations result in a relatively small cycle error rate of 12%.

6.4 Overall Flow and the Design Space Exploration

The overall flow is shown in Fig. 6.11. The input code is first profiled with the Vivado HLS software simulation flow. In this stage, the loop trip counts TC and the number of loop occurrences OCC are recorded. Next, possible transformed codes are generated from the input code discussed in Section 6.2. Next, R design points for each loop are synthesized with the HLS tool (Section 6.3.1). Based on the synthesized result, cycle count and resource consumption are estimated as discussed in Section 6.3. Among possible design points that satisfy the resource constraint, the one with the least latency is chosen and presented as the final output.

Following Lin-analyzer [138], the design parameters evaluated in the framework are shown in Table 6.5. Loops can be unrolled to their maximum loop bound, and the array can be partitioned

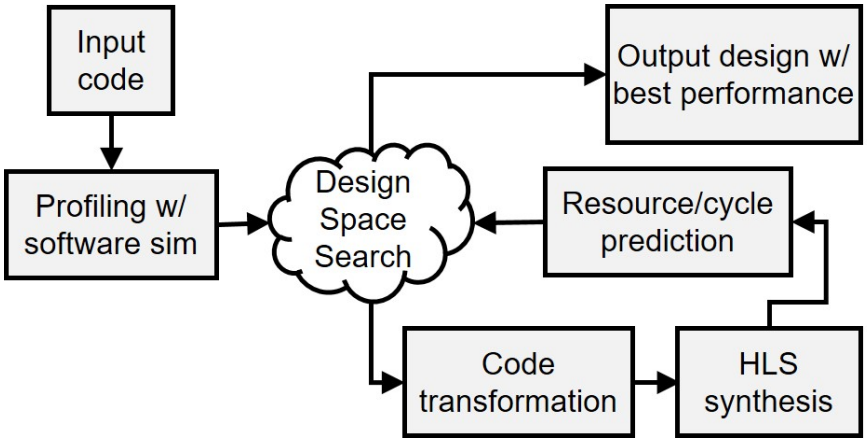


Figure 6.11: Overall DSE framework

to its array size. For simplification, the loop unrolling factor and the array partitioning factor are explored in power of 2s (1, 2, 4 ...). We also explore the loop pipelining. As mentioned in Section 2.5, the optimization is performed on the innermost loops for fine-grain parallelization and pipelining.

Table 6.5: Design parameters evaluated in DSE

Parameter	Range
Loop unrolling factor	1 : Max loop bound (pow of 2)
Pipelining	True, False
Array partitioning factor	1 : Array size (pow of 2)

In order to reduce the design space, we prune away design spaces that are not promising. First, we do not consider array partitions that are larger than the number that can be simultaneously consumed or produced by PEs. Second, we do not consider pairs of optimizations that require partial partitioning on multiple dimensions of a local memory, because this complicates routing, and the number of BRAM instances increases rapidly.

Table 6.6: Effect of the proposed code transformations (unit: cycles)

	Baseline	+Part Unr	+Var Rdct	+Var PSum
Cholesky	404M (1.0X)	-	14.2M (28X)	-
LU	942M (1.0X)	2.96M (317X)	-	-
Trisolv	2.50M (1.0X)	-	63.9K (39X)	-
RotIntImg	11.1M (1.0X)	1.61M (6.9X)	-	246K (45X)
Durbin	5.25M (1.0X)	-	-	522K (10X)
Dynprog	872M (1.0X)	-	-	98.8M (8.8X)

6.5 Experimental Result

6.5.1 Experimental Setup

For evaluation, we use the Polybench benchmark [99] that was also used in Lin-analyzer [138] and COMBA [137]. To demonstrate the effectiveness of our framework, five benchmarks with variable loop bounds have been chosen—Cholesky, LU, Trisolv, Durbin, and Dynprog. We also constructed rotated integral image benchmarks from [81]. The matrix size is set to 512, and the variable types are set to single-precision floating-point for all benchmarks, except Dynprog which was set to have a matrix size of 128 to fit FPGA. The design is synthesized using Vivado HLS 2018.2 [130] software. For the platform, we target the ADM-PCIE-KU3 board [3] with Xilinx's Ultrascale KU060 FPGA [129] at 250MHz. The FPGA resource (DSP/FF/LUT) limit is set to half of the total resource on KU060 to ease the place-and-route process.

6.5.2 Performance

The performance after applying the proposed transformations to the unmodified baseline code is 8.8X to 317X, as shown in Table 6.6. LU has a large speedup due to the abundant parallelism; other applications have loop-carried dependencies (reduction or prefix sum patterns) that limit performance improvement. On average, the proposed transformations achieved a 75X speedup.

Table 6.7: Comparison of the performance, design space exploration speed, and the prediction accuracy among proposed, COMBA, and AutoAccel flows (the performance and the prediction error rates are that of the final output design)

Application	Flow	Exec Time (cycles)	Design Space Exploration			Prediction Error Rates			
			# Design	# HLS Runs	Expl Time	Exec Time	DSP	FF	LUT
Cholesky	Proposed	14.2M	100	11	625s	12%	0%	7.7%	1.3%
	COMBA	21.1M	NA	0	872s	-86%	NA	NA	NA
	AutoAccel	180M	32	NA	252s	500%	0%	0%	0%
LU	Proposed	2.96M	100	11	451s	-3.7%	0%	0.89%	-5.5%
	COMBA	1.01B	NA	0	416s	-99.9%	NA	NA	NA
	AutoAccel	403M	16	NA	145s	201%	0%	0%	0%
Trisolv	Proposed	63.9K	10	5	232s	14%	0%	-1.7%	-0.85%
	COMBA	103K	NA	0	431s	-98%	NA	NA	NA
	AutoAccel	2.10M	32	NA	131s	101%	0%	0%	0%
RotIntImg	Proposed	247K	1000	20	1370s	-29.0%	0%	-18%	-20%
	COMBA	21.5M	NA	0	1,490s	-99.9%	NA	NA	NA
	AutoAccel	2,13M	32	NA	103s	98%	0%	0%	0%
Durbin	Proposed	522K	10	5	245s	-2.5%	0%	1.2%	4.4%
	COMBA	8.12M	NA	0	2,540s	-99%	NA	NA	NA
	AutoAccel	1.08M	32	NA	203s	96%	0%	0%	0%
Dynprog	Proposed	98.8M	64	9	436s	10%	0%	0.54%	2.4%
	COMBA	1.87B	NA	0	2,200s	-99.9%	NA	NA	NA
	AutoAccel	234M	32	NA	121s	454%	0%	0%	0%
Average	Proposed	1.0X	-	-	-	12%	0%	5.1%	5.7%
	COMBA	78X	-	-	-	97%	NA	NA	NA
	AutoAccel	32X	-	-	-	241%	0%	0%	0%

For comparison, we obtained access to the source code for two of the latest DSE works, AutoAccel [37] and COMBA [137], and produced the output for the same benchmarks. We adjusted the parameters in AutoAccel and COMBA to match the characteristics of the KU060 FPGA. Since COMBA does not provide code on how to unroll variable bound loops, we applied the conventional unrolling with maximum bound (Section 6.2.1) with the unrolling factor instructed by the tool. COMBA had a tendency to over-unroll the loops beyond the given resource threshold—probably because its resource estimation is mostly based on the operators only. In this case, we manually reduced the unrolling factor to fit the given threshold. For the LU benchmark, Vivado HLS was unable to unroll the loops (as discussed in Section 6.2.2), and achieved no speedup when using the configuration suggested by COMBA. Similarly, for prefix sum patterns, the unroll directive infers fixed-length serialized addition (Section 6.2.4) which does not improve the performance.

