ARCHITECTURAL FRAMEWORKS FOR AUTOMATED DESIGN AND OPTIMIZATION OF HARDWARE ACCELERATORS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Tao Chen

May 2018

ARCHITECTURAL FRAMEWORKS FOR AUTOMATED DESIGN AND OPTIMIZATION OF HARDWARE ACCELERATORS

Tao Chen, Ph.D.

Cornell University 2018

As technology scaling slows down and only provides diminishing improvements in general-purpose processor performance, computing systems are increasingly relying on customized accelerators to meet the performance and energy efficiency requirements of emerging applications. For example, today's mobile SoCs rely on accelerators to perform compute-intensive tasks, and datacenters are starting to deploy accelerators for applications such as web search and machine learning. This trend is expected to continue and future systems will contain more specialized accelerators. However, the traditional hardware-oriented accelerator design methodology is costly and inefficient because it requires significant manual effort in the design process. This development model is unsustainable in the future where a wide variety of accelerators are expected to be designed for a large number of applications. To solve this problem, the development cost of accelerators must be drastically reduced, which calls for more productive design methodologies that can create *high-quality* accelerators with *low manual effort*.

This thesis addresses the above challenge with architectural frameworks that combine *novel accelerator architectures* with *automated design and optimization frameworks* to enable designing high-performance and energy-efficient accelerators with minimal manual effort. Specifically, the first part of the thesis proposes a framework for automatically generating accelerators that can effectively tolerate long, variable memory latencies, which improves performance and reduces design effort by removing the need to manually create data preloading logic. The framework leverages architecture mechanisms such as memory prefetching and access/execute decoupling, as well as automated compiler analysis to generate accelerators that can intelligently preload data needed in the future from the main memory.

The second part of the thesis proposes a framework for building parallel accelerators that leverage concepts from task-based parallel programming, which enables software programmers to quickly create high-performance accelerators using familiar parallel programming paradigms, without needing to know low-level hardware design knowledge. The framework uses a computation model that supports dynamic parallelism in addition to static parallelism, and includes a flexible architecture that supports dynamic scheduling to enable mapping a wide range of parallel applications to hardware accelerators and achieve good performance. In addition, we designed a unified language that can be mapped to both software and hardware, enabling programmers to create parallel software and parallel accelerators in a unified framework.

The third part of the thesis proposes a framework that enables accelerators to perform intelligent dynamic voltage and frequency scaling (DVFS) to achieve good energy-efficiency for interactive and real-time applications. The framework combines program analysis and machine learning to train predictors that can accurately predict the computation time needed for each job, and adjust the DVFS levels to reduce the energy consumption.

BIOGRAPHICAL SKETCH

Tao Chen attended Fudan University from the year 2008 to 2012, where he received his Bachelor of Science degree (with distinction) in Microelectronics. After graduation from Fudan University, he began pursuing his Ph.D. degree in the School of Electrical and Computer Engineering at Cornell University, where he worked with his advisor, Professor G. Edward Suh, on topics in the field of computer architecture, with a focus on hardware accelerators.

This dissertation is dedicated to my parents.

ACKNOWLEDGEMENTS

Six years ago, I arrived at Cornell to pursue my Ph.D. degree. At that time, I was a young student who was nervous about the challenges ahead, and was uncertain if I could make it to the end. Six years later, I have successfully completed this dissertation and become a doctor. I am extremely grateful to have so many people help me along this exciting and rewarding journey.

First and foremost, I would like to express my sincerest gratitude to my advisor, Professor G. Edward Suh. Throughout my Ph.D. study, Ed has supported me without reservations and provided valuable guidance, advice, encouragement, and help whenever I needed them. Ed gave me the freedom to pursue research directions that I am passionate about, and at the same time providing necessary guidance so that I can stay on the right path. Ed is always ready to offer his generous help, whether it is about brainstorming ideas, revising a paper, or perfecting a conference talk. Ed is also always encouraging when I face difficulties, which helped me stay optimistic and motivated through the challenging journey of working towards a Ph.D. degree. I am deeply grateful to him.

I would like to thank my committee members, Professor David H. Albonesi and Professor Zhiru Zhang. Dave is a role model to me as a great computer architect who is passionate about research and teaching. Dave's course on memory systems is one of the most exciting classes that I took, and inspired me to pursue the research on memory optimizations for accelerators. Zhiru's vision and his pioneering work on high-level synthesis is a major source of inspiration for my research. He also provided many helpful suggestions and comments that greatly improved my work.

I would like to thank Professor Christopher Batten for his guidance and support, and for generously sharing the research infrastructure that his group developed. Chris also mentored me on the parallel accelerator project and provided many insightful suggestions and advice. I am sincerely grateful to him.

Special thanks to my friends and colleagues at CSL who helped me tremendously both with my research and with navigating graduate school. I would like to thank members of the Suh Research Group. I want to thank Daniel Lo for providing many helpful comments and insights that greatly helped my research. I would like to thank Ruirui (Raymond) Huang and Wing-kei (KK) Yu for sharing their experiences as senior Ph.D. students. Special thanks to Yao Wang for providing great suggestions and directions throughout my Ph.D. journey. I would also like to thank Andrew Ferraiuolo, Mohamed Ismail, Benjamin Wu, Weizhe (Will) Hua, and Mulong Luo for their support and friendship, which made my life as a Ph.D. student a lot more enjoyable. Special thanks to Shreesha Srinath for being both a mentor and a good friend. I enjoyed discussing and debating research ideas with him, and also benefited from his suggestions and guidance as a senior student. I would also like to thank Xiaodong Wang, Gai Liu, Steve Dai, and all other CSL students, whom I am fortunate to be friends with. I am proud to be a member of this brilliant community.

I would like to thank my girlfriend Lin, for being caring and supportive for my life and research. Her encouragement helped me push forward in times of difficulties, and her warmth made me feel delighted every day.

Finally, I would like to express my deepest gratitude to my parents, Xin Chen and Meihua Liu, for their unconditional love and support. They taught me to be persistent and optimistic, and that hard work pays off, which got me this far in my academic endeavor. They encouraged me to think independently, and supported me no matter what decisions I have made in my life. I am proud to have them as my parents, and I hope I have made them proud of me too.

TABLE OF CONTENTS

Biographical Sketch iii
Dedication iv
Acknowledgements v
Table of Contents vii
List of Tables x
List of Figures xi

1 Introduction **1**
1.1 Background 1
1.2 Design Complexity of Accelerators 3
1.3 Thesis Contributions and Organization 6

2 Memory Optimization Framework for Efficient Data Supply **9**
2.1 Introduction 9
2.2 Overview 12
2.2.1 System Architecture 12
2.2.2 High-Level Synthesis 13
2.2.3 Impact of Memory Accesses on Accelerator Performance . 14
2.2.4 Data Preloading Framework 16
2.3 Prefetching 18
2.4 Decoupled Access/Execute 19
2.4.1 Access Unit 23
2.4.2 Memory Units 24
2.4.3 Execute Unit 26
2.4.4 Deadlock Avoidance 26
2.4.5 Customization of Memory Units 27
2.4.6 Automated DAE Accelerator Generation 28
2.5 Evaluation 29
2.5.1 Methodology 30
2.5.2 Experimental Setup 31
2.5.3 Baseline Validation 33
2.5.4 Performance Results 34
2.5.5 Area, Power, and Energy Results 36
2.5.6 Design Space Exploration: Queue Size 42

3 Parallel Accelerator Framework **44**
3.1 Introduction 44
3.2 Computation Model for Dynamic Parallelism 48
3.2.1 Primitives 48
3.2.2 Continuation Passing 50
3.2.3 Scheduling the Computation 54
3.2.4 Function Calls 56

3.3 Accelerator Architecture 56
3.3.1 FlexArch Tile and PE Architecture 58
3.3.2 LiteArch Tile and PE Architecture 64
3.3.3 Networks 64
3.3.4 Memory Hierarchy 65
3.3.5 CPU-Accelerator Interface 66
3.4 Design Methodology and Framework 67
3.4.1 Architectural Template 67
3.4.2 Algorithm Description Format 68
3.4.3 Accelerator RTL Generation 69
3.5 Unified Framework for Parallel Accelerators and Software 71
3.5.1 CPPWD-TBB Library 72
3.5.2 Programmability 74
3.6 Evaluation 75
3.6.1 Benchmarks 76
3.6.2 Design Effort Comparison 79
3.6.3 Hardware Prototype on Today's FPGA 81
3.6.4 Simulation Methodology 82
3.6.5 Performance Results 84
3.6.6 Resource Utilization 95
3.6.7 Power and Energy Efficiency 96
3.6.8 Cache Size Customization 97
3.6.9 Parallel Software with Unified Description 98

4 Predictive DVFS Framework for Energy Efficiency 101
4.1 Introduction 101
4.2 Fine-grained DVFS for Hardware Accelerators 104
4.2.1 System Setup 104
4.2.2 Tasks and Jobs 104
4.2.3 Execution Time Variation 105
4.2.4 Current Approaches to DVFS 106
4.3 Predictive DVFS Framework for Hardware Accelerators 109
4.3.1 Source of Execution Time Variation 112
4.3.2 Features from Hardware Accelerators 113
4.3.3 Identifying and Obtaining Features 115
4.3.4 Prediction Model 116
4.3.5 Hardware Slicing 119
4.3.6 DVFS Model 120
4.3.7 Predictor Operation Modes 121
4.3.8 Case Study 122
4.4 Evaluation 124
4.4.1 Methodology 125
4.4.2 Experimental Setup 126
4.4.3 Results for ASIC Accelerators 128

4.4.4 Results for FPGA-based Accelerators 135
4.4.5 Extensions 136

5 Related Work **139**
5.1 High-Level Design Methodologies for Accelerators 139
5.1.1 High-Level Synthesis 139
5.1.2 Hardware Generation Languages 142
5.2 Data Supply for Accelerators 142
5.2.1 Data Supply for In-Core Accelerators 143
5.2.2 Memory Architecture for Standalone Accelerators 144
5.2.3 Memory Optimizations in High-Level Synthesis 145
5.3 Parallel Accelerators 146
5.3.1 Task-Based Parallel Programming 146
5.3.2 Design Methodologies for Parallel Accelerators 147
5.4 Power Management for Systems with Time Constraints 148
5.4.1 Dynamic Voltage and Frequency Scaling 148
5.4.2 Execution Time Prediction 150

6 Conclusion **151**
6.1 Summary 151
6.2 Future Directions 152
6.2.1 Compiler Support for Parallel Accelerators 152
6.2.2 Hybrid GPP-Accelerator Work-Stealing Architecture . . . 153

Bibliography **155**

LIST OF TABLES

2.1 Comparison of prefetching and DAE. 18
2.2 Summary of benchmarks. 31
2.3 Lines of code (LOC) comparison between the input to the framework (C++ source code) and the generated Verilog code. Blank lines and comments are not counted. 32
2.4 Experiment parameters. 32
2.5 Area and power of the baseline and DAE accelerators. The Abs column shows absolute numbers, and the Norm column shows results normalized to the baseline. 37
2.6 Impact of the queue size customization on area and performance. Numbers are normalized to LQ16/SQ08. 43

3.1 Comparison between tile architectures. 58
3.2 Summary of task APIs. 68
3.3 Programmability comparison. M: Manual. A: Assisted by the compiler. 74
3.4 Summary of benchmarks. PA: Parallelization Approach, PF=parallel-for, FJ=fork-join, CP=continuation passing. R/N: Recursive/Nested Parallelism. DP: Data-Dependent Parallelism. MP: Memory Access Pattern. MI: Memory Intensity. . . . 77
3.5 Lines of code (LOC) comparison between Cilk Plus, CPPWD (the input to the framework), and the generated Verilog code. Blank lines and comments are not counted. 80
3.6 Platform configuration. 83
3.7 Scalability of Cilk Plus. The numbers are the speedup of a n-core implementation over a single core implementation. 85
3.8 Scalability of the accelerators. The numbers are the speedup of a n-PE implementation over a single PE implementation. 86
3.9 FlexArch accelerators resource utilization. Each tile consists of four PEs and a cache. DSPs are shown in the number of DSP48 slices. BRAMs are shown in the number of RAM18's (each RAM36 counts as two RAM18's). 94
3.10 LiteArch accelerators resource utilization. Each tile consists of four PEs and a cache. 95

4.1 Summary of features in prediction model. 115
4.2 Variables in prediction model. 118
4.3 Summary of benchmarks. 127
4.4 Summary of workloads. 127
4.5 Summary of ASIC implementation results. 128

LIST OF FIGURES

1.1 Current accelerator design flow. 3

2.1 System architecture. 13
2.2 High-level synthesis flow. 14
2.3 The inner loop of sparse matrix vector multiplication. 15
2.4 Example schedule of an HLS SpMV accelerator with (a) ideal memory (b) cache that has a miss when accessing `vec`. 15
2.5 Modified memory request message format. 19
2.6 Example schedule of the access part of a decoupled SpMV accelerator when there is a cache miss. 21
2.7 Architecture of access/execute decoupled accelerators. 22
2.8 Hardware structure of the memory unit. 24
2.9 High-level flow of decoupled access/execute accelerator generation. 28
2.10 Comparison of baseline accelerators performance with processors. 34
2.11 Performance of the proposed schemes normalized to the baseline accelerator. 35
2.12 Area breakdown of DAE accelerators. The baseline area is shown for comparison. 37
2.13 Power breakdown of DAE accelerators. The baseline power is shown for comparison. 39
2.14 Energy comparison. 40
2.15 Energy breakdown of DAE accelerators. The baseline energy is shown for comparison. If baseline energy is higher than DAE, it is annotated with a number on top of the bar. 41
2.16 Performance comparison when varying load queue (LQ) sizes. . 42

3.1 Continuation passing for (a) sequential composition of tasks, (b) fork-join. Downward arrows represent spawning tasks. Horizontal arrows represent creating successor tasks. Dotted arrows represent returning values (arguments). 51
3.2 Task graphs constructed using continuation passing. (a) Vector-vector add. Node labels represent the start and end indices of the sub-vectors. (b) Fibonacci. Each numbered node represents the task for `fib(n)`. Nodes labeled `S` represent the successor (sum) tasks. (c) Dynamic programming. Solid arrows represent spawns along which the continuation is passed. Result values are passed along dotted arrows, with the final result sent to the continuation *k*. 52
3.3 System architecture. The accelerator is shown in the shaded box. 57
3.4 FlexArch tile. 59
3.5 P-Store architecture. 62

3.6 LiteArch tile. 64
3.7 Accelerator design flow. 67
3.8 C++-based worker description for Fibonacci. 70
3.9 Unified parallel accelerator and software flow. 71
3.10 Hardware and software implementation stacks. 73
3.11 Accelerators performance compared to parallel software on Zedboard. 82
3.12 Normalized accelerator performance. The x-axis is the number of workers (PEs). The y-axis is performance normalized to a single OOO core. The horizontal bar indicates the performance of an eight-core Cilk Plus implementation. 88
3.13 Execution time breakdown for FlexArch accelerators. Busy: PE is actively processing a task. Wait: PE is waiting to steal a task. Steal: PE is performing task stealing. 90
3.14 Maximum Task Queue and P-Store Occupancy for FlexArch accelerators. The x-axis is the number of PEs. The y-axis is the maximum number of occupied entries in any task queue or P-store at any point of execution. 93
3.15 Normalized performance and energy efficiency. Energy efficiency is the inverse of energy consumption. Both performance and energy efficiency are normalized to the Cilk Plus implementation on 8 OOO cores. Points to the right of the vertical line have better performance. Points above the horizontal line have better energy efficiency. The diagonal line represents the iso-power line. Points above the diagonal line have lower power. Points for the same benchmark are linked. Note that both axes are in log scale. 97
3.16 Performance when varying accelerator L1 cache size. 98
3.17 Performance comparison between parallel software implementations. 99

4.1 A sequence of jobs for a task. 105
4.2 Execution time of H.264 decoder for three video clips at 60fps. Each point is one job (frame). 106
4.3 Actual execution time and execution time predicted by PID controller for H.264. 108
4.4 Operation of predictive DVFS. 110
4.5 Accelerator with execution time prediction-based DVFS. 110
4.6 Execution time prediction flow. 111
4.7 Control-Datapath structure of an accelerator. 112
4.8 Example Finite State Machine in control unit. 113
4.9 Predictor operation modes. 122
4.10 Architecture of H.264 decoder. 122

4.11 Errors of slice-based execution time prediction. The box extends from the 25% to 75%, with a line at the median. The whiskers show the range of the data. Outliers are shown as individual points. 130
4.12 Normalized energy and deadline misses of different DVFS schemes. 130
4.13 Area, energy and execution time overhead of prediction slice. . . 131
4.14 Normalized energy and deadline when overhead is removed. . . 132
4.15 Normalized energy and deadline misses with voltage boosting. . 133
4.16 Normalized energy and deadline misses when varying deadlines (averaged across all benchmarks). 134
4.17 Normalized energy and deadline misses for FPGA-based accelerators. 135
4.18 Area, energy and execution time overhead of prediction slice for FPGA accelerators. 136
4.19 Comparison of prediction errors and deadline misses between slicing at RTL and HLS level. 137
4.20 Comparison of area, energy and execution time overhead between slicing at RTL and HLS level. 138

CHAPTER 1

INTRODUCTION

As technology scaling slows down and only provides diminishing improvements in general-purpose processor performance, computing systems are increasingly relying on customized accelerators to meet the performance and energy efficiency requirements of emerging applications. For example, today's mobile SoCs rely on accelerators to perform compute-intensive tasks such as multimedia processing and face recognition [10,89], and datacenters are starting to deploy FPGA and ASIC accelerators for applications such as web search [87] and machine learning [56]. This trend is expected to continue and future systems will contain more specialized accelerators. However, the traditional hardware-oriented accelerator design methodology is costly and inefficient because it requires significant manual effort in the design process. As a result, only the most widely used applications today are able to amortize the high development cost and benefit from accelerators. This development model is unsustainable in the future where a wide variety of accelerators are expected to be designed for a large number of applications. To address this problem, the development cost of accelerators must be drastically reduced, which calls for more productive design methodologies that can create *high-quality* accelerators with *low manual effort*.

1.1 Background

In the past five decades, the landscape of semiconductor technology scaling has been largely governed by two laws: *Moore's law* [71] and *Dennard scaling* [38]. Moore's law states that the number of transistors that can be economically fit onto an integrated circuit doubles with each technology generation. Dennard scaling states that when voltages are scaled along with transistor dimensions, metal-oxide semiconductor field-effect transistors (MOSFETs) can be made to switch faster, and at the same time, consume less power. For decades, computer architects have successfully harnessed architecture scaling, enabled by Moore's law, and frequency scaling, enabled by Dennard scaling, to design microprocessors with exponentially higher clock frequencies and new architecture features, achieving tremendous growth in performance and energy efficiency.

Unfortunately, technology scaling is becoming increasingly more difficult today. Dennard scaling started to break down around the mid-2000s, when it became clear that scaling supply voltage proportionally with transistor feature size is no longer feasible due to the difficulties in scaling the threshold voltage. As a result, continuing to scale transistor frequency would lead to drastically increased power density, which is impractical due to cooling constraints. This is known as the *power wall*, which effectively ended the exponential increase of microprocessor clock frequency, and prompted the industry to shift to multi-core architectures. Moore's law has significantly slowed down too: as transistor feature sizes approach physical limits, developing new technology generations is becoming increasingly difficult and costly. It is likely that in the near future, we will enter an era without Moore's law or Dennard scaling. As a result, continuing to scale the number of cores on a chip is unsustainable without Moore's law. To make things worse, not all cores can be powered on at the same time due to power limits [39]. As a result, computer architects have to turn to other approaches to continue improving performance and energy efficiency. Among many candidates, accelerators are emerging as a promising solution. Accelerators trade off generality for efficiency by specializing the hardware for a reduced

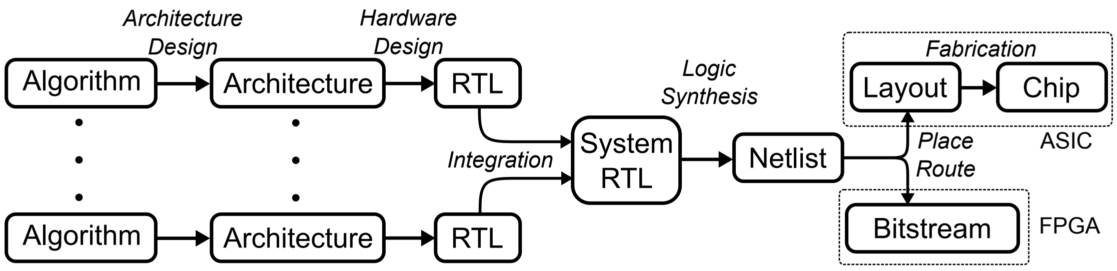


Figure 1.1: Current accelerator design flow.

set of applications. Accelerators have been shown to deliver orders of magnitude better performance and energy efficiency compared to general-purpose processors [24, 56], and offer a promising way going forward in the absence of technology scaling benefits.

1.2 Design Complexity of Accelerators

Accelerators achieve efficiency through specialization. By focusing on a reduced set of applications (often just one), the data path and memory circuitry of an accelerator can be specialized and optimized, thus achieving better performance and/or energy efficiency. It has been shown that many of the optimizations in accelerators are algorithm-specific [50], which means that different applications will need to use different accelerators to in order to achieve the best efficiency. For this reason, we focus on accelerators that are custom designed for their target applications in this thesis. Future systems need to use an increasing number of accelerators for various applications in order to sustain the growth in performance and energy efficiency. This trend can already be seen in the mobile space, where each generation of SoCs integrates more accelerators [27, 102].

Unfortunately, an increasing number of accelerators creates a design complexity problem. Most of today's accelerators are designed using a low-level hardware design methodology that involves significant manual effort and incurs high non-recurring engineering (NRE) costs per accelerator. As a result, the design complexity and cost grows quickly with the number of accelerators, which limits the number of accelerators that can be economically designed, and thus limiting the number of applications that can be accelerated. We call this problem the *design complexity wall*. Just as the power wall ended microprocessor's frequency increase, the design complexity wall may end the growth of accelerators too. Figure 1.1 shows the major steps of today's accelerator design flow. For each algorithm that needs to be accelerated, designers first conceive the architecture of the accelerator, and then perform hardware design by writing register-transfer-level (RTL) descriptions of the accelerators using hardware description languages such as Verilog or VHDL, or using software tools to generate the RTL. Then the designers integrate the RTL descriptions of the accelerators into a system which may contain other components such as general-purpose processor cores and memory controllers, and then run the standard synthesis, place and route tools to obtain the physical layout for ASIC implementation, or the bitstream for FPGA implementation. During each step of the process, designers also need to test the accelerators and repeat the step until the design meets the performance and functionality specifications. The poor productivity of this flow, especially the process of converting each algorithm to efficient hardware architectures and RTL, is a major contributor to the design complexity problem. Studies have shown that designing accelerators using this flow often takes months, which can be an order of magnitude longer compared to software implementations [36, 65, 109]. We observe that one primary reason for the low productivity is the high manual effort involved in the design process. Today's accelerator design methodologies require designers to manually describe the low-level details of an accelerator's operations statically at design time. In contrast, modern computer architectures and system software support many dynamic features that automatically adapts to application behavior at runtime for improved programmer productivity, performance, and/or energy efficiency. For example, dynamic scheduling, automatically managed memory hierarchy, and data prefetching free programmers from needing to manually manage instruction scheduling and data movements. Power management techniques such as dynamic frequency and voltage scaling (DVFS) adaptively adjust to application behavior to achieve energy savings. Task-based parallel programming frameworks provide abstractions for easily expressing diverse types of parallelism, and adaptively schedules the computation to allows programmers to write efficient parallel programs without needing to worry about low-level details.

Solving the design complexity problem would require hardware design to be much more productive, and more like modern software design. Just like high-level languages, reusable libraries, and optimizing compilers freed software programmers from needing to write low-level assembly code and drastically improved productivity, today's accelerator design also needs *high-level abstractions*, *reusable architectures*, and *automated design frameworks* in order to be productive and efficient. In addition, just as optimizing compilers needed to generate code with comparable quality to hand-written assembly, high-level accelerator design frameworks also need to generate high-quality hardware in order to be widely used.

1.3 Thesis Contributions and Organization

This thesis proposes to address the design complexity problem with architectural frameworks that combine *novel accelerator architectures* with *automated design and optimization frameworks* to enable designing high-performance and energy-efficient accelerators with minimal manual effort. The accelerator architectures provide high-level abstractions and reusable architecture templates that offer dynamic and adaptive features which can be applied to the design of high-quality accelerators for a wide range of applications. The automated frameworks include a *structured HLS* design methodology that combines the benefits of high-level synthesis and hardware generation techniques to enable designing accelerators with high productivity while retaining the flexibility of quality advantages of RTL designs. In addition, the frameworks leverage program analysis and compiler optimization techniques to enable generating optimized accelerators with novel architectural features. Specifically, the thesis proposes three such architectural frameworks to address concrete challenges in today's accelerator design, and achieve reduced design complexity, improved performance, and better energy efficiency.

1. The thesis proposes an architectural framework for automatically generating accelerators that can effectively tolerate long, variable memory latencies, which improves performance and reduces design effort by removing the need to manually create data preloading logic [23]. The framework leverages architecture mechanisms such as memory prefetching and access/execute decoupling, as well as compiler analysis to generate accelerators that can intelligently preload data needed in the future from the main memory. The framework uses program slicing and architecture templates

together with high-level synthesis to enable generating fully synthesizable accelerators automatically from high-level languages.

2. The thesis proposes an architectural framework for building parallel accelerators that leverages concepts from task-based parallel programming, which enables programmers to quickly create high-performance accelerators using familiar parallel programming paradigms, without needing to know low-level hardware design knowledge [22]. The framework uses a computation model that supports dynamic parallelism in addition to static parallelism, and includes a flexible architecture that supports dynamic scheduling to enable mapping a wide range of parallel applications to accelerators and achieve good performance. The framework includes an architecture template and uses hardware generation to automatically generate accelerators with the proposed architecture from high-level descriptions. In addition, the thesis proposes a unified language that can be mapped to both software and hardware, enabling programmers to create parallel software and parallel accelerators in a unified framework.

3. The thesis proposes an architectural framework that enables accelerators to perform intelligent dynamic voltage and frequency scaling (DVFS) to achieve good energy-efficiency for interactive and real-time applications [21]. The framework combines program analysis and machine learning to train predictors that can accurately predict the computation time needed for each job, and adjust the DVFS levels to reduce the energy consumption. The framework automatically generates the prediction hardware from the RTL description of an accelerator.

The rest of the thesis is organized as follows. Chapter 2 describes the framework for generating accelerators with efficient data supply. Chapter 3 describes the framework for generating parallel accelerators that support dynamic parallelism, and the unified language for describing both parallel hardware and software. Chapter 4 describes the framework for generating accelerators that can perform intelligent dynamic voltage and frequency scaling. Finally, Chapter 5 summarizes related work, and Chapter 6 concludes the thesis and discusses future directions.

CHAPTER 2

MEMORY OPTIMIZATION FRAMEWORK FOR EFFICIENT DATA SUPPLY

2.1 Introduction

This chapter proposes a framework to automatically optimize hardware accelerators and enable them to effectively hide long, variable memory latencies of an SoC memory hierarchy by preloading data in parallel to computations. The effective data preloading is achieved through hardware prefetching and design transformations to decouple memory accesses and computations. This framework is generally applicable to accelerator designs that are attached to the memory bus or the last-level cache and have their own memory access logic. This accelerator design style is widely adopted both in the industry and the research community [21, 24, 32, 66, 89, 113]. While the techniques proposed in this framework can be applied to any accelerator in general, the framework is designed to target accelerators that are generated using *High-Level Synthesis* (HLS). HLS compiles high-level languages such as C/C++ [20, 114], OpenCL [54], or domain-specific languages [11, 58] into RTL. HLS is becoming an increasingly popular approach to design accelerators because it raises the level of abstraction of hardware design, and is used in both industry designs [42] as well as hardware accelerator research [92, 98].

Unfortunately, even with HLS, data supply from memory often needs to be carefully coordinated with manual optimizations in order to achieve high performance in hardware accelerators. For example, today's HLS tools assume a fixed latency of all memory accesses, and rely on accelerator designers to write explicit logic to manage the communication between DRAM and on-chip scratchpad memory. This approach requires serious manual design efforts, and the resulting management logic is accelerator-specific and not reusable for other designs. Alternatively, designers can use caches to ease communication management given locality in memory accesses [8]. However, we found that caches are not sufficient to provide high performance without carefully orchestrated data supply. Unlike modern processors with expensive latency-hiding mechanisms such as dynamic scheduling, typical accelerators rely on a static pipeline schedule and a cache miss stalls the entire pipeline.

This framework proposed in this chapter aims to enable efficient data supply for HLS-based accelerators without manual efforts necessary today. To achieve this goal, we remove inefficiencies in today's cache-based accelerators in two ways. First, we use a prefetch engine to remove cache misses for easy-to-predict memory accesses. The prefetch engine is general and common across accelerators. For example, we use a stride prefetcher in our experiments. Second, to handle complex memory access patterns, we propose to decouple memory access logic of an accelerator from the main computation pipeline. For many accelerators, memory addresses of data that need to be accessed are often independent of main computations and can be computed ahead to fetch data in parallel to the main computation. Thus, by decoupling the memory access logic from the computation pipeline, it can run ahead to fetch and buffer the data to be consumed by the computation pipeline. As long as the access logic runs sufficiently ahead, it can absorb cache misses without stalling the computation pipeline, this hiding memory latency. In fact, data supply in manually optimized accelerators relies on such decoupling and preloading. In this chapter, we show that this decoupling can be done automatically using program slicing on a high-level accelerator design. The framework can be applied to existing high-level synthesis tools with minimal manual efforts.

While prefetching and access/execute decoupling have been studied for processing cores, we found that applying them to accelerators introduce new challenges. For prefetching, unlike processing cores, accelerators do not have program counters (PCs) that can be used to easily distinguish different sources of memory accesses. In order to apply traditional prefetch algorithms, we augment our accelerator generation process to automatically add additional tags.

We also found that simply decoupling memory accesses from main computations alone does not significantly improve the performance of accelerators unless independent accesses can be overlapped. The decoupled access/execute (DAE) architecture on processing cores rely on expensive out-of-order or dataflow execution to perform multiple accesses in parallel. For hardware accelerators with static pipelines, we show that simple decoupling of memory accesses through dedicated forwarding logic is sufficient to achieve good performance with minimal overhead in most cases.

In order to evaluate the effectiveness of the proposed framework, we applied prefetching and access/execute decoupling to eight HLS-based accelerators. The experimental results show that the proposed framework can be applied to accelerators with minimal manual efforts and significantly improve the performance compared to the baseline accelerator. The DAE architecture alone improved performance by 1.89x on average while the average speedup increased to 2.28x when prefetching was added. The optimizations also reduce energy consumption for many accelerators, by 15% on average.

The rest of the chapter is organized as follows. Section 2.2 provides an overview of the accelerator data supply problem and briefly discusses the proposed solution. Section 2.3 describes prefetching as an approach to improve accelerator data supply and a technique to enable efficient prefetching for hardware accelerators. Section 2.4 describes the architecture of access/execute decoupled accelerators as well as a framework to automatically generate them from a high-level description. Section 2.5 discusses our evaluation methodology, experimental setup, and evaluation results.

2.2 Overview

2.2.1 System Architecture

Figure 2.1 shows the high-level system architecture that we assume in this chapter. The system is a heterogeneous SoC that consists of general-purpose processing engines such as processor cores and GPGPUs as well as a large number of accelerators. We consider stand-alone accelerators that are loosely-coupled to the cores and have their own memory interfaces to access main memory. A processing core configures and initiates an accelerator, then the accelerator performs its computation without intervention from the core. Each accelerator has its own compute pipeline and accesses memory through an on-chip cache.

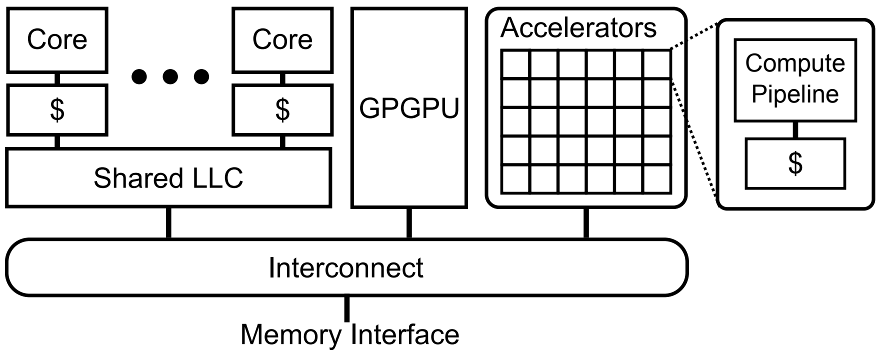
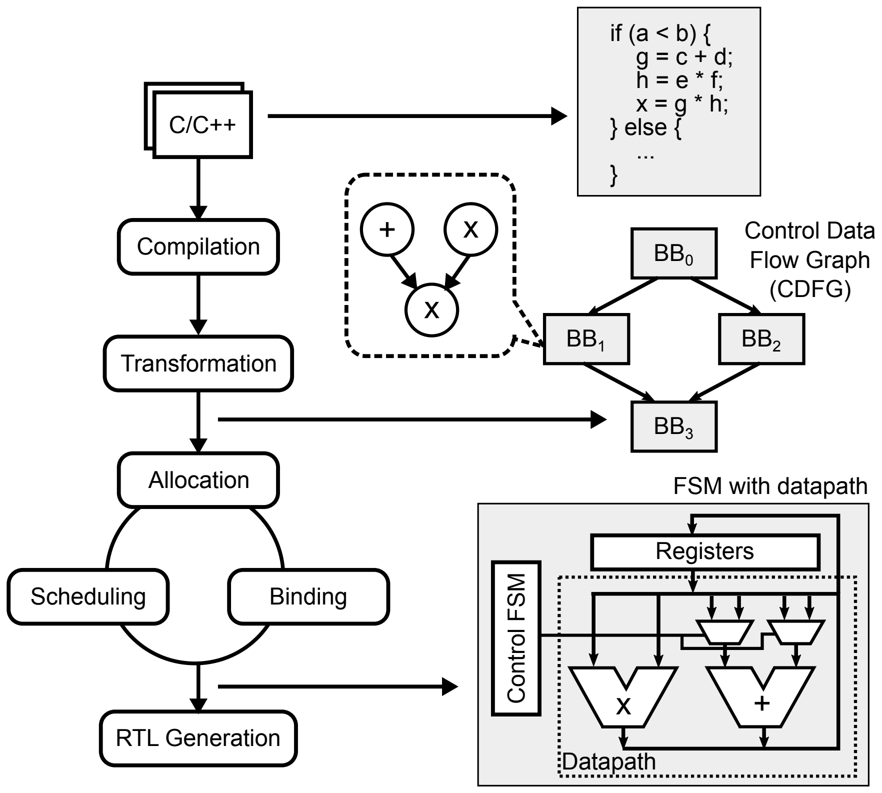


Figure 2.1: System architecture.

