LIBRA: ACHIEVING EFFICIENT INSTRUCTION- AND DATA- PARALLEL EXECUTION FOR MOBILE APPLICATIONS

by

Yongjun Park

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Electrical Engineering)
in The University of Michigan
2013

To my family

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the guidance and support of many people. First, I would like to thank my advisor, Scott Mahlke. His insight, expertise, enthusiasm, and encouragement played a large part in my success in graduate school. Without his guidance and long endurance, this dissertation would not exist.

I would also like to thank my thesis committee members, Professor Trevor Mudge, David Blaauw, and Vineet Kamat. They donated their time, providing valuable comments and suggestions that helped me refine my thesis.

The research presented in this dissertation is not the work of one person; I was fortunate to have the assistance of a number of other students in the Compilers Creating Custom Processors research group. In particular, Hyunchul Park is the most thankful person for my whole successful graduate student life: research, writing papers, and also valuable help in all aspects of Ann Arbor life. Sangwon Seo also contributed significantly, helping me understand mobile applications, SIMD architectures, and rules of football. Hyoun Kyu Cho also gave me valuable help on power measurement on SIMD papers. More recently, Jason Jong Kyu Park has been great support in Libra accelerator related project.

In addition to this, I appreciate the opportunity to have worked with a group of people at Samsung Advanced Institute of Technology: Sukjin Kim, Kiseok Kwon, Jaeun Park, and

Taewook Oh. They provided their framework with advanced mobile application benchmarks which were used everywhere in my thesis, and free food.

I would also like to thank several people in special: Mark Woh, Gaurav Chadha, and Anoushe Jamshidi. They helped me improve initial research ideas into reasonable and well-structured products with discussion. I am also specially grateful to Shantanu Gupta, Ganesh Dasika, Amin Ansari, and Shuguang Feng, for encouraging me not to loose my confidence in research.

Beyond those who helped in technical way, those who offered engaging conversation and moral support were crucial to my graduate school life, namely: Mehrzad Samadi, Daya Khudia, Janghaeng Lee, Andrew Lukefahr, Shruti Padmanabha, Ankit Sethia, Amir Hormati, Mojtaba Mehrara, Yuan Lin, Kevin Fan, Griffin Wright, Po-Chun Hsu, Jeff Hao, Davood Salehabadi, and Manjunath Kudlur. I have shared offices, homes with these friends, and have had joyful group lunch meetings, and therfore, my time in Ann Arbor would not have been the same without them.

I would like to thank my family for their support, encouragement, and advice. My parents, Chanyoung Park and Eunji Byeun, and my brother Myoungjun Park provided their unconditional love and support throughout this whole process.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

LIBRA: ACHIEVING EFFICIENT INSTRUCTION- AND DATA- PARALLEL EXECUTION FOR MOBILE APPLICATIONS

by

Yongjun Park

Chair: Scott A. Mahlke

Mobile computing as exemplified by the smart phone has become an integral part of our daily lives. The next generation of these devices will be driven by providing richer user experiences and compelling capabilities: higher definition multimedia, 3D graphics, augmented reality, and voice interfaces. To meet these goals, the core computing capabilities of the smart phone must be scaled. But, the energy budgets are increasing at a much lower rate, thus fundamental improvements in computing efficiency must be garnered. To meet this challenge, computer architects employ hardware accelerators in the form of SIMD and VLIW. Single-instruction multiple-data (SIMD) accelerators provide high degrees of scalability for applications rich in data-level parallelism (DLP). Very long instruction word (VLIW) accelerators provide moderate scalability for applications with high degrees of instruction-level parallelism (ILP). Unfortunately, applications are not so nicely partitioned into two groups: many applications have some DLP, but also contain significant fractions of code with low trip count loops, complex control/data dependences, or non-uniform execution behavior for which no DLP exists. Therefore, a more adaptive accelerator is required to be able to deploy resources as needed: exploit DLP on SIMD when it's available, but fall back to ILP on the same hardware when necessary.

In this thesis, we first focus on various compiler solutions that solve inefficiency problem in both VLIW and SIMD accelerators. For SIMD accelerators, a new vectorization pass, called SIMD Defragmenter, is introduced to uncover hidden DLP using subgraph identification in SIMD accelerators. CGRA express effectively accelerates sequential code regions using a bypass network in VLIW accelerators, and Resource Recycling leverages stream-graph modulo scheduling technique for scheduling of multiple code regions in multi-core accelerators.

Second, we explore potential solutions in the context of mobile applications for scaling the performance of tiled accelerators in an energy efficient manner: homogeneous versus heterogeneous functionality, interconnect topologies, simple versus complex processing elements, and scalar versus vector memory support. We then propose the new scalable multicore accelerator referred to as *Libra* for mobile systems, which can support execution of code regions having both DLP and ILP, as well as hybrid combinations of the two. We believe that as industry requires higher performance, the proposed flexible accelerator and compiler support will put more resources to work in order to meet the performance and power efficiency requirements.

CHAPTER 1

Introduction

The mobile devices market, including cell phones, netbooks, and personal digital assistants, is one of the most highly competitive businesses. The computing platforms that go into these devices must provide ever increasing performance capabilities while maintaining low energy consumption in order to support advanced multimedia and signal processing applications. Application-specific integrated circuits (ASICs) were the most common solutions for the heavy lifting, performing the most compute intensive kernels in a high performance but energy-efficient manner. However, several features push designers to a more flexible and programmable solution: supporting multiple applications or variations of applications, providing faster time-to-market, and enabling algorithmic changes after the hardware is constructed.