The performance comparison is shown in the "Exec Time" column of Table 6.7. Our framework outperforms COMBA and AutoAccel by 78X and 32X on average. The main reason is that COMBA and AutoAccel do not perform code transformation that can solve the PE efficiency problem of the variable loops (Section 6.2). Thus, the design space explored by these tools is limited

and results in a relatively worse performance. Another reason is that their cycle estimation model does not properly consider variable loop bounds.

6.5.3 Exploration Speed and Prediction Accuracy

The execution time, DSE result, and the prediction error rates are shown in Table 6.7. The execution times of the variable loops are measured using the method proposed in [24], and the resource usage is compared to the Vivado HLS synthesized result.

The result shows that the number of HLS synthesis performed is on average only 23% of the entire design spaces explored. The performance and the resource consumption of the rest of the design points are estimated using the model presented in Section 6.3. Thus, the exploration time is maintained at a few hundreds of seconds.

The table also shows that the prediction error rate of execution time, DSP, FF, LUT, with the proposed model is on average 12%, 0%, 5.1%, 5.7%, respectively. Such a low error rate helps the DSE process find the best design point accurately.

The exploration time and the prediction accuracy for AutoAccel and COMBA is also shown in Table 6.7. The execution time error of COMBA is probably caused by the mispredicted II and IL compared to the actual Vivado HLS synthesized result. This is due to the fact that COMBA does not reference the HLS report. On the other hand, AutoAccel does refer to the HLS report—however, its execution time is also not very accurate. The reason is that Vivado HLS reports the cycle based on the maximum of the variable loop. The resource estimated for AutoAccel is the result given by the HLS tool itself, and thus has an error rate of 0%.

6.6 Comparison with Related Work

The review of other DSE works can be found in Section 2.5, and the comparison with our proposed framework was explained in Section 6.1. Other related works include [29] and ElasticFlow [119] which provide efficient HLS-based methodologies to distribute the dynamic workload among coarse-grain PEs. However, many examples, such as those found in Polybench benchmarks, do

not have coarse-grain parallelism in outer loops (*e.g.,* row-wise parallelism in sparse matrix-vector multiplication [29, 119]). Instead, there are several variable loops that are executed in serial, similar to the examples presented in [79]. Thus, our DSE work is more focused on optimizing these innermost loops by exploiting fine-grain parallelism and pipelining, accurately estimating resource sharing among these serial loops, and efficiently allocating non-sharable resource for overall latency minimization. Another difference that we see in these works is that they require modification of HLS scheduling and binding kernels—whereas our DSE work is based on source-to-source transformation to produce HLS codes that can be easily integrated into existing HLS frameworks.

6.7 Conclusion

Optimization of variable loop bounds with conventional HLS directives for pipelining and unrolling often leads to a low PE utilization problem. We have shown that techniques such as partial unrolling with pipelining or loop early termination will help solve this problem. HLS-based code transformations were devised to demonstrate how these techniques can be applied to common computational patterns. Also, we have proposed a resource estimation method that models operator sharing with a small number of HLS syntheses. The experimental result shows that a 75X speedup was achieved compared to the baseline implementation. As a future work, we are considering to support more patterns for loops with variable bounds beyond those in the Polybench benchmarks.

CHAPTER 7

Quantitative Comparison among Proposed Frameworks and Concluding Remarks

7.1 Quantitative Comparison with Sparse Matrix-Vector Multiplication Benchmark

In previous chapters, we have described our performance debugging frameworks for various stages of the design process. However, it was difficult to compare each framework's accuracy and the required time. In this section, we will provide a quantitative comparison among the proposed framework based on a common benchmark—sparse matrix-vector multiplication (SpMV).

There are two main reasons we have chosen this benchmark. The first reason is the importance of the sparse matrix processing. Sparse matrix processing has been widely used in many applications such as computer vision, linear algebra, chemical engineering, medical imaging, and circuit simulation (please refer to the University of Florida sparse matrix collection [44] for a comprehensive list). Recently, a greater emphasis is being placed on the sparse matrix processing to reduce the computational complexity in deep learning applications [55, 56]. The second reason is the potential large benefit of customized computing in SpMV. It is well known that GPU cannot achieve its peak performance for irregular applications (e.g., SpMV) due to the thread divergence and load balancing problems [112, 136]. FPGA, on the other hand, can customize its architecture for various irregular applications [19–21, 29, 34, 50, 78, 119].

In Section 7.1.1, we will provide background information on the recurrent neural network and the sparse matrix-vector multiplication architecture. In Section 7.1.2, we will provide a quantitative comparison among the proposed frameworks—HLScope-S, FLASH, and HLScope-M. The

performance comparison with CPU and GPU implementations is provided in Section 7.1.3.

7.1.1 Sparse Matrix-Vector Multiplication

The SpMV kernel performs multiplication between a sparse matrix A and an input array x and produces an output array y. The sparse matrix can be represented in various formats such as coordinate list (COO), compressed sparse row (CSR), and compressed sparse column (CSC). Among them, we choose the CSR format because of its most frequent use [50]. The CSR format compresses the non-zero elements and the row address of the matrix elements. The CSR format is composed of array $rows$ (list of the starting address of the matrix elements in each row of matrix A), val (value of the matrix elements), and col (column address of the matrix elements). The SpMV kernel in CSR format is computed as follows:

```
for(r = 0; r < N; r++) //L1
  for(c = rows[r]; c < rows[r+1]; c++) //L2
    y[r] += val[c] * x[col[c]];
```

Figure 7.1: The SpMV kernel

The existence of the input-dependent and irregular loop bound in $L2$ ($rows[r]$~$rows[r+1]$) of Fig. 7.1 often creates challenges in accelerating SpMV applications.

The dataflow architecture for SpMV computation is shown in Fig. 7.2. One row of matrix will be distributed in a round-robin fashion to PE groups. Each PE group is composed of four modules. Based on the column range of each row ($rows[r]$~$rows[r+1]$), the first module reads the matrix element value ($val[c]$), and the second module reads the column value ($col[c]$). The third module uses $col[c]$ as a read address of x, that has been pre-loaded into local BRAM. In order to achieve high throughput, $x[col[c]]$ and $val[c]$ are packed and sent to the fourth module in a chunk of UF, which corresponds to the fine-grain unrolling factor in the accumulation PE. The fourth module accumulates the multiplied value of $val[c]$ and $x[col[c]]$ using the reduction technique introduced in Section 6.2.3. Finally, $y[r]$ is collected and written to DRAM. The modules communicate through FIFOs, and the depth of each FIFO is set to 4.