2.2.2 High-Level Synthesis

In this work, we target accelerators that are generated using High-Level Synthesis (HLS). Figure 2.2 shows a typical HLS flow that automatically transforms a functional description of the accelerator written in a high-level language such as C or C++ into a register-transfer level (RTL) description. To achieve this, HLS tools first transform source code into control data flow graphs (CDFG), and then perform allocation, scheduling, and binding to generate the final RTL. HLS tools usually pipeline the computation in order to achieve high performance. The pipeline is generated using a static schedule, where each operation is placed in a fixed slot determined at compile time. This approach works well if all functional units and memory operations have a short fixed latency. For operations with an uncertain latency, the HLS tool has to use a best guess for scheduling. For example, cache accesses are usually assumed to be a hit in order to generate a compact pipeline schedule. Then, the pipeline is stalled at run-time if an access turns out to be a cache miss.



```
if (a < b) {
    g = c + d;
    h = e * f;
    x = g * h;
} else {
    ...
}
```

Figure 2.2: High-level synthesis flow.

2.2.3 Impact of Memory Accesses on Accelerator Performance

We use an example to illustrate how a long memory access latency on a cache miss can impact accelerator performance. The code in Figure 2.3 shows the inner loop of a sparse matrix vector multiplication (SpMV) accelerator. Note that the access to the `vec` array is an indirect memory access that has an irregular access pattern, and is likely to miss in the cache.

An example pipeline schedule for the corresponding accelerator is shown in Figure 2.4. The pipeline has an initiation interval (II) of one, that is, a new iteration can begin execution every clock cycle in the ideal case, as illustrated in Figure 2.4(a). The three load operations in each iteration are to `val`, `cols`, and

```
    for (j = begin; j < end; j++) {
        #pragma HLS pipeline
        Si = val[j] * vec[cols[j]];
        sum = sum + Si;
    }
```

Figure 2.3: The inner loop of sparse matrix vector multiplication.

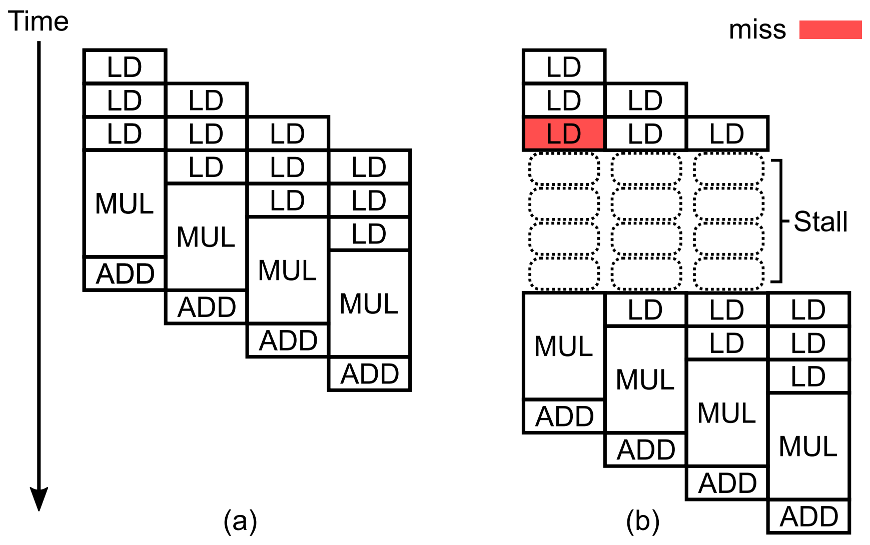


Figure 2.4: Example schedule of an HLS SpMV accelerator with (a) ideal memory (b) cache that has a miss when accessing vec.

vec, respectively. Figure 2.4(b) shows an actual pipeline operation when accessing vec in the first iteration incurs a cache miss. Since the schedule is static, the entire pipeline has to stall until the miss is resolved, even though the memory accesses of later iterations might have been hits. The stall due to a long memory access latency can have a large impact on the accelerator's performance. For example, in Figure 2.4(b), although only one out of four iterations has a cache miss, the effective initiation interval for the four iterations is increased from one to two, essentially lowering the throughput by half. The impact can be even larger for accelerators with deeper pipelines where one cache miss can potentially stall many more operations than what is shown in the example.

Our experimental results on a set of HLS-based hardware accelerators suggest that the performance loss due to long memory accesses is significant. There exists a large performance gap between accelerators with ideal memory (1-cycle) and a realistic cache-based memory hierarchy. This work aims to bridge this gap by developing techniques to automatically preload data for accelerators. An ideal preloading scheme would effectively eliminate cache misses, and allow the pipeline to run at the full throughput possible with the ideal memory.

2.2.4 Data Preloading Framework

There are a few challenges in developing a data preloading scheme to enable efficient data supply for accelerators. First, the scheme needs to accurately predict future data needs of an accelerator so that data can be preloaded. Second, the prediction needs to be early enough to hide memory latency. Third, the prediction and memory accesses need to be decoupled from computation so that accesses and computation can happen in parallel. Fourth, all the above need to be performed automatically with minimal manual efforts.

In this work, we use two data preloading techniques to hide long memory accesses: (1) prefetching and (2) access/execute decoupling. These two techniques have complementary characteristics, and can both be applied with minimal manual efforts.

Hardware prefetchers predict likely memory addresses to be accessed in the future by observing a sequence of memory accesses at run-time. For example, a stride prefetcher is widely used to detect and preload streaming memory accesses with a fixed stride. In our example, simple strided accesses such as `val[j]` and `cols[j]` can easily be detected and preloaded by a hardware prefetch engine. Moreover, the prefetch engine is inherently decoupled from accelerators and can perform multiple prefetching operations in parallel. On the other hand, on-line prefetching often cannot accurately predict complex memory access patterns such as the indirect accesses (`vec[cols[j]]`) in our example.

For difficult-to-predict memory accesses, we use decoupled access/execute (DAE) architecture. In this approach, we observe that program slicing techniques can be used to automatically separate parts that are necessary to compute addresses for memory accesses (*access part*) from the rest that performs main computations (*execute part*). Then, the access part can run ahead of the execute part to preload data. In a sense, the DAE approach provides a perfectly accurate predictor for future memory accesses. However, decoupling and providing early predictions can be more difficult in the DAE architecture compared to prefetching. In DAE, address generations must be exact (binding) unlike prefetching whose predictions may be incorrect (non-binding). Also, in certain cases, it may be difficult to decouple the access and execute parts due to dependencies. Table 2.1 summarizes the characteristics of prefetching and DAE in terms exactness in address generation, accuracy, and timeliness.

Our experiments show that prefetching and DAE can complement each other. DAE enables accurate preloading of memory addresses when possible. Prefetching provides speculative preloading for simple access patterns when DAE cannot generate exact addresses early enough.

Table 2.1: Comparison of prefetching and DAE.

	Binding	**Accuracy**	**Timeliness**
Prefetch	No	Good when regular	Good
DAE	Yes	Good	Depends

2.3 Prefetching

As we mentioned in the previous section, hardware prefetchers observe the memory address stream and predict the addresses that are likely to be referenced in the future. In most cases, just looking at a global address stream is not enough to make good predictions, as the global stream is usually a mixture of multiple data streams with different strides as well as irregular accesses, making it difficult to learn the access pattern and make predictions. Thus, most hardware prefetchers perform stream localization to separate a global address stream into multiple local address streams that can be learned and predicted effectively, and to exclude irregular accesses with poor predictability. Since most hardware prefetchers are designed for general-purpose processing cores, they often use the program counter (PC) of load and store instructions as a hint for stream localization [13, 75], with the intuition that different streams come from different instructions in the program. In addition, the PC is also used for other purposes such as spatial correlation prediction [103] to improve the accuracy of prefetching. Hardware accelerators, on the other hand, usually do not have a PC. Thus, traditional hardware prefetchers that rely on a PC would not be effective when used naively with hardware accelerators.

We observe that for hardware prefetchers, the fundamental role of a PC is to

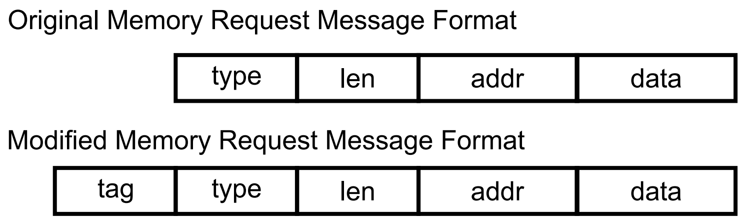

Original Memory Request Message Format
type
len
addr
data
Modified Memory Request Message Format
tag
type
len
addr
data

Figure 2.5: Modified memory request message format.

indicate which memory instruction in a program a memory access comes from. If we replace the PC with a unique identifier for each memory instruction, the prefetcher would work equally well as the identifier provides the same amount of information for stream localization. Thus, we propose to tag each memory access operation in a hardware accelerator with a unique identifier that is sent to a prefetch engine in place of the PC for each memory access. In our implementation, we modified the memory request message format of the accelerators to include a tag field, as shown in Figure 2.5. To generate the tags, we add an extra pass to the HLS compiler frontend, which traverses all basic blocks in the code, and tags each memory operation with a unique identifier that emulates a PC. The pseudo-code of the pass is shown in Algorithm 1. Using the tag, features such as PC-based stream localization would work correctly, and the hardware prefetcher is able to effectively prefetch memory addresses of hardware accelerators.

2.4 Decoupled Access/Execute

While hardware prefetchers are effective in prefetching regular memory accesses, they work less well for complex access patterns or short streams that do

Algorithm 1: Generate tags for memory accesses

```
1: procedure GenerateTags
2:     $t \leftarrow 0$
3:     for all basic blocks do
4:         for all operations in the basic block do
5:             if $op.type = load$ or $op.type = store$ then
6:                 $op.tag \leftarrow t$
7:                 $t \leftarrow t + 4$
8:             end if
9:         end for
10:    end for
11: end procedure
```

not trigger hardware prefetching. The fundamental limit of hardware prefetchers is the lack of semantic information about the computation. Previous studies have proposed various techniques to employ semantic information to enable more accurate prefetching for software programs. For example, software prefetching [72] allows programmers or compilers to embed prefetch instructions into the code, which provide hints to the hardware about the addresses to be accessed in the future. Helper thread [35] and runahead execution [73] pre-execute a part of the program or a specially crafted program slice to bring data into the cache. All these techniques rely on the assumption that memory addresses can often be computed well ahead of when the data are needed. Decoupled access/execute (DAE) [101] materializes this assumption to a full extent by allowing the memory access part, where memory addresses are computed and data accesses are performed, to run ahead of the execute part, where data are

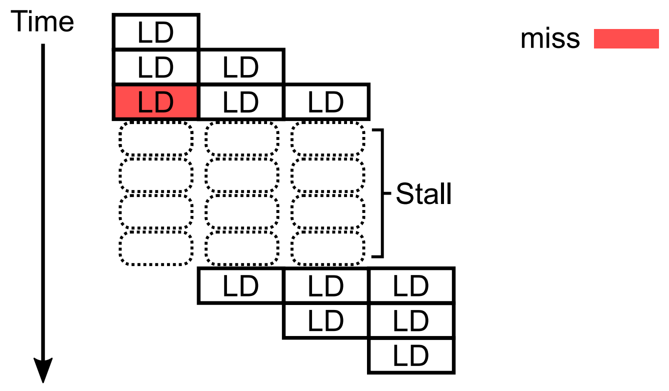


Figure 2.6: Example schedule of the access part of a decoupled SpMV accelerator when there is a cache miss.

consumed. In a typical access/execute decoupled architecture, the access part manages all communications with the memory and supplies data to the execute part; the execute part does not have a memory interface.

A key requirement for achieving performance improvements with DAE is that the access part in the decoupled architecture must run faster than the non-decoupled architecture, otherwise the performance is limited by the access part. However, in highly pipelined accelerators, this is unlikely true. Figure 2.6 shows an example schedule of the access part of a decoupled SpMV accelerator where the same miss occurs as in the non-decoupled version shown in Figure 2.4. The miss has the same performance impact on the access part as in the non-decoupled version. Thus, simply dividing the accelerator pipeline into access and execute parts is unlikely to improve performance significantly when the access part has the same rigid pipeline that cannot tolerate memory latencies. Allowing the access pipeline to tolerate cache misses is a key challenge in designing the DAE accelerator architecture.

Figure 2.7 shows the architecture of the proposed access/execute decoupled

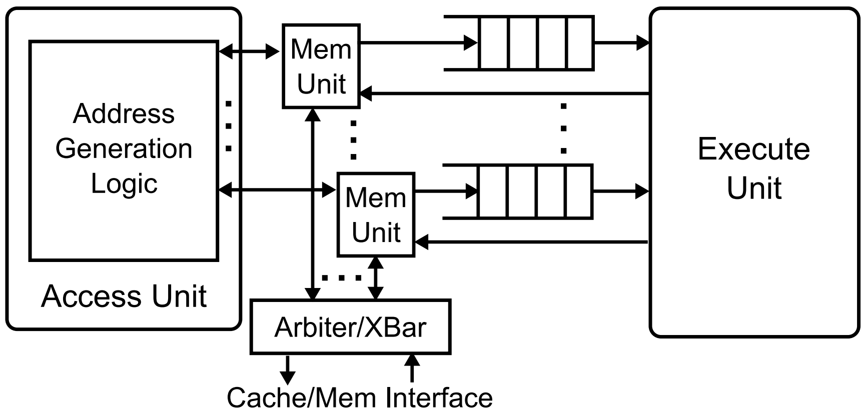


Figure 2.7: Architecture of access/execute decoupled accelerators.

accelerator, consisting of the Access Unit, Execute Unit, Memory Units, and decoupling queues. A visible difference from classic access/execute decoupled architectures is the added memory units, which is a proxy through which memory accesses are performed. Later we will show that this is necessary for tolerating the memory latency. The access unit generates memory addresses and request types, and then sends them to the memory unit to be forwarded to memory. For load operations, once responses come back, the memory unit enqueues the data into the Load Queue (LQ) to be read by the execute unit. For store operations, the memory unit combines the address from the access unit and data from the execute unit, and then sends the request to memory. An access/execute decoupled accelerator can have multiple memory units, which share the cache interface.

2.4.1 Access Unit

In a simple DAE accelerator implementation, the access unit is responsible for address generation, handling memory requests/responses, and forwarding data to the execute unit, all in a single static schedule generated by the HLS tool. Among these tasks, address generation and sending out memory requests usually have a fixed latency and thus would work well under the static schedule. Handling memory responses and forwarding data, however, have variable latencies depending on when the response comes back from memory. This has two implications. First, they cannot be executed efficiently under the static schedule generated by HLS. Second, they may stall address generation and sending out requests for other independent accesses. To address this problem, we propose to decouple memory response handling and data forwarding from address generation and sending out requests. Specifically, we delegate these tasks to the memory unit, which handles them independently, decoupled from the access unit.

The result of a load operation can either be used by the execute unit for data computation or by the access unit for address computation. In the first case, the access unit is not involved in handling the load result. This type of load operations are called terminal loads [49]. In the second case, however, the access unit would need to wait for the load result and thus its pipeline could be stalled if the load is a miss. One way to enable the access unit to continue to perform independent operations is to employ an out-of-order core as the access unit, or use dataflow execution for memory accesses [51]. Though these approaches can achieve higher performance, we choose not to employ them because we observe that the load dependency chains in many accelerators are short. In fact, a large

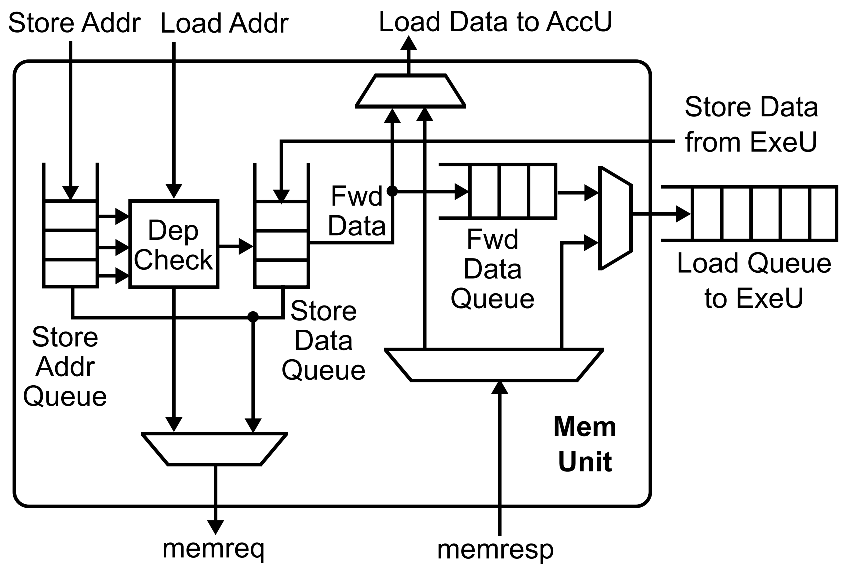


Figure 2.8: Hardware structure of the memory unit.

portion of load operations are terminal loads. This is because many accelerators mostly perform parallel operations, instead of serial operations through memory such as pointer chasing. Decoupling just terminal loads, i.e. the last node of a load dependency chain, provides most of the benefits with a low cost. Hence, our architecture would work reasonably well for short memory dependency chains, and we trade off the ability to handle long chains for low hardware complexity.

2.4.2 Memory Units

Figure 2.8 shows the hardware structure of the memory unit. It mainly consists of load queue, store queue, forward data queue, dependency checking logic, and memory request/response routing logic.

The Store Address Queue (SAQ) contains store addresses that are not yet sent to memory, either because the store data have not been computed yet, or because it is waiting for access to the memory interface. The Store Data Queue (SDQ) buffers store data from the execute unit. The head entries of SAQ and SDQ are paired to form a store request to be sent to memory.

Each load request from the access unit contains a `dest` field indicating whether the result is used by the access unit or the execute unit. The field is kept in the corresponding response to the request. When the memory unit receives a load request, it checks whether there is an entry in the SAQ matching the store address. If there is a match, the load waits until the corresponding entry in the SDQ is valid, and data is forwarded from the SDQ to either the Forward Data Queue (FwdQ) or the access unit, depending on whether the load operation is a terminal load or not. If there is no match, the load request is sent to the memory. The load and store requests share the memory port. Load requests are given priority over stores to reduce load latency.

The Load Queue (LQ) contains data to be forwarded to the execute unit. When a load response returns from memory, its `dest` field is inspected to route the response data to the LQ, the access unit, or both. Because the execute unit consumes data from memory in a program order, the LQ entries are reserved and maintained in the request order. For example, responses from the memory and the Forward Data Queue are placed in the LQ in the program order. The memory unit supports multiple in-flight requests. If the cache returns responses out-of-order, the LQ is used to reorder and return them in order.

2.4.3 Execute Unit

The execute unit is generated using HLS from the execute slice, and mainly consists of the data computation pipeline.

2.4.4 Deadlock Avoidance

There exist two possible deadlock situations in the proposed access/execute decoupled architecture. Here we describe them and discuss how to prevent them.

Pipelining-Induced Deadlocks: A deadlock may occur when accelerator pipeline interacts with a store queue of insufficient size. Suppose the execute unit pipeline has latency L and initiation interval II, it needs to consume $N = \lceil L/II \rceil$ inputs before producing the first output. If the store queue size is less than N, it may fill up and block the access pipeline. Because the execute pipeline depends on the access pipeline for data supply, it also blocks and the accelerator deadlocks. The deadlock occurs because pipelines generated by most HLS tools do not support flushing by default. That is, a blocking operation stalls the entire pipeline, not just subsequent iterations. This restriction enables HLS tools to generate simple pipelines without complex control logic and buffering, but causes deadlocks in this situation.

Pipeline synthesis techniques that support flushing [37] can be used to avoid this deadlock. If the HLS tool does not support flushing, another approach is to ensure that size of the SAQ is larger than $N = \lceil L/II \rceil$, so that it would not become full before the execute pipeline produces the first output. Often the SAQ size required for performance reasons is already greater than N, then no additional changes are needed in this case.

Deadlock Due to Full Load/Store Queues: A deadlock can occur when the queues are full and form a circular dependency. For example, a load response returns from memory when the load queue and store queue are both full, and the memory system cannot accept another request because it has reached the maximum number of in-flight requests. In this situation, the load queue cannot be drained because the execute unit is stalled trying to write to the full store queue, which is waiting for the memory system, which in turn is waiting for the load queue. This creates a circular dependency, causing a deadlock. This deadlock can be avoided by ensuring that not all queues can become full at the same time. Specifically, we track the number of in-flight load operations and free entries in the load queue, and delay issuing a load if the response would cause the load queue to become full.

2.4.5 Customization of Memory Units

The memory unit design described in Section 2.4.2 can be customized to fit the needs of a particular accelerator, providing just enough resources and features but not more. The sizes of various queue structures can be adjusted to fit the accelerator's memory characteristics. For example, if the accelerator rarely performs stores, sizing down the store queue would help save area and energy. If a certain feature is unused by an accelerator or does not help too much, it can be removed. For example, if a memory port is read-only, the memory unit can be made much simpler by removing any store-related features such as store queue and dependency checking logic. As another example, store to load forwarding

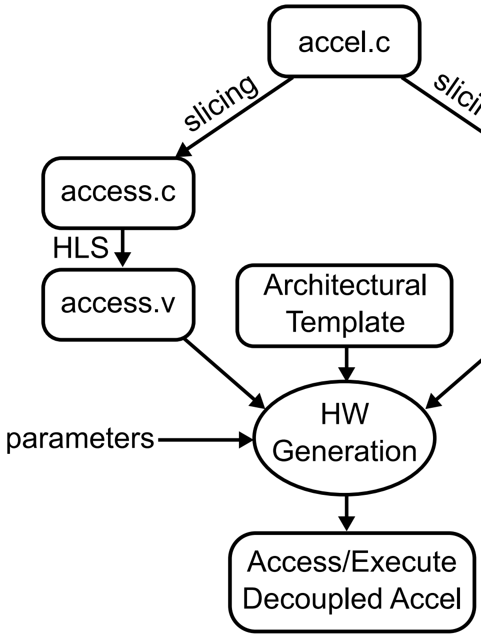


Figure 2.9: High-level flow of decoupled access/execute accelerator generation.

can be removed if the accelerator does not need it.

2.4.6 Automated DAE Accelerator Generation

Figure 2.9 shows the high-level flow used by the framework for automatically generating accelerators with access/execute decoupling. Starting from a single source code written in a high-level language, the framework uses program slicing [110], which is built in modern optimizing compilers, to generate access and execute slices. To generate the access slice, the program slicing algorithm backtracks from loads and stores in the Control Data Flow Graph (CDFG) of an accelerator and keeps all necessary operations for computing memory addresses, while removing others. To generate the execute slice, the algorithm backtracks from stores and finds all operations needed to compute store values. The slicing process also performs transformations to enable decoupling. In the access slice, stores are transformed into `store_addr` operations, which only have address but not data. Terminal loads are identified as loads that are not in the back slice of address calculations, and are transformed into `load_fwd` operations which indicate that the result should be consumed by the execute slice. In the execute slice, all loads and stores are replaced with queue reads and writes. The framework then synthesizes the resulting access and execute slices into Verilog RTL using HLS.

The framework implements the decoupled accelerator architecture shown Figure 2.7 in as an architectural template written in a hardware generation language PyMTL [69]. The template includes RTL implementations of the components such as memory unit, queue structures, arbiters, and memory crossbars. The architectural template is designed to be fully configurable to allow customization of the modules as described in Section 2.4.5. For example, the sizes of the load and store queues, as well the architecture features can be configured.

To generate the final RTL of the accelerator, the framework elaborates the template with the parameters specified by the designer, and combines it with the access and execute slices to output the RTL of the access/execute decoupled accelerator.

2.5 Evaluation

In this section, we present the evaluation results for the proposed data supply framework for accelerators. We first discuss our evaluation methodology and experimental setup. Then, we show the performance, area, and energy results.

2.5.1 Methodology

We use an integrated evaluation methodology that combines cycle-level, register-transfer-level, and gate-level modeling.

Cycle-level modeling is used to model the performance of the system components including caches, interconnect, memory controller, and main memory. We use gem5 [15] for this purpose.

Register-transfer-level modeling is used to accurately model the performance of hardware accelerators. Vivado HLS 2015.2 is used to synthesize a C-based description of the accelerators into Verilog. For DAE accelerators, RTL of the memory unit and queue structures are generated from the architectural template. Verilator [3] is used for RTL simulation. We integrated Verilator with gem5 for co-simulation of accelerators and system components.

Gate-level modeling is used to build accurate area and energy models for the accelerators. We synthesized, placed and routed each accelerator using Synopsys Design Compiler and IC Compiler with the TSMC 65nm standard cell library to obtain area numbers. Design Compiler automatically inserts clock gating logic for all designs. Power and energy analysis were performed using Synopsys PrimeTime PX with the switch activity factors obtained from simulations of the place and routed netlist.

Table 2.2: Summary of benchmarks.

Benchmark	Description
bbgemm	Blocked matrix multiplication
bfsbulk	Breadth-first search
gemm	Dense matrix multiplication
mdknn	Molecular dynamics (K-nearest neighbor)
nw	Needleman-Wunsch algorithm
spmvcrs	Sparse matrix vector multiplication
stencil2d	2D stencil computation
viterbi	Viterbi algorithm

2.5.2 Experimental Setup

We use a set of eight benchmark accelerators adapted from MachSuite [92] in our experiments. Table 2.2 summarizes the accelerators. For each benchmark, we use the framework to generate the Verilog RTL of the accelerator with DAE and prefetching from a C++ source code. Each accelerator has a private L1 cache connected to the DRAM controller through the system bus. For prefetching, we use a stride hardware prefetcher. Table 2.3 compares the lines of code between the C++ source code and the generated Verilog RTL. The results demonstrate that the framework is able to generate high-quality accelerators from a small amount of high-level code.

Table 2.4 shows the detailed experiment parameters. We compare the following schemes:

1. **Baseline** is the original accelerator without prefetching or DAE.

Table 2.3: Lines of code (LOC) comparison between the input to the framework (C++ source code) and the generated Verilog code. Blank lines and comments are not counted.

Benchmark	LOC (C++)	LOC (Verilog)	Ratio
bbgemm	74	6,494	87.8
bfsbulk	74	3,148	42.5
gemm	75	6,246	83.3
mdknn	96	7,601	79.2
nw	133	9,080	68.3
spmvcrs	89	6,640	74.6
stencil2d	81	7,095	87.6
viterbi	89	7,053	79.2

Table 2.4: Experiment parameters.

Technology	65nm
Frequency	500MHz
DAE MemUnit	16-entry LQ, 8-entry SQ
Cache	16KB / 2-way / 32B line size / 1 cycle latency / 4 MSHRs
Prefecher	Stride prefetcher, degree=8
DRAM	Single-channel 32-bit LPDDR3-1600, 6.4GB/s BW

2. **Stride** has the stride prefetcher enabled but not DAE. The memory accesses are tagged to facilitate prefetching.

3. **DAE** is the access/execute decoupled implementation, but without the stride prefetcher.

4. **DAE+stride** adds stride prefetching to DAE.

2.5.3 Baseline Validation

HLS-based accelerators have a large design space. Depending on the parameters used, the same accelerator can be synthesized to have different area, performance, and power. We use the same set of parameters when synthesizing the baseline and DAE versions to exclude the possibility that the improvement comes from different synthesis parameters. To ensure that the improvement is not from poorly optimized baseline, we apply most HLS optimizations including pipelining and unrolling so that baseline accelerators have best performance within the system-level constraints (such as the number of memory ports or memory bandwidth).

To validate the performance of the baseline, we simulated the performance of functionally equivalent software implementations of these accelerators. Figure 2.10 shows the performance comparison between in-order, 4-wide out-of-order processors, and the baseline accelerators. Note that `mdknn` and `viterbi` are not included because we use custom-precision fixed-point arithmetic in their implementations, which would be inefficient to emulate in software on processors. On average, the performance of the baseline accelerators is about 2x of an in-order processor, and is comparable to an out-of-order processor. These numbers are roughly in line with previous studies [43, 108]. The performance of the baseline accelerators is mainly limited by the memory bottleneck. We will show that with the proposed techniques to enable efficient data supply, the accelerators could achieve much higher performance.

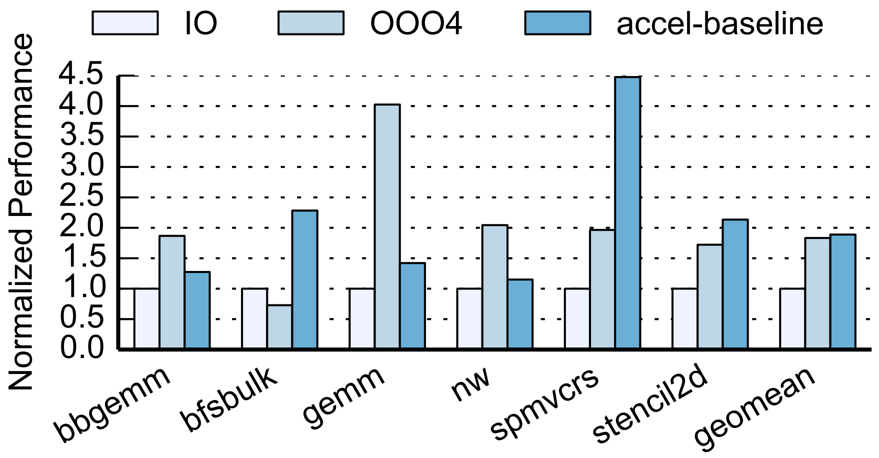


Figure 2.10: Comparison of baseline accelerators performance with processors.

2.5.4 Performance Results

Figure 2.11 compares the performance of the proposed optimization schemes normalized to the baseline accelerator. Overall, the stride prefetcher with memory access tags improves the performance by 45% on average over the baseline. DAE alone achieves an average speedup of 1.89x, while DAE combined with stride prefetching achieves a 2.28x speedup.

Comparing stride prefetching and DAE, DAE usually achieves higher performance due to decoupling and having more precise knowledge about the addresses to be accessed next. One such case is when the access pattern is irregular, but the addresses can be computed early. For example, `mdknn` computes the force between a molecule and its N nearest neighbors. The access pattern is highly irregular because the addresses of the N neighbors in memory usually do not have a pattern. However, the addresses can be computed early because the indices of these N neighbors are known. Hence, the access unit can send out

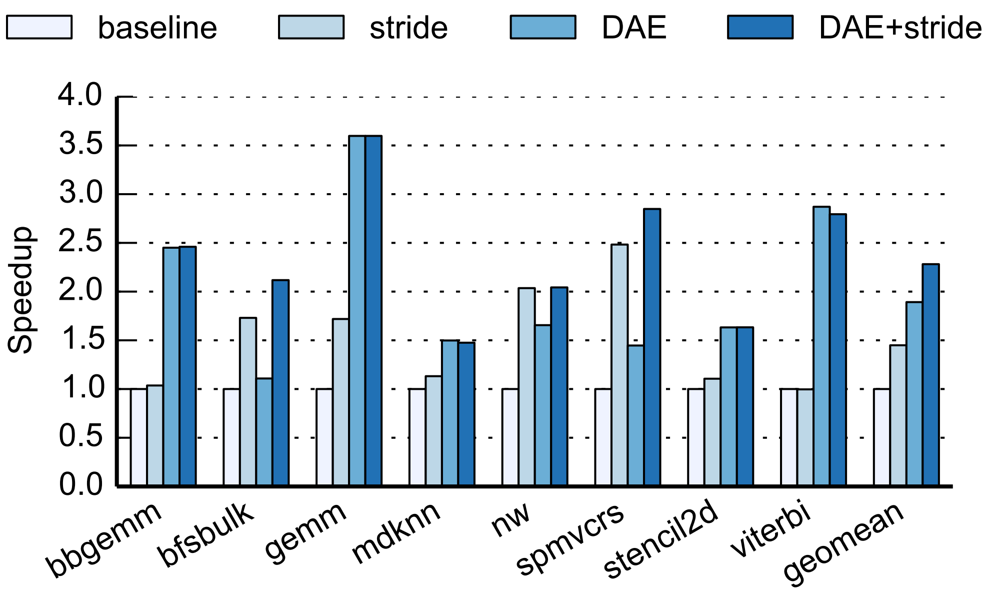


Figure 2.11: Performance of the proposed schemes normalized to the baseline accelerator.

load requests early to hide memory latency. In contrast, the stride prefetcher is unable to predict the addresses and thus unable to prefetch them.

DAE also has advantages when the accesses consist of regular but short streams. One example is `viterbi`. The stride prefetcher needs warm-up, thus is too late in sending out prefetch requests. It also prefetches beyond the end of the stream before realizing the stream has ended, wasting memory bandwidth and causing cache pollution. In contrast, DAE has precise information about when the stream begins and ends, thus is able to preload data effectively.

There are some cases where prefetching is more effective than DAE. For example, `bfsbulk` performs a graph traversal, which is dominated by memory accesses with dependencies. As a result, the access unit in the decoupled architecture is not able to pre-calculate the addresses. Prefetching, on the other hand, speculatively fetch data from memory without computing the exact addresses, which improves performance in this case because there is regularity in the access pattern even though the addresses cannot be determined early.

It is also clear from the results that prefetching and DAE can often complement each other, providing a higher speedup compared to using only one of them. For example, in the inner loop of `spmvcrs` (code shown in Section 2.2.3), DAE is able to hide the memory latency for accesses to `val` and `vec`, but not `cols`. Because `cols` is used by the access unit to calculate the address to `vec`, a cache miss for `cols` stalls the access unit. However, the prefetcher can easily detect the strided accesses to `cols` and prefetch it into the cache. As a result, we observe the combined scheme with both DAE and stride prefetching achieves the speedup of 2.85x, which is higher than the speedups, 1.45x and 2.48x respectively, when DAE and prefetching are applied separately.

We note that adding prefetching to DAE does not always yield higher performance. For example, we see a slight degradation in performance for `viterbi` when prefetching is enabled. This is because DAE alone can already hide most of the memory latency, while prefetching, due to its imprecise nature, can pollute the cache and contend for memory bandwidth.