Traditionally, the design of programmable mobile computing platforms has focused on software defined radio [17, 16, 33, 59, 89]. These systems are geared towards wireless signal processing that contains vast amounts of vector parallelism. As a result, wide single-instruction multiple-data (SIMD) hardware is recognized as an effective strategy to achieve both high-performance and programmability. SIMD provides high efficiency because of its regular structure, ability to scale lanes, and low control cost. However, mobile computing systems are not limited to wireless signal processing. High-definition video, audio, 3D graphics, and other forms of media processing are high value applications for mobile terminals. In fact, many believe the quality and types of media support will be the key differentiating factors of future mobile terminals.

Such media applications in a mobile environment offer a different challenge than wireless signal processing. First, the complexity of media processing algorithms is typically higher than signal processing. Computation is no longer dominated by simple vectorizable loops. Instead, current media processing algorithms are more like general-purpose programs with data-level parallelism (DLP) available selectively and to varying degrees. Second, significant amounts of control/data dependencies to handle the complexity of media coding also reduce the fraction of SIMDizable loops. Finally, various application domains have totally different amounts of SIMD parallelism. As a result, the applications are more dependent on the instruction-level parallelism (ILP) for performance. Coarse-grained reconfigurable architectures (CGRA) are a variant of VLIW processors that exploit high degrees of ILP with low cost/energy implementations [61, 85, 65, 66, 61, 78]. Loops are modulo scheduled onto the CGRA to utilize the large number of resources and achieve high performance [65, 72, 73].

To support both ILP- and DLP-rich applications, today's smart phones simply use multiple different types of accelerators: a baseband accelerator for DLP and a media accelerator for ILP. This is because running DLP-rich applications on VLIW accelerators is energy-inefficient due to massive hardware overhead such as register file (RF) and interconnect complexity, and running ILP-rich applications on SIMD accelerators is also ineffective as available SIMD resources cannot be fully utilized and a substantial portion of resources are idle at runtime. However, using multiple solutions still incurs three critical problems: 1) static power dissipation and poor area utilization due to presence of multiple separate hardware accelerators, 2) poor execution efficiency as applications are typically not solely ILP or DLP applications, but rather contain hybrid forms of parallelism that force some execution on mismatched hardware, and 3) higher software development costs as applications must be partitioned and customized to separate accelerators.

Based on the above observation, the fundamental sources of inefficiency range from a mismatch between program characteristics and the target accelerator, to a heterogeneous system which incur multiple idle hardware instances. In the context of program-architecture mismatch, we first attack current challenges for efficiently utilizing existing mobile media accelerators. The specific purpose of this effort is to find the potential code region which will not fully utilize the given resources on a target accelerator, and optimize the region to be favorable to the accelerator. In this thesis, three compilation techniques with small architectural modifications for efficient mapping of applications onto three DLP-, ILP-, and task level parallelism-based accelerators are proposed: 1) the SIMD Defragmenter to uncover hidden DLP that lurks below the surface in the form of ILP, 2) the sub-cycle modulo scheduler to effectively accelerate latency-constrained code regions using a bypass network , and 3) a compilation framework to maximize application throughput with hybrid resource partitioning of a flexible multi-core system.

While these compiler backend optimizations show substantial performance improvement with higher resource utilization on existing accelerators, architecture-specific optimizations are likely insufficient for solving the fundamental problem of heterogeneous systems-multiple idle hardwares, but only improve execution efficiency when some code region is executed on mismatched hardware. This motivates a designing of a unified accelerator that can support multiple forms of parallelism by dynamically tuning execution strategy. Therefore, the second overarching objective of this thesis is to design and evaluate a mobile unified accelerator with high scalability, flexibility, and energy efficiency. To achieve this, we find several reasons why current tiled accelerators fail to meet future performance requirements and discuss the feasibility of their potential solutions. Based on these intuitions, we then propose a unified multi-core accelerator that is capable of customizing its execution strategy to the running application, referred to as *Libra*. The above compiler optimizations can be directly applied to the Libra accelerator since the basic building blocks of the Libra accelerator can support all three levels of parallelism.

1.1 Compiler Support for Various Accelerator Models

1.1.1 Improving DLP Performance

Single-instruction multiple-data (SIMD) accelerators provide an energy-efficient platform to scale the performance of mobile systems while still retaining post-programmability. The central challenge is translating the parallel resources of the SIMD hardware into real application performance. In scientific applications, automatic vectorization techniques have proven quite effective at extracting large levels of data-level parallelism (DLP). However, vectorization is often much less effective for media applications due to low trip count loops, complex control flow, and non-uniform execution behavior. As a result, SIMD lanes remain idle due to insufficient DLP.

To attack this problem, Chapter 2 proposes a new vectorization pass called *SIMD Defragmenter* to uncover hidden DLP that lurks below the surface in the form of instruction-level parallelism (ILP). The difficulty is managing the data packing/unpacking overhead that can easily exceed the benefits gained through SIMD execution. The SIMD defragmenter overcomes this problem by identifying groups of compatible instructions (subgraphs) that can be executed in parallel across the SIMD lanes. By SIMDizing in bulk at the subgraph level, packing/unpacking overhead is minimized