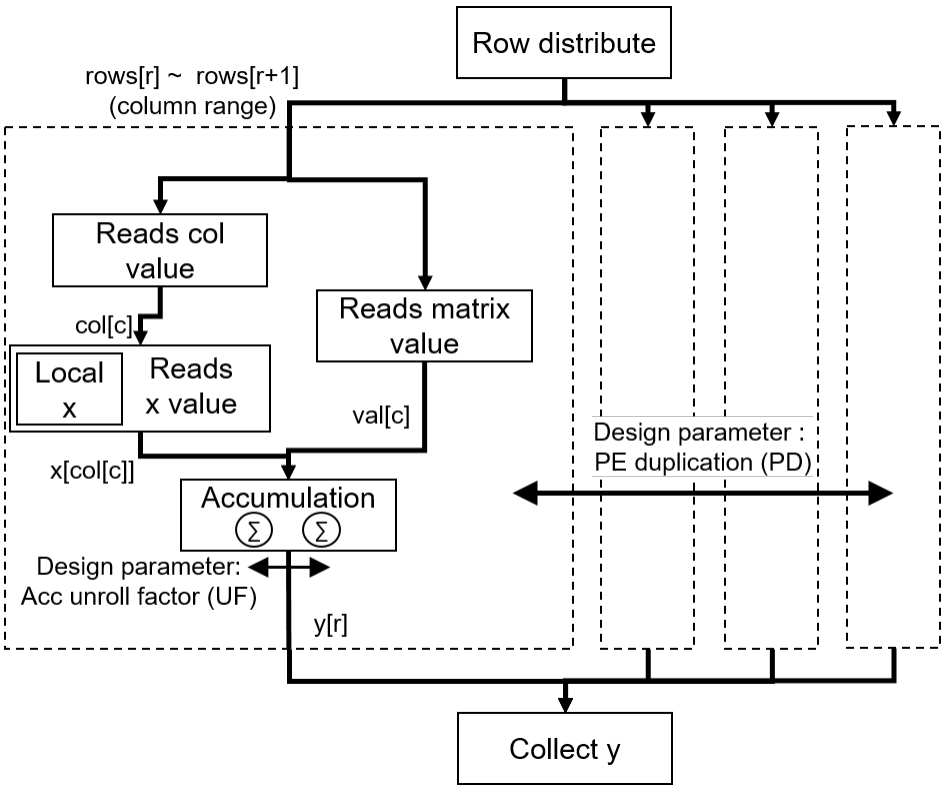


Figure 7.2: Overall architecture for SpMV computation

As shown in Fig. 7.2, we simplify the design space into two parameters: coarse-grain PE duplication (PD) and fine-grain accumulator unrolling factor (UF). The coarse-grain PE duplication exploits the row-wise parallelism [29, 119, 136] in $L1$ of Fig. 7.1, and the fine-grain unrolling exploits the parallelism in the reduction tree (Section 6.2.3).

The SpMV dataset we use for the case study is from electroencephalogram (EEG), which monitors the electrical signal of the brain [16]. It can be used to diagnose various neurological diseases such as epilepsy or depression. From the raw EEG signal, recurrent neural network (RNN) [67] is used to extract analytic signal. RNN consists of several hidden nodes, and the interconnection among the nodes is represented by a sparse matrix A. The output y of the hidden layer is computed by performing sparse matrix-vector multiplication (SpMV) between A and the internal signal x of each node. Matrix A has a dimension of 500×500, and the sparsity of the matrix (proportion of zero elements among all matrix elements) is 82%. The EEG signal was obtained at a 32kHz sampling rate, and we perform SpMV on 100,000 samples, which corresponds to an observation

duration of 3.1s (=100K/32KHz).

Using the EEG dataset and the SpMV architecture in Fig. 7.2, we have performed the design space exploration process explained in Chapter 6. We set the resource usage limit to 60% of the Xilinx's Ultrascale KU060 FPGA [129] to ease the placement and the routing process. The result of exhaustive search among all design spaces (PD, UF) is shown in Figure 7.3. The result shows that the configuration with the best performance is achieved with PE duplication factor of 32 and fine-grain reduction unrolling factor of 1. The FPGA resource consumption of this design point is shown in Table 7.1.

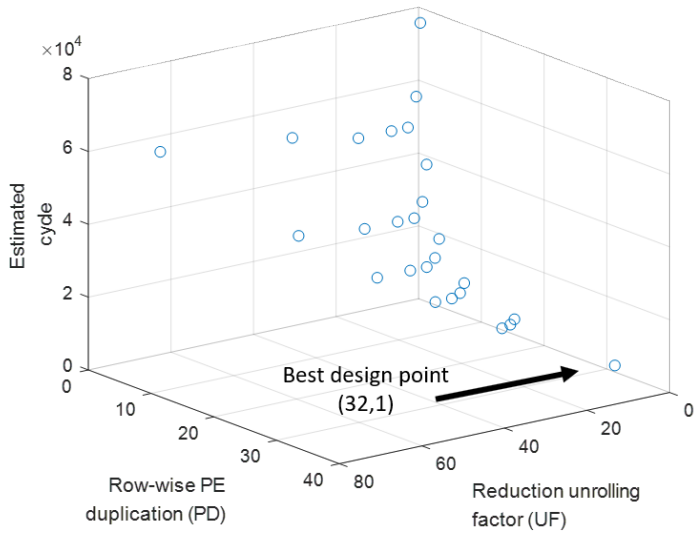


Figure 7.3: Design space exploration result of SpMV (cycles per sample)

Table 7.1: FPGA resource consumption of the design with the best performance

	BRAM	DSP	FF	LUT
# of resource used	334	288	145k	130K
Utilization	(15%)	(10%)	(21%)	(39%)

7.1.2 Comparison of the Proposed Performance Debugging Frameworks

In this subsection, we compare the performance debugging frameworks proposed in previous chapters. We have chosen the design point obtained in Section 7.1.1 that provides the best performance—

(PD, UF)=(32, 1). In order to clearly show the difference in cycle estimation accuracy between HLScope-S and FLASH, we have intentionally reduced the buffer size of all FIFOs to 2. Table 7.2 shows the comparison of the cycle estimation accuracy and the time taken to obtain the result. In addition to the result from HLScope-S, FLASH, and HLScope-M, we also show the result from the HLS synthesis report and the HLS RTL co-simulation for comparison. The simulation time is obtained from running 100,000 EEG signal samples as an input (x) for SpMV. Due to the long time required for the RTL simulation, the RTL simulation time is estimated by running SpMV on one sample and multiplying by 100,000 times.

Table 7.2: Comparison of the proposed performance debugging frameworks for the best SpMV design space (100,000 samples, FIFO depth set to 2)

	Synth Rpt (Viv HLS)	HLScope-S (Chapter 3)	FLASH (Chapter 4)	RTLSim (Viv HLS)	HLScope-M (Chapter 5)
Cycles for one sample (Error Rate)	9281 (257%)	1487 (-43%)	2594 (0%)	2594 (0%)	2594 (0%)
S2S transf time	-	2.0min	15min	-	1.3m
HLS synth time	13min	16min	15min	13min	16min
PnR time	-	-	-	-	215min
Sim (exec) time	-	15min	54min	$1.3 * 10^6$min	0.04min
Total time	13min	33min	84min	$1.3 * 10^6$min	232min

The synthesis report provides the cycle estimate based on the maximum loop trip count. In SpMV, the number of non-zero elements varies per row. As a result, the synthesis report cycle estimate is much larger than the actual cycle count (257% over-estimation). HLScope-S does not have this problem, since it is based on the software simulation of the input dataset. However, the reduction of FIFO depth has induced frequent FIFO stalls, and the HLS software simulator cannot estimate the FIFO stalls (Section 4.2.2). As a result, HLScope-S has cycle under-estimation of -43%. FLASH, on the other hand, is based on a cycle-accurate software simulation and provides accurate estimation (0% error rate). HLScope-M accurately measures the cycle count with its on-board monitors.