2.5.5 Area, Power, and Energy Results

Table 2.5 shows the area and power numbers for the baseline and DAE accelerators. The area of DAE accelerators is larger than the baseline by 14% (`mdknn`) to 129% (`spmvcrs`). We note that our area and power analysis only includes the accelerator itself but not the cache, which includes a prefetch unit. The relative

Table 2.5: Area and power of the baseline and DAE accelerators. The Abs column shows absolute numbers, and the Norm column shows results normalized to the baseline.

Benchmark	Base Area (μm^2)	Base Power (mW)	DAE Area (μm^2)		DAE Power (mW)	
			Abs	Norm	Abs	Norm
bbgemm	25,191	4.15	52,943	2.10x	8.11	1.96x
bfsbulk	11,507	1.22	14,437	1.25x	1.41	1.16x
gemm	22,127	1.87	47,305	2.14x	3.39	1.81x
mdknn	170,312	32.58	194,034	1.14x	48.40	1.49x
nw	49,094	4.54	89,396	1.82x	8.81	1.94x
spmvcrs	18,686	2.54	42,736	2.29x	4.04	1.59x
stencil2d	27,579	3.88	49,567	1.80x	7.69	1.98x
viterbi	42,963	4.78	80,982	1.88x	11.30	2.36x

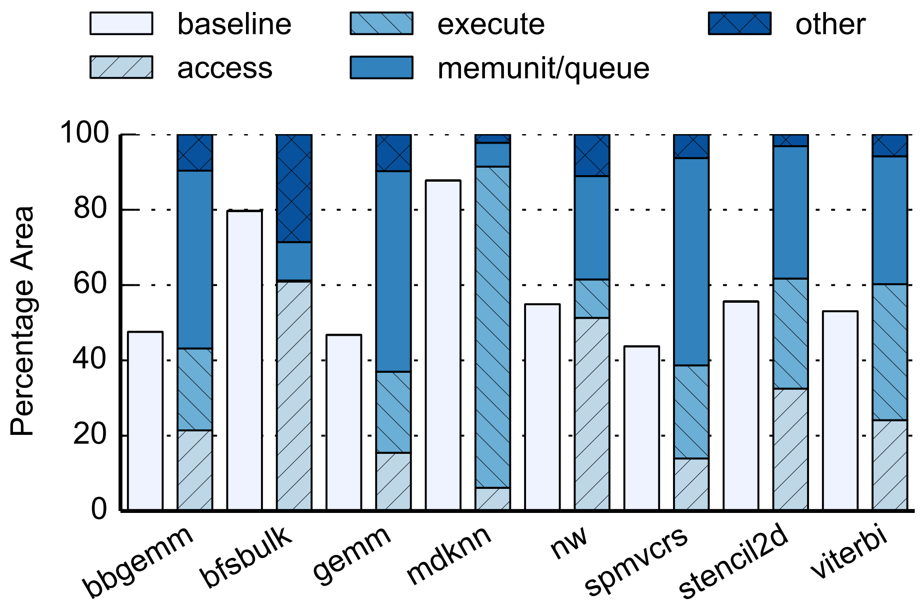


Figure 2.12: Area breakdown of DAE accelerators. The baseline area is shown for comparison.

overhead will be much lower when the cache, which exists in both the baseline and our architecture, is included.

The area increase comes from several factors: First, the DAE architecture adds additional queues and memory units to accelerators. The area for the queues and memory units are similar across accelerators given that we used the same queue size for all benchmarks. As a result, this overhead represents a large relative overhead for small accelerators such as `spmvcrs`. For larger accelerators such as `mdknn`, the area overhead for queues and memory units only represents a small percentage. Later, we show that this overhead can be reduced significantly by customizing the size of queues for each accelerator.

Second, area overhead can come from the reduced resource sharing between the access part and the execute part in the DAE architecture. During the synthesis process, the HLS tool tries to share resources between various parts of the accelerator to reduce area. In the baseline accelerators, such optimizations can be performed across the entire accelerator. For example, a multiplier may be shared between memory address computation logic and value computation logic. In the DAE architecture, such sharing is not possible between the access and execute units because they need to be decoupled and synthesized separately. We note that while reduced resource sharing increases area, it also allows more operations to be performed in parallel and improve performance. The impact of reduced resource sharing is lower for larger accelerators where there are abundant opportunities for resource sharing within the access unit or execute unit.

Figure 2.12 shows the breakdown of area of the DAE accelerators compared to the baseline. The area is broken down into access unit, execute unit, memory

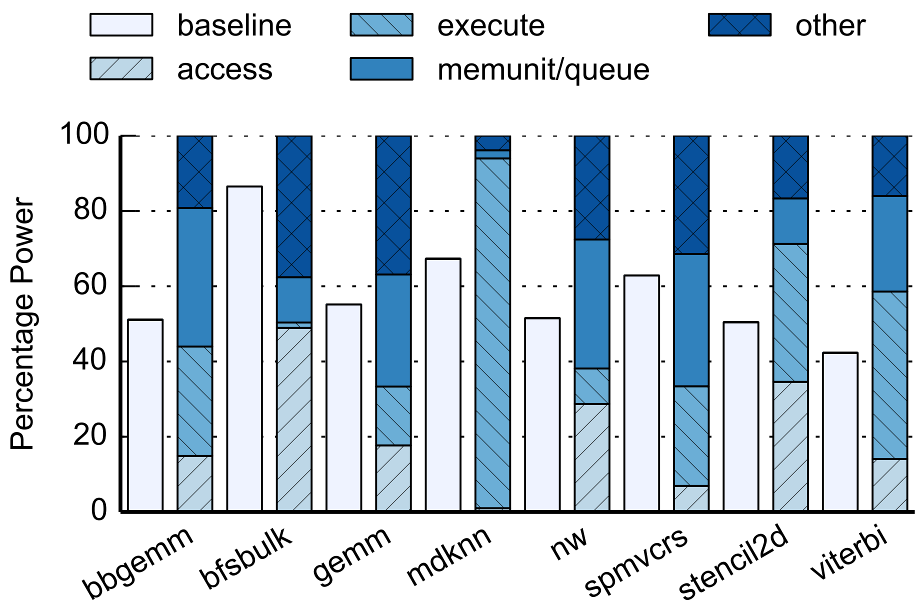


Figure 2.13: Power breakdown of DAE accelerators. The baseline power is shown for comparison.

units (including queues), and other components such as configuration registers, miscellaneous control logic, buffers inserted during place and route, etc. The results indicate that the main area overhead comes from the memory units and queues. The combined area of the access unit, the execute unit and other components, which have corresponding logic in the baseline accelerator, is only 13% higher than the area of the baseline on average, indicating the impact of reduced resource sharing is low.

Figure 2.13 shows the breakdown of power consumption. The percentage of power consumed by memory units and queues ranges from a few percent to around 35%. For `mdknn`, which is relatively large, the power consumption of memory units and queues is only 2.2% of the total power consumption. In addition to the added operations for memory units and queues, the DAE archi-

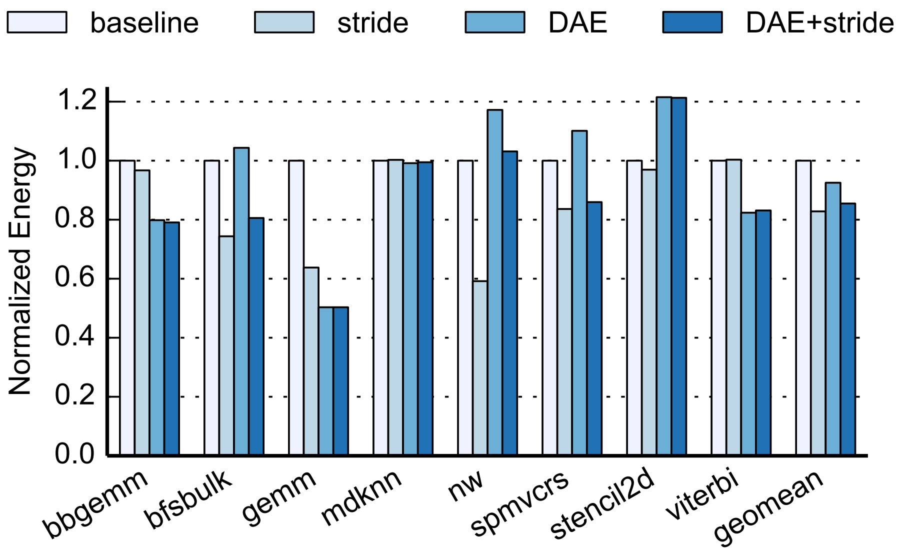


Figure 2.14: Energy comparison.

tecture also has higher power consumption compared to the baseline because it has higher activity factors. DAE reduces pipeline stalls waiting for memory accesses and allows accelerators to perform more computations per unit time.

Figure 2.14 shows the energy consumed by the baseline and DAE accelerators to complete the computation. On average, the stride prefetcher is able to reduce energy for both the baseline and DAE accelerators, by 17.1% and 7.6% respectively. This is because the stride prefetcher is able to reduce memory stalls, leading to shorter execution time. As a result, the accelerators spend less time burning energy without doing useful work.

Compared to the baseline, DAE accelerators often use less or a comparable amount of energy even though they have significantly higher power, which indicates that the higher power mostly comes from doing more useful work per unit time because of reduced pipeline stalls. In cases where DAE accelerators

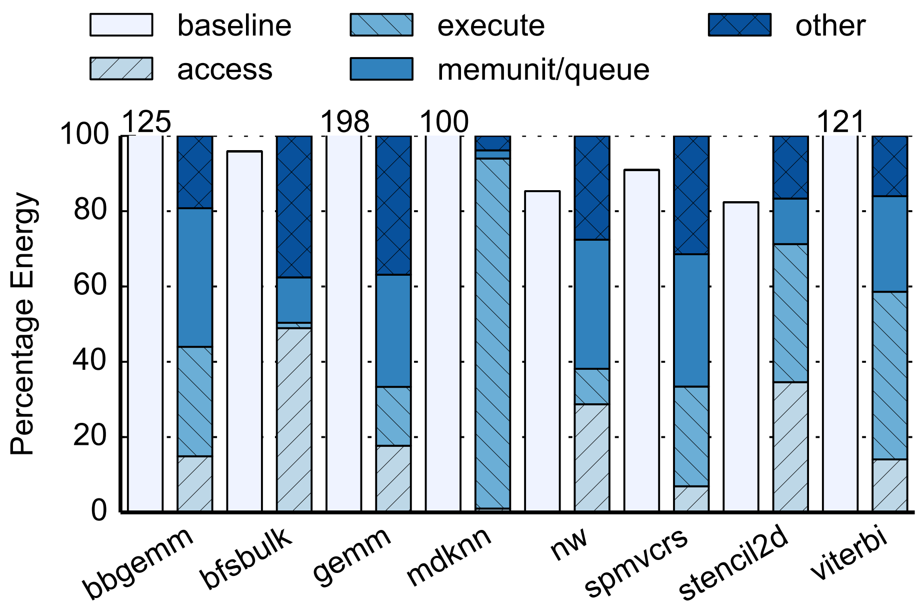


Figure 2.15: Energy breakdown of DAE accelerators. The baseline energy is shown for comparison. If baseline energy is higher than DAE, it is annotated with a number on top of the bar.

use less energy, it is likely to be because the energy spent while stalling is significant in the baseline. While our design flow automatically inserts clock gating, it does not completely remove static power consumption. Most of these stalls are removed in the DAE accelerators, resulting in lower energy.

Figure 2.15 shows the breakdown of the energy consumption. The energy numbers are for the baseline and the DAE architecture without prefetching. The percentage of energy consumed by the memory units and queues ranges from 2.2% to 36.8%. Again, the relative overhead is lower for large accelerators compared to small accelerators.

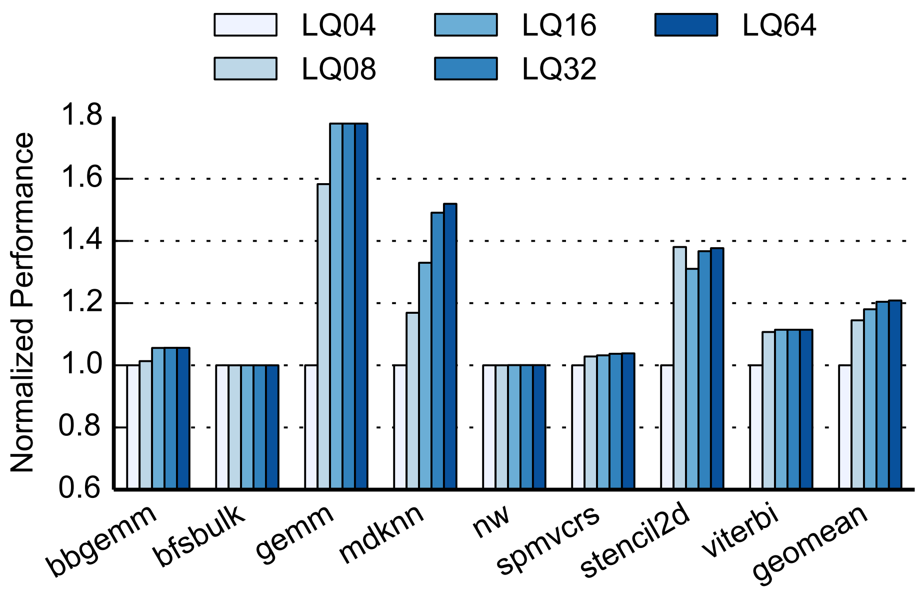


Figure 2.16: Performance comparison when varying load queue (LQ) sizes.

2.5.6 Design Space Exploration: Queue Size

The size of queue structures in the decoupled accelerator can impact the performance, area, and energy consumption of the accelerator. Larger queues can provide more decoupling, thus potentially better performance, but may also have a larger area and consume more energy. Figure 2.16 shows the normalized performance of the accelerators when varying load queue sizes from 4 entries to 64 entries. On average, larger load queues yield higher performance, but the improvement diminishes as the queue size increases. This indicates that as long as the access unit runs sufficiently ahead of the execute unit, it can provide the decoupling needed to hide the latency of occasional cache misses.

The results also indicate that some accelerators are less sensitive to queue sizes than others. Thus, accelerator-specific optimization of the queue sizes can

Table 2.6: Impact of the queue size customization on area and performance. Numbers are normalized to LQ16/SQ08.

Benchmark	**Custom Size**	**Total Area (μm^2) Abs**	**Total Area (μm^2) Norm**	**Queue Area (μm^2) Abs**	**Queue Area (μm^2) Norm**	**Norm Perf**
bbgemm	LQ04/SQ08	40,442	0.76	12,504	0.50	0.94
bfsbulk	LQ02/SQ08	14,437	1.00	1,473	1.00	1.00
gemm	LQ08/SQ02	33,535	0.71	11,423	0.45	0.89
mdknn	LQ16/SQ04	191,183	0.99	9,441	0.77	1.00
nw	LQ04/SQ08	82,449	0.92	17,594	0.72	0.97
spmvcrs	LQ08/SQ02	29,788	0.70	10,582	0.45	0.97
stencil2d	LQ08/SQ02	39,372	0.79	7,232	0.41	1.05
viterbi	LQ08/SQ08	72,661	0.90	19,183	0.70	0.99

be used to reduce the area overhead of decoupled accelerators, with minimal degradation in performance.

Table 2.6 shows the impact of customizing queue sizes on the area and performance of DAE accelerators. For each accelerator, we choose a queue size that has lower area but not significantly lower performance, and normalize the area and performance to the configuration with constant 16-entry load queues and 8-entry store queues across all accelerators. On average, the customization reduces queue area and total area by 37% and 15% respectively, while lowering performance by only 2.3%.

CHAPTER 3

PARALLEL ACCELERATOR FRAMEWORK

3.1 Introduction

This chapter proposes an architectural framework for generating parallel accelerators from high-level descriptions of parallel algorithms. Leveraging concepts from task-based parallel programming, the framework enables software programmers to quickly create high-performance accelerators using familiar parallel programming paradigms, without needing to know low-level hardware design knowledge. The framework uses a computation model that supports dynamic parallelism, and includes a flexible architecture that supports dynamic scheduling to enable mapping a wide range of parallel applications and achieve good performance. In addition, the framework proposes a unified language mapped to both software and hardware, enabling programmers to create parallel software and parallel accelerators in a unified framework.

Ever since the exponential growth of microprocessor frequency stopped, computing systems have heavily relied on parallelism to achieve performance gains. Parallelism comes in many forms, such as instruction-level parallelism, data-level parallelism, and task-level parallelism. In addition, *dynamic parallelism*, where work is generated at run-time rather than statically at compile time, is inherent in many modern applications and algorithms, and is widely used to write parallel software for general-purpose processors. For example, hierarchical data structures such as trees, graphs, or adaptive grids often have data-dependent execution behavior, where the computation to be performed is determined at run-time. Recursive algorithms such as many divide-and-conquer algorithms have dynamic parallelism for each level of recursion. Algorithms that adaptively explore space for optimization or process data as in physics simulation also generate work dynamically.

Modern applications, regardless of being implemented on general-purpose processors or as specialized hardware, often need to exploit multiple types of parallelism to achieve good performance. Effectively exploiting parallelism in accelerators is particularly important for two reasons. First, hardware is inherently parallel. A central problem in accelerator design is how to turn the raw hardware parallelism into good application performance. Second, reconfigurable hardware such as field-programmable gate-arrays (FPGAs) are becoming an increasingly popular general-purpose acceleration platform because they are flexible, easily accessible (e.g., through the cloud [6]), and increasingly integrated with general-purpose cores. FPGAs provide a vast amount of programmable resources, but can only operate at a frequency that is much lower than general-purpose processors or accelerators implemented as ASICs. As a result, it is crucial to exploit parallelism in order to achieve performance on FPGA-based accelerators.

Unfortunately, today's accelerator design methodologies do not provide adequate support for productively exploiting various types of parallelism, especially dynamic parallelism. For example, high-level design frameworks for accelerators such as C/C++-based *High-level synthesis (HLS)* [28, 115] and *OpenCL* [54] are mostly designed to exploit static data-level or thread-level parallelism that can be determined and scheduled at compile time and mapped to a fixed pipeline. Domain-specific languages such as Liquid Metal [11] and Delite [58] raise the level of abstraction but also only support static parallel patterns. A recent study explored dynamically extracting parallelism from irregular applications on FPGAs [64], but still only supports a limited form of pipeline parallelism and does not provide efficient scheduling of dynamically generated work on multiple processing elements. Low-level *Register-transfer-level (RTL)* designs, on the other hand, provide flexibility to implement arbitrary features, but require long design cycles and significant manual effort, making them unattractive especially when targeting a diverse range of applications. This chapter proposes a framework that solves this problem and enables programmers to productively express diverse types of parallelism. The framework includes three key innovations: (1) a new parallel computation model that is general enough while suitable for hardware, (2) an architecture that efficiently realizes the new computation model in hardware, and (3) a productive design methodology to automatically generate RTL.

As a parallel computation model, we propose to adopt a tasked-based programming model with explicit continuation passing. Task-based parallel programming is an increasingly popular approach to write parallel software that achieves this goal, with many frameworks introduced for chip-multiprocessors (e.g., Intel Cilk Plus [4, 62], Intel Threading Building Blocks (TBB) [93], and OpenMP [5]). These task-based frameworks allow diverse types of parallelism to be expressed using a unified *task* abstraction. They also provide good support for dynamic parallelism by allowing a task to generate child tasks at run-time. To support a wide range of communication patterns among tasks and enable efficient hardware implementations, we use *explicit continuation passing* to encode inter-task synchronization.

Then, we propose a novel architecture that can execute an arbitrary computation described using the explicit continuation passing model. The architecture works as a configurable template that provides a platform to dynamically create and schedule tasks, and supports a trade-off between generality and efficiency. For irregular workloads, the architecture can adaptively schedule independent tasks to a pool of processing elements using *work-stealing* [16,17,40], and supports fine-grained load balancing. When the computation exhibits a simple static parallel pattern (e.g., only data-parallel), the architecture can use static scheduling for efficiency. The architecture separates the logical parallelism of the computation from the physical parallelism of the hardware, and enables programmers to express the computation as tasks without worrying about the low-level details of how these tasks are mapped to the underlying hardware.

To minimize the manual effort for accelerator designers, we propose a new design methodology that combines HLS with the proposed computation model and architecture template. The design methodology uses (1) HLS to generate the application-specific worker from a C++-based description, and (2) a parameterized RTL implementation of the architecture template to generate the final accelerator RTL with the desired architecture features and configuration. The designer does not need to write any RTL in order to use the framework.

We built a prototype on the Xilinx Zynq FPGA and implemented a number of parallel accelerators to demonstrate that our framework can indeed handle a wide range of application and provide performance improvements on FPGAs today. We further evaluate the proposed framework in the context of a future SoC with multiple general-purpose cores and an integrated reconfigurable fabric with a cache-coherent memory system using detailed simulations. The results suggest that our framework can generate accelerators that are scalable, and achieve significant speedup compared to a parallel software implementation using Intel Cilk Plus across eight cores.

The rest of the chapter is organized as follows. Section 3.2 gives an overview of the computation model based on explicit continuation passing that support dynamic parallelism. Section 3.3 describes the proposed accelerator architecture. Section 3.4 describes the design methodology for the proposed accelerators. Section 3.5 introduces a unified design flow that enables mapping a single description to both parallel accelerators and software. Section 3.6 describes our evaluation methodology, experimental setup, and evaluation results.

3.2 Computation Model for Dynamic Parallelism

In this section, we introduce the computation model that we use in our framework. The model is based on explicit continuation passing, inspired by task-based parallel programming languages such as MIT Cilk [16, 40], and allows diverse types of parallelisms to be expressed and scheduled under a common framework.

3.2.1 Primitives

A *task* is a piece of computation that takes as input a number of arguments, as well as a *continuation*. More formally, a task is a tuple $(f, args, k)$, where f is the function, $args$ is a list of arguments to f, and k is the continuation which points to another task that should continue after the current task finishes. Formally, a continuation is a tuple $k = (c, p)$ where c is a pointer to a task, and p is an index into the pending task's *args* list. Intuitively, a task is analogous to a function call in software, where f and *args* are a function pointer and the arguments to the function, and k points to the caller, which receives the function's return value and continues execution.

A task can *spawn* new tasks while it is executing. The spawned tasks are called *child* tasks. Spawning tasks is similar to function calls except that the parent and child tasks are allowed to run concurrently. The spawned tasks eventually need to be *joined*, so we know that they finished and subsequent computation (that potentially depends on the output of the child tasks) can proceed. Today's software frameworks use special join commands that call into a sophisticated runtime to perform synchronization, which is difficult to implement in hardware. Instead, our model uses explicit continuation passing that leads to a simpler hardware architecture.

A task can be either *ready* or *pending*. A task is ready if it has received all of its arguments, and thus is ready to execute. A task is pending if some of its arguments are still missing, for example, because the tasks that produce them have not completed yet. Each pending task is associated with a *join counter* j, whose value is the number of missing arguments. A task returns a value by sending it to the pending task pointed by its continuation. In our model, task return values and arguments are two sides of the same coin: a task's return value is simply another task's argument. Upon receiving the value, the join counter j of the pending task is decremented. When the join counter reaches zero, the pending task has received its last missing argument, and becomes ready.

3.2.2 Continuation Passing

Today's parallel programming frameworks for software uses a runtime system to manage synchronizations among tasks. Whenever a task needs to perform a synchronization operation such as a join, it transfers control to the runtime, which then checks the state of other tasks and decides the action to perform. This approach is challenging to implement in hardware. First, software can perform control transfers between the user code and runtime with function calls or `setjmp/longjmp`, and easily save and restore the program state using stack frames. These capabilities are not present in hardware accelerators. Second, the runtime logic is often quite complex, which would incur high overhead if implemented in hardware. We address these challenges by using *explicit continuation passing* for synchronization instead of a runtime system.

Continuation passing style (CPS) is a style of programming where control is passed explicitly in the form of a continuation that represents what should be done with the result that the current procedure generates. Our framework uses continuation passing to express computation as a dynamic task graph with explicit dependence. The continuation passing serves as the foundation and can be used to construct other abstractions such as data-parallel loops and fork-join patterns.

In our model, the continuation of a task points to a pending task (more precisely, one of the pending task's arguments) that should receive the current task's return value. The simplest use of continuation passing is to implement sequential composition of tasks. Suppose we want to execute tasks A and B sequentially and return the result to continuation k. Using continuation passing, we can invoke task A with k as its continuation. When A finishes, it spawns

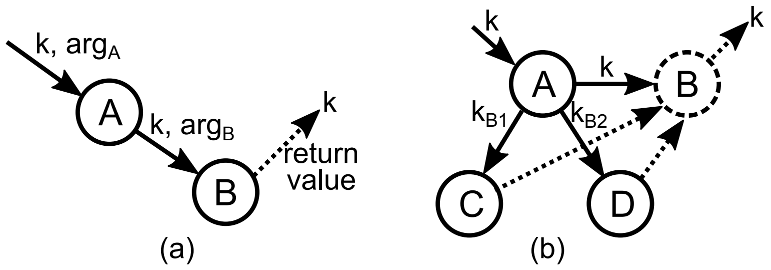


Figure 3.1: Continuation passing for (a) sequential composition of tasks, (b) fork-join. Downward arrows represent spawning tasks. Horizontal arrows represent creating successor tasks. Dotted arrows represent returning values (arguments).

task *B*, passing its own continuation *k* to *B*. When *B* finishes, it returns its result to *k*. Figure 3.1(a) illustrates this operation.

Continuation can also be used to implement the fork-join pattern, which is a common pattern for dynamically generating parallel tasks and perform synchronization. Suppose we would like to run two parallel tasks and combine their results. In this case, task *A* creates a pending task *B* called the *successor*, spawns two child tasks *C* and *D*, and points their continuations to *B*. This completes the fork step. When the child tasks finish, they send their result values to *B*. *B* becomes ready once it receives the results of both *C* and *D*. This completes the join step. Figure 3.1(b) illustrates the fork-join operation.

Task Graph Examples

Using continuation passing combined with task spawning, both static and dynamic parallel algorithms can be expressed in our model. When the algorithm executes, the tasks form a graph that dynamically unfolds. Here we show three example task graphs from different algorithms.

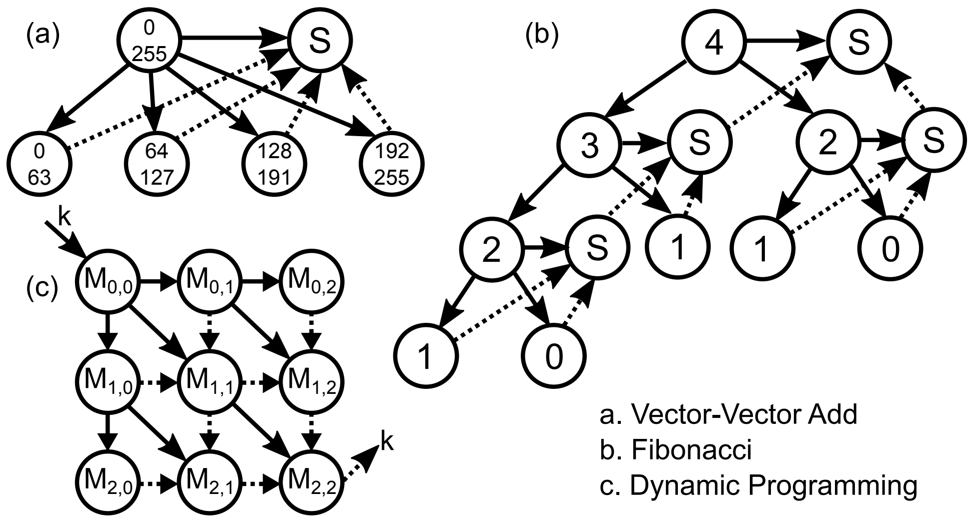


Figure 3.2: Task graphs constructed using continuation passing. (a) Vector-vector add. Node labels represent the start and end indices of the sub-vectors. (b) Fibonacci. Each numbered node represents the task for `fib(n)`. Nodes labeled `S` represent the successor (sum) tasks. (c) Dynamic programming. Solid arrows represent spawns along which the continuation is passed. Result values are passed along dotted arrows, with the final result sent to the continuation k.

The first example computes the sum of two vectors of length 256. Suppose we allow the vector to be divided into chunks of length 64. Figure 3.2(a) shows the task graph. As the computation is data-parallel, the task graph is very regular, and can (though does not have to) be constructed using only a single level of child tasks. In addition, only the child tasks perform actual work, and the parent and successor tasks are only for synchronization because the results of child tasks do not need to be combined.

Consider another example that calculates the n^{th} Fibonacci number by recursively applying the formula $fib(n) = fib(n-1) + fib(n-2)$, and calculate $fib(n-1)$ and $fib(n-2)$ concurrently. This is a typical fork-join pattern. Here we need two types of tasks: `fib` and `sum`. The `fib` task takes a number we want to calculate Fibonacci for, and if it is the base case, it sends the result to the continuation. Otherwise, it performs a fork-join operation, with the successor task performing the sum operation. For example, calculating `fib(4)` results in the graph shown in Figure 3.2(b).

This example illustrates why it is challenging to map computations with dynamic task-level parallelism to a hardware accelerator. First, tasks are created dynamically during execution, which makes it difficult to enumerate them statically. Second, the computation involves recursion, which is often not supported in current hardware design tools. Third, the task tree is often imbalanced, which makes techniques that partition work statically inefficient.

Now consider a third example, in which an algorithm fills a matrix with values. Each element depends on its left and top neighbor. This pattern is common in many dynamic programming algorithms. Figure 3.2(c) shows the task graph for filling a simple 3x3 matrix. This type of general task-parallel pattern be expressed using continuation passing, but cannot be easily expressed in frameworks that only support fork-join. This example shows that using continuation passing can also improve expressive power in some cases.

Nesting and Composability

It is apparent from the examples above that the computation model naturally supports nested parallelism, where a spawned task can recursively spawn parallel sub-tasks. Nesting is important for achieving good parallelization for many algorithms. For example, in Fibonacci, the degree of parallelism at each task is only two. Without nesting, only the root task can be parallelized, yielding a maximum speedup of two. With nesting, all non-overlapping subtrees of the

fib task tree can run in parallel, significantly increasing the amount of parallelism.

Another feature of the computation model is composability, which means the computation can be expressed in a combination of data-parallel, fork-join, or general task-parallel patterns, and the resulting program should still work. This is from the fact that all higher-level patterns are ultimately transformed to the continuation passing primitives.

3.2.3 Scheduling the Computation

The model described above enables expressing concurrency in the computation. However, it is up to the *scheduler* to schedule the tasks onto processing elements at run-time for parallel execution. Our framework supports both dynamic work scheduling using work stealing [16, 17] for general dynamic computation, and static task distribution for simple data-parallel computations. Here, we briefly describe the work stealing model.

We model each processing element as a datapath that can process tasks, a local task queue that stores ready tasks, and a pending task storage that holds pending tasks. Each processing element operates on its own task queue in a LIFO (Last-In First-Out) manner, that is, operating on the tail of the queue. When a processing element is idle, it first tries to dequeue and execute a task from the tail of its local task queue. If a task is spawned or a pending task becomes ready, it is appended to the tail of the task queue. If the task queue is empty, the processing element (thief) begins work stealing by randomly selecting another processing element (victim) and trying to steal a task from the head of victim's task queue, that is, the oldest task in the queue. If the continuation of a task refers to a pending task on another processing element, and sending the task's return value caused the pending task to become ready, the newly created task is transferred back to the original processing element to be executed. This is needed to implement *greedy* scheduling, which means the processing element that produces the last missing argument of a pending task should continue execution with the successor task [94]. Greedy scheduling is important for the space bound.

A task graph is said to be *fully strict* if every task sends its result only to its parent's successor task. Both the vector-addition and Fibonacci graphs are fully strict by this definition. In fact, any fork-join computation described using the above approach is fully strict. It can be shown that for fully strict computations, the scheduling policy described above has the same behavior as Cilk's scheduler [16], which is provably efficient. Specifically, it can be shown that the space to store the tasks required for an execution with P processing elements is bounded by $S_P \leq S_1 P$, where S_1 is the space required for a serial execution on one processing element [17]. This bound is important to put a limit on the task queue sizes required for the parallel execution. This limit guarantees that the parallel execution is space efficient, which avoids the space explosion issue seen in certain scheduling policies. Also, the space bound is crucial for accelerators as the task queues are likely hardened rather than being a data structure in the memory.

3.2.4 Function Calls

Traditional HLS tools provide insufficient support of function calls, especially the ones that are deeply nested or recursive. This is not surprising because recursive function calls require a stack, which a hardware accelerator does not have. As a result, these tools can only handle simple functions that can be inlined. Our computation model naturally supports recursive function calls by reusing the task spawn and continuation mechanisms. This is similar to how classic continuation passing style programs handle function calls.

3.3 Accelerator Architecture

The accelerator architecture implements the computation model described in Section 3.2. Specifically, the architecture is designed to fulfill two major goals: (1) implementing task spawning and explicit continuation passing in hardware, and (2) scheduling the computation. Figure 3.3 shows the high-level system architecture, with the accelerator shown in the shaded box. The accelerator consists of multiple tiles, and each tile is composed of a configurable number of processing elements (PEs), each with a unique ID. A tile serves as a basic building block in the architecture, which is a fully-functional task processing engine. The accelerator can consist of any number of tiles without changing its functionality. This tile-based architecture enables an accelerator designer to easily scale the number of tiles and PEs, and also reduces design effort as one component is reused multiple times. The tiles are connected together using two on-chip networks: an argument/task network, and a work stealing network. Each tile has an L1 cache, shared by the PEs in that tile. The accelerator also has an interface block (IF) that communicates with the host CPU to receive commands.

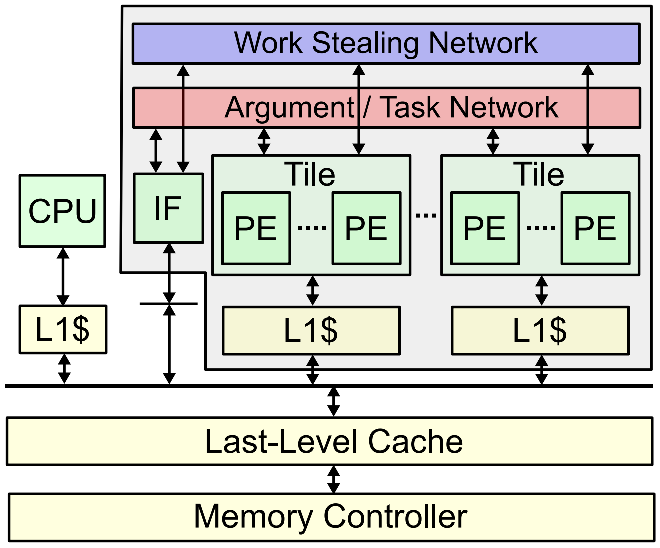


Figure 3.3: System architecture. The accelerator is shown in the shaded box.