Table 7.2 also provides the breakdown of the time taken to apply each framework. First, the table shows the time taken to apply the source-to-source (S2S) code transformation process. Unlike HLScope-S and HLScope-M that performs code transformation in loops or functions, FLASH

performs transformation for each statement. This makes the code transformation time of FLASH about 10X slower compared to HLScope-S and HLScope-M.

The next row in the table shows the Vivado HLS synthesis time and the Vivado placement and route time (if applicable). Since HLScope-S and FLASH makes code modification to the original source code, the synthesis time for HLScope-S and FLASH differs slightly from that of the unmodified code. HLScope-S and FLASH only require HLS synthesis run, whereas HLScope-M requires place and route (PnR) process to generate the FPGA bitstream. As a result, HLScope-M requires much more initial preparation time than HLScope-S and FLASH to begin its performance debugging process.

In terms of simulation (execution) time, HLScope-M is several orders of magnitude faster than HLScope-S or FLASH because it takes advantage of the FPGA acceleration. HLScope-S and FLASH simulation speed is much faster than RTL simulation because they are based on HLS software simulation. HLScope-S is slightly faster than FLASH, because the performance estimation code is instrumented in a higher level (granularity of loops or functions) than FLASH (granularity of statements).

Note that the total time taken to apply HLScope-S (33 mins) and FLASH (84 mins) on SpMV is slightly longer than the time needed for most designs (reported as 1–10 mins in Table 1.1). The reason is that this particular SpMV design is composed of a large number ($130 = 32 \times 4 + 2$ — see Fig. 7.2) of modules.

In summary, HLScope-S is most advantageous in obtaining a fast performance estimation—however, its accuracy drops in cases where the FIFO depth is inadequate. FLASH requires long source-to-source transformation time and simulates slower than HLScope-S, but provides cycle-accurate result. HLScope-M requires a long time for bitstream generation and typically does not have advantage over HLScope-S or FLASH—however, this flow may be necessary for cases where the input dataset does not exist and on-board measurement is required.

Table 7.3: Performance comparison of SpMV among CPU, GPU, and FPGA implementations

Compute Node (Year, Frequency, Technology)	# Thrds/PEs	Exec Time (Spdup, GFLOPs)
CPU Intel Xeon E5-2666 (2015, 2.9GHz, 22nm)	36	3.1s (1.0X, 3.1GF)
GPU NVIDIA Tesla M60 (2015, 899MHz, 28nm)	100K $\times$ 512	0.71s (4.4X, 13GF)
FPGA Xilinx Ultrascale KU060 (2015, 196MHz, 20nm)	32 $\times$ 1	1.4s (2.2X, 6.8GF)

7.1.3 Comparison with GPU and Multithreaded CPU Implementation

We will compare the performance of SpMV on different architectures in this subsection. For FPGA implementation, we use the design found by the DSE process in Section 7.1.1. For CPU implementation, we use OpenMP to exploit the row-wise parallelism of SpMV. Dynamic scheduling is applied to balance the irregular workload. For GPU implementation, we use OpenCL. Each sample is processed by a workgroup (coarse-grain parallelism) and each row is processed by a workitem (fine-grain parallelism). Array x is stored in the local memory so that the array can be shared among multiple threads.

Compute nodes of similar technology and release date have been selected for comparison. For CPU, we use Intel Xeon E5-2666 [65]. For GPU, we use NVIDIA Tesla M60 [96]. For FPGA, we use Xilinx's Ultrascale KU060 FPGA [129]. Although Tesla M60 has the best performance among the Maxwell Tesla processors, Xeon E5-2666 only has 73% performance of E5-2699 (best among Haswell Xeon E5 processors) [41] and Ultrascale KU060 only has the half the resource of Ultrascale KU115 (best among Kintex Ultrascale FPGAs) [129]. Thus, we only use one out of two GM204 chips on Tesla M60 GPU for fair comparison.

The performance comparison is shown in Table 7.3. Although the GPU has a theoretical single-precision peak performance of 3.7 TFLOPs, it only achieved 13GFLOPs (0.4% of peak performance) for the SpMV example. The main reason is that the number of floating-point multiplications and additions is different on each row, and this leads to thread divergence [95]. Another reason can be found in the access pattern to x (Fig. 7.1)—the random local memory access leads to frequent local memory bank conflict.

In the FPGA implementation, the efficiency of floating-point multipliers is 56% and the efficiency of floating-point adders is 19%. The operator efficiency is relatively higher than that of

GPU. The reason is that FPGA can efficiently process the irregular loop bounds of SpMV by customizing its architecture to process the next row as soon as the previous row is computed. Also, the local memory architecture can be customized to provide high-throughput data (row, col, val, x) to the accumulation modules. However, the FPGA implementation only has a comparable performance to the implementation in other platforms due to the low clock frequency (196MHz).

7.2 Concluding Remarks and Future Work

In Chapter 3, we have described our performance debugging flow. In order to provide an accurate cycle estimation for performance debugging, we have also proposed a performance estimation flow called HLScope-S. We have shown that the performance estimation accuracy of the designs with input-dependent behavior can be improved by instrumenting analytical models into the source code and running the HLS software simulation. We have also described methods to extract performance-related information that has been abstracted by the HLS tool. Moreover, we have proposed high-level model for various external memory access patterns. Experiments on an ADM-PCIE-7V3 board showed that HLScope-S on average has an estimation error rate of 1.1% in compute-bound modules and 5.0% for performance-criticial DRAM-bound modules, with 4% time overhead in software simulation. This shows that the proposed modeling techniques can be used to rapidly provide accurate performance estimation for input-dependent HLS applications.

In Chapter 4, we have proposed a new HLS software simulation flow (FLASH) that extracts the scheduling information from the HLS synthesis report and automatically generate a cycle-accurate simulation model. The experimental results shows that the proposed flow simulates three order of magnitude faster than the RTL simulation, because FLASH is not slowed by the allocation / binding information and the component library in the RTL files. This confirms that simulating based on the scheduling information can help solve the correctness issue of HLS software simulators and rapidly provide accurate performance estimation. We also proposed debugging features that allow users to describe their needs in high-level and enabled live capturing of variables to reduce the debugging effort of HLS users. We expect that our findings and the techniques in this chapter could

motivate commercial HLS tool developers to adopt a similar simulation and debugging flows and significantly decrease the validation time of the HLS customers.

Chapter 5 describes an on-board performance debugging framework (HLScope-M) that could be used in the case where input dataset is not available for simulation. The monitor used for debugging was described in pure HLS C language for easy integration with existing HLS flows and only consumed few hundreds of LUTs. In addition to the automated cycle measurement flow, we have also proposed a method to trace the reason for stall in dataflow applications. This could be used to help programmers to quickly identify the performance bottleneck of a HLS design.