In this study, we present the architecture in the context of an SoC where the general-purpose cores and accelerators share a single address space as well as the last-level cache through a cache-coherent interconnect. The accelerators can be implemented either as ASIC or using on-chip FPGA fabric. In the latter case, the architecture fits well into future integrated CPU-FPGA SoCs, which are increasingly popular and attractive for general-purpose acceleration because it can be easily reconfigured, and it allows fine-grained data sharing between the CPU and accelerators. For applications that do not need fine-grained sharing, the proposed architecture can also be adapted to discrete FPGAs with changes to the memory hierarchy.

We present two variants of our architecture, named FlexArch and LiteArch, which support different trade-off points between flexibility and overhead. FlexArch supports the full continuation passing model, and allows programmers to implement algorithms using many parallel patterns including data-

Table 3.1: Comparison between tile architectures.

Pattern	**FlexArch**	**LiteArch**
Data-Parallel	Yes	Yes
Fork-Join	Yes	No
General Task-Parallel	Yes	No
Task Scheduling	Work-Stealing	Static Distribution

parallel and fork-join patterns as well as nesting in flexible ways. It uses work-stealing for task scheduling. In comparison, LiteArch only supports the data-parallel pattern, and uses static task distribution for task scheduling. LiteArch is intended as a lightweight alternative for applications where a static data-parallel pattern is sufficient and does not need advanced load balancing capabilities. Table 3.1 summarizes the features of the two architectures. An algorithm using the data-parallel pattern can map to either architecture without code changes. In contrast, an algorithm using the fork-join or continuation passing pattern maps naturally to FlexArch, but can only be mapped to LiteArch if it can be rewritten using the data-parallel pattern.

3.3.1 FlexArch Tile and PE Architecture

Figure 3.4 shows the architecture of a FlexArch tile. A tile contains multiple processing elements, as well as a pending task storage (P-Store), an argument/task router, and network interfaces (Net IF). These components are connected via intra-tile buses. Each PE consists of a worker and a task management unit (TMU).

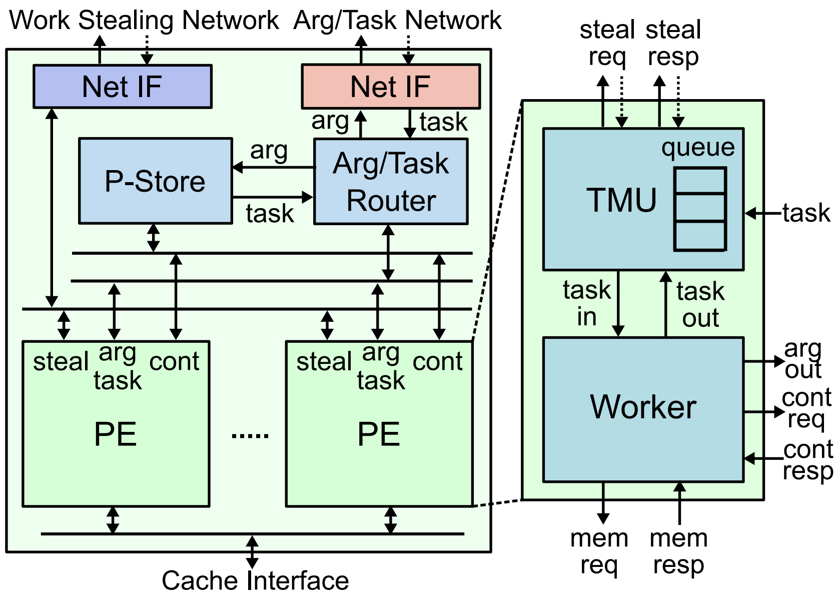


Figure 3.4: FlexArch tile.

The worker performs task-specific computations. Because this is the part that an accelerator designer needs to describe for each application, the architecture is designed to keep the worker simple. We factor out common functionalities such as task management into separate modules that can be reused, and provide the worker an interface to communicate with these modules by sending and receiving messages. It has a `task_in` port for receiving a task, a `task_out` port for spawning a task, an `arg_out` port for returning a result value, and a pair of `cont_req` and `cont_resp` ports for creating a successor task and receiving a continuation that points to it. The worker also has a memory port. The architecture does not stipulate how the worker is implemented as long as it follows the interface protocol. For example, it can be implemented in either HLS or RTL. Our architecture currently uses homogeneous workers that can run any task in the computation graph. The type of a task is identified by the `type` field in the task message, which corresponds to the f in the computation model. It is also possible to extend the framework to use heterogeneous workers where each worker is used to process different tasks, and is shared by the tile. This allows coarse-grained resource sharing at the tile level. In contrast, when using homogeneous workers, HLS tools perform fine-grained resource sharing between the logic for different tasks at the worker level.

The task management unit (TMU) is responsible for feeding the worker with tasks. Internally, it has a task queue that stores ready tasks. The task queue is implemented as a double-ended queue that supports operations on both ends. The worker enqueues and dequeues tasks at the tail of the queue in a LIFO order, which is important because it results in much better task locality than FIFO order, by traversing the task graph in a depth-first manner. When the task queue becomes empty, the TMU initiates work stealing. It uses a linear feedback shift register (LFSR) to pick a random PE as the victim. Then it sends a steal request to the victim through the network. When the TMU on the victim PE receives the request (shown as dotted arrows in Figure 3.4), it dequeues a task from the head of its queue and sends it back to the stealing PE. Stealing from the head is important for efficiency because it enables stealing a larger chunk of work with each request (i.e. the task at the head is closer to the root of the task spawn tree).

During low-parallelism phases of the computation, there may not be enough parallel tasks available to keep all processing elements busy. As a result, the TMUs will repeatedly send out work stealing requests, potentially causing congestion in the work stealing network. To avoid this problem, the TMUs implement an exponential backoff scheme. We add a waiting period, which represents the number of cycles the TMU needs to wait before sending out a work stealing request. Initially, the waiting period is set to a single cycle. After each unsuccessful steal, which indicates that there are not enough parallel tasks available, the waiting period is doubled until it reaches a limit (e.g., 128 cycles). In this way, the work stealing activity is throttled during low-parallelism phases, which not only alleviates network congestion, but also reduces energy consumption. On the other hand, when the computation enters a high-parallelism phase, we would like the PEs to quickly pick up the parallel tasks and process them. To achieve this goal, we improve the design so that when the TMU performs a successful steal, the waiting period is reset to a single cycle. This allows the TMUs to quickly exit the exponential backoff process and begin processing the parallel tasks that become available.

The P-Store holds pending tasks that are waiting for arguments, and keeps track of whether they are ready for execution. Its function is analogous to the reservation stations in an out-of-order processor. A straightforward design is to implement the P-Store as a centralized structure for the entire accelerator, where all pending tasks are kept. However, this would lead to severe contention when scaling up the number of PEs. We address this challenge by proposing a distributed architecture, where each tile has a local P-Store, but is still able to access P-Stores on other tiles over the network. Because of locality in the processing of the task graph, pending tasks created by a tile are likely to be consumed within that tile, which means most accesses would go to the local P-Store without incurring network traffic.

Figure 3.5 shows the P-Store architecture. Each P-Store consists of a control unit, a free list, a join counter array, a metadata array, and argument arrays. The free list keeps track of the available entries in the P-Store. When a PE requests to create a pending task, an entry is allocated, and a continuation ID is returned.

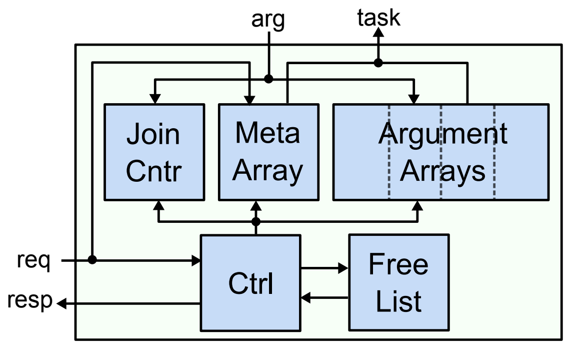


Figure 3.5: P-Store architecture.

The join counter array stores the number of missing arguments for each pending task. When an argument is received, the data is written to the argument array specified by the continuation, and the join counter is decremented. If the counter reaches zero, the task that became ready is sent to the PE that produced the last argument, and the entry is deallocated.

The task queue and P-Store implemented in hardware have a maximum capacity that is fixed at design time. Thus it is important to make sure that there is enough space to hold the ready and pending tasks. There are two aspects of this issue. First, there needs to be enough space for a serial execution of the computation with a single PE. For fully strict computations, the space needed is proportional to the critical path of the task graph. For most parallelizable applications, the critical path is much smaller than the number of tasks in the graph (the total amount of work), because this is a key requirement for the application to be parallelizable. In fact, when the task spawn tree is balanced, the critical path can be derived from the height of the tree, which grows logarithmically with the number of tasks. As a result, the amount of space required is small even for very large problem sizes. Second, when the number of PEs grows, the space needed for a parallel execution should not exceed the available task queue and P-Store space. The space bound discussed in Section 3.2.3 guarantees that the space required for a parallel execution with P processing elements is at most S_1P, where S_1 is the space required for a serial execution. Assuming we fix the number of PEs per tile, the available task queue and P-Store space also grows linearly with the number of tiles. As a result, we can scale the number of tiles without increasing the number of entries of the task queue or P-Store.

For highly unbalanced workloads, it is possible that the required space may exceed what can economically fit in the on-chip task queue and P-Store. In this case, we can extend the architecture to make the task queue and P-Store backed by data structures in the main memory. The on-chip task queue and P-Store would then cache the most recently used entries of the in-memory task data structures.

The argument/task router steers argument and task messages between local and remote tiles. This is needed for two reasons. First, when a worker returns an argument, it may refer to a pending task on a remote tile. Second, when the P-Store receives an argument from a remote PE and outputs a task, the task needs to be sent back to the remote PE in order to implement *greedy* scheduling, which is critical for guaranteeing the asymptotic bound on space [16]. The structure of the argument/task router consists of two funnel/router pairs. The funnel combines two message streams into one, and the router splits a stream into two depending on whether the destination is local or remote.

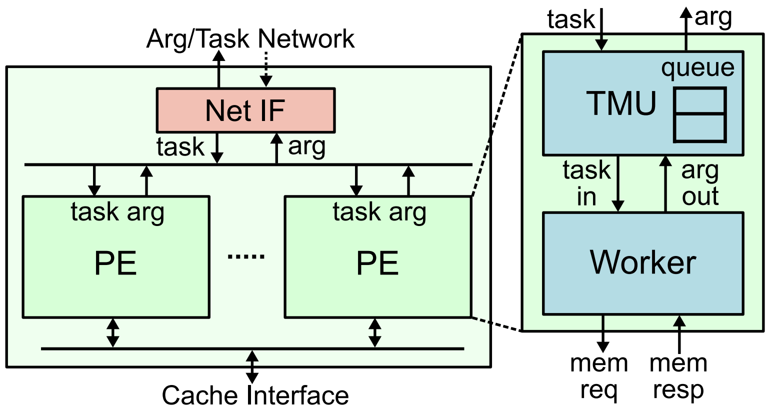


Figure 3.6: LiteArch tile.

3.3.2 LiteArch Tile and PE Architecture

Figure 3.6 shows the architecture of a LiteArch tile. Compared to a FlexArch tile, it does not have a P-Store or argument/task router as there is no support for creating pending tasks or routing arguments and tasks between tiles. In this architecture, the accelerator does not have a work stealing network. Within a PE, the TMU is simplified to remove work stealing capabilities, and the worker does not have P-Store ports. This architecture supports the data-parallel pattern with the host CPU splitting the range into smaller subranges, and enqueuing the tasks for execution on the PEs.

3.3.3 Networks

The argument and work stealing networks shown in Figure 3.3 are two logical networks. Our architecture does not specify the physical implementation of these two networks, as long as they are compatible with the network interface protocol. They can be implemented with different topologies, or even be combined into one physical network. In our implementation, we use crossbars for the networks.

3.3.4 Memory Hierarchy

In our architecture, the accelerators are integrated into the general-purpose memory hierarchy via the last-level cache, and share the same address space as the general-purpose cores. Integrating accelerators into the memory hierarchy represents a challenge for many HLS tools. Traditional HLS tools assume a fixed latency for all memory accesses, and generate large monolithic designs which struggle when facing the variable memory latency of a general-purpose memory system; a delay in any memory response would cause the entire design to stall. Our accelerator architecture overcomes this problem by making PEs independent so that one stalled PE would not affect others, and using dynamic work scheduling to balance the load on the PEs should any imbalance arise due to memory latencies.

The accelerator has a number of L1 caches, one per each tile. The caches are kept coherent among themselves and with the last-level cache. We found that the cache coherence support is not necessary for many applications, but use coherent caches in our architecture to support a wider range of applications including the ones that require fine-grained sharing among PEs. For FPGA implementations, the accelerator caches can be implemented using the block RAMs. Future FPGAs can also include hardened L1 cache blocks. Because the block RAMs can be clocked at a considerably higher frequency than the accelerator logic [29], the L1 caches can be double-pumped. Also note that the workers can have local memory structures such as scratchpads that are not a part of the cache-coherent memory system. Some accelerators rely on the massive internal memory bandwidth provided by such local memory to achieve high performance. When a task is stolen, data movement is performed transparently through shared memory with coherent caches. Note that scratchpads in PEs are only used to store temporary state, local to each task. We investigate the single address space, cache-based memory system for two reasons. First, caches reduce programming effort by removing the need to manually orchestrate data transfers, which is a significant portion of the design efforts in today's hardware accelerators. Second, caches and a single address space enable fine-grained data sharing between the CPU and FPGA, e.g. sharing pointer-based data structures, which widens the range of applications that can be mapped to the architecture. The proposed framework is also applicable to the DMA-based accelerators if fine-grained data sharing is not needed, and designers are willing to explicitly control data transfers. A PE can initiate DMA transfers to read input / write output data for a task.

3.3.5 CPU-Accelerator Interface

The accelerator contains an interface (IF) block that serves as the interface between the CPU and the accelerator. The IF block implements a memory-mapped interface. The CPU can send tasks to the accelerator and read results back using memory-mapped accesses. Once the IF receives a task, it needs to pass it to the PEs for processing. For FlexArch, we leverage work stealing for this purpose. A PE can steal a task from IF via the work stealing network. For LiteArch, the IF passes the task via the argument/task network, which is then consumed by

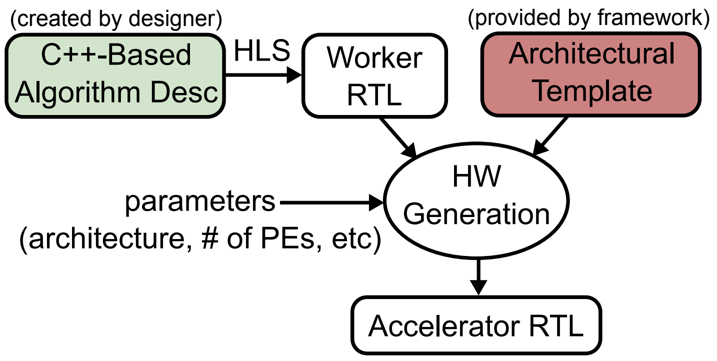


Figure 3.7: Accelerator design flow.

one of the PEs.

3.4 Design Methodology and Framework

In this section, we discuss the methodology and framework we developed for designing accelerators with low manual effort. Figure 3.7 shows the overall flow of the framework. Accelerator designers describe the algorithm using a C++-based worker description format, then the framework synthesizes the worker RTL using HLS. Next, the framework combines worker RTL with an architectural template it provides, and generates the final RTL of the accelerator.

3.4.1 Architectural Template

We implemented the proposed accelerator architecture as an architecture template in PyMTL [69], a Python-based hardware generation language. The template is parameterized so that the designer can configure the architecture (FlexArch or LiteArch), the number of tiles and PEs, the number of entries of

Table 3.2: Summary of task APIs.

Name	Arguments	Description
`spawn`	`t`: task to be spawned `task_out`: task spawn port	Spawn child task `t` through port `task_out`
`spawn_next`	`t`: task to be spawned `task_out`: task spawn port	Similar to `spawn`, but the task is treated as a successor rather than a child
`make_successor`	`t`: task type `k`: continuation `j`: join count `cont_req`: request port `cont_resp`: response port	Create a successor task of type `t` with `j` missing arguments through the `cont_req` and `cont_resp` port pairs, and pass continuation `k` to it
`send_arg`	`k`: continuation `arg`: argument value `arg_out`: output port	Return argument `arg` to the task pointed by `k` through port `arg_out`

the task queue and P-Store, as well as the cache size.

3.4.2 Algorithm Description Format

While the accelerator architecture does not specify how the task processing logic (worker) should be implemented, in practice, high-level synthesis is usually preferred because of its productivity compared to RTL design. We support the HLS approach by defining a C++-based worker description (CPPWD) format and a set of task APIs. Using CPPWD and the task APIs, designers can describe the worker logic and express operations in the computation model such as task spawning and synchronization. Table 3.2 summarizes the task APIs. The APIs are implemented as a library, which enables them to be easily integrated into existing HLS tools.

As an example, Figure 3.8 shows the CPPWD code for the Fibonacci algorithm described in Section 3.2. The worker is defined as a function, and the arguments of the function are the ports of the worker. The function header is standard for all workers, except for the function name and task type, which are defined by the designer. The body of the function defines the Fibonacci algorithm, which recursively splits the problem into sub-problems by dynamically spawning child tasks until reaching the base case, and then merging the results back to obtain the answer. This algorithm is challenging to express using today's accelerator design methodologies because it involves dynamically bounded parallel recursion, but is trivial to express using our framework.

For the data-parallel pattern, the framework provides a helper function (`parallel_for`) similar to Intel TBB [93], which wraps the details of implementing dynamic spawning/joining of tasks in an easy-to-use interface. CPPWD also supports the `blocked_range` concept, which allows splitting a linear range into blocks of configurable size.

3.4.3 Accelerator RTL Generation

The framework generates the accelerator RTL by combining the synthesized worker RTL with the architecture template according to the parameters specified by the designer, including the choice of the architecture, the number of PEs, the number of task queue entries, cache size, etc. The framework then elabo-

```
void FibWorkerHLS
(
  TaskInPort<FibTaskType>  task_in,
  TaskOutPort<FibTaskType> task_out,
  ContReqPort              cont_req,
  ContRespPort             cont_resp,
  ArgOutPort               arg_out
) {
  const FibTaskType task = task_in.read();

  // continuation
  task_k_t k = task.k;

  if (task.type == FIB) {
    int n = task.x;
    if (n < 2)
      send_arg(Argument(k, n), arg_out);
    else {
      // create successor task
      k = make_successor(SUM, k, 2, cont_req, cont_resp);

      // spawn child tasks
      spawn(FibTaskType(FIB, k, 1, n-2, 0, 0), task_out);
      spawn(FibTaskType(FIB, k, 0, n-1, 0, 0), task_out);
    }
  } else if (task.type == SUM) {
    int sum = task.x + task.y;
    send_arg(Argument(k, sum), arg_out);
  }
}
```

Figure 3.8: C++-based worker description for Fibonacci.

rates the template and perform hardware generation to output the final RTL of the accelerator. Design space exploration can be done easily by changing the parameters given to the framework, without rewriting any code.

Using the design methodology, accelerator designers only need to write a small amount of code describing a parallel algorithm, and the framework can automatically generate the RTL description of a parallel accelerator.

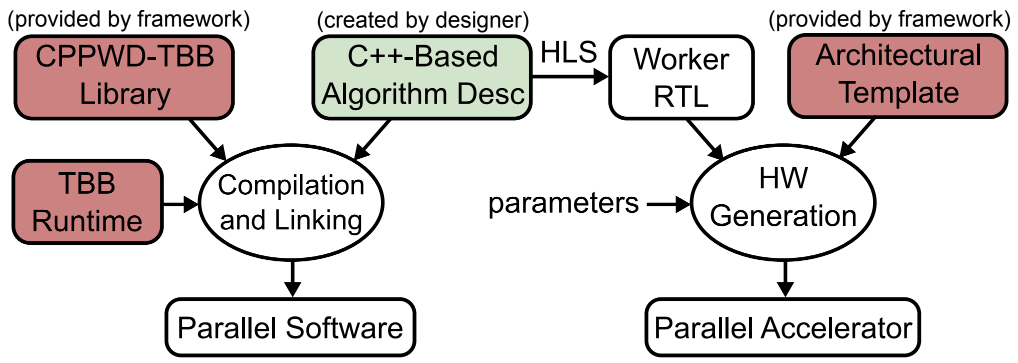


Figure 3.9: Unified parallel accelerator and software flow.

3.5 Unified Framework for Parallel Accelerators and Software

In this section, we propose a design flow that enables mapping a single description of a parallel algorithm to both software and accelerators. Traditionally, software programmers and accelerator designers rely on different languages and frameworks. For example, to create parallel software, programmers usually use frameworks such as Cilk Plus [4], Threading Building Blocks [93], or OpenMP [5]. To create parallel accelerators, designers either use hardware description languages such as Verilog, or use specialized languages such as OpenCL [54]. These two distinct sets of languages and tools have very different semantics, programming model, and features sets, which makes porting applications between software and hardware time-consuming and error-prone. In addition, using software programming frameworks and hardware design tools require different knowledge and skill sets. As a result, it is difficult for programmers with only software expertise to use the hardware design tools, or vice versa. It is desirable to have a single framework with a unified language that both software programmers and hardware designers can use, which would improve productivity, and expedite the adoption of accelerators.

We have shown in previous sections of this chapter that algorithms described using a task-based parallel computation model can be mapped to efficient hardware architectures. In this section, we show that the same description can also be mapped to parallel software runtimes. Figure 3.9 shows the unified flow that has both a parallel accelerator backend and a parallel software backend. Starting from a C++-based algorithm description, designers can use the hardware flow (described in the previous section) to generate a parallel accelerator, and can also use the software flow to generate parallel software that targets the Threading Building Blocks (TBB) runtime. To achieve this, the framework provides a library that turns task operations into calls to the TBB runtime. The primary reason we choose a library-based approach over a language-based approach is that it does not require modifications to the compiler. We leverage TBB's support for continuation passing and its work-stealing runtime instead of creating our custom runtime, which has two benefits. First, the TBB runtime is highly optimized, which ensures that the generated parallel software is efficient. Second, it enables the flow to take advantage of TBB's extensive support for various operating systems and ISAs.

3.5.1 CPPWD-TBB Library

The framework is able to generate both hardware and software from the same CPPWD-based algorithm description by providing two implementations of the task APIs. The task APIs are used for operations such as spawning child tasks and returning arguments. Figure 3.10 shows the hardware and software implementation stacks. In the hardware implementation, the task APIs are translated into messages sent and received on the worker's ports. To keep the algorithm

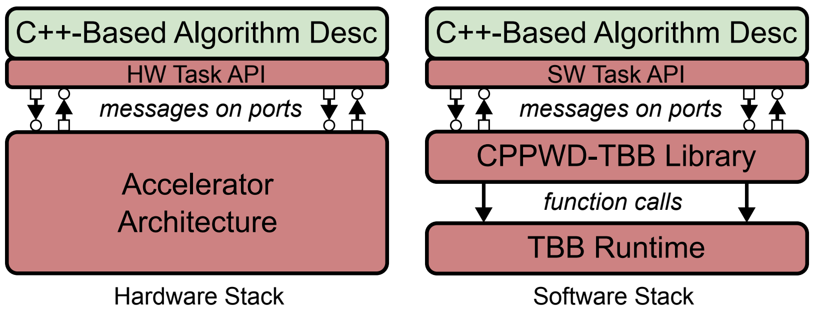


Figure 3.10: Hardware and software implementation stacks.

description reusable, we implemented *virtual ports* in the software implementation of the workers, which are objects with `read` or `write` methods. We implemented a CPPWD-TBB library that the worker communicates with by sending and receiving messages on the virtual ports. The library then converts messages into calls to the TBB runtime.

Here we describe the implementation of task APIs for the software backend.

- `spawn` is implemented by sending a child task message on the `task_out` virtual port of the worker. The CPPWD-TBB library receives and parses the task message, and calls the TBB runtime's `tbb::task::allocate_additional_child_of` method to allocate the child task, and then calls `tbb::task::spawn` to perform the spawn operation.
- `spawn_next` is similar to `spawn`, but the task is allocated using `tbb::task::allocate_continuation` instead.
- `make_successor` is implemented by sending a successor task message on the `cont_req` virtual port of the worker. The CPPWD-TBB library receives and parses the message, and then calls the TBB runtime's `tbb::task::allocate_continuation` method to allocate the succes-

Table 3.3: Programmability comparison. M: Manual. A: Assisted by the compiler.

Step	**CPPWD**	**TBB**	**Cilk Plus**
Algorithm Parallelization	M	M	M
Code Restructuring	M	M	A
Task Input/Output	M	M/A	A

sor task.

- `send_arg` is implemented by writing the argument value to the argument slot pointed by a task's continuation, without involving TBB.

3.5.2 Programmability

Here we describe the programmability of writing parallel programs using the proposed unified framework, and compare it to native TBB and Cilk Plus.

To create a parallel program using the proposed unified framework, programmers need to write the code in a continuation passing style. Compared to a sequential program, this involves three extra steps: (1) programmers need to conceive an approach to parallelize the algorithm; (2) the program needs to be restructured into several pieces (tasks), and the tasks need to be spawned and synchronized using the task API; (3) programmers need to specify the input and output data passed between the tasks. Table 3.3 shows whether these three steps need to be manually performed or assisted by the compiler.

In all three frameworks, parallelizing the algorithm needs to be performed manually, as none of them intend to perform automatic parallelization. In terms of code restructuring, both our framework (using CPPWD) and TBB require the programmers to perform the step manually because they are both library-based approaches. In Cilk Plus, this step is assisted by the compiler. Programmers only need to annotate the code with certain keywords, and the compiler frontend automatically transforms the code. In terms of passing input/output data between the tasks, both our framework and TBB (prior to C++11) require the programmers to manually specify the data to be passed. TBB (C++11 and later) leverages the lambda capture feature of C++11 to assist the programmers in this step. In Cilk Plus, this step is automated by the compiler. This comparison indicates that our framework is somewhat a lower-level framework compared to TBB and Cilk Plus. However, we found that in practice this does not significantly affect productivity. The reason is that the most challenging part of creating a parallel program is usually algorithm parallelization, while the other two steps are fairly simple and mechanical. In addition, when the parallel algorithm uses `parallel_for`, our framework also provides a convenient wrapper that alleviates the need to perform code restructuring and task input/output handling. Future work may also look into providing compiler support for our framework to automate these two steps in the general case.

3.6 Evaluation

In this section, we present the evaluation results for the proposed accelerator framework. We first present a hardware prototype of accelerators on today's FPGA platform. Then, in order to perform a more detailed study of the architecture, and to avoid the limitations of the current FPGA platform, we present a simulation-based study of the accelerators in the context of a future integrated

CPU-FPGA SoC. Finally, we present an evaluation of the performance of parallel software built using the unified description.

3.6.1 Benchmarks

We use a set of ten benchmark algorithms that cover a variety of application domains, including linear algebra, graph search, sorting, combinatorial optimization, image processing, and bioinformatics. Some of the benchmarks are developed in-house, while others are adapted from benchmark suites such as Cilk apps [40], Unbalanced Tree Search [77]. and MachSuite [92]. We coded parallel implementations of these algorithms using the unified description format provided by our framework. Task granularity depends on application characteristics, but is chosen to strike a balance between parallelization overhead and load balancing. Table 3.4 summarizes the benchmarks and shows the characteristics of each benchmark. Among them, the ones that are recursive, or have nested or data-dependent parallelism are especially challenging to express in existing accelerator design frameworks, but our framework allows writing these algorithms easily.

Here we give a brief description of each benchmark algorithm, as well as the approach we take to parallelize them:

1. `nw` implements the Needleman-Wunsch algorithm, which is a dynamic programming algorithm that aligns two DNA sequences. The algorithm fills values of a two-dimensional matrix, where the value of each element depends on its neighbors on the north, west, and northwest. We parallelize `nw` by blocking the matrix, and using continuation passing to con-

Table 3.4: Summary of benchmarks. PA: Parallelization Approach, PF=parallel-for, FJ=fork-join, CP=continuation passing. R/N: Recursive/Nested Parallelism. DP: Data-Dependent Parallelism. MP: Memory Access Pattern. MI: Memory Intensity.

Name	From	PA	R/N	DP	MP	MI
nw	In-house	CP	Yes	Yes	Regular	Medium
quicksort	In-house	FJ	Yes	Yes	Regular	Medium
cilksort	Cilk apps	FJ	Yes	Yes	Regular	Medium
queens	Cilk apps	FJ	Yes	Yes	Regular	Low
knapsack	Cilk apps	FJ	Yes	Yes	Regular	Low
uts	UTS	FJ	Yes	Yes	Regular	Low
bbgemm	MachSuite	PF	Yes	No	Regular	Medium
bfsqueue	MachSuite	PF	No	No	Irregular	High
spmvcrs	MachSuite	PF	No	No	Irregular	High
stencil2d	MachSuite	PF	No	No	Regular	High

struct the task graph, similar to Figure 3.2(c).

2. `quicksort` implements the classic Quicksort algorithm, which is a divide and conquer algorithm that recursively partitions an array into two smaller arrays and sorts them. We use the Hoare partition scheme [52] in our implementation, and use fork-join to parallelize across the divide-and-conquer tree.

3. `cilksort` is a parallel merge sort algorithm first described in [9]. It recursively divides an array into smaller arrays and sorts them, and also performs the merging in parallel. When the sub-array size gets small, it uses quicksort to sort the sub-array, which in turn partitions and sorts the sub-arrays, and uses insertion sort when the sub-array size becomes sufficiently small (tens of elements). We use fork-join to parallelize across the

divide-and-conquer tree.

4. queens solves the classic N-queens problem. We use fork-join to parallelize searching of the solution space.

5. knapsack solves the 0-1 knapsack problem. Our implementation uses a branch-and-bound algorithm, and is parallelized using fork-join.

6. uts is a benchmark that dynamically constructs and searches an unbalanced tree. The unbalanced nature of the tree stresses the load balancing capability of the architecture. We use fork-join to parallelize across the subtrees.

7. bbgemm is a matrix multiplication kernel that uses blocking to achieve good memory locality [61]. We use a block size of 32 and parallelize the loop nest with two nested parallel-for's.

8. bfsqueue is a breadth-first search algorithm that uses a queue to store frontier nodes. We parallelize across the frontier with a parallel-for loop.

9. spmvcrs is a sparse matrix-vector multiplication algorithm using compressed row storage format. We parallelism across the matrix rows using parallel-for.

10. stencil2d performs stencil computation on a 2D image. We break the image into blocks and use parallel-for to parallelize across the blocks.

For each worker that is generated by HLS, we applied standard HLS optimization techniques such as loop pipelining and unrolling, and use application-specific local memory structures such as scratchpads and buffers to achieve high internal memory bandwidth when possible. In that sense, a single PE in our architecture can be considered to represent optimized accelerators designed using today's HLS tools without additional parallelization support.

For benchmarks that use fork-join or continuation passing, we also tried to implement a version that only uses parallel-for, targeting the LiteArch. The high-level idea is to use multiple rounds, with each round processing one level of the task graph using a parallel-for, and at the same time constructing the next level. This requires the tasks in the same level to be homogeneous. For benchmarks that cannot be parallelized this way, we also tried to rewrite the algorithm using a different approach if that helps mapping it to parallel-for. In the end, we were able to implement parallel-for versions of `nw`, `quicksort`, `queens` and `knapsack`, but not `cilksort`, due to the complexity and irregularity of its dynamic task graph.

We also coded a parallel software implementation for each algorithm using Intel Cilk Plus [4], and compiled with `-O3` optimization and auto-vectorization targeting NEON SIMD extensions.

3.6.2 Design Effort Comparison

Table 3.5 compares the lines of code between the Cilk Plus source code, the CP-PWD source code and the generated Verilog RTL. The number of lines of the generated Verilog code is intended to serve as a rough estimate of the design effort needed to describe the accelerators using a hardware description language. Manual designs would have a different number of lines from the generated Verilog code, but is likely to be the same order of magnitude because manual designs also need to implement the features needed to support dynamic work generation and scheduling in order to be efficient. We can see that our framework provides at least an order of magnitude reduction in code size because

Table 3.5: Lines of code (LOC) comparison between Cilk Plus, CPPWD (the input to the framework), and the generated Verilog code. Blank lines and comments are not counted.

Benchmark	Lines of Code			Ratio
	Cilk Plus	CPPWD	Verilog	Verilog / CPPWD
nw	131	147	9,534	64.9
quicksort	87	121	8,960	74.0
cilksort	207	366	11,339	31.0
queens	68	80	7,863	98.3
knapsack	41	74	7,922	107.1
uts	92	108	8,801	81.5
bbgemm	28	136	10,206	75.0
bfsqueue	60	72	8,350	116.0
spmvcrs	30	64	8,401	131.3
stencil2d	51	136	10,235	75.3

the framework handles most of the low-level details, rather than requiring the accelerator designers to handle them. In addition, writing high-level CPPWD code typically requires less effort compared to writing RTL because the high-level code is untimed. From the above analysis, we estimate that our framework can provide an order of magnitude gain in accelerator design productivity compared to manually writing RTL. This estimation is consistent with our experience using the framework: most of the benchmark accelerators can be created in one or two days, rather than requiring several weeks as in RTL design.

Compared to writing parallel programs using Cilk Plus, designing accelerators using the framework with CPPWD only requires a small amount of extra code. As we discussed earlier, the extra code is mainly from the needed to manually perform code restructuring and task input/output handling. There are a few benchmarks that show a relatively large increase in code size compared to Cilk Plus, most notably `bbgemm` and `stencil2d`. The reason is that for these two benchmarks we implement algorithm-specific on-chip memory structures (scratchpads and line buffers) in CPPWD in order to optimize the performance of the accelerators, which lead to increased code size.

3.6.3 Hardware Prototype on Today's FPGA

To demonstrate the proposed framework, we implemented a prototype system using the Xilinx Zynq-7000 [7] FPGA SoC on Zedboard. The SoC includes two ARM Cortex-A9 cores and an integrated FPGA fabric equivalent to Artix-7. We implemented the FlexArch template for the FPGA and generated accelerators using the flow described in Section 3.4.3. The Zynq-7000 platform has some limitations compared to future integrated CPU-FPGA platforms that we envision (Figure 3.3). First, the FPGA fabric does not have a shared-cache interface that can be used to implement coherent caches on the FPGA. As a result, we implemented stream buffers instead of L1 caches to connect PEs to the L2 cache, and a few benchmarks that rely on fine-grained cache accesses were not implemented. Second, the bandwidth from the FPGA to the L2 cache is limited by a single ACP port and is much lower than the CPU-to-L2 bandwidth. The memory bandwidth becomes a bottleneck when scaling up the number of PEs.

We compare the performance of the accelerators to an optimized parallel Cilk Plus implementation of the benchmarks running on the two ARM cores on the SoC. Figure 3.11 shows the performance of FPGA accelerators with 4

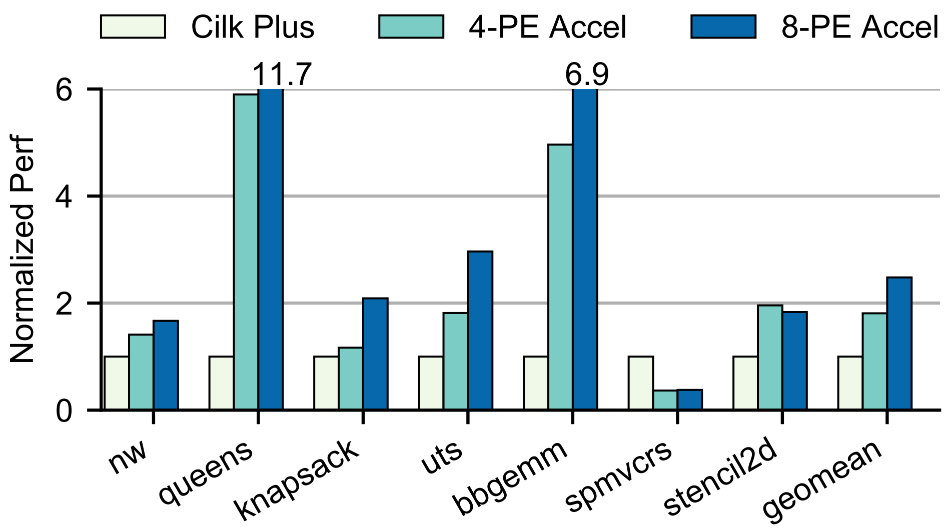


Figure 3.11: Accelerators performance compared to parallel software on Zedboard.

PEs and 8 PEs, normalized to the parallel software implementation. The results show that the 4-PE accelerators achieve up to 5.9x speedup over parallel software (geomean 1.8x), and the 8-PE accelerators achieve up to 11.7x speedup (geomean 2.5x). The results also reveal the limitations of the Zynq-7000 platform. For example, the accelerators show a slowdown for `spmvcrs`, which is a memory-bound benchmark, because the FPGA has lower memory bandwidth to the L2 cache compared to the ARM cores. Similarly, there is little performance improvement for `nw`, `spmvcrs`, and `stencil2d` when increasing the number of PEs, again due to limited memory bandwidth.

3.6.4 Simulation Methodology

The limitations of today's FPGA platform makes it difficult to evaluate the proposed architecture in the context of future integrated CPU-FPGA platforms with

Table 3.6: Platform configuration.

Technology	28nm
CPU	ARM ISA, eight-core, four-issue, out-of-order, 32 entries IQ, 96 entries ROB, 1GHz
CPU L1 Cache	L1I/L1D: 32KB, 2-way, 64B line size, 1-cycle hit latency, next-line prefetcher
Accel logic	In FPGA fabric, 200MHz
Accel L1 cache	32KB, 2-way, 64B line size, 400MHz, 1-cycle hit latency, next-line prefetcher
L2 Cache	2MB, 8-way, 1GHz, 10-cycle hit latency, shared between cores and accelerator
DRAM	Single-channel 64-bit DDR3-1600, 12.8GB/s peak bandwidth

support for cache coherent accelerators [104] and higher memory bandwidth. For the rest of the section, we present a simulation-based study, which allows us to further explore the design space and perform a more detailed evaluation.

We model a future integrated CPU-FPGA SoC where the CPU and FPGA share a cache-coherent memory system. The parameters of the platform are shown in Table 3.6. We use gem5 [15] to model the integrated CPU-FPGA SoC. To simulate the accelerators, we modified gem5 by integrating an RTL simulator (Verilator) into gem5 as a `ClockedObject` that is ticked every cycle, similar to gem5's CPU models. We wrote adapters to perform synchronization between gem5's event-based components (memory-system) and the cycle-based accelerator RTL simulator. In this way, we can perform detailed RTL simulations of the accelerators, while retaining the flexibility in configuring the system components such as cores, caches, interconnect, and DRAM.

We estimate FPGA resource utilization by synthesizing the RTL using Vivado targeting Xilinx's 7-series FPGA to obtain LUT/FF count, the number of DSP slices, and the number of block RAMs. We estimate the resource utilization of the accelerator caches using numbers from Xilinx's cache IP [1].

To estimate the energy of the accelerators, we run Vivado's power estimation tool on the synthesized netlist using signal activity factors from RTL simulation. We model the energy of the cores using McPAT [63], using event statistics from gem5 simulations.

3.6.5 Performance Results

Scalability

Here we present the scalability of the proposed accelerator architecture using parallel speedup, which is the speedup of a n-PE implementation over a single PE implementation. In our experiments, we configure each tile to have 4 PEs and simulate up to 8 tiles (32 PEs) for both FlexArch and LiteArch. For comparison, we also show the scalability of the Cilk Plus baseline on 1 to 8 cores. Because a PE is much smaller and lower-power than an out-of-order core, we can fit more PEs than cores in the same area and power budget. On the memory system side, the 8-tile and 8-core configurations have the same number of L1 caches. Table 3.7 and Table 3.8 shows the scalability results for the Cilk Plus software implementation and the accelerators, respectively. Comparing the soft-

Table 3.7: Scalability of Cilk Plus. The numbers are the speedup of a n-core implementation over a single core implementation.

Benchmark	**Cilk Plus on OOO CPU**			
	1-C	2-C	4-C	8-C
nw	1.00	1.74	3.21	5.54
quicksort	1.00	1.91	3.42	5.40
cilksort	1.00	1.98	3.78	7.05
queens	1.00	1.99	3.92	7.65
knapsack	1.00	2.05	3.92	8.20
uts	1.00	1.75	2.81	3.91
bbgemm	1.00	1.99	3.85	7.04
bfsqueue	1.00	1.77	3.11	4.64
spmvcrs	1.00	1.95	3.50	5.45
stencil2d	1.00	1.99	3.85	7.04
geomean	1.00	1.91	3.52	6.04

ware and accelerators results, the accelerators achieve similar speedups (from 1 to 8 cores/PEs) compared to Cilk Plus, which is a state-of-the-art task-based parallel programming framework and runtime. In addition, the accelerators continue to get more speedups with more PEs for most benchmarks. This shows that the proposed accelerator architecture is effective in harnessing the parallelism in applications.

Comparing the two accelerator architectures, LiteArch accelerators match the scalability of the FlexArch accelerators when algorithms map naturally to the data-parallel pattern (`bbgemm`, `bfsqueue`, `spmvcrs` and `stencil2d`). However, for benchmarks that have dynamic data-dependent parallelism or are irregular (parallelized with fork-join or explicit continuation passing), FlexArch

Table 3.8: Scalability of the accelerators. The numbers are the speedup of a n-PE implementation over a single PE implementation.

Benchmark	**Flex Accelerator**						**Lite Accelerator**					
	1-PE	2-PE	4-PE	8-PE	16-PE	32-PE	1-PE	2-PE	4-PE	8-PE	16-PE	32-PE
nw	1.00	1.98	3.69	7.11	13.23	21.19	1.00	1.81	3.09	5.10	7.54	9.90
quicksort	1.00	1.89	3.24	5.15	6.52	6.81	1.00	1.61	2.54	3.46	4.55	5.17
cilksort	1.00	1.99	3.50	6.94	13.66	26.20	N/A	N/A	N/A	N/A	N/A	N/A
queens	1.00	1.89	3.10	6.20	12.12	24.20	1.00	2.00	3.96	7.45	12.08	13.21
knapsack	1.00	1.97	3.22	6.13	12.55	23.94	1.00	1.93	3.80	7.64	15.15	29.99
uts	1.00	1.95	3.66	6.50	11.32	15.64	1.00	1.92	3.52	5.76	7.51	7.44
bbgemm	1.00	1.99	3.88	7.50	13.38	17.48	1.00	1.95	3.42	6.39	11.29	18.27
bfsqueue	1.00	1.78	3.36	6.13	9.93	12.40	1.00	1.56	4.23	6.95	9.99	12.55
spmvcrs	1.00	1.99	3.59	6.86	13.16	16.51	1.00	1.93	2.91	5.52	10.16	17.42
stencil2d	1.00	1.99	3.17	6.22	12.12	20.13	1.00	1.98	2.73	5.36	10.32	17.35
geomean	1.00	1.94	3.43	6.44	11.57	17.35	1.00	1.85	3.31	5.82	9.37	12.98

accelerators generally achieves better scalability, except for `knapsack`. The `knapsack` implementation on LiteArch uses a different algorithm that sacrifices algorithmic efficiency in order to map to parallel-for. Though it has good scalability, we will see later that the absolute performance is actually much lower. These results indicate that LiteArch is adequate to support regular data-parallel algorithms. FlexArch, on the other hand, is a better fit for most other parallel algorithms. This is because although some of these algorithms can be rewritten to map to LiteArch, their dynamic and irregular nature makes the implementation less efficient, due to less effective load balancing and/or reduced algorithmic efficiency.

The results also show that some benchmarks have better scalability than others. For example, the two sorting algorithms, `cilksort` and `quicksort`, show similar speedups when there are only a small number of cores/PEs. However, when the number of cores/PEs increases, `cilksort` can continue to scale its performance, achieving 26.20x speedup with 32 PEs using FlexArch, while the performance of `quicksort` quickly tapers off. The reason is that these two algorithms have a different amount of dynamic parallelism. `quicksort` has a significant non-parallelizable portion. Specifically, the partitioning step is performed serially, thus the achievable speedup is limited by Amdahl's law. In contrast, `cilksort` (a.k.a. parallel merge sort) generates a large number of parallel tasks during execution, hence it achieves better scalability.

The results also indicate that the FlexArch architecture achieves good load balancing using its hardware-based work stealing mechanism. For example, `uts` (Unbalanced Tree Search) is particularly difficult to load balance and requires frequent work stealing operations. Cilk Plus only achieves 3.91x speedup with 8 cores. In comparison, the FlexArch accelerator achieves 6.50x speedup with 8 PEs, and is able to continue to scale the performance with more PEs, which demonstrates the efficiency of the hardware-based work stealing mechanism.

Normalized Performance

Figure 3.12 shows the performance of FPGA accelerators normalized to a single out-of-order core. The horizontal line represents the performance of parallel software using Cilk Plus running on eight cores. The crosspoint represents the number of PEs that is needed to achieve the performance of the 8-core parallel software implementation. The results show that the accelerators outperform the

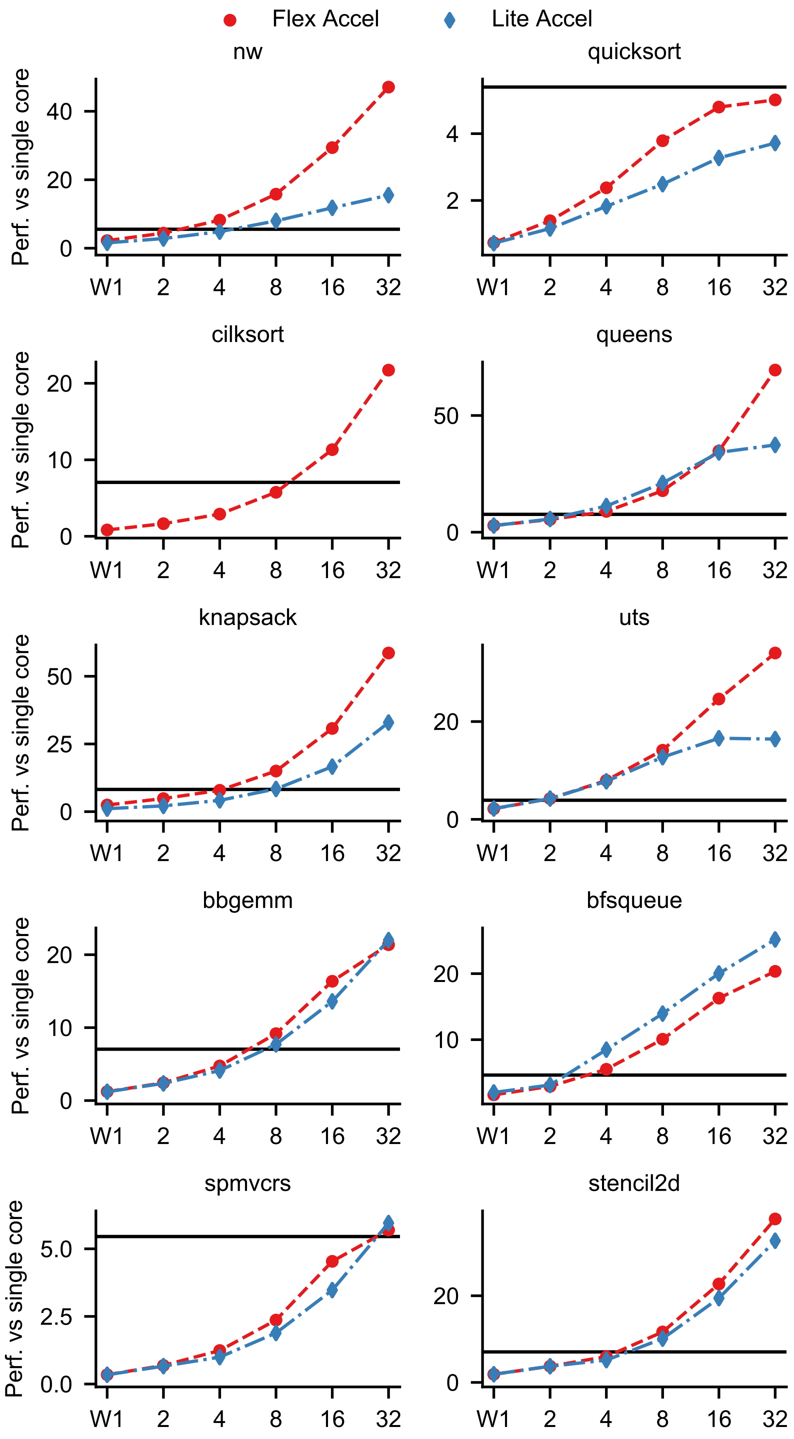


Figure 3.12: Normalized accelerator performance. The x-axis is the number of workers (PEs). The y-axis is performance normalized to a single OOO core. The horizontal bar indicates the performance of an eight-core Cilk Plus implementation.

8-core software implementation for most benchmarks. When using 32 PEs, the FlexArch accelerators are up to 9.1x (geomean 4.0x) faster than eight cores, and up to 69.5x (geomean 24.1x) faster than a single core. The accelerators cannot significantly outperform the 8-core software implementation for `quicksort` and `spmvcrs`. As discussed earlier, `quicksort` has a significant serial portion, so the processor with a high frequency runs faster. `spmvcrs` is limited by memory bandwidth, as a result all implementations eventually reach similar performance.

The LiteArch accelerators achieve similar performance as the FlexArch accelerators for data-parallel benchmarks. For most other benchmarks, FlexArch significantly outperforms LiteArch, especially with a large number of PEs. Also note that the performance difference of `knapsack` comes from the algorithmic inefficiency as discussed earlier.

These results also demonstrate that FPGA accelerators need to exploit parallelism in order to provide performance benefits over parallel software. Traditional HLS tools that use sequential C/C++ code as input can only generate accelerators that roughly match the performance of a single PE, which is often slower than parallel software. Our framework enables mapping parallel algorithms to FPGA easily and achieves compelling performance (and shown later, energy) advantages over parallel software.

Execution Time Breakdown

Figure 3.13 shows the execution time breakdown of the FlexArch accelerators. The percentage is calculated by aggregating the execution time across all PEs.

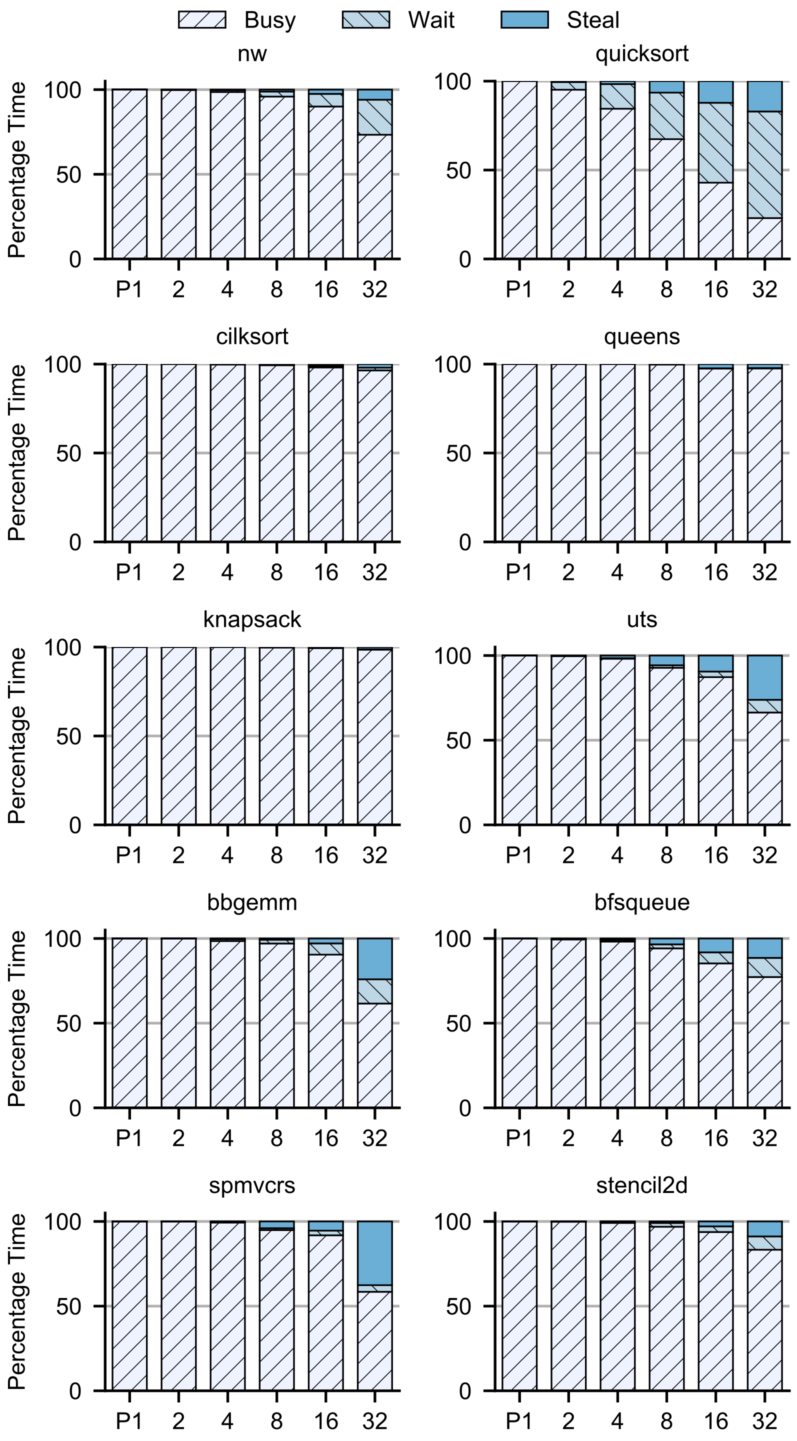


Figure 3.13: Execution time breakdown for FlexArch accelerators. Busy: PE is actively processing a task. Wait: PE is waiting to steal a task. Steal: PE is performing task stealing.

Execution time is divided into three components: busy, wait, and steal. The busy component represents that a PE is actively processing a task. The wait component represents that a PE is waiting to initiate task stealing. This is because task stealing is throttled when there are few available tasks in the system to prevent congestion in the work stealing networks. The steal component represents that a PE is performing task stealing. The overall trend seen from the graphs is that as we increase the number of PEs, the percentage of time spent in work stealing operations (wait and steal) increases. There are mainly two reasons. First, as we increase the number of processing elements, the serial portion of a program will become a larger percentage of overall execution time according to Amdahl's law. This shows up as time spent in work stealing because the PEs repeatedly try to steal work from each other but there are simply not enough parallel tasks available in the system. Second, as the number of PEs increases, it becomes more challenging to balance the load on the PEs, and thus require more work stealing operations.

Among the benchmarks, `quicksort` spends the most time in work stealing operations when increasing the number of PEs because it has a significant serial portion. `uts` and `spmvcrs` also spend a sizeable amount time in work stealing because of the irregularities in the computation. `nw` and `bbgemm` spends more time in work stealing for large accelerator configurations (32-PE) because there is not enough parallelism to keep all PEs busy. Note that we use the same problem size for all accelerator configurations in order to keep the results consistent with the performance numbers shown earlier. In reality, as pointed out in Gustafson's law [47], programmers tend to use larger problem sizes as more computing resources become available. The reasoning is that programmers will use the improved computing power to solve larger problems in the same amount of time, rather than trying to minimize the execution time for a fixed-size problem. Because the available parallelism in an application grows with the problem size, we expect that in reality less percentage of execution time will be spent in the work stealing operations compared to shown in the figure for large accelerator configurations.

Task Queue and P-Store Occupancy

Figure 3.14 shows the maximum number of occupied entries in any task queue or P-Store at any point of execution for the FlexArch accelerators. Most benchmarks only need a small number of task queue and P-Store entries. This is because when the task graph is balanced, the space required is logarithmic to the problem size, and thus is small even for very large problems. The only benchmark that requires significantly more space is `uts`, because its task graph is highly unbalanced. Nonetheless, the number of entries it requires can still easily fit into on-chip memories. For example, we use the block RAMs on FPGAs to implement the task queues and P-Stores. The smallest configuration of the block RAMs has 512 entries [116], which is already sufficient to meet the requirements of all benchmarks we experimented with. When needed, the blocks RAMs can also be sized up to support larger task queue and P-Store sizes.

Another trend that we can see from the figure is that increasing the number of PEs does not require larger task queues. This confirms the space bound property discussed in Section 3.2.3 and Section 3.3.1, which states that the total space required for a parallel execution with P processing elements is at most S_1P, where S_1 is the space required for a serial execution. Because the total number of available task queue entries grows with the number of PEs, we can

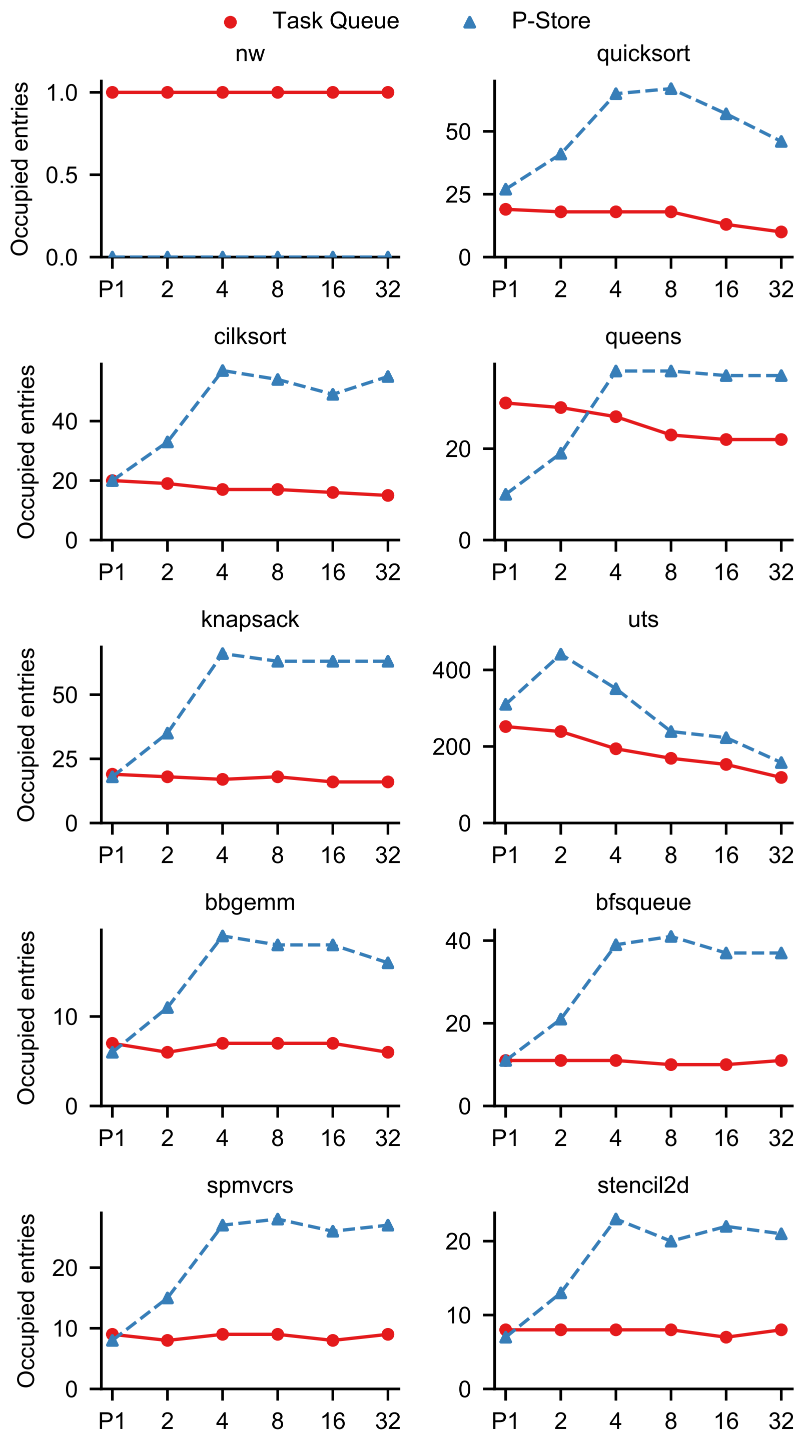


Figure 3.14: Maximum Task Queue and P-Store Occupancy for FlexArch accelerators. The x-axis is the number of PEs. The y-axis is the maximum number of occupied entries in any task queue or P-store at any point of execution.

Table 3.9: FlexArch accelerators resource utilization. Each tile consists of four PEs and a cache. DSPs are shown in the number of DSP48 slices. BRAMs are shown in the number of RAM18's (each RAM36 counts as two RAM18's).

Benchmark	**Flex PE**				**Flex Tile (incl. Cache)**			
	LUT	FF	DSP	RAM	LUT	FF	DSP	RAM
nw	1487	1547	3	7	8914	8668	12	51
quicksort	1828	1484	0	6	10618	8484	0	47
cilksort	5961	3785	0	8	27233	17622	0	58
queens	549	535	0	4	5744	4684	0	40
knapsack	737	770	5	5	6083	5674	20	45
uts	2227	2216	0	5	11510	11438	0	44
bbgemm	1551	1789	15	19	9671	9620	60	100
bfsqueue	1481	1190	0	6	9353	7348	0	48
spmvcrs	1441	1273	3	13	9303	7660	12	76
stencil2d	1741	2334	12	10	10316	11905	48	64

increase the number of PEs without increasing the number of entries in each PE's task queue.

The space bound also applies to the P-Store. In our experiments, each tile has up to four PEs. Because the P-Store is shared among the PEs in a tile, as we increase the number of PEs from one to four, the P-Store occupancy grows because of the increasing sharing. As we further increase the number of PEs, the P-Store occupancy does not grow further because of the space bound. Note that `nw` does not use the P-Store, so the occupancy is always zero.

Table 3.10: LiteArch accelerators resource utilization. Each tile consists of four PEs and a cache.

Benchmark	Lite PE				Lite Tile (incl. Cache)			
	LUT	FF	DSP	RAM	LUT	FF	DSP	RAM
nw	1273	1346	1	4	6431	6838	4	36
quicksort	1857	1490	0	2	8665	7387	0	28
cilksort	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
queens	704	606	0	0	4164	3851	0	20
knapsack	575	466	0	0	3591	3295	0	20
uts	2541	2158	0	0	10997	10063	0	20
bbgemm	1019	1361	15	14	5401	6736	60	76
bfsqueue	887	822	0	1	4901	4791	0	24
spmvcrs	875	905	3	8	4777	5119	12	52
stencil2d	1200	1964	12	5	6175	9359	48	40

3.6.6 Resource Utilization

Table 3.9 and Table 3.10 shows the per-PE and per-tile resource utilization of the FlexArch and LiteArch accelerators, respectively. Each tile consists of four PEs and a cache. The DSP blocks are mainly used to implement multipliers, and the BRAMs are used as local scratchpads and buffers, task storage, and caches. The results show that the LiteArch accelerators generally use fewer resources than the FlexArch accelerators. The reduction is most apparent for regular data-parallel benchmarks (`bbgemm`, `bfsqueue`, `spmvcrs`, and `stencil2d`) whose tasks graphs can be determined statically. The reason is that these benchmarks consist of a single parallel range whose task graph can be determined statically, thus the LiteArch accelerators avoid the cost of dynamically splitting the range and generating the task graph in hardware, as done in the FlexArch accelerators. As a result, LiteArch is a good fit for data-parallel benchmarks. On the other hand, the resource reduction by using LiteArch for other benchmarks is less significant, as the task graphs need to be constructed dynamically in both architectures. The FlexArch architecture is a better fit for these benchmarks, due to its ability to perform efficient load balancing and thus achieve significantly better performance.

To put the resource utilization numbers into context, we studied how many PEs can be mapped to typical FPGA devices. We experimented with two FPGA devices: a low-cost FPGA (Artix XC7A75T) similar to the one on Zedboard, and a mainstream FPGA (Kintex XC7K160T). The low-cost FPGA can fit on average 4 tiles (16 PEs) for FlexArch, and 5 tiles (20 PEs) for LiteArch. The mainstream FPGA can fit 8 tiles (32 PEs) for most benchmarks (except for `cilksort`) for both FlexArch and LiteArch.

3.6.7 Power and Energy Efficiency

Figure 3.15 shows the performance and energy efficiency of the accelerators (16-PE configuration) normalized to a Cilk Plus implementation on eight out-of-order cores. The results show that the proposed accelerators are lower power and more energy efficient for all benchmarks, with most benchmarks showing more than 10x gain in energy efficiency. Comparing the two accelerator architectures, there exists a clear trend in the performance/energy efficiency profile: FlexArch usually achieves better performance, while LiteArch often has better energy efficiency. On average, FlexArch achieves a normalized energy efficiency of 11.8x compared to the out-of-order cores, while LiteArch achieves 15.3x.

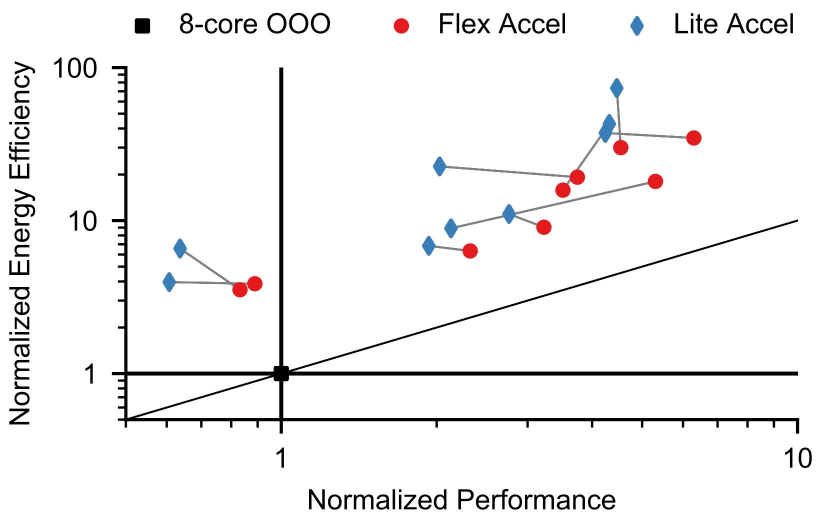


Figure 3.15: Normalized performance and energy efficiency. Energy efficiency is the inverse of energy consumption. Both performance and energy efficiency are normalized to the Cilk Plus implementation on 8 OOO cores. Points to the right of the vertical line have better performance. Points above the horizontal line have better energy efficiency. The diagonal line represents the iso-power line. Points above the diagonal line have lower power. Points for the same benchmark are linked. Note that both axes are in log scale.

3.6.8 Cache Size Customization

In our evaluation, the accelerator L1 cache (tile cache) are built using the BRAMs in the FPGA fabric. The size of the cache can be customized according to application characteristics. For benchmarks that are not memory intensive, or have good locality, the cache sizes can be made smaller to reduce BRAM usage without significantly degrading performance. Figure 3.16 shows the performance of the FlexArch accelerators (16-PE configuration) when varying the L1 cache size from 4kB to 32kB. The benchmarks that have an irregular memory access pattern (`bfsqueue` and `spmvcrs`) show the largest performance loss. `nw` and bbgemm also showed some performance loss because of the reduced temporal reuse with smaller cache sizes. The other benchmarks perform relatively well even with a small cache size. Among them, cilksort, quicksort, and stencil2d have good locality, and the other three have relatively low memory intensity.

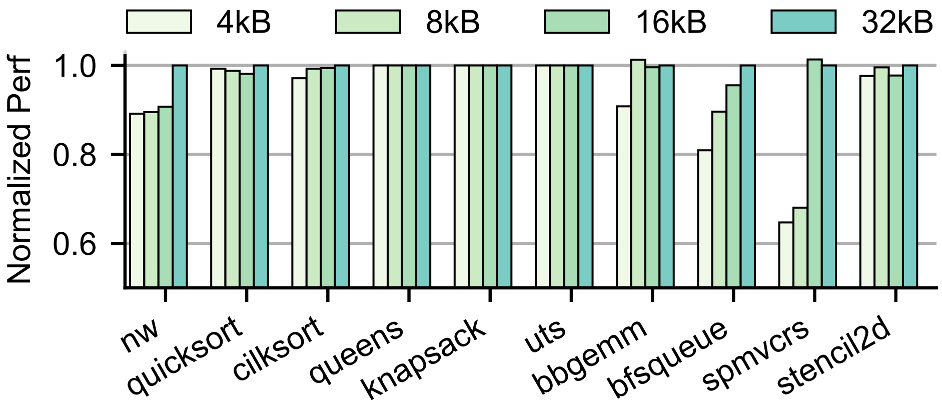


Figure 3.16: Performance when varying accelerator L1 cache size.