Based on the performance estimation techniques in Chapter 3, we have devised a DSE framework which is described in Chapter 6. The proposed method achieves a large speedup of 75X even on applications with variable loop bounds and outperformed state-of-the-art DSE frameworks. We have shown that conventional method of applying HLS directives for pipelining and unrolling on applications with variable loop bounds often leads to a low PE utilization problem. Instead, we have shown that techniques such as partial unrolling with pipelining or loop early termination helps solve this problem. HLS-based code transformations were devised to demonstrate how these techniques can be applied to common computational patterns such as reduction and prefix sum. Also, we have proposed methods to predict execution time and resource usage for multiple design points based on few HLS synthesis runs.

One of the limitation of the HLS performance estimation and the simulation frameworks in Chapters 3 and 4 is that they are based on single-threaded software simulation. As a result, the current flows are not suitable for benchmarks that require long simulation time. To solve this problem, we plan to add parallelization using Pthread/OpenMP in the future so that large-scale simulation can be accelerated by exploiting the multicore architecture. One challenge in applying parallelization to the simulation framework is the frequent synchronization that is needed after the module and the FIFO simulation loops (lines 8-11 in Fig. 4.10). Asynchronous execution and localized clock would be needed to increase the granularity of workload for each thread. Also, we would need a mapping of modules and FIFOs to the threads that reduces the inter-thread communication and the false sharing.

In terms of coverage, HLScope-M and HLScope-S is comprehensive because HLScope-M monitors any type of modules and HLScope-S models the DRAM access latency. However, FLASH and the DSE framework only models computation and intra-FPGA communication. One of our future plan is to generalize FLASH and the DSE framework so that they can model the DRAM access latency.

Some of the techniques in the HLScope-S framework in Chapter 3 was applied to the DSE framework in Chapter 6 to provide fast performance estimation. However, HLScope-S may provide inaccurate performance estimation when FIFOs do not have adequate FIFO depth (Section 4.2.2). This may result in the DSE process providing a design space with low performance. The FIFO depth problem can be solved by exploiting the FLASH framework (Chapter 4), since it accurately models the FIFO depth and provides cycle-accurate result. However, in contrast to HLScope-S which performs code instrumentation in the granularity of loop or module level, FLASH performs instrumentation on each statements. As demonstrated in Section 6.3.2, it is often enough to collect the input-dependent behavior information only on the loop level for performance estimation in DSE. Therefore, additional code instrumentation routines would be needed to extract a program's execution information on a higher level than that of the current FLASH framework, so that we can accelerate the DSE process that includes the FIFO depth as a design space.

Another limitation of the DSE framework is that the design space is limited the fine-grain unrolling, fine-grain pipelining, and array partitioning. As demonstrated in COMBA [137], better performance can be achieved with additional design space such as function pipelining, function inlining, and dataflow (coarse-grain pipelining). Our goal for future work is to improve the DSE framework so that it would apply most of the known FPGA optimization techniques and rapidly find the best design space even for input-dependent benchmarks.

The proposed performance debugging frameworks can benefit both experienced HLS programmer and novice HLS programmer with algorithmic expertise. For an experienced HLS programmer, HLScope-S can quickly provide a feedback with an accurate performance estimation for each optimization the programmer has applied. HLScope-S also provides the stall rate (e.g. Fig. 3.5) which helps the programmer to focus on optimizing the module with low utilization. The FLASH

simulator may be used instead of HLScope-S for more accurate performance estimation if the design has a feedback path or frequent FIFO stalls (Section 4.2). If a bitstream has been generated and the on-board performance result does not match the expectation, the programmer could use HLScope-M to analyze the reason. Our frameworks help shorten such FPGA development process.

A novice HLS programmer with algorithmic expertise would need a help of DSE work such as the one proposed in Chapter 6 or the work in [37, 72, 100, 111, 137, 138] to obtain an optimized HLS design. Then our performance debugging frameworks can be used to open up a wider possibility for further performance optimization with the expertise of algorithm developers. The DSE work optimizes the FPGA design by applying a different combination of directives or by varying the coding structure of loops or functions. Our framework, on the other hand, reports the stall rate and the reason for the stall (Chapter 3.1), which are used to calculate the usage rate of the FPGA compute and memory resources. Such analysis could motivate a programmer to develop a new algorithm that use less of the resource that has been identified as the bottleneck. For example, although the LUT has the highest resource consumption rate (Table 7.1) in the SpMV benchmark, the usage rate (efficiency) of the compute units is only 56% for floating-point multipliers and 19% for floating-point adders (Section 7.1.3). In this case, a programmer could try to increase the efficiency of the FPGA compute units with a modified algorithm that interleaves the computation between the rows [46]. As another example, the performance bottleneck in the final version of the quicksort example in Section 3.1.3 was analyzed to be the DRAM access module (Table 3.6). In this case, a programmer may try to develop an algorithm with a higher data reuse ratio. With such help from both DSE work and performance debugging framework, algorithmic developers with little previous HLS experience will also succeed in obtaining the best design for their applications.

REFERENCES

[1] R. Allen and K. Kennedy, *Optimizing compilers for modern architectures: a dependence-based approach*, Morgan Kaufmann, San Francisco, CA, 2002.

[2] Alpha Data, Alpha Data ADM-PCIE-7V3 Datasheet, 2017, *http://www.alpha-data.com/pdfs/adm-pcie-7v3.pdf*.

[3] Alpha Data, Alpha Data ADM-PCIE-KU3 Datasheet, 2017, *http://www.alpha-data.com/pdfs/adm-pcie-ku3.pdf*.

[4] Altera, SignalTap II with Verilog Designs, 2012, *http://www.altera.com/*.

[5] Amazon, Amazon EC2 F1 Instance, 2019, *https://aws.amazon.com/ec2/instance-types/f1/*.

[6] Apache Spark examples, *http://spark.apache.org/examples.html*.

[7] K. Asanovic, *et al.*, "A view of the parallel computing landscape," *Commun. ACM*, 52(10), pp. 56–67, 2009.

[8] M. Beister, D. Kolditz, and W. Kalender, "Iterative reconstruction methods in X-ray CT," *Physica Medica*, vol. 28, no. 2, pp. 94–108, 2012.

[9] L. Benini, *et al.*, "SystemC cosimulation and emulation of multiprocessor SoC designs," *Computer*, 36(4), 53–59, 2003.

[10] F. Bouchhima, et al., "A SystemC/Simulink co-simulation framework for continuous/discrete-events simulation," *IEEE Int. Behavioral Modeling and Simulation Workshop*, pp. 1–6, 2006.

[11] Cadence, "Incisive Enterprise Simulator," *http://www.cadence.com*, 2019.

[12] Cadence, "Palladium Z1 Enterprise Emulation Platform," *http://www.cadence.com*, 2019.

[13] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *Proc. Int. Conf. Hardware/software Codesign and System Synthesis*, 19–24, 2003.

[14] A. Canis, *et al.*, "From software to accelerators with LegUp high-level synthesis," in *Proc. Int. Conf. CASES*, 18–26, 2013.