3.6.9 Parallel Software with Unified Description

The benchmark applications presented earlier in this evaluation are already written in the unified description format. The only exception is UTS, which contains an RTL component for SHA-1 calculation. We compiled all benchmarks except for UTS with the CPPWD-TBB library and the TBB runtime to obtain the executables. To evaluate the performance of parallel programs built using the unified description, we compare them with two other parallel software implementations. The first is the Cilk Plus implementation described earlier. The second is a native TBB implementation we coded without using the unified description. All three implementations are compiled with the same compiler flags

(`-O3` optimization and auto-vectorization).

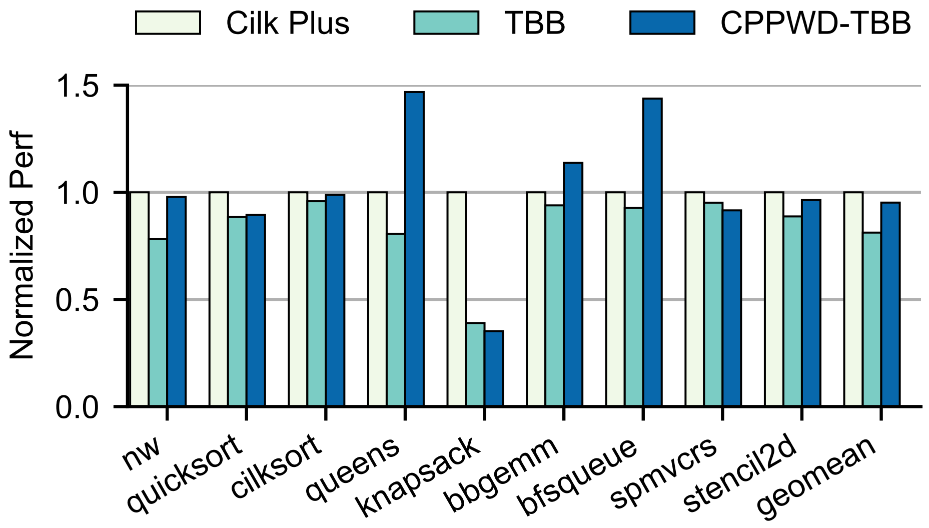


Figure 3.17: Performance comparison between parallel software implementations.

Figure 3.17 compares the performance of all implementations normalized to Cilk Plus. Overall, Cilk Plus has the highest performance. The performance of the implementation using the unified description (CPPWD) on TBB runtime is about 5% lower than Cilk Plus, while native TBB is about 18% lower. Native TBB has lower performance likely because it uses stalling scheduling, which is known to less efficient [94].

For most benchmarks, the performance of all implementations is similar. There are a few exceptions. Cilk Plus has much higher performance than both TBB and CPPWD-TBB on `knapsack`, likely because it uses continuation stealing, which has lower overhead. CPPWD-TBB has higher performance than Cilk Plus and TBB on `queens` and `bfsqueue`. This is because the CPPWD version initially written for hardware implementation copies data into local arrays, which turns out to have better locality when implemented in software as well.

In summary, the evaluation results show that the performance of the unified description running on TBB runtime is competitive with state-of-the-art task-based parallel programming frameworks.

CHAPTER 4

PREDICTIVE DVFS FRAMEWORK FOR ENERGY EFFICIENCY

4.1 Introduction

This chapter proposes an architectural framework that enables accelerators to perform intelligent dynamic voltage and frequency scaling (DVFS) to achieve good energy-efficiency for interactive and real-time applications. Energy savings are achieved by accurately adjusting the voltage and frequency levels of accelerators to match the computation demand. One key challenge of performing DVFS is to accurately predict the computation demand ahead of time. The framework automatically generates application-specific predictors by analyzing the source code of accelerators and build prediction models using machine learning techniques, which removes the need to manually tune the predictor, and at the same time achieves better accuracy and more energy savings.

Modern SoCs often contain hardware accelerators that interacts with the user or perform real-time processing of tasks, such as video codecs, image signal processors in cameras, and computer vision accelerators in self-driving cars. Although more energy-efficient than general-purpose processors, the power consumption of hardware accelerators is becoming a concern as applications demand more computational power. For example, the DaDianNao machine learning accelerator consumes 15.97W [25]. The FPGAs used to accelerate the Bing web search engine consumes up to 22.7W per chip [87], and the FPGAs used to accelerate neural networks in Microsoft's datacenters consume 125W per chip [31]. The power consumption of accelerators is especially a concern for battery-powered mobile devices, where power directly impacts battery life, as well as datacenters, where power not only affects electricity costs, but also affects cooling and power distribution costs. Fortunately, the interactive nature of many accelerated applications, as well as the variations in the workload, means that the accelerators do not need to operate at the peak performance levels at all times. Thus, it is important to dynamically adjust the performance level of accelerators to achieve good energy efficiency.

Dynamically adjusting the performance level is subject to a few constraints. Accelerators used in interactive or real-time applications have response time requirements. Interactive applications are required to respond to user inputs by a certain deadline for responsiveness. Real-time applications such as frame-based applications need to process each frame before the screen refresh deadline, otherwise the frame will be dropped, and the application will feel sluggish. In both cases, meeting response time requirements is essential for good user experience. On the other hand, finishing tasks earlier than the response time requirements does not improve user experience due to the limits of human perception. In other words, there often exists slack in interactive or real-time applications, which can be exploited to improve energy-efficiency by lowering the performance level of the system, using techniques such as dynamic voltage and frequency scaling (DVFS). Unfortunately, today's software/hardware abstractions do not offer applications an easy way to express their timing requirements. As a result, the hardware accelerators/IP blocks are usually agnostic to the timing requirement of the applications, and operate in best-effort mode. When there is a high variation in the workload, the hardware accelerators need to operate conservatively to make sure they meet deadlines even in the worst-case. This means they often run at unnecessarily high performance levels compared to what is needed to meet deadlines in the average case, which leads to wasted energy.

Various techniques exist for exploiting such slack in the software part of applications, using either reactive approaches [18, 30, 45, 68, 83] or predictive approaches [46, 53, 119, 120]. However, exploiting slack to inform fine-grained DVFS has not been studied much for hardware accelerators.

In this chapter, we present an architectural framework for automatically generating fine-grained DVFS controllers for accelerators that exploits input-dependent variations. We observe that input-dependent control decisions are the major source of execution time variations. A good estimate of an accelerator's execution time can be obtained if its control decisions for a certain input are known. In order to do this, we propose an automatic flow to generate a minimal version of a hardware accelerator from its RTL description, which computes the control decision features given an input. A model based on convex optimization is developed and trained to map these features to the accelerator's execution time and the most efficient DVFS level to run the accelerator at.

Our approach is general and applicable to a wide range of hardware accelerators. We implemented and tested this predictive DVFS framework on a number of accelerators including video decoding, image processing, encryption, physics computation, etc. Our results show the framework can generate DVFS controllers that make more accurate predictions and achieve more energy savings than manually designed controllers, while being fully automatic.

The rest of this chapter is organized as follows. Section 4.2 discusses execution time variations in hardware accelerators and existing DVFS controllers. Section 4.3 describes our predictive DVFS framework, including the features used, prediction model, method to generate the predictor, and DVFS model. We also include a case study on using the framework for an example accelerator. Section 4.4 discusses our evaluation methodology, experimental setup, and evaluation results.

4.2 Fine-grained DVFS for Hardware Accelerators

4.2.1 System Setup

The system we consider in this chapter consists of general-purpose processor cores, caches, main memory, and hardware accelerators. The cores and accelerators are loosely coupled. The accelerators access memory through the shared last-level cache or memory bus. Each accelerator contains computation logic, and often internal scratchpad memories to store the working set. We assume each accelerator's DVFS level can be controlled individually.

4.2.2 Tasks and Jobs

Here we define some terminologies used in this chapter. A *task* is a piece of work that has an associated deadline. For example, for a video player, decoding and rendering a frame is a task. In this case, the deadline associated with a task is determined by the frame rate of the video. A *job* is a dynamic instance of a task. For example, decoding a video at 60fps executes 60 jobs every second. Figure 4.1 shows a sequence of jobs for a task.

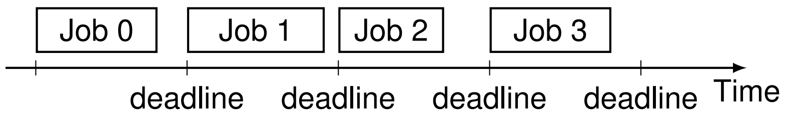


Figure 4.1: A sequence of jobs for a task.

4.2.3 Execution Time Variation

The execution time for each job can vary depending on the job's input. For example, Figure 4.2 shows the execution time for a hardware H.264 decoder when decoding three video clips of the same resolution. We can see that even though all frames have the same resolution, there is a large variation in job execution time for frames in different videos, or even between frames in the same video. The reason for such large execution time variations is that for each frame, depending on the content in it, the H.264 algorithm chooses different modes to encode each macroblock in a frame, which leads to different computation complexity for decoding, and thus different execution time. Note that if we take into account videos of different resolutions, the execution time variation will be even larger. If we can lower the frequency for frames with shorter execution time, significant energy savings can be achieved.

However, setting an appropriate DVFS level for each job is not easy. The large and irregular variations in workload make it difficult to predict the next job's execution time. Without accurate prediction, the DVFS controller has to be conservative and use higher DVFS levels to avoid deadline misses, missing opportunities for energy reduction. Otherwise, the DVFS controller risks missing deadlines when there is a sudden increase in job execution time.

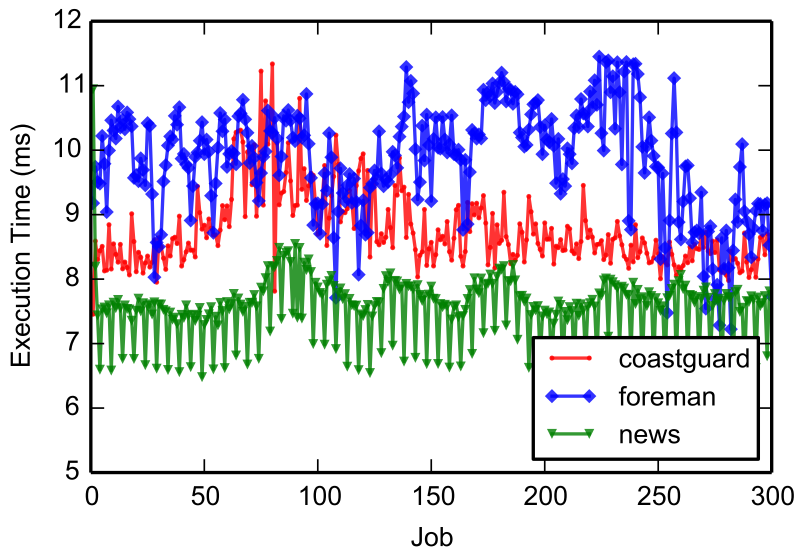


Figure 4.2: Execution time of H.264 decoder for three video clips at 60fps. Each point is one job (frame).

4.2.4 Current Approaches to DVFS

DVFS is widely used for reducing the energy of computation. The key idea of DVFS is to reduce voltage and frequency to provide "just enough" performance to the application. An important part of a DVFS controller is the prediction of future workload, which allows the voltage and frequency to be lowered to the minimum required level while satisfying response time requirements.

For applications without response time requirements, simple interval-based scheduling algorithms [111] can be used. These algorithms usually divide time into fixed-length intervals and measure the utilization of the previous interval and set DVFS level for the next interval. Since response time is not a requirement, some level of performance degradation caused by workload misprediction can be tolerated. These algorithms are widely used in operating systems.

For example, the Linux power governors [18] are interval-based.

For applications with response time requirements, misprediction has to be minimized since it degrades the quality-of-service (QoS). There are many approaches in literature and industry practice to perform DVFS under response time requirements. The following summarizes approaches that can be applied to hardware accelerators.

Table-based: Some hardware accelerators, including the Multi-Format Codec (MFC) in Samsung Exynos Series SoCs, use a lookup table to determine the DVFS level [2]. The table is addressed by a coarse-grained parameter, such as the resolution of a video. Before decoding a video, the driver will look into the table and set a DVFS level for the entire video sequence. Researchers have also studied using the type of frames as inputs to the table [105]. However, these approaches do not take into account fine-grained job-to-job execution time variations. Essentially, these approaches set DVFS levels to the worst case for that coarse-grained parameter used to index the table. As can be seen in Figure 4.2, though all jobs have the same coarse-grained parameter (resolution in this case), most jobs have much shorter execution time than the worst-case. Thus the coarse-grained approach misses opportunities for energy reduction.

Reactive Control: A number of previous studies proposed using reactive control approaches to adjust DVFS levels. Some studies investigated using job execution time history to predict future job execution time, and set DVFS levels accordingly [30, 74]. Others proposed using control theory-based approaches, such as PID control [45, 68, 83]. Most of these studies target software-based systems, but some of them also consider hardware accelerators [74, 105]. These approaches are simple to implement, and work well for applications whose ex-

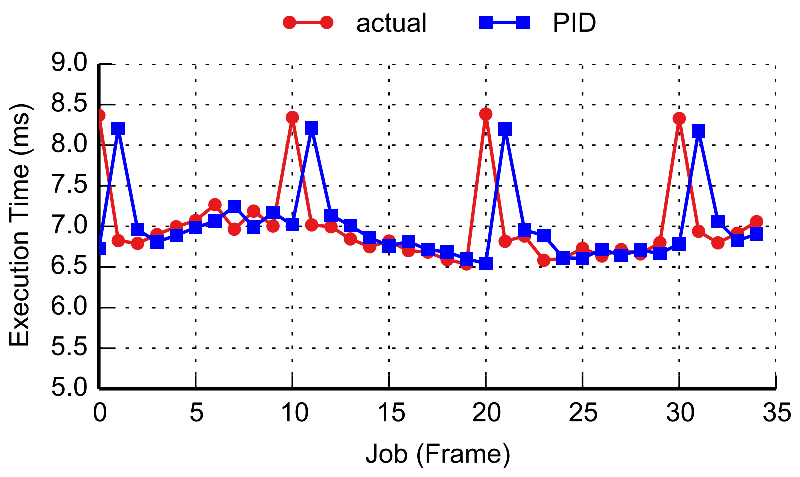


Figure 4.3: Actual execution time and execution time predicted by PID controller for H.264.

ecution time varies slowly with time. However, many applications and accelerators do not fall into this category. For applications with rapid changes in job-to-job execution time, reactive decisions to adjust DVFS levels tend to lag behind actual changes, leading to deadline misses. Figure 4.3 shows an example how a PID-based controller mis-predicts job execution time for H.264 video decoding. When actual execution time shows spikes occasionally, the PID controller's prediction lags behind by one frame, causing one under-prediction and one over-prediction, which leads to one job missing deadline and one job running at unnecessarily high frequency around the spike. Apart from lagging behind in decision making, reactive control approaches can not be applied in some cases. For example, when browsing a website, the images processed by the JPEG decoding accelerator usually do not show correlation with previous or next images, rendering any reactive control approaches ineffective.

Predictive Control: There have been a few studies that looked at using predictive approaches to predict execution time and set DVFS levels accordingly.

The target applications include interactive games [46], video players [95], web browsers [119, 120] and servers [53]. Predictive control has been shown to outperform reactive control for these applications. However, all of these studies target software-based systems. Moreover, these approaches use application-specific features for prediction, which require domain-specific knowledge and manual effort to identify and obtain. Predictive control of DVFS for hardware accelerators is largely unexplored. As more and more hardware accelerators are deployed in applications today, it is necessary to take them into account in controlling DVFS. In this chapter, we propose a predictive DVFS framework for hardware accelerators. In addition, our prediction framework uses features automatically extracted from hardware, which eliminates the need for domain-specific knowledge or manual effort in obtaining features.

4.3 Predictive DVFS Framework for Hardware Accelerators

In this section, we propose a framework for controlling accelerator DVFS based on execution time prediction. At high level, our goal is to predict the lowest DVFS level each job can run at without violating response time requirements. This can be done in two steps: first, we predict the execution time for each job at a nominal frequency (i.e. without doing DVFS). After that, we predict what the execution time would be at each DVFS level, and choose the lowest level that meets response time requirements.

Figure 4.4 illustrates how accelerators operate with predictive DVFS. For each job, a predictor is run first to obtain an estimate of the execution time of the job. Then the best DVFS level is set according to predicted execution time. Af-

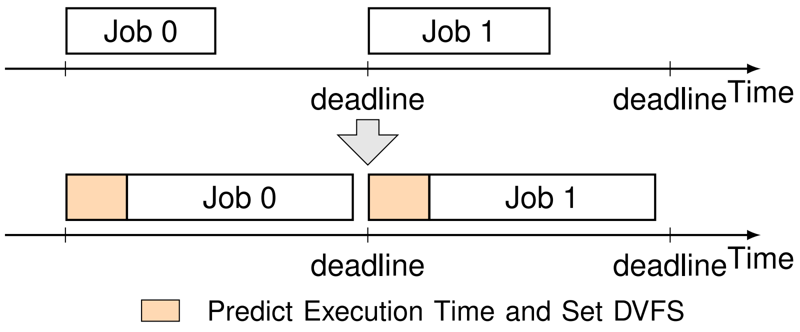


Figure 4.4: Operation of predictive DVFS.

ter frequency and voltage change stabilizes, the accelerator's main logic starts execution. Figure 4.5 shows the block diagram of an accelerator with execution time prediction-based DVFS controller. For each job, the predictor informs the clock generator and voltage regulator the frequency and voltage to be used. Access to the scratchpad memory is time-multiplexed between the predictor and the main computation logic. Although in our implementation the predictor directly controls DVFS circuitry, the control can also be done in software through the operating system.

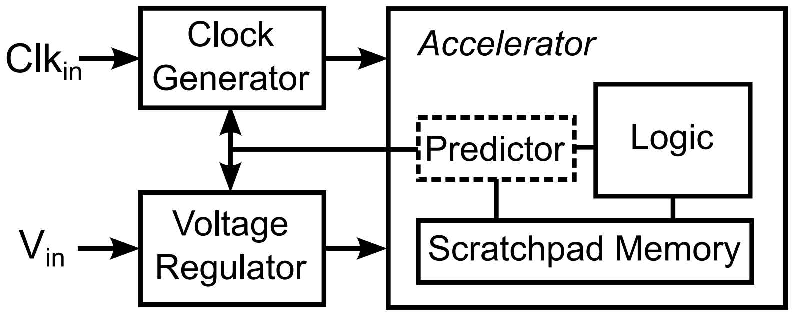


Figure 4.5: Accelerator with execution time prediction-based DVFS.

We have the following design goals for the DVFS framwwork:

- **Look-ahead**: Instead of reacting to changes in job execution time, the

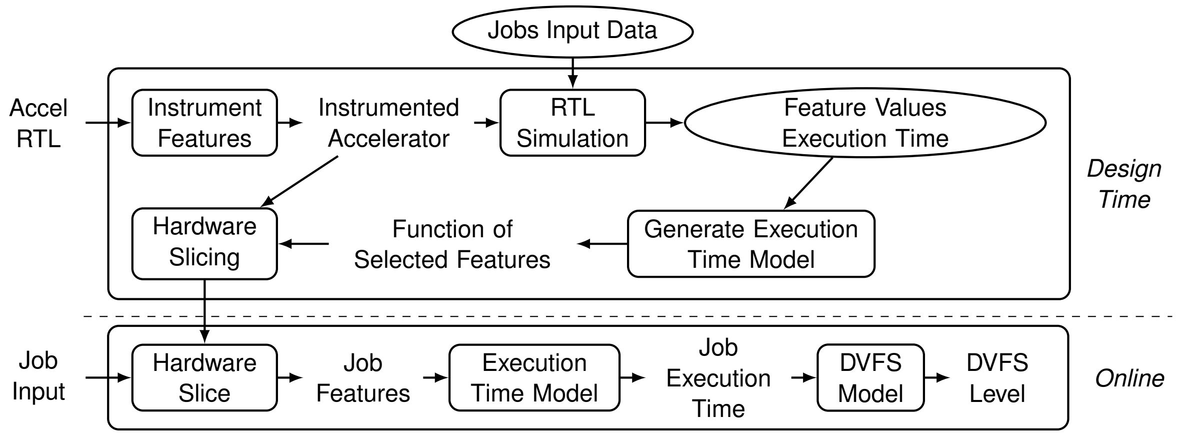


Figure 4.6: Execution time prediction flow.

DVFS controller looks ahead into upcoming jobs and predicts what the execution time would be before actually running the jobs.

- **General**: The DVFS framework should be general and applicable to a wide range of accelerators. To this end, the framework does not use application-specific knowledge.
- **Automated**: The DVFS controller should be generated by an automated flow with minimal manual effort.
- **Low overhead**: The DVFS controller should have low overhead in terms of area and energy, or increased design complexity.

Figure 4.6 shows the high-level flow for our DVFS framework based on execution time prediction. It consists of two parts. The offline part works during the design process of the accelerator to generate the predictor. The online part shows the operation of the predictor during accelerator execution.

Although we only investigate DVFS in this chapter, this approach can also be applied to other methods for performance-energy trade-off, such as dynamically reconfiguring accelerators to different performance-energy points, etc.

4.3.1 Source of Execution Time Variation

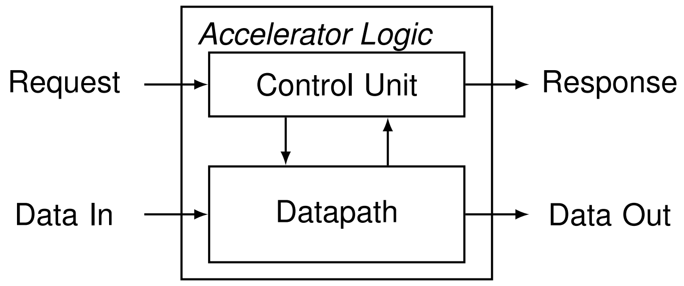


Figure 4.7: Control-Datapath structure of an accelerator.

In Section 4.2.3, we showed that accelerators can have significant input-dependent execution time variations. Here we describe where these variations come from. Figure 4.7 shows a typical high-level structure of an accelerator. It mainly consists of two parts: control unit and datapath. The control unit is responsible for handling requests and responses, as well as orchestrating computation in the datapath by generating various control signals. The datapath performs computation on the input data to generate the output, and also generates various signals for the control unit to make decisions.

The control unit is usually composed of one or more Finite State Machines (FSMs). Figure 4.8 shows an example FSM from the control unit of an accelerator. In state S1, the accelerator reads a piece of input data. Then, depending on the value, the FSM transitions to either state S2 or S3 to perform computation. When computation is done, the FSM transitions to state S4 to generate an output, then transitions back to S1 to process the next input. The computation

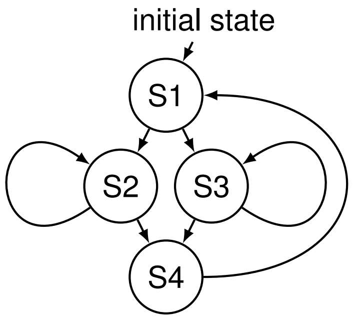


Figure 4.8: Example Finite State Machine in control unit.

in state S2 and S3 can take different number of cycles (for example, 50 and 100 respectively). This is the major source of execution time variation.

A job usually uses multiple inputs. For example, an image consists of multiple macroblocks. If we know the decisions made by the control FSMs when processing each input, and the processing time of each computation state, we can predict execution time, and consequently the best DVFS level to run the job at.

4.3.2 Features from Hardware Accelerators

In this section, we describe the features we use to represent the decisions of the control unit. Here a feature refers to a measurable property that can be extracted during accelerator execution. We also show how these features and the control decisions they represent correlate with execution time.

We observe that control unit decisions are embedded in the state transitions of the control unit FSM. For example, in Figure 4.8, a state transition from S1 to S2 means the control unit decides to perform some computation associated with state S2. If we count the number of state transitions from S1 to S2, we can know the number of times computation associated with S2 has taken place. Thus, we use *state transition count* (STC) as a type of feature in our prediction model.

However, knowing state transition counts is not enough. We also need to know how each state transition impacts execution time of the accelerator. This can be divided into two cases: If the latency of a computation state is fixed, we can use statistical regression to figure out how much time the computation takes given enough training data. If the latency is variable depending on input, statistical regression can only estimate the average latency, which is not enough to make good predictions. We observe that in this case, there is usually a counter to keep track of whether the computation is finished. For example, when the computation starts, the control unit sets the counter to the latency of the computation. In each cycle the counter is decremented until it reaches zero, signaling the end of computation. The *range* of the counter correlates with the computation's impact on execution time. In a decrementing counter, range can be obtained from the counter's initial value. In an incrementing counter, range can be obtained from the counter's final value before a reset. Thus, we use several counter-related features: 1) *initialization count* (IC), which is the number of times each counter is initialized. 2) *average initial value* (AIV), which is the average value a counter is initialized to. 3) *average pre-reset value* (APV), which is the average of a counter's final value before a reset. The last two features capture the range of each counter. Table 4.1 summarizes the features we use in our prediction model.

Table 4.1: Summary of features in prediction model.

Feature	**Source**	**Granularity**
STC	FSM	Each source-destination states pair
IC	Counter	Each counter
AIV	Counter	Each counter
APV	Counter	Each counter

4.3.3 Identifying and Obtaining Features

Manually annotating and modifying FSMs and counters in hardware accelerator designs would be too tedious and not feasible for large designs. Moreover, many accelerators are third-party IPs and system designers are not familiar with their internals. Thus, we propose an automated approach to identify and extract such features in hardware accelerators based on a static analysis of RTL code of accelerators.

The first step of the analysis is to identify FSMs and counters in the RTL, as these are the sources of features. To achieve this, a behavioral RTL of hardware accelerators is first transformed to a structural RTL using a synthesis tool. We use Yosys [112], which is an open-source synthesis suite. After that, we use an algorithm to find FSMs in the design based on techniques from a previous study [99] on extracting FSMs from a gate-level netlist. The algorithm works by analyzing the RTL and look for specific structures related to FSMs. Similar to identifying FSMs, counters are also identified by RTL analysis.

The next step is to instrument the accelerator so that it records feature values during its operation, as illustrated in the offline stage of Figure 4.6. This is done through RTL analysis and instrumentation. For state transition counts, we extract each FSM's transition table and compute the criteria for each state transition to take place. For each source-destination pair, we instrument the RTL to generate a signal whenever the transition criteria is met, and record the number of times the signal is asserted using a register. With this, we can simply read out the register's value to get a state transition count. Similarly, for initialization counts, we compute the criteria for the counter to be initialized and instrument the RTL to generate a signal when the criteria is met. To keep track of an average initial value and an average pre-reset value, we use registers which are controlled by the initialization criteria. Note that we do not actually have to calculate the average, it is sufficient to record the sum of these values and the prediction model will take care of scaling the values to obtain average. All these steps are done automatically without manual effort using our RTL analysis and instrumentation framework implemented inside the Yosys open-source Verilog analysis and synthesis suite.

After instrumenting the accelerator, we run RTL simulations with a training set of job input data to obtain the feature values as well as execution time for each job.

4.3.4 Prediction Model

After obtaining the features and execution time for each job, we develop a model that takes feature values and maps them to execution time. The model is then trained using the feature values and execution time data from training runs. We have the following requirements for our prediction model: (1) Accurate prediction. (2) Low overhead in terms of time, area, and energy. A simple model which uses a small number of features is preferred. (3) Conservative prediction, which means when there is a trade-off to be made between a deadline miss and less energy savings, the model should avoid deadline miss even though it may use more energy.

With these requirements in mind, we use a linear model to map feature values to execution time. Linear models are very simple to evaluate at runtime by calculating the dot product of feature values and model coefficients, which is just a series of multiply accumulate operations. The linear model can be written as

$$\bar{y} = \mathbf{x}\boldsymbol{\beta}$$

where $\bar{y}$ is the predicted execution time, $\mathbf{x}$ is a vector of feature values, and $\boldsymbol{\beta}$ is a vector of model coefficients. Table 4.2 summarizes the variables in our prediction model.

To train a linear model, the most common way is to use a least squared error as a metric. That is, we try to minimize the following term

$$\underset{\boldsymbol{\beta}}{\text{minimize}} \quad \|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|^2$$

However, this commonly used approach has major drawbacks in the context of DVFS control: first, it uses all feature values to calculate the target function, despite the fact that only a few features are often sufficient to predict the execution time. Second, this approach tries to minimize both positive and negative errors. However, in the context of DVFS, it is more important to minimize negative errors to reduce deadline misses.

To address the first issue, we use Lasso [107] to minimize the number of

Table 4.2: Variables in prediction model.

Variable	Type	Description
$\bar{y}$	Scalar	Predicted execution time
$\mathbf{x}$	Vector	Feature values
$\boldsymbol{\beta}$	Vector	Model coefficients
$\mathbf{y}$	Vector	Profiled execution times
$\mathbf{X}$	Matrix	Profiled feature values
$\mathbf{X}\boldsymbol{\beta} - \mathbf{y}$	Vector	Prediction errors
α	Scalar	Under-predict penalty weight
γ	Scalar	Number of terms penalty weight
$\|\cdot\|$	Scalar	L2-norm (Euclidean distance)
$\|\cdot\|_1$	Scalar	L1-norm (sum of absolute values)

non-zero coefficients in our model by adding a penalty term to our model:

$$\underset{\boldsymbol{\beta}}{\text{minimize}} \quad \|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|^2 + \gamma\|\boldsymbol{\beta}\|_1$$

where γ is parameter empirically determined to reduce the number of non-zero coefficients without impacting modeling accuracy too much. To address the second issue, we separate positive and negative errors:

$$\underset{\boldsymbol{\beta}}{\text{minimize}} \quad \|pos(\mathbf{X}\boldsymbol{\beta} - \mathbf{y})\|^2 + \alpha\|neg(\mathbf{X}\boldsymbol{\beta} - \mathbf{y})\|^2 + \gamma\|\boldsymbol{\beta}\|_1$$

where $pos(x) = max(x, 0)$ and $neg(x) = max(-x, 0)$. We set $\alpha > 1$ to place more weight on negative errors.

It can be shown that the objective function we try to minimize above is convex. Thus, we can use a convex optimization solver to fit the model.

4.3.5 Hardware Slicing

Now we have a model to predict execution time from features. However, to obtain feature values for a job at run-time, we need to run the hardware accelerator with the job's input. This is not feasible since our goal is to predict execution time before running the hardware accelerator. To address this issue, we propose to generate a minimal version of the hardware accelerator, which we call a *hardware slice*. During runtime, the slice can be executed with the job's input to quickly calculate the feature values.

To generate such a slice, we use program slicing techniques on hardware description languages [34] to only keep the part of the original accelerator that computes the features of interest, while removing other parts of the hardware. This allows us to obtain a slice that is much smaller than the original hardware accelerator in terms of area.

However, executing such a slice would take the same number of cycles as the original hardware accelerator. This is not fast enough since we need to make predictions before running the hardware accelerator. The reason why a slice can not run faster is that the control unit is not aware that some parts of the hardware were removed, and still waits in certain states as if the original computation is still taking place. For example, in Figure 4.8, the FSM still transitions to S2, waits for a number of cycles, and then transitions to S4. This inefficiency can be removed by modifying the FSM transition table to remove the waiting behavior. This optimization does not affect the accuracy of the prediction because we still have the information about how long the FSM would stay in waiting states from counter initial values and pre-reset values. The resulting hardware slice efficiently calculates the control flow features of the original hardware accelerator.

4.3.6 DVFS Model

After obtaining an execution time prediction for a job under the nominal frequency, a DVFS model is used to predict what the execution time would be under different frequencies. We use a common model in literature [70] that decomposes execution time into memory time and compute time:

$$T = T_{memory} + C/f$$

where T is execution time, T_{memory} is the part of execution time when the accelerator is stalled waiting for memory, which does not scale with accelerator frequency. C is the number of cycles when the accelerator is not stalled, and f is the frequency of the accelerator. From the execution time prediction, we know

$$T_0 = T_{memory} + C/f_0$$

where T_0 is the predicted execution time, and f_0 is the nominal frequency. To predict the execution time under a different frequency, we need to know T_{memory} and C. We found that for the many accelerators, T_{memory} is negligible, because most accelerators preload data from memory ahead of time in parallel to the computation. This can be either done using DMA or using the data supply framework described in Chapter 2. Thus, the equation above can be simplified as

$$T_0 = C/f_0$$

Assuming T_{budget} is the time budget for the job, we can run the accelerator at

$$f = C/T_{budget} = f_0 T_0/T_{budget}$$

to minimize energy while meeting deadline.

In real hardware, however, there are only a few discrete frequencies the accelerator can run at. As a result, we need to round up f to the nearest frequency level. Also, executing the hardware slice and switching voltage/frequency takes some time, which needs to be deducted from T_{budget}. We can also add a margin to the predicted execution time to be conservative. After taking all these into account, we set the final frequency level to be

$$f = \lceil f_0(T_0 + T_{margin})/(T_{budget} - T_{slice} - T_{DVFS}) \rceil$$

and execute the accelerator, where $\lceil \cdot \rceil$ represents rounding to the nearest frequency level above.

4.3.7 Predictor Operation Modes

There are a few options regarding how to run the predictor and main computation job. In Figure 4.4, we have shown a simple sequential mode that runs the predictor first, sets a DVFS level, and then runs the accelerator. Figure 4.9 shows two alternative modes. In *pipelined mode*, prediction for job $i + 1$ runs while job i is running. This ensures the DVFS decision is ready at the start of a job without incurring time overheads from predictor execution. However, this requires that the prediction for job $i + 1$ does not use the output of job i. In *parallel mode*, the main jobs start running at a conservative DVFS level with predictor for that job running alongside. After predictor finishes execution, the DVFS level is set according to the prediction, and the main job continues to execute at the predicted DVFS level. Parallel mode does not require independent jobs, while still removing most of the time overheads of prediction execution.

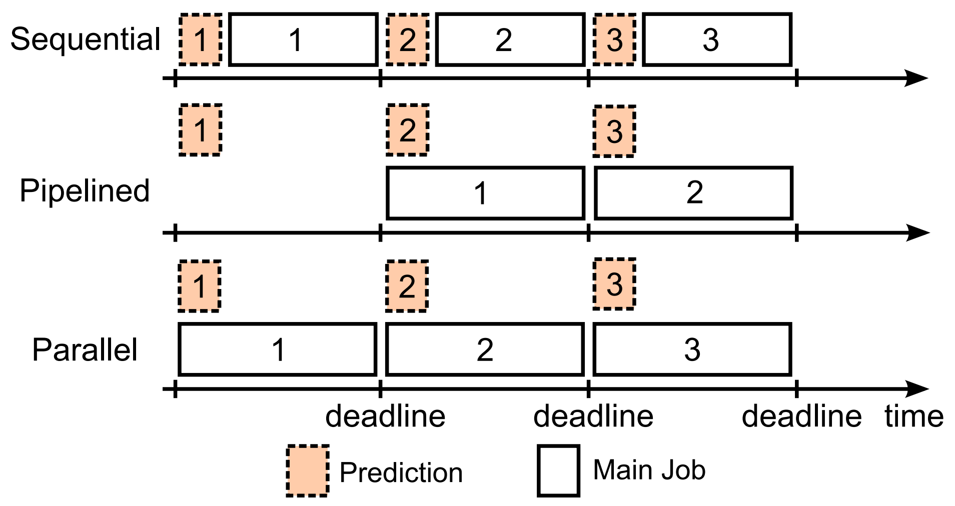


Figure 4.9: Predictor operation modes.

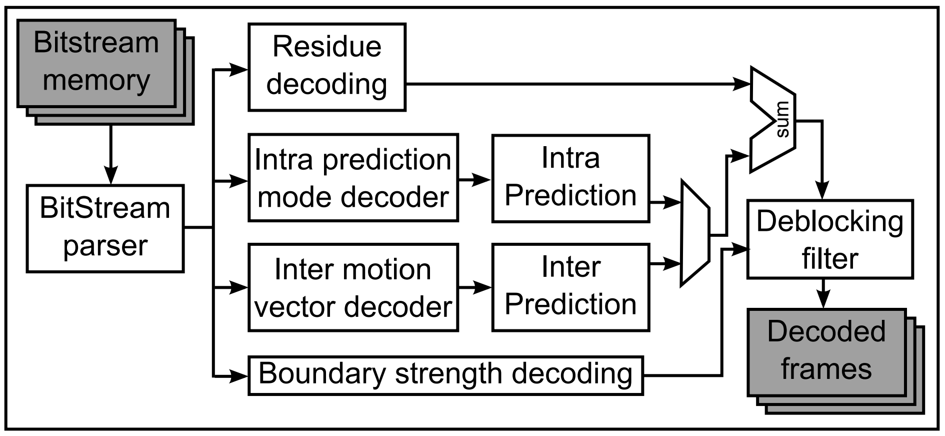


Figure 4.10: Architecture of H.264 decoder.

4.3.8 Case Study

In this section, we present a case study on execution time prediction using the H.264 decoder as an example. We discuss the features chosen by the framework and why these features can be used to predict execution time. We also discuss the details of the hardware slice, and show which parts of the original accelerator were kept and which parts were sliced out.

Figure 4.10 shows the high-level architecture of the H.264 decoder [117]. During decoding, the bitstream parser reads an H.264 bitstream from memory, and performs parsing and entropy decoding. Then, according to the prediction type, each macroblock is either sent to the intra prediction or inter prediction pipeline. The prediction output is then combined with the output of residue decoding. The result is then processed by the deblocking filter to generate the final picture. Each block in Figure 4.10 has its control unit and datapath.

In feature detection step, our framework detected 257 features related to FSMs and counters. Using Lasso, the number of features is reduced to only 7 while still maintaining good prediction accuracy with a worst-case error around 3%. Among these features, two of them are FSM transitions related to the number of transform coefficients in the residue decoding of a macroblock. The other 5 features come from the inter prediction (motion compensation) pipeline. They are counters that control the preloading of data used by inter prediction, as well as counters that control the computation of macroblocks. All these features are in the control unit of the corresponding blocks of the H.264 decoder, and thus do not involve the main computation datapath.

Since the most computation-intensive parts of the decoder are not involved in generating the features, the hardware slicing step of the framework was able to remove them, such as the datapath of intra and inter prediction, deblocking filter, etc. The slice only contains the bitstream parser and the control units of intra and inter prediction pipeline. As a result, the hardware slice was very small and energy-efficient compared to the full decoder. The area of the slice is 37,713μm^2, which is only 5.7% of the full decoder. In addition, the execution time optimization step of our framework was able to remove empty waiting states in the hardware slice, thus the slice only takes 5%-15% percent of the full decoder's execution time to generate features. As a result, the hardware slice only consumes 2.8% of the energy compared to the full decoder. Furthermore, predicting the execution time from features is very fast since the model is linear with only 7 coefficients.

For comparison, we also built a predictor based on application-specific features that we manually identified for H.264 using an approach proposed in a previous study [95]. These features were obtained using an H.264 bitstream analysis software. Surprisingly, the predictor using manually identified features had a worst-case prediction error around 10%, compared to 3% for our automatically generated predictor. Further inspection revealed that a subtle effect (long latency for blocks with quarter-pixel motion vectors) was not captured by the manually identified features. While carefully chosen application-specific features may improve prediction accuracy, obtaining them requires a deep understanding of the algorithm, which often can only be done by domain experts.

4.4 Evaluation

In this section, we present the evaluation results for the proposed DVFS framework. We first discuss our evaluation methodology and experimental setup. Then, we show evaluation results for ASIC and FPGA-based accelerators, and discuss some extensions.

4.4.1 Methodology

We use an integrated evaluation methodology that uses a combination of circuit-level, gate-level and register-transfer-level modeling. Circuit-level modeling is used to characterize the voltage-frequency relationship. Gate-level modeling is used to build accurate area and energy models. And register-transfer-level modeling is used to accurately model the performance of hardware accelerators.

Circuit-level modeling: For ASIC accelerators, we characterize the relationship between voltage and frequency for our accelerators using SPICE simulations based on a method reported in literature [41]. For each accelerator, we used a chain of FO4 loaded inverters such that the total delay of the chain matches the cycle time of the accelerator at nominal voltage. Then we change the supply voltage and measure the voltage-delay curve and use that to model the accelerator's frequency under different voltage levels. For FPGA accelerators, the voltage-frequency relationship is obtained from published characterizations [14].

Gate-level modeling: For ASIC accelerators, we implemented each hardware accelerator using Synopsys Design Compiler, IC Compiler, PrimeTime PX with the TSMC 65nm standard-cell library characterized at 1 V. Detailed post-place-and-route gate-level simulations were used to obtain the power and energy of the accelerator's execution for a subset of the jobs at 1 V. Then we apply the voltage-frequency model and the frequency-execution time model to estimate the power and the energy consumption under different DVFS levels. For FPGA accelerators, the synthesis and place-and-route flow is based on Vivado. Post-place-and-route simulations are used to obtain power and energy estimations, which are then scaled to different DVFS levels.

Register-transfer-level modeling: We use RTL simulations to determine the execution time of each job for our accelerators. We assume the accelerators are not bandwidth-limited and a DMA controller transfers data from the main memory to the accelerator's scratchpad before executing each job.

4.4.2 Experimental Setup

We use a set of seven benchmark accelerators in our evaluation, including an H.264 video decoder [117], a JPEG encoder [81], a JPEG decoder [79], a molecular dynamics accelerator [92], a stencil filtering accelerator used in image processing [92], an Advanced Encryption Standard (AES) accelerator [78], and a Secure Hash Algorithm (SHA) accelerator [80]. Note that even though some hardware accelerators are traditionally throughput-oriented, they can have response time requirements when used as a part of a frame-based or interactive application. For example, when a user is playing a DRM-protected video, a crypto accelerator has to decrypt the data for each frame before a certain deadline. As another example, when a smartphone camera shoots in a burst mode, the JPEG engine has to encode each picture before a certain deadline. Table 4.3 lists the accelerators and describes what a task is in each accelerator. Table 4.4 describes the workloads we use to train and test the DVFS controller.

For ASIC accelerators, we use six equally-spaced voltage levels that span from the nominal voltage at 1 V (high performance/energy point) to 0.625 V (low performance/energy point). The frequency corresponding to a voltage is determined using the voltage-frequency relationship obtained from circuit-level modeling. For FPGA accelerators, we use seven equally-spaced voltage levels

Benchmark	Description	Task
h264	H.264 video decoder	Decode one frame
cjpeg	JPEG encoder	Encode one image
djpeg	JPEG decoder	Decode one image
md	Molecules/physics simulation	Simulate one timestep
stencil	Image filtering	Filter one image
aes	Advanced Encryption Standard	Encrypt a piece of data
sha	Secure Hash Function	Hash a piece of data

Table 4.3: Summary of benchmarks.

Benchmark	Workload (Train)	Workload (Test)
h264	2 videos (600 frames, same size)	5 videos (1500 frames, same size)
cjpeg	100 images (various sizes)	100 images (various sizes)
djpeg	100 images (various sizes)	100 images (various sizes)
md	200 steps (particle pos. changes)	200 steps (particle pos. changes)
stencil	100 images (various sizes)	100 images (various sizes)
aes	100 pieces of data (various sizes)	100 pieces of data (various sizes)
sha	100 pieces of data (various sizes)	100 pieces of data (various sizes)

Table 4.4: Summary of workloads.

from 1 V to 0.7 V. We assume voltage regulators are off-chip. Switching time for off-chip voltage regulators are typically in the range of 10μs [57]. In our evaluation, switching time is conservatively set to 100μs to account for potential overheads in case changing DVFS levels involves device drivers. However, we do note that there are faster DVFS switching techniques in literature [41, 57], which could further reduce DVFS switching overhead to the range of tens of nanoseconds.

We set the deadline for each job at 16.7ms, which corresponds to the 60fps screen refresh rate in most of today's devices. We compare our prediction-based DVFS controller with the following schemes:

Bench-	Area	Frequency	Execution Time (ms)		
mark	(μm^2)	(MHz)	Max	Avg.	Min
h264	659,506	250	11.46	7.56	6.50
cjpeg	175,225	250	13.90	5.22	0.88
djpeg	394,635	250	14.79	3.78	1.82
md	31,791	455	15.52	7.11	0.80
stencil	10,140	602	15.97	5.92	1.41
aes	56,121	500	16.19	4.62	1.94
sha	19,740	500	12.94	4.11	1.11

Table 4.5: Summary of ASIC implementation results.

1. **baseline**: The baseline runs at constant voltage and frequency. Each accelerator runs at 1 V and the frequency it is synthesized at.
2. **pid**: The PID-based controller uses prediction errors from previous jobs together with a control-theory based algorithm to determine the execution time of the next job. For each accelerator, we tuned the PID controller's parameters to achieve the best prediction accuracy. A margin is added to PID controller's output to reduce the number of deadline misses. We tried different margins and chose 10% as the amount that balances deadline miss rate and energy savings.
3. **prediction**: This is our prediction-based DVFS controller. A 5% margin is added to its prediction. Its predictions are usually fairly accurate so only a small margin is needed.

4.4.3 Results for ASIC Accelerators

Implementation Results Table 4.5 shows the area, frequency, and execution time statistics for the benchmark accelerators. The area numbers are from place-

and-route results. The frequency numbers are shown for nominal voltage at 1 V. The execution time statistics are obtained at nominal voltage and frequency. Large execution time variations are observed.

Execution Time Prediction Accuracy Figure 4.11 shows box-and-whisker plots of prediction error of our scheme for each benchmark. The box extends from the 25% to 75%, with a line at the median. The whiskers extend from the box to show the range of the data. Positive numbers correspond to over-prediction and negative numbers correspond to under-prediction. For most benchmarks, the prediction error is negligible, indicating the effectiveness of our approach. The JPEG decoder showed higher prediction error. This is because some of its execution time variations cannot be accurately modeled using the extracted features. Specifically, some of the FSMs in the decoder stay in a state for a variable number of cycles which cannot be obtained using a corresponding counter. However, our slice-based predictor still captured the majority of its execution time variations. In addition, the convex optimization-based prediction framework is conservative and leads to very few under-predictions.

Energy Savings and Deadline Misses Figure 4.12 shows the comparison of normalized energy and deadline misses between different DVFS schemes. The energy numbers are normalized to the baseline. The baseline always runs at constant frequency and thus has the highest energy but no deadline misses. On average, our schemes achieved 36.7% energy savings across all benchmarks. PID-based DVFS controller has 4.3% higher energy consumption than our scheme. In addition, the PID controller often chooses lower DVFS levels than needed which leads to many deadline misses. On average, the PID-based

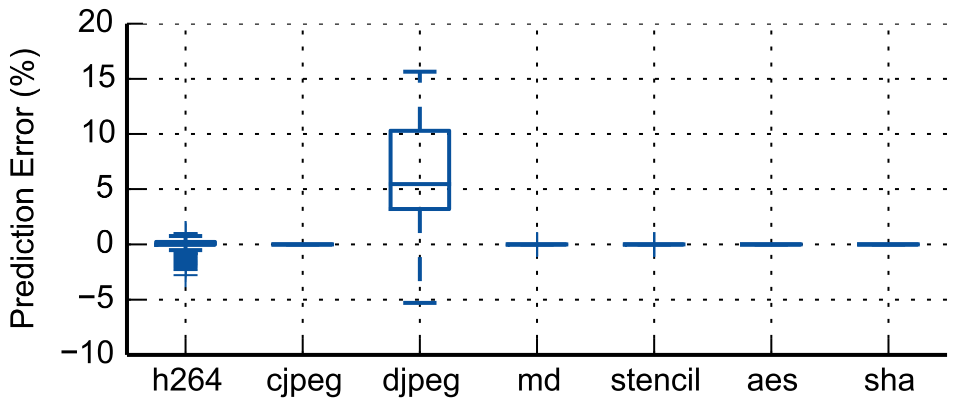


Figure 4.11: Errors of slice-based execution time prediction. The box extends from the 25% to 75%, with a line at the median. The whiskers show the range of the data. Outliers are shown as individual points.

controller misses 10.5% of the deadlines while our prediction-based controller misses only 0.4% of the deadlines.

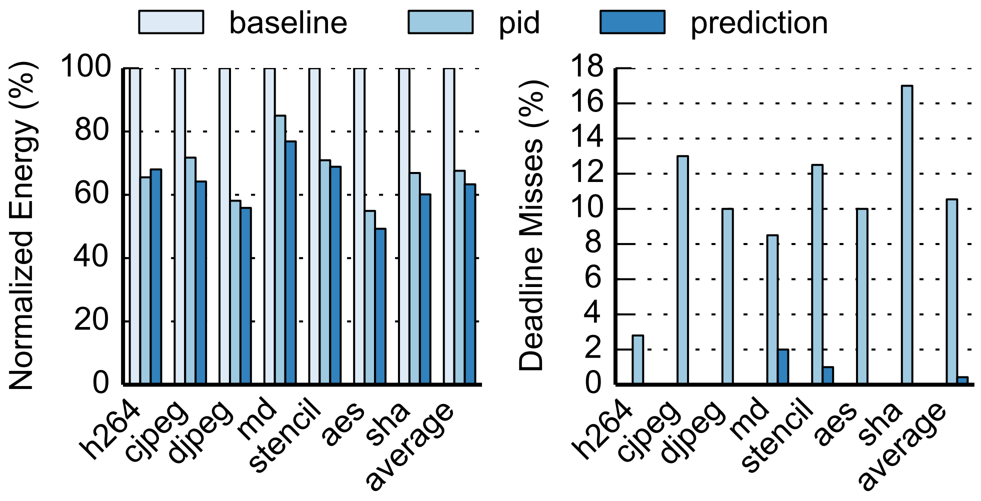


Figure 4.12: Normalized energy and deadline misses of different DVFS schemes.

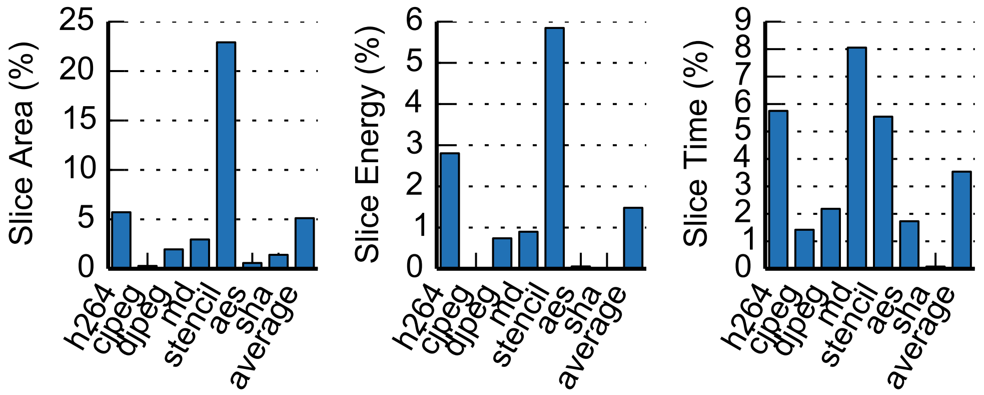


Figure 4.13: Area, energy and execution time overhead of prediction slice.

Overheads of Execution Time Prediction The hardware slice for our prediction-based DVFS has overheads in terms of area, energy, and time. The prediction slice takes up extra die area, and consumes energy during execution. Also, since the prediction slice is run before the actual job, the time needed to run the slice reduces the amount of time left to run the job, which in turn reduces the opportunity to run the job at a lower frequency. Figure 4.13 shows the overheads of the slice. Slice energy is normalized to the actual job's energy. Slice area is normalized to the accelerator's area. Slice time is normalized to the job's deadline. On average, the prediction slice adds 5.1% area overhead to the baseline accelerator. Running the prediction slice takes about 3.5% of the time budget, while adding 1.5% energy overhead to the job on average. The energy overhead is low because the slice is small compared to the full accelerator, and only runs for a short period. Besides the overheads introduced by the hardware slice, DVFS switching also has overheads since the voltage and frequency takes time to stabilize. Note that the results shown in Figure 4.12 includes these overheads.

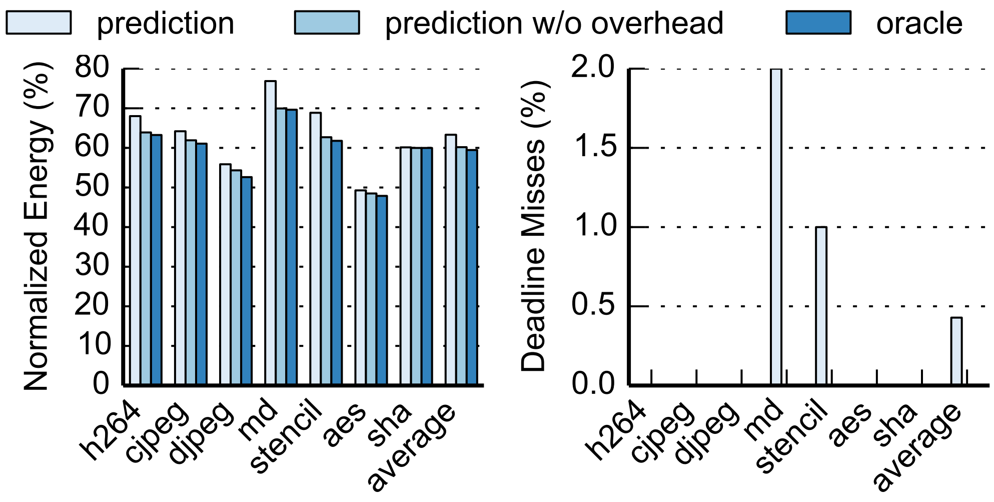


Figure 4.14: Normalized energy and deadline when overhead is removed.

To better understand how these overheads impact energy savings and deadline misses, we show the results for the prediction scheme when the overheads of hardware slice and DVFS switching are removed. In addition, we also show the results for an oracle scheme that always sets a best DVFS level for each job, and without DVFS switching overhead. Figure 4.14 shows that by removing these overheads, energy savings are improved by 3.1%, from 36.7% to 39.8%. Deadline misses are reduced from 0.4% to 0%. The oracle scheme has 40.5% energy savings, which is 0.7% higher than the prediction scheme without overheads. Both the oracle scheme and the prediction scheme without overhead have zero deadline misses. This shows that the prediction scheme with overhead removed is very close to optimal. It also indicates that the deadline misses we see in the prediction scheme without overhead is not due to misprediction. Instead, it is because insufficient time budget is left after the slice finishes execution, so that even running at highest frequency will miss the deadline. This only happens to jobs whose execution time is very close to, or even longer than the length of the deadline, which are usually rare.

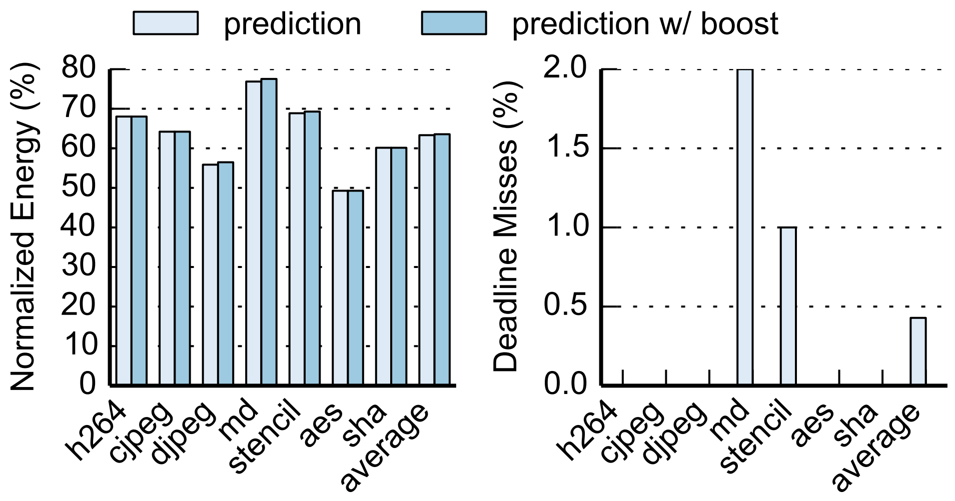


Figure 4.15: Normalized energy and deadline misses with voltage boosting.

These rare misses can be eliminated by boosting voltage temporarily for these jobs. With execution time prediction, the DVFS controller knows when the time budget left is not enough, and can boost DVFS level accordingly. Figure 4.15 shows normalized energy and deadline misses when we introduce a boost level at 1.08 V. With voltage boosting, deadline misses are eliminated while only increasing normalized energy by 0.24%.

Sensitivity Study on Varying Deadlines Figure 4.16 shows the normalized energy and deadline misses when we vary the job deadline from 0.6x to 1.6x of the deadline used before. For conciseness, we only show results averaged across all benchmarks. Since our DVFS model is aware of the deadline, it will use lower DVFS levels to save energy when deadlines are longer, and use higher DVFS levels trying to meet response time requirements when deadlines are

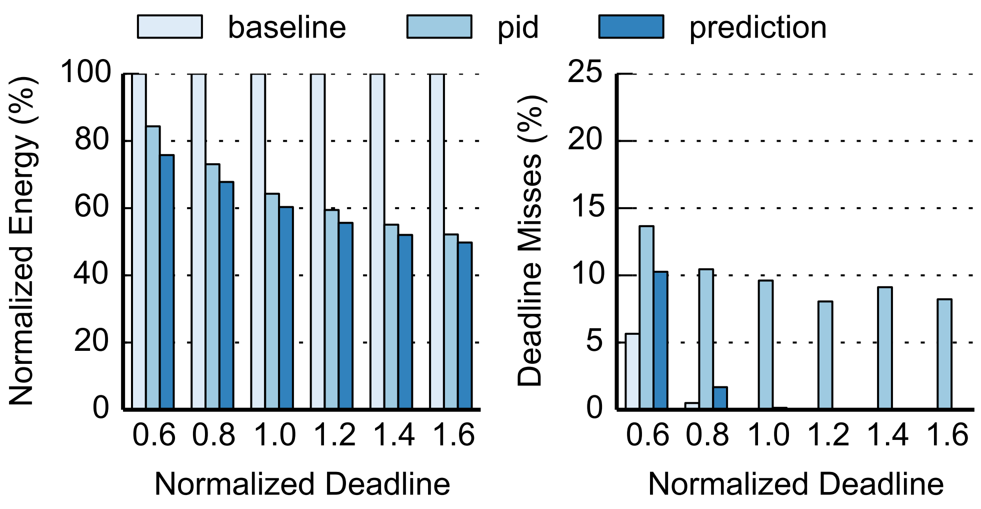


Figure 4.16: Normalized energy and deadline misses when varying deadlines (averaged across all benchmarks).

shorter. When the deadline is shorter than 1.0x, the prediction-based DVFS controller starts showing misses. This is mostly due to the deadlines being too short to meet for some jobs even when running at highest frequency, which is why the baseline policy also shows misses. When the deadline is increased, the prediction-based DVFS controller achieves more energy savings while still meeting all deadlines. Note that the execution time predictor does not need to be retrained when the deadline changes. Only a new deadline needs to be set in the DVFS model. The PID-based controller, on the other hand, still shows misses even with longer deadlines because it often uses the wrong DVFS level due to low prediction accuracy.

4.4.4 Results for FPGA-based Accelerators

In this section, we show results for FPGA-based accelerators. The target FPGA device we use is Xilinx Kintex-7. The execution time prediction accuracy for FPGA accelerators is similar to the accuracy for ASIC accelerators because the features used for prediction are from RTL level, and the prediction model is able to adapt to differences in operation frequency.

Figure 4.17 shows normalized energy and deadline misses of different DVFS schemes for FPGA-based accelerators. Overall, FPGA-based accelerators achieved 35.9% energy savings with the predictive DVFS framework, while missing 0.4% of the deadlines. The numbers are comparable to ASIC results.

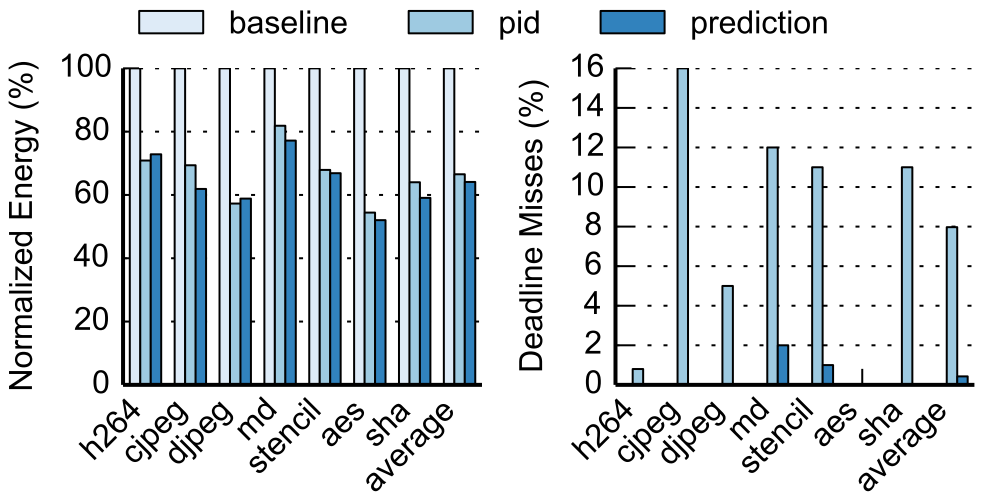


Figure 4.17: Normalized energy and deadline misses for FPGA-based accelerators.

Figure 4.18 shows the overheads of the slice for FPGA-based accelerators. On average, the prediction slice use 9.4% resources (average of LUT/DSP/BRAM) of the baseline accelerator. Running the prediction slice consumes 2% of the energy of the job on average, while using about 3.5% of the time budget. The resource overhead for `stencil` appears large because the accelerator only uses a small number of LUTs for control logic while using DSP blocks for main computations. As a result, the relative overhead is shown to be high even though the absolute resource usage of the prediction slice is quite low.

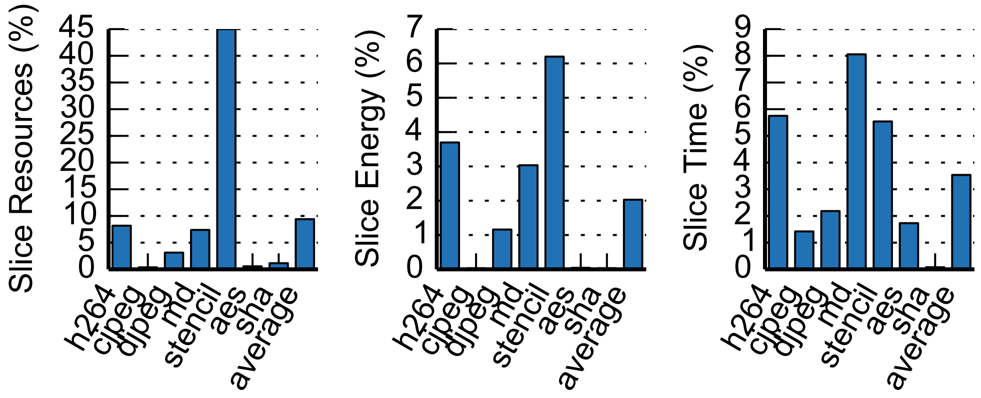


Figure 4.18: Area, energy and execution time overhead of prediction slice for FPGA accelerators.

4.4.5 Extensions

Accelerators Generated using High-Level Synthesis High-Level Synthesis (HLS) allows designers to write accelerators in a high-level programming language such as C, and synthesizes them into RTL descriptions. To use our framework with HLS-generated accelerators, one way is to analyze the RTL generated by HLS, extract features, and build a predictor by slicing the RTL. However, if analysis can be done during the HLS process, it can potentially enable optimizations that can not be easily performed at RTL level. For example, instead of slicing the RTL to obtain a hardware slice, we can use program slicing [110] on the C code to obtain a software slice that calculates the control flow features, and then synthesize the sliced C code into hardware. The HLS tool can perform optimizations during the synthesis, resulting in a slice that calculates feature values faster. This leaves more time budget to run the accelerator itself, reducing the possibility of deadline misses due to insufficient time budget after slice execution.

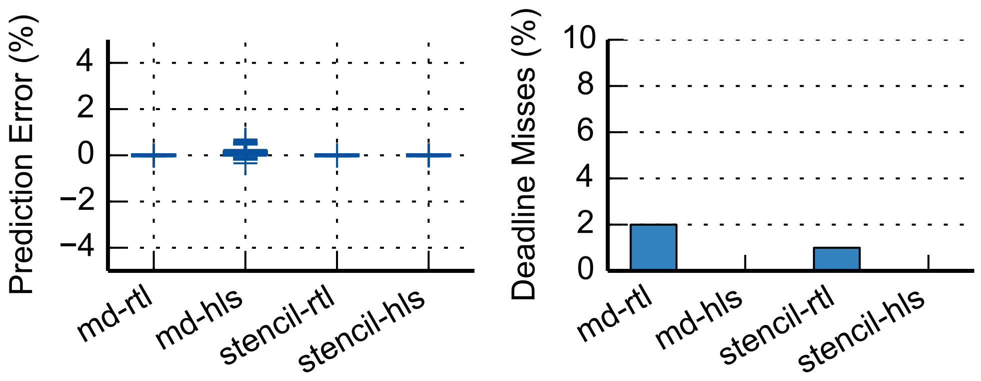


Figure 4.19: Comparison of prediction errors and deadline misses between slicing at RTL and HLS level.

We compared these two approaches using the `md` and `stencil` accelerators [92], which have C versions available. Figure 4.19 shows that the prediction accuracy of both approaches are very high, but when the hardware slice is generated using HLS from sliced C code, the deadline misses are gone. This implies the slice generated using HLS runs faster because we know from Section 4.4.3 that the deadline misses in `md` and `stencil` are caused by insufficient time budget left after the slice finishes execution rather than mispredictions. This can be verified by looking at Figure 4.20, which shows the slice execution time for the HLS approach is much shorter.

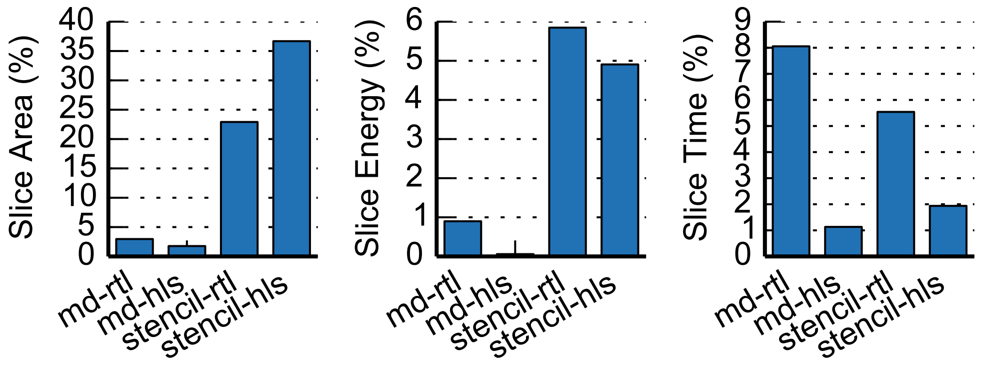


Figure 4.20: Comparison of area, energy and execution time overhead between slicing at RTL and HLS level.

Software-based Predictors Some accelerators have a software version with the same function, either because they are generated using HLS, or because they have a software implementation (e.g. ffmpeg for H.264). In these cases, instead of building hardware predictor, we can run a software predictor on the CPU to predict the execution time of the accelerator. We experimented with this idea on the H.264 decoder, and achieved good prediction accuracy.

CHAPTER 5

RELATED WORK

This chapter describes the existing work related to this thesis. Section 5.1 discusses high-level design methodologies for accelerators. Section 5.2 discusses previous work on data supply for accelerators. Section 5.3 discusses related work on parallel programming and design methodologies for parallel accelerators. Section 5.4 discusses related work on dynamic voltage and frequency scaling and execution time prediction.

5.1 High-Level Design Methodologies for Accelerators

Traditionally, accelerators are designed using a hardware-oriented flow that involves writing low-level register transfer level (RTL) code using hardware description languages such as Verilog and VHDL. The high complexity and poor productivity of this flow have lead researchers to look into design methodologies that provide a higher level of abstraction.

5.1.1 High-Level Synthesis

High-Level Synthesis (HLS) compiles high-level languages such as C/C++ [20, 114] into RTL descriptions. A major reason that HLS improves productivity is that it decouples the *functional specification* from the *timing specification* of hardware design. Most high-level languages are *untimed*, meaning that programmers only need to describe the functional behavior of an algorithm or application. In contrast, RTL design is *timed*, meaning that the designer needs to specify both the functionality and exact time each operation takes place, which is tedious to perform manually. HLS tools allow programmers to focus on the functional specification, and uses sophisticated algorithms to automatically generate the timing schedule. Many HLS tools allow programmers to issue high-level *directives* to guide the scheduling process, which enables generating multiple designs with different performance, power, and area profiles from the same functional description. This ability to quickly explore the design space is another reason why HLS achieves higher productivity compared to RTL design. Many high-level synthesis frameworks have been proposed, with support for various input languages. Here we describe a number of widely used frameworks.