[15] R. Chandra, *et al.*, *Parallel Programming in OpenMP*, Morgan Kaufmann, San Francisco, CA, 2001.

[16] Z. Chen, A. Howe, H. T. Blair, and J. Cong, "CLINK: Compact LSTM inference kernel for energy efficient neurofeedback devices," in *Proc. Int. Symp. Low Power Electronics and Design (ISLPED)*, 2018.

[17] Y. Chi, J. Cong, P. Wei, and P. Zhou, "SODA : stencil with optimized dataflow architecture," *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD)*, 2018.

[18] Y. Chi, Y. Choi, J. Cong, and J. Wang, "Rapid cycle-accurate simulator for high-level synthesis," *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, pp. 178–183, 2019.

[19] Y. Choi, K. You, J. Choi, and W. Sung, "A real-time FPGA-based 20,000-word speech recognizer with optimized DRAM access," *IEEE Trans. Circuits and Systems I: Regular Papers (TCAS1)*, 57(8), pp. 2119–2131, 2010.

[20] Y. Choi, J. Cong, and D. Wu, " FPGA implementation of EM algorithm for 3-D CT reconstruction," in *Proc. 22nd IEEE Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*, pp. 157–160, 2014.

[21] Y. Choi and J. Cong, "Acceleration of EM-based 3D CT reconstruction using FPGA," *IEEE Trans. Biomedical Circuits and Systems (TBCAS)*, 10(3), pp. 754–767, 2016.

[22] Y. Choi, *et al.*, "A quantitative analysis on microarchitectures of modern CPU-FPGA platforms," in *Proc. DAC*, pp. 109–114, 2016.

[23] Y. Choi and J. Cong, "HLScope: High-Level performance debugging for FPGA designs," in *Proc. IEEE Int. Symp. Field-Programmable Custom Computing Machines (FCCM)*, pp. 125–128, 2017.

[24] Y. Choi, P. Zhang, P. Li, and J. Cong, "HLScope+: Fast and accurate performance estimation for FPGA HLS," in *IEEE/ACM Int. Conf. Computer Aided Design (ICCAD)*, pp. 691–698, 2017.

[25] Y. Choi and J. Cong, "HLS-based optimization and design space exploration for applications with variable loop bounds," in *IEEE/ACM Int. Conf. Computer Aided Design (ICCAD)*, 2018.

[26] Y. Choi, *et al.*, "In-depth analysis on microarchitectures of modern heterogeneous CPU-FPGA platforms," *ACM Trans. Reconfigurable Technology and Systems (TRETS)*, 12(1), 2019.

[27] N. Chong, A. Donaldson, and J. Ketema, "A sound and complete abstraction for reasoning about parallel prefix sums," *ACM SIGPLAN Notices*, 49(1), pp. 397–409, 2014.

[28] M. Chung, J. Kim, and S. Ryu, "SimParallel: A high performance parallel SystemC simulator using hierarchical multi-threading," *IEEE Int. Symp. Circuits and Systems (ISCAS)*, pp. 1472–1475, 2014.

[29] J. Cong, and Y. Zou, "A comparative study on the architecture templates for dynamic nested loops," *IEEE Ann. Int. Symp. Field-Programmable Custom Computing Machines (FCCM)*, pp. 251–254, 2010.

[30] J. Cong, V. Sarkar, G. Reinman, and A. Bui, "Customizable domain-specific computing," *IEEE Design & Test of Computers*, 28(2) pp. 6–15, 2010.

[31] J. Cong, *et al.*, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 30(4), pp. 473–491, 2011.

[32] J. Cong, *et al.*, An optimal microarchitecture for stencil computation acceleration based on non-uniform partitioning of data reuse buffers, in *Proc. DAC*, 1–6, 2014.

[33] J. Cong, Z. Fang, H. Kianinejad, and P. Wei, "Revisiting FPGA acceleration of molecular dynamics simulation with dynamic data flow behavior in high-level synthesis," *ArXiv Preprint*, 2016.

[34] J. Cong, *et al.*, "CPU-FPGA co-optimization for big data applications: A case study of in-memory Samtool sorting," in *Proc. Int. Symp. FPGA*, 291, 2017.

[35] J. Cong, P. Wei, C. Yu, and P. Zhou, "Bandwidth optimization through on-chip memory restructuring for HLS," *Proc. Ann. Design Automat. Conf.*, 2017.

[36] J. Cong and J. Wang, "PolySA: polyhedral-based systolic array auto compilation," *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD)*, 2018.

[37] J. Cong, P. Wei, C. Yu, and P. Zhang, "Automated accelerator generation and optimization with composable, parallel and pipeline," *Proc. Ann. Design Automat. Conf.*, pp. 154–159, 2018.

[38] J. Cong, *et al.*, A multi-paradigm programming infrastructure for heterogeneous architectures, 2017, *www.nsf.gov*.

[39] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms," *The MIT Press*, Cambridge, MA, 2005.

[40] P. Coussy, et al., "An introduction to high-level synthesis," *IEEE Design & Test of Comput.*, 26(4), pp. 8–17, 2009.

[41] CPU Benchmarks, *https://www.cpubenchmark.net*, 2019.

[42] J. Curreri, *et al.*, "Performance analysis framework for high-level language applications in reconfigurable computing," *ACM Trans. Reconfigurable Technology and Systems*, 3(1), 2009.

[43] S. Dai, M. Tan, K. Hao, and Z. Zhang, "Flushing-enabled loop pipelining for high-level synthesis," *Proc. Ann. Design Automation Conf. (DAC)*, 2014.

[44] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, 38(1), 2011.

[45] R. Deville, I. Troxel, and A. George, "Performance monitoring for run-time management of reconfigurable devices," in *Proc. IEEE Int. Conf. Engineering of Reconfigurable Systems and Algorithms*, 175–181, 2005.

[46] B. Dickov, *et al.*, "Row-interleaved streaming data flow implementation of sparse matrix vector multiplication in FPGA," in *4th Workshop on Reconfigurable Computing*, 104, 109–118, 2010.

[47] H. Esmaeilzadeh, *et al.*, "Dark silicon and the end of multicore scaling," *IEEE Ann. Int. Symp. Computer Architecture (ISCA)*, 365–376, 2011.

[48] Falcon Computing Solutions, "Merlin Compiler," *https://www.falconcomputing.com/merlin-fpga-compiler/*, 2019.

[49] D. Finley, Optimized QuickSort, 2007, *http://alienryderflex.com/quicksort*.

[50] J. Fowers, *et al.*, "A high memory bandwidth FPGA accelerator for sparse matrix-vector multiplication," *IEEE Ann. Int. Symp. Field-Programmable Custom Computing Machines (FCCM)*, pp. 36–43, 2014.

[51] J. Goeders and S. J. Wilton, "Effective FPGA debug for high-level synthesis generated circuits," *IEEE Int. Conf. Field Programmable Logic and Appl. (FPL)*, 2014.

[52] J. Goeders and S. J. Wilton, "Using dynamic signal-tracing to debug compiler-optimized HLS circuits on FPGAs," *IEEE Ann. Int. Symp. Field-Programmable Custom Computing Machines (FCCM)*, pp. 127–134, 2015.

[53] J. Goeders and S. J. Wilton, "Signal-tracing techniques for in-system FPGA debugging of high-level synthesis circuits," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 36(1), pp. 83–96, 2017.

[54] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*, Kluwer Academic Publishers, 2002.

[55] S. Han, *et al.*, "EIE: efficient inference engine on compressed deep neural network," *ACM/IEEE Ann. Int. Symp. Computer Architecture (ISCA)*, 2016.

[56] S. Han, *et al.*, "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, pp. 78–84, 2017.

[57] M. Harris, S. Sengupta, and J. Owens, "Parallel Prefix Sum (Scan) with CUDA", *https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch39.html*, 2008.

[58] IBM, Application Note: Understanding DRAM Operation, 1996.

[59] Intel, Intel Quickpath interconnect FPGA core cache interface specification.

[60] Intel, Intel HLS Compiler, 2019, *https://www.altera.com/products/design-software/high-level-design/intel-hls-compiler/overview.html*.

[61] Intel, V-Series Transceiver PHY IP Core User Guide, 2017, *https://www.altera.com/en_US/pdfs/literature/ug/xcvr_user_guide.pdf*.

[62] Intel, "Intel FPGA SDK for OpenCL Pro Edition," *https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf*, 2019.

[63] Intel, "Quartus Prime Pro Edition Handbook," *https://www.intel.com/content/www/us/en/programmable/products/design-software/fpga-design/quartus-prime/user-guides.html*, 2019.

[64] Intel, Intel VTune Amplifier, 2019, *http://www.intel.com/*.

[65] Intel, "Intel Xeon Processor E5-2666 v3," *www.intel.com/*, 2016.

[66] Intel, "Intel Xeon Processor E5-2680 v4," *www.intel.com/*, 2016.

[67] H. Jaeger and H. Haas, "Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication," *Science*, 304(5667):78–80, 2004.

[68] J. Jang, S. Choi, and V. Prasanna, "Energy-and time-efficient matrix multiplication on FPGAs," *IEEE T. VLSI*, 13(11):1305–1319, 2005.

[69] R. Kastner, M. Matai, and S. Neuendorffer, "Parallel Programming for FPGAs", *ArXiv E-prints*, http://kastner.ucsd.edu/hlsbook/, 2018.

[70] Kingston, KVR13LSE9/8 memory module specifications, 2012, *http://www.kingston.com/datasheets/*.

[71] S. Koehler, J. Curreri, and A. George, "Performance analysis challenges and framework for high-performance reconfigurable computing," *Parallel Computing*, 34(4–5):217–230, 2007.

[72] D. Koeplinger, *et al.*, "Automatic generation of efficient accelerators for reconfigurable hardware," in *Proc. ISCA*, 115–127, 2016.

[73] P. Kogge, and H. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Trans. Computers*, 100(8), pp. 786–793, 1973.

[74] S. Kundu and I. F. Akyildiz, "Deadlock free buffer allocation in closed queueing networks," *Queueing Systems*, 4(1), pp. 47–56, 1989.

[75] S. Lahti, P. Sjövall, and J. Vanne, "Are we there yet? A study on the state of high-level synthesis," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 38(5), pp. 898–911, 2019.

[76] J. Lancaster, J. Buhler, and R. Chamberlain, "Efficient runtime performance monitoring of FPGA-based applications," in *Proc. IEEE Int. System-on-Chip Conf.*, 2009.

[77] C. Lee, O. Mutlu, V. Narasiman, and Y. Patt, "Prefetch-aware DRAM controllers," in *Proc. Int. Symp. Microarchitecture*, 200–209, 2008.

[78] J. Lei, Y. Chen, Y. Li, and J. Cong, "A high-throughput architecture for lossless decompression on FPGA designed using HLS," in *Proc. Int. Symp. FPGA*, 277, 2016.

[79] P. Li, P. Zhang, L. Pouchet, and J. Cong, "Resource-aware throughput optimization for high-level synthesis," in *Proc. Int. Symp. FPGA*, 200–209, 2015.

[80] Y. Liang, S. Wang, and W. Zhang, "FlexCL: a model of performance and power for OpenCL workloads on FPGAs," *IEEE Trans. Computers*, 67(12), pp. 1750–1764, 2018.

[81] R. Lienhart, A. Kuranov, and V. Pisarevsky, "Empirical analysis of detection cascades of boosted classifiers for rapid object detection," *Joint Pattern Recognition Symp.*, pp. 297–304, 2003.

[82] J. Liu and J. Cong, "Dataflow systolic array implementations of matrix decomposition using high level synthesis," *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, pp. 187–187, 2019.

[83] A. Mahapatra, Y. Liu, and B. C. Schafer, "Accelerating cycle-accurate system-level simulations through behavioral templates," *Integration*, 62, 282–291, 2018.

[84] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design & Test of Computers*, 26(4), 18–25, 2009.

[85] B. Meng, G. Pratx, and L. Xing, "Ultrafast and scalable cone-beam CT reconstruction using MapReduce in a cloud computing environment," *Med. Physics*, vol. 38, no. 12, pp. 6603–6609, 2011.

[86] Mentor Graphics, "ModelSim PE," *http://s3.mentor.com/public_documents/datasheet/products/fv/modelsim_pe_ds.pdf*, 2019.

[87] Mentor Graphics, "Veloce Emulation Platform," *https://www.mentor.com/products/fv/emulation-systems/*, 2019.

[88] Microsoft, "Microsoft Azure", https://azure.microsoft.com/, 2009.

[89] J. S. Monson and B. L. Hutchings, "New approaches for in-system debug of behaviorally-synthesized FPGA circuits," *IEEE Int. Conf. Field Programmable Logic and Appl. (FPL)*, 2014.

[90] J. S. Monson and B. L. Hutchings, "Using source-level transformations to improve high-level synthesis debug and validation on FPGAs," *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, pp. 5–8, 2015.

[91] G. Morris and V. Prasanna, "An FPGA-based floating-point Jacobi iterative solver," *Proc. Int. Symp. Parallel Architectures, Algorithms and Networks (ISPAN)*, 2005.

[92] M. Nanjundappa, et al., "SCGPSim: A fast SystemC simulator on GPUs," *Proc. Asia and South Pacific Design Automation Conf.*, pp. 149–154, 2010.

[93] R. Nane, et al., "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 35(10), 1591–1604, 2015.

[94] NVIDIA, NVIDIA Nsight, 2019, *http://www.nvidia.com/*.

[95] NVIDIA, CUDA C Programming Guide, 2019, *http://www.nvidia.com/*.

[96] NVIDIA, NVIDIA Tela M60 GPU Accelerator, 2016, *https://www.nvidia.com/*.

[97] J. Park, P. Diniz, and K. Shayee, "Performance and area modeling of complete FPGA designs in the presence of loop transformations," *IEEE T. Computers*, 53(11):1420–1435, 2004.

[98] D. Pellerin and S. Thibault *Practical FPGA programming in C*, Prentice Hall, 2005.

[99] L. Pouchet, "PolyBench/C", http://web.cse.ohio-state.edu/ pouchet.2/software/polybench/, 2015.