Bluespec Bluespec is a framework which includes a language and a set of tools for hardware design. The language, Bluespec SystemVerilog (BSV), extends SystemVerilog to support describing hardware as a set of *Guarded Atomic Actions*, which specifies the operations and the rules under which they should fire. The Bluespec compiler then synthesizes BSV into RTL (Verilog) by finding a schedule for the operations that satisfies the rules. The BSV language also supports many advanced features of modern programming languages, such as object-oriented interfaces, polymorphic types, static type checking, and first class parameterization [76].

HLS from C Family Languages C family languages such as C/C++, SystemC, and OpenCL are among the most widely supported languages for HLS, with various tools developed by commercial companies [54, 114] and academic institutions [20]. HLS compilers for these languages perform a series of steps including code transformation, resource allocation, operation scheduling, and resource binding to generate RTL. Due to the procedural nature of these languages, directly mapping the statements in a program to hardware would often be overly serialized and inefficient. Thus, HLS tools often use compiler analysis as well as additional user-provided directives to control how the procedural constructs such as loops are mapped to efficient hardware pipelines.

HLS from Domain Specific Languages Domain-specific languages (DSLs) trade off generality for efficiency. DSLs often provide a higher level of abstraction than general-purpose languages, and potentially enable programmers to more productively express certain domain-specific algorithms and allow HLS tools to generate more efficient hardware. Delite Hardware Definition Language (DHDL) [58] is a DSL for generating accelerators based on a collection of parallel patterns inspired by functional programming languages. Halide [90] is a DSL for generating efficient image processing pipelines, and can be synthesized into hardware [84].

The architectural frameworks proposed in this thesis research leverage HLS, but in addition address some challenges faced by existing HLS frameworks, such as not able to tolerate variable memory latency, and insufficient support for dynamic parallelism. Furthermore, the design methodology proposed in this thesis combines the benefits of both HLS and RTL designs, while avoiding their shortcomings. HLS is used to allow accelerator designers to productively describe application logic, and the frameworks use RTL to implement highly optimized and configurable accelerator architectures that are challenging to implement in existing HLS tools. The reusable architecture templates allow designers to use the accelerator architectures without needing to manually write any RTL.

5.1.2 Hardware Generation Languages

Hardware Generating Languages (HGLs) aim to enable rapid design spacing exploration for hardware designs. In contrast to HLS, HGLs are typically used with RTL designs. Genesis2 [97] combines SystemVerilog with Perl scripts to enable creating highly parameterizable hardware designs (“chip generators”), which are templated designs that encapsulate designer knowledge and design trade-offs. Chisel [12] is a hardware construction language embedded in Scala that enables rapid hardware generation using highly parameterized generators and layered domain-specific hardware languages.

The architectural frameworks proposed in this thesis leverage HGLs for implementing the parameterizable architecture templates. We use PyMTL [69], which is a hardware generation language embedded in Python. Our frameworks extend the HGL approach by allowing designers to use HLS to synthesize application logic from high-level languages, which is more productive than writing hardware generation languages.

5.2 Data Supply for Accelerators

Efficient data supply is a fundamental requirement for accelerators to achieve good performance. Depending on how tightly the accelerators are integrated with general-purpose processor cores, and the way the accelerators are designed, researchers have proposed different approaches to address the data supply problem.

5.2.1 Data Supply for In-Core Accelerators

Accelerators that are tightly integrated into a processor core often rely on the processor pipeline to perform memory accesses. A number of proposals perform memory accesses in a decoupled fashion, following the Decoupled Access/Execute (DAE) paradigm. DAE [101] was originally proposed for in-order processors as a complexity-effective mechanism to address the memory latency problem by dividing a program's instructions into an access stream and an execute stream that run in a decoupled fashion and communicate through architecturally visible queues. Later work extended DAE to out-of-order processors and found that DAE can use two small instruction windows to achieve the effect of a single large instruction window, but with less complexity [55]. In recent work, DeSC [49] explored DAE for heterogeneous architectures and proposed to use an out-of-order processor core to supply data to a hardware accelerator. MAD [51] proposed to use dataflow to build a specialized engine that executes memory access phases efficiently, which can also be used to supply data to hardware accelerators.

The data supply framework proposed in this thesis differs from previous work as we target stand-alone accelerators that are not tightly integrated with a processor core or dedicated memory access engine. We employ DAE as a paradigm to design accelerators that effectively tolerate the memory latency and thus remove the burden of hand-crafting dedicated memory management logic from accelerator designers.

5.2.2 Memory Architecture for Standalone Accelerators

CoRAM [33] is a memory architecture for FPGA-based accelerators. In CoRAM, designers write *control threads* in a C-like language that manages the communication between DRAM and on-chip scratchpad memories. The data supply framework proposed in this thesis differs from CoRAM in that we provide a framework to automatically transform accelerators into a decoupled architecture, instead of relying on the designer to write application-specific control threads manually.

LEAP [8] is a compiler framework that transforms accelerators that use arbitrary-size scratchpads to use small caches backed-up by a memory hierarchy. It was later extended to handle prefetching [118] but can only use address-based stream localization since accelerators do not have a PC. The data supply framework proposed in this thesis can improve LEAP by providing better latency tolerance using access/execute decoupling, and enabling more effective prefetching by tagging memory accesses.

CHIMPS [86, 88] is a memory architecture and compilation framework for FPGA accelerators that use many small, distributed caches implemented using block RAMs. The cache coherence issue is avoided by statically partitioning the memory address space between caches. Our data supply framework can work with this many-cache architecture by connecting each memory unit to a cache and use the same address partitioning technique. This can potentially lead to better performance utilizing higher memory bandwidth.

5.2.3 Memory Optimizations in High-Level Synthesis

Deep pipelining is an HLS technique that allocates extra pipeline stages for memory operations in order to tolerate memory latency. However, in cache-based accelerators, it may lower pipeline throughput as it always targets the worst case even though most memory accesses are cache hits.

Tan et al. proposed to synthesize multithreaded pipelines with HLS to tolerate memory latency [106]. The approach mainly targets loop pipelining and allocates a thread for each iteration of a loop. Threads are switched out on a cache miss and stored in a context buffer, and woken up to continue execution when the memory response comes back. This approach achieves good speedup with low resource overhead, but is only applicable to data-parallel kernels where each loop iteration is independent. In contrast, our data supply framework is applicable to more general programs.

Decoupled pipelining was first proposed as a technique to parallelize single-threaded programs. DSWP [82] is a compiler framework that extracts coarse-grained pipeline parallelism from single-thread code and executes using multiple threads. The framework analyzes the program dependence graph and partitions the graph between threads. The threads communicate using message passing. Later work [26] extended it to HLS where an accelerator is transformed into multiple decoupled pipeline stages that communicate through FIFOs. As a result, the impact of a variable-latency memory access can be limited to one stage. Coarse-Grained Pipelined Accelerators (CGPA) [67] extends decoupled pipelining by using multiple workers for pipeline stages that are parallelizable. In comparison, our data supply framework uses DAE as the decoupling mechanism and combines hardware prefetching with decoupling to enable more efficient data supply for accelerators.

5.3 Parallel Accelerators

Both general-purpose processors and accelerators need to exploit parallelism to achieve good performance. There is a rich history of research on parallel programming, and a number of proposals on generating parallel accelerators.

5.3.1 Task-Based Parallel Programming

Task-based parallel programming was first proposed in [19], and recently gained popularity as the technology matures and with the introduction of languages and frameworks such as Cilk [62] and Intel TBB [93]. Task-based programming has been shown to allow programmers to think at a higher level in addition to its performance benefits such as matching parallelism to available hardware resources and improved load balancing. Carbon [59] implements hardware task queues in a processor that can be accessed using special instructions. The parallel accelerator framework proposed in this thesis is inspired by task-based parallel programming, but leverages it for designing hardware accelerators. The framework includes a hardware architecture that implements a task-based parallel computation model, and a unified language that can be mapped to both accelerators and parallel software.

Work stealing was developed along with task-based programming and has been extensively studied [16,17,93]. It has been shown to have provable bounds in terms of the space and time needed for a parallel execution compared to serial execution [17], and also works well in practice. We implement work stealing in hardware and show that it can efficiently distribute and balance load in parallel accelerators.

5.3.2 Design Methodologies for Parallel Accelerators

Generating parallel accelerators from a high-level description was explored and implemented in several languages and frameworks [11,28,44,54,58]. For example, OpenCL [54] has been adopted for generating accelerators based on data parallelism. Delite Hardware Definition Language (DHDL) [58] is a domain-specific language for generating accelerators based on a collection of parallel patterns. Liquid Metal [11] extends Java to support accelerators with pipeline parallelism. Legup [28] supports a subset of POSIX threads. Kiwi [44] extends C# to generate accelerators with threads and channels. These existing frameworks require parallelism to be specified at compile time and statically scheduled. As a result, it is difficult to map dynamic or irregular algorithms to these frameworks. The parallel accelerator framework proposed in this thesis supports dynamic parallelism with dynamic work generation and dynamic scheduling, enabling mapping a wider range of algorithms and achieving good load balancing.

A few prior studies explored dynamic parallelism in hardware. Li et al. [64] propose to extract parallelism from irregular applications dynamically on FPGAs [64], but only supported limited pipeline parallelism. Our framework supports broader types of parallelism including data-parallel, fork-join, and general task-parallel patterns, and also support scalable scheduling of dynamically generated work on multiple processing elements using work stealing. Ramanathan et al. [91] explored implementing software-based work stealing runtimes on FPGAs using OpenCL atomic operations, which incurs high performance and resource overheads. In contrast, we propose a hardware architecture that implements native support for work stealing, which is more efficient, more scalable, and uses fewer resources.

5.4 Power Management for Systems with Time Constraints

Computing systems often employ power management techniques, which improves energy efficiency by dynamically adjusting the performance level of a system to match application demand. Power management techniques for interactive or real-time systems also need to take into account the response time requirements of such systems.

5.4.1 Dynamic Voltage and Frequency Scaling

Dynamic Voltage and Frequency Scaling (DVFS) is a widely studied technique for reducing the energy of computing systems. For applications without response time requirements, simple interval-based scheduling algorithms [111] are often used. For example, the Linux CPU power governors [18] are interval-based. DVFS has also been studied in the context of hardware accelerators. Linux implements interval-based governors in its `devfreq` framework [48] for controlling DVFS of hardware accelerators. As with other reactive approaches, interval-based scheduling algorithms do not work well for applications that have response time requirements, because their reactions lag behind the workload changes. In contrast, the proposed DVFS framework in this thesis is able to predict the optimal DVFS level by look ahead into the upcoming workload.

Researchers have studied DVFS in the context of hard real-time systems. One approach uses worst-case execution time (WCET) analysis of tasks to guide DVFS settings [100]. Although it guarantees that deadlines are met, it can be overly conservative since actual execution time can be much shorter than worst-case execution time. As a result, this approach is limited to hard real-time systems where deadlines must be strictly met. Our framework targets interactive and soft real-time systems which are more widely used in practice.

DVFS has been studied in the context of resource management in datacenters to achieve energy proportionality, as well as controlling tail latency. PEGASUS [68] is a feedback-based DVFS controller that utilizes request latency statistics to make power management decisions. However, it only responds to slowly-changing workload variations. Adrenaline [53] uses workload metrics to predict tail queries for web services, and boosts DVFS levels accordingly. A number of studies have also investigated using workload metrics to predict the execution time in order to inform DVFS decisions for interactive games [46], video decoding [95], and web browsing [119,120]. However, most of these studies use metrics that are application-specific and manually identified, which requires domain experts to obtain and does not generalize to other applications. In contrast, our framework obtains workload metrics with predictors automatically generated using program analysis and transformation. As a result, our framework does not require domain-specific knowledge to use. We have also shown that our approach can sometimes generate predictors with the same or better quality than those obtained manually.

5.4.2 Execution Time Prediction

Worst-case execution time analysis is a well-studied topic in hard real-time systems [85] and has been applied to DVFS for these systems [100]. WCET tries to calculate an upper bound of a job's execution time under all possible inputs. However, it does not estimate a job's execution time given a specific input. In contrast, the execution time prediction technique presented in this thesis is able to predict input-dependent execution time variations.

Mantis [60] is an execution time prediction framework for smartphone applications, which uses automatically-extracted program features and thus is general across different applications. The high-level approach of Mantis is similar to our work. However, Mantis only considers software programs. Our work proposes an execution time prediction framework for hardware accelerators and investigates its application to DVFS.

CHAPTER 6

CONCLUSION

6.1 Summary

This thesis introduces architectural frameworks that combine *novel accelerator architectures* with *automated design and optimization frameworks* to enable designing high-performance and energy-efficient accelerators with minimal manual effort.

First, the thesis proposes a framework for automatically generating accelerators that can effectively tolerate long, variable memory latencies, which improves performance and reduces design effort by removing the need to manually create data preloading logic. The framework leverages architecture mechanisms such as memory prefetching and access/execute decoupling, as well as automated compiler analysis to generate accelerators that can intelligently preload data needed in the future from the main memory.

Second, the thesis introduces a framework for building parallel accelerators that leverages concepts from task-based parallel programming, which enables software programmers to quickly create high-performance accelerators using familiar parallel programming paradigms, without needing to know low-level hardware design knowledge. The framework uses a computation model that supports dynamic parallelism in addition to static parallelism, and includes a flexible architecture that supports dynamic scheduling to enable mapping a wide range of parallel applications and achieve good performance. In addition, we designed a unified language that can be mapped to both software and hardware, enabling programmers to create parallel software and parallel accelerators in a unified framework.

Third, the thesis proposes a framework that enables accelerators to perform intelligent dynamic voltage and frequency scaling (DVFS) to achieve good energy-efficiency for interactive and real-time applications. The framework combines program analysis and machine learning to train predictors that can accurately predict the computation time needed for each job, and adjust the DVFS levels to reduce the energy consumption.

Our evaluation results show that the proposed frameworks allow designers to productively create high-quality accelerators for a diverse range of applications with very little manual effort. We believe the thesis provides a promising way to address the design complexity problem of accelerators, which is becoming increasingly important as more and more applications will need to rely on customized accelerators to achieve performance and energy-efficiency gains in the future.

6.2 Future Directions

6.2.1 Compiler Support for Parallel Accelerators

The parallel accelerator framework proposed in this thesis requires programmers to describe the parallel computation using an explicit continuation passing style. Although this facilitates an efficient hardware implementation, it requires more effort compared to modern task-based parallel programming frameworks with compiler support (e.g., Cilk Plus) because programmers need to restructure the code and handle task input and output manually.

To further improve accelerator design productivity, an interesting direction of future research is to investigate compiler support for the proposed framework. For example, one approach is for the compiler to generate continuation passing style code from a more traditional fork-join code. The compiler would need to perform program analysis to determine the input and output of the tasks, and perform code transformations to convert the code into a continuation passing style. Another approach is to leverage existing work on supporting parallel tasks in a compiler's intermediate representation (IR), which embeds logically parallel tasks in a program's control flow graph [96]. The proposed framework can then serve as a backend for the intermediate representation by generating hardware from the IR.

6.2.2 Hybrid GPP-Accelerator Work-Stealing Architecture

In most of today's systems, accelerators and the general-purpose processors (GPPs) work separately on different tasks. Usually, the accelerators are controlled by the GPPs. After launching a computation job on an accelerator, the GPPs either need to be idle and wait for the job to complete, or find some other work to do while waiting. This separation is inefficient because the processing power of the GPPs is often left unused. Hence, one interesting research direction is to investigate enabling GPPs and accelerators to work collaboratively on a problem. Task-based parallel programming provides a good foundation for achieving this goal because it enables decomposing a problem into many tasks that can potentially run in parallel. Toward this direction, future research can look into designing a hybrid software runtime and accelerator architecture where the GPPs and accelerators can execute a task-based parallel program collaboratively using work stealing. In this architecture, the GPP cores can not only steal tasks from other GPP cores, but also from the accelerator. Similarly, the accelerator can also steal tasks from the GPP cores. In this way, the architecture can achieve high utilization of the available computation resources. In addition, because the accelerator may be better at processing certain types of tasks while the GPPs are better at other types, it will be interesting to look into support task affinity in the architecture, where programmers can express whether it is better to execute certain tasks on the GPPs or accelerator, which the work-stealing algorithm can take into account when scheduling the tasks.

BIBLIOGRAPHY

[1] LogiCORE IP system cache v3.0. Xilinx Product Guide. https://www.xilinx.com/support/documentation/ip_documentation/system_cache/v3_0/pg118_system_cache.pdf.

[2] Samsung Exynos Linux kernel drivers. Online Webpage. https://github.com/hardkernel/linux/blob/odroidxu3-3.10.y/arch/arm/mach-exynos/include/mach/exynos-mfc.h.

[3] Verilator. Online Webpage. http://www.veripool.org/wiki/verilator.

[4] Intel Cilk Plus language extension specification, version 1.2. Intel Reference Manual, 2013. https://www.cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_1.2.htm.

[5] OpenMP application program interface, version 4.0. OpenMP Architecture Review Board, 2013. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[6] Amazon EC2 F1 instances. Online Webpage, 2017 (accessed Apr 17, 2018). https://aws.amazon.com/ec2/instance-types/f1/.

[7] Zynq-7000 all programmable SoC. Online Webpage, 2017 (accessed Apr 17, 2018). https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html.

[8] Michael Adler, KE Fleming, and Angshuman Parashar. LEAP scratchpads: Automatic memory and cache management for reconfigurable logic. In *Proceedings of the 19th International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 25–28, 2011.

[9] Selim G. Akl and Nicola Santoro. Optimal parallel merging and sorting without memory conflicts. *IEEE Transactions on Computers*, 36(11):1367–1369, 1987.

[10] Apple. The future is here: iPhone X. Online Webpage, 2017 (accessed Apr 9, 2018). https://www.apple.com/newsroom/2017/09/the-future-is-here-iphone-x/.

[11] Joshua S. Auerbach, David F. Bacon, Ioana Burcea, Perry Cheng, Stephen J. Fink, Rodric M. Rabbah, and Sunil Shukla. A compiler and

runtime for heterogeneous computing. In *Proceedings of the 49th Annual Design Automation Conference 2012 (DAC)*, pages 271–276, 2012.

[12] Jonathan Bachrach, Huy Vo, Brian C. Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: Constructing hardware in a Scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference (DAC)*, pages 1216–1225, 2012.

[13] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 176–186, 1991.

[14] Arash Beldachi and José L. Núñez-Yáñez. Run-time power and performance scaling in 28 nm FPGAs. *IET Computers & Digital Techniques*, 8(4):178–186, 2014.

[15] Nathan L. Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali G. Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[16] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 207–216, 1995.

[17] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[18] Dominik Brodowski. CPU frequency and voltage scaling code in the Linux^{TM}kernel. Online Webpage. https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.

[19] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 187–194, 1981.

[20] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason Helge Anderson, Stephen Dean Brown, and Tomasz S. Czajkowski. LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 33–36, 2011.

[21] Tao Chen, Alexander Rucker, and G. Edward Suh. Execution time prediction for energy-efficient hardware accelerators. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, pages 457–469, 2015.

[22] Tao Chen, Shreesha Srinath, Christopher Batten, and G. Edward Suh. An architectural framework for accelerating dynamic parallel algorithms on reconfigurable hardware. In submission.

[23] Tao Chen and G. Edward Suh. Efficient data supply for hardware accelerators with prefetching and access/execute decoupling. In *Proceedings of the 49th International Symposium on Microarchitecture (MICRO)*, pages 46:1–46:12, 2016.

[24] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 269–284, 2014.

[25] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 609–622, 2014.

[26] Shaoyi Cheng and John Wawrzynek. Architectural synthesis of computational pipelines with decoupled memory access. In *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, pages 83–90, 2014.

[27] Chipworks. Inside the Apple A7 from the iPhone 5s - updated. Online Webpage, 2013 (accessed Apr 9, 2018). https://www.chipworks.com/about-chipworks/overview/blog/inside-apple-a7-iphone-5s-updated.

[28] Jongsok Choi, Stephen Dean Brown, and Jason Helge Anderson. From pthreads to multicore hardware systems in LegUp high-level synthesis

for FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2867–2880, 2017.

[29] Jongsok Choi, Kevin Nam, Andrew Canis, Jason Helge Anderson, Stephen Dean Brown, and Tomasz S. Czajkowski. Impact of cache architecture and interface on performance and area of fpga-based processor/parallel-accelerator systems. In *Proceedings of the 20th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 17–24, 2012.

[30] Kihwan Choi, Karthik Dantu, Wei-Chung Cheng, and Massoud Pedram. Frame-based dynamic voltage and frequency scaling for a MPEG decoder. In *Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, 2002.

[31] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengil, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Christian Boehn, Oren Firestein, Alessandro Forin, Kang Su Gatlin, Mahdi Ghandi, Stephen Heil, Kyle Holohan, Tamas Juhasz, Ratna Kumar Kovvuri, Sitaram Lanka, Friedel van Megen, Dima Mukhortov, Prerak Patel, Steve Reinhardt, Adam Sapek, Raja Seera, Balaji Sridharan, Lisa Woods, Phillip Yi-Xiao, Ritchie Zhao, and Doug Burger. Accelerating persistent neural networks at datacenter scale. In *HotChips*, 2017.

[32] Eric S. Chung, John D. Davis, and Jaewon Lee. LINQits: Big data on little clients. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, pages 261–272, 2013.

[33] Eric S Chung, James C Hoe, and Ken Mai. CoRAM: An in-fabric memory architecture for FPGA-based computing. In *Proceedings of the 19th International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 97–106, 2011.

[34] Edmund M. Clarke, Masahiro Fujita, Sreeranga P. Rajan, Thomas W. Reps, Subash Shankar, and Tim Teitelbaum. Program slicing of hardware description languages. In *Proceedings of the 10th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 298–312, 1999.

[35] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher J. Hughes, Yong-Fong Lee, Daniel M. Lavery, and John Paul Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings*

of the 28th International Symposium on Computer Architecture (ISCA), pages 14–25, 2001.

[36] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees A. Vissers, and Zhiru Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[37] Steve Dai, Mingxing Tan, Kecheng Hao, and Zhiru Zhang. Flushing-enabled loop pipelining for high-level synthesis. In *Proceedings of the 51st Design Automation Conference (DAC)*, pages 76:1–76:6, 2014.

[38] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct 1974.

[39] Hadi Esmaeilzadeh, Emily R. Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011.

[40] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 212–223, 1998.

[41] W. Godycki, C. Torng, I. Bukreyev, A. Apsel, and C. Batten. Enabling realistic fine-grain voltage scaling with reconfigurable power distribution networks. In *Proceedings of the 47th International Symposium on Microarchitecture (MICRO)*, 2014.

[42] Google. VP9 video hardware RTLs. Online Webpage. http://www.webmproject.org/hardware/vp9/.

[43] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *Proceedings of the 17th International Conference on High-Performance Computer Architecture (HPCA)*, pages 503–514, 2011.

[44] David J. Greaves and Satnam Singh. Kiwi: Synthesis of FPGA circuits from parallel programs. In *Proceedings of the 16th IEEE International Symposium on Field-Programmable Custom Computing Machines*, pages 3–12, 2008.

[45] Yan Gu and Samarjit Chakraborty. Control theory-based DVS for interactive 3D games. In *Proceedings of the 45th Annual Design Automation Conference (DAC)*, 2008.

[46] Yan Gu and Samarjit Chakraborty. A hybrid DVS scheme for interactive 3D games. In *14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2008.

[47] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[48] MyungJoo Ham. devfreq: Generic dynamic voltage and frequency scaling (DVFS) framework for non-CPU devices. Online Webpage. https://github.com/torvalds/linux/tree/master/drivers/devfreq.

[49] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. DeSC: Decoupled supply-compute communication management for heterogeneous architectures. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, pages 191–203, 2015.

[50] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th International Symposium on Computer Architecture (ISCA)*, pages 37–47, 2010.

[51] Chen-Han Ho, Sung Jin Kim, and Karthikeyan Sankaralingam. Efficient execution of memory access phases using dataflow specialization. In *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, pages 118–130, 2015.

[52] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, 1961.

[53] Chang-Hong Hsu, Yunqi Zhang, Michael A. Laurenzano, David Meisner, Thomas Wenisch, Jason Mars, Lingjia Tang, and Ronald G. Dreslinski. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[54] Intel Corporation. *Intel FPGA SDK for OpenCL Programming Guide*.

[55] G. P. Jones and Nigel P. Topham. A comparison of data prefetching on an access decoupled and superscalar machine. In *Proceedings of the 30th International Symposium on Microarchitecture (MICRO)*, pages 65–70, 1997.

[56] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2017.

[57] Wonyoung Kim, M.S. Gupta, Gu-Yeon Wei, and D. Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *Proceedings of the IEEE 14th International Symposium on High Performance Computer Architecture (HPCA)*, 2008.

[58] David Koeplinger, Raghu Prabhakar, Yaqi Zhang, Christina Delimitrou, Christos Kozyrakis, and Kunle Olukotun. Automatic generation of efficient accelerators for reconfigurable hardware. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, pages 115–127, 2016.

[59] Sanjeev Kumar, Christopher J. Hughes, and Anthony D. Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *34th International Symposium on Computer Architecture*, pages 162–173, 2007.

[60] Yongin Kwon, Sangmin Lee, Hayoon Yi, Donghyun Kwon, Seungjun Yang, Byung-Gon Chun, Ling Huang, Petros Maniatis, Mayur Naik, and Yunheung Paek. Mantis: Automatic performance prediction for smart-

phone applications. In *Proceedings of the 2013 USENIX Annual Technical Conference*, 2013.

[61] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Forth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 63–74, 1991.

[62] Charles E. Leiserson. The Cilk++ concurrency platform. In *Proceedings of the 46th Design Automation Conference*, pages 522–527, 2009.

[63] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd International Symposium on Microarchitecture (MICRO-42)*, pages 469–480, 2009.

[64] Zhaoshi Li, Leibo Liu, Yangdong Deng, Shouyi Yin, Yao Wang, and Shaojun Wei. Aggressive pipelining of irregular applications on reconfigurable hardware. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 575–586, 2017.

[65] Yun Liang, Kyle Rupnow, Yinan Li, Dongbo Min, Minh N. Do, and Deming Chen. High-level synthesis: Productivity, performance, and software constraints. *Journal of Electrical and Computer Engineering*, 2012:649057:1–649057:14, 2012.

[66] Kevin T. Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. Thin servers with smart pipes: Designing SoC accelerators for memcached. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, pages 36–47, 2013.

[67] Feng Liu, Soumyadeep Ghosh, Nick P. Johnson, and David I. August. CGPA: coarse-grained pipelined accelerators. In *Proceedings of the 51st Annual Design Automation Conference*, pages 78:1–78:6, 2014.

[68] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014.

[69] Derek Lockhart, Gary Zibrat, and Christopher Batten. PyMTL: A unified framework for vertically integrated computer architecture research.

In *Proceedings of the 47th International Symposium on Microarchitecture (MICRO)*, pages 280–292, 2014.

[70] Rustam Miftakhutdinov, Eiman Ebrahimi, and Yale N Patt. Predicting performance impact of DVFS for realistic memory systems. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.

[71] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, April 1965.

[72] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 62–73, 1992.

[73] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 129–140, 2003.

[74] Nachiappan Chidambaram Nachiappan, Praveen Yedlapalli, Niranjan Soundararajan, Anand Sivasubramaniam, Mahmut T. Kandemir, Ravi Iyer, and Chita R. Das. Domain knowledge based energy management in handhelds. In *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[75] Kyle J. Nesbit and James E. Smith. Data cache prefetching using a global history buffer. In *Proceedings of the 10th International Conference on High-Performance Computer Architecture (HPCA)*, pages 96–105, 2004.

[76] Rishiyur S. Nikhil. Bluespec system verilog: efficient, correct RTL from high level specifications. In *Proceedings of the 2nd International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 69–70, 2004.

[77] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. UTS: an unbalanced tree search benchmark. In *19th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 235–250, 2006.

[78] OpenCores. AES (Rijndael) IP core. Online Webpage. http://opencores.org/project,aes_core.

[79] OpenCores. JPEG decoder in Verilog. Online Webpage. http://opencores.org/project,djpeg.

[80] OpenCores. SHA cores. Online Webpage. http://opencores.org/project,sha_core.

[81] OpenCores. Video compression systems. Online Webpage. http://opencores.org/project,video_systems.

[82] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th International Symposium on Microarchitecture (MICRO)*, pages 105–118, 2005.

[83] Vinicius Petrucci, Michael A. Laurenzano, John Doherty, Yunqi Zhang, Daniel Mossé, Jason Mars, and Lingjia Tang. Octopus-Man: QoS-driven task management for heterogeneous multicores in warehouse-scale computers. In *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[84] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming heterogeneous systems from an image processing DSL. *ACM Transactions on Architecture and Code Optimization*, 14(3):26:1–26:25, 2017.

[85] Peter P. Puschner and Alan Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2):115–128, 2000.

[86] Andrew Putnam, Dave Bennett, Eric Dellinger, Jeff Mason, Prasanna Sundararajan, and Susan J. Eggers. CHiMPS: A C-level compilation flow for hybrid CPU-FPGA architectures. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 173–178, 2008.

[87] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James R. Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24, 2014.

[88] Andrew Putnam, Susan J. Eggers, Dave Bennett, Eric Dellinger, Jeff Mason, Henry Styles, Prasanna Sundararajan, and Ralph Wittig. Performance and power of cache-based reconfigurable computing. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, pages 395–405, 2009.

[89] Qualcomm. Snapdragon 820 processor product brief. Qualcomm Product Brief, 2016. https://www.qualcomm.com/documents/snapdragon-820-processor-product-brief.

[90] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 519–530, 2013.

[91] Nadesh Ramanathan, John Wickerson, Felix Winterstein, and George A. Constantinides. A case for work-stealing on fpgas with opencl atomics. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 48–53, 2016.

[92] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David M. Brooks. MachSuite: Benchmarks for accelerator design and customized architectures. In *Proceedings of the 2014 IEEE International Symposium on Workload Characterization (IISWC)*, 2014.

[93] James Reinders. *Intel Threading Building Blocks: Outfitting C++ For Multi-Core Processor Parallelism*. O'Reilly, 2007.

[94] Arch Robison. A primer on scheduling fork-join parallelism with work stealing. Technical report, The C++ Standards Committee, 01 2014.

[95] Michael Roitzsch, Stefan Wächtler, and Hermann Härtig. ATLAS: Look-ahead scheduling using workload metrics. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

[96] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding fork-join parallelism into LLVM's intermediate representation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 249–265, 2017.

[97] Ofer Shacham, Omid Azizi, Megan Wachs, Stephen Richardson, and Mark Horowitz. Rethinking digital design: Why design must change. *IEEE Micro*, 30(6):9–24, 2010.

[98] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David M. Brooks. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, pages 97–108, 2014.

[99] Yiqiong Shi, Chan Wai Ting, Bah-Hwee Gwee, and Ye Ren. A highly efficient method for extracting FSMs from flattened gate-level netlist. In *International Symposium on Circuits and Systems (ISCAS)*, 2010.

[100] Dongkun Shin, Jihong Kim, and Seongsoo Lee. Low-energy intra-task voltage scheduling using static timing analysis. In *Proceedings of the 38th Annual Design Automation Conference (DAC)*, 2001.

[101] James E. Smith. Decoupled access/execute computer architectures. In *Proceedings of the 9th International Symposium on Computer Architecture (ISCA)*, pages 112–119, 1982.

[102] Ryan Smith. Chipworks disassembles Apple's A8 SoC. Online Webpage, 2014 (accessed Apr 9, 2018). http://www.anandtech.com/show/8562/chipworks-a8.

[103] Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. Spatial memory streaming. In *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA)*, pages 252–263, 2006.

[104] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel. CAPI: A coherent accelerator processor interface. *IBM Journal of Research and Development*, 59(1):7:1–7:7, Jan/Feb 2015.

[105] Vivienne Sze, Daniel F Finchelstein, Mahmut E Sinangil, and Anantha P Chandrakasan. A 0.7-V 1.8-mW H.264/AVC 720p video decoder. *IEEE Journal of Solid-State Circuits*, 44(11):2943–2956, 2009.

[106] Mingxing Tan, Bin Liu, Steve Dai, and Zhiru Zhang. Multithreaded pipeline synthesis for data-parallel kernels. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 718–725, 2014.

[107] Robert Tibshirani. Regression shrinkage and selection via the Lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 1996.

[108] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: Reducing the energy of mature computations. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 205–218, 2010.

[109] Kazutoshi Wakabayashi. C-based behavioral synthesis and verification analysis on industrial design examples. In *Proceedings of the 2004 Conference on Asia South Pacific Design Automation*, pages 344–348, 2004.

[110] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, 1981.

[111] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced CPU energy. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 1994.

[112] Clifford Wolf. Yosys Open SYnthesis Suite. Online Webpage, 2018 (accessed Apr 18, 2018). http://www.clifford.at/yosys/.

[113] Lisa Wu, Raymond J. Barker, Martha A. Kim, and Kenneth A. Ross. Navigating big data with high-throughput, energy-efficient data partitioning. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, pages 249–260, 2013.

[114] Inc. Xilinx. Vivado high-level synthesis. Online Webpage, 2018 (accessed Apr 12, 2018). https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[115] Xilinx, Inc. *Vivado Design Suite User Guide: High-Level Synthesis*.

[116] Xilinx, Inc. *UG473: 7 Series FPGAs Memory Resources*, 2016.

[117] Ke Xu and Chiu-sing Choy. Low-power H.264/AVC baseline decoder for portable applications. In *Proceedings of the 2007 International Symposium on Low Power Electronics and Design*, 2007.

[118] Hsin-Jung Yang, Kermin Fleming, Michael Adler, and Joel S. Emer. Optimizing under abstraction: Using prefetching to improve FPGA performance. In *Proceedings of the 23rd International Conference on Field programmable Logic and Applications (FPL)*, pages 1–8, 2013.

[119] Yuhao Zhu, Matthew Halpern, and Vijay Janapa Reddi. Event-based scheduling for energy-efficient QoS (eQoS) in mobile web applications. In *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[120] Yuhao Zhu and Vijay Janapa Reddi. High-performance and energy-efficient mobile web browsing on big/little systems. In *Proceedings of the 19th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2013.