[100] R. Prabhakar, *et al.*, "Generating configurable hardware from parallel patterns," in *Proc. ASPLOS*, 50(2), 651–665, 2016.

[101] J. Pu, *et al.*, "Programming heterogeneous systems from an image processing DSL," *ACM Trans. Architecture and Code Optimization*, 14(3), 26, 2017.

[102] A. Putnam, *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," *ACM SIGARCH Computer Architecture News*, 42(3), 13–24, 2014.

[103] W. Qadeer, *et al.*, "Convolution engine: balancing efficiency & flexibility in specialized computing," *ACM SIGARCH Computer Architecture News*, 41(3), 24–35, 2013.

[104] J. Ragan-Kelley, *et al.*, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *ACM SIGPLAN Notices*, 48(6), 519–530, 2013.

[105] B. Reagon, *et al.*, "Machsuite: Benchmarks for accelerator design and customized architectures," in *Proc. IISWC*, 110–119, 2014.

[106] ROSE compiler infrastructure, *http://rosecompiler.org/*.

[107] R. Sampson, *et al.*, "Sonic millip3de: A massively parallel 3d-stacked accelerator for 3d ultrasound," in *IEEE Int. Symp. High Performance Computer Architecture (HPCA)*, 318–329, 2013.

[108] A. Schmidt, N. Steiner, M. French, and R. Sass, "HwPMI: an extensible performance monitoring infrastructure for improving hardware design and productivity on FPGAs," *Int. J. Reconfigurable Computing*, 2012.

[109] T. Schmidt, G. Liu, and R. Dömer, "Exploiting thread and data level parallelism for ultimate parallel SystemC simulation," *Proc. Ann. Design Automation Conf. (DAC)*, 2017.

[110] O. Segal, et al., "Sparkcl: A unified programming framework for accelerators on heterogeneous clusters," *ArXiv Preprint*, 2015.

[111] Y. Shao, *et al.*, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *Proc. ISCA*, 97–108, 2014.

[112] Y. Shan, *et al.*, "FPGA and GPU implementation of large scale SpMV," in *IEEE Symp. Application Specific Processors (SASP)*, 64–70, 2010.

[113] W. Snyder, "Verilator: Speedy Reference Models, Direct from RTL," *https://www.veripool.org/papers/Verilator_Modeling_UMass2017b_pres.pdf*, 2017.

[114] L Soule and T. Blank, "Parallel logic simulation on general purpose machines," *Proc. ACM/IEEE Design Automation Conf. (DAC)*, pp. 166–171, 1988.

[115] E. Sozzo, et al., "A common backend for hardware acceleration on FPGA," *IEEE Int. Conf. Comput. Design (ICCD)*, pp. 427–430, 2017.

[116] J. Sun, H. Shum, and N. Zheng, "Stereo matching using belief propagation," *IEEE Trans. Pattern Anal. Mach. Intell.*, 25(7), pp. 787–800, 2003.

[117] Synopsys, "VCS Functional Verification Solution," *https://www.synopsys.com/verification/simulation/vcs.html*, 2019.

[118] Synopsys, "ZeBu Fast Emulation," *https://www.synopsys.com/verification/emulation.html*, 2019.

[119] M. Tan, *et al.*, "Elasticflow: A complexity-effective approach for pipelining irregular loop nests," *Proc. IEEE/ACM Int. Conf. Computer Aided Design (ICCAD)*, pp. 78–85, 2015.

[120] A. Verma, *et al.*, "Developing dynamic profiling and debugging support in OpenCL for FPGAs," in *Proc. DAC*, 56–61, 2017.

[121] Z. Wang, B. He, W. Zhang, and S. Jiang, "A performance analysis framework for optimizing OpenCL applications on FPGAs," in *Proc. High Performance Computer Architecture (HPCA)*, pp. 114–125, 2016.

[122] D. J. Warne, N. A. Kelson, and R. F. Hayward, "Comparison of high level FPGA hardware design for solving tri-diagonal linear systems," *Procedia Computer Science*, 29, pp. 95–101, 2014.

[123] M. Wissolik, D. Zacher, A. Torza, and B. Day, Virtex UltraScale+ HBM FPGA: A revolutionary increase in memory performance, 2017, *https://www.xilinx.com/support/documentation/white_papers/wp485-hbm.pdf*.

[124] All Programmable FPGAs and 3D ICs, Xilinx, 2015, Available: http://www.xilinx.com/products/silicon-devices/fpga.html.

[125] Xilinx, AXI Reference Guide UG761, 2012, *http://www.xilinx.com/*.

[126] Xilinx, "ChipScope Pro Software and Cores (UG029)," *https://china.xilinx.com/support/documentation/sw_manuals/xilinx14_7 /chipscope_pro_sw_cores_ug029.pdf*, 2012.

[127] Xilinx, "FIFO Generator v13.2 (PG057)," *https://www.xilinx.com/support/documentation/ip_documentation/fifo_generator/v13_2/pg057-fifo-generator.pdf*, 2017.

[128] Xilinx, SDAccel Development Environment, 2016, *http://www.xilinx.com/*.

[129] Xilinx, "UltraScale architecture and product data sheet: overview (DS890)", https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf, 2018.

[130] Xilinx, Vivado High-level Synthesis UG902, 2018, *http://www.xilinx.com/*.

[131] F. Xu and K. Mueller, "Real-time 3D computed tomographic reconstruction using commodity graphics hardware," *Physics in Med. Biol.*, 52, pp. 3405–3417, 2007.

[132] Y. Xu, *et al.*, "Belief propagation implementation using CUDA on an NVIDIA GTX 280," *Advances in Artif. Intell.*, pp. 180–189, 2009.

[133] M. Yan and L. Vese, "Expectation maximization and total variation-based model for computed tomography reconstruction from undersampled data," in *SPIE Med. Imag.*, pp. 79612X–1–8, 2011.

[134] C. Yu, et al., "S2FA: an accelerator automation framework for heterogeneous computing in datacenters," *Proc. Ann. Design Automation Conf.*, 2018.

[135] C. Zhang, *et al.*, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. Int. Symp. FPGA*, 161–170, 2015.

[136] Y. Zhang, et al., "FPGA vs. GPU for sparse matrix vector multiply," *IEEE Int. Conf. Field-Programmable Technology (FPT)*, pp. 255–262, 2009.

[137] J. Zhao, *et al.*, "COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications," *Proc. Int. Conf. Computer Aided Design (ICCAD)*, pp. 430–437, 2017.

[138] G. Zhong, *et al.*, "Lin-analyzer: A high-level performance analysis tool for FPGA-based accelerators," in *Proc. DAC*, 136–141, 2016.

[139] G. Zhong, *et al.*, "Design Space exploration of FPGA-based accelerators with multi-level parallelism," in *Proc. DATE*, 2017.

[140] L. Zhou and V. Prasanna, "Sparse matrix-Vector multiplication on FPGAs," *Proc. Int. Symp. Field-Programmable Gate Arrays (FPGA)*, pp. 63–74, 2005.

[141] Q. Zhu and M. Tatsuoka, "High quality IP design using high-level synthesis design flow," *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 212–217, 2016.

[142] H. Zohouri, *et al.*, "Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs," *Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 409–420, 2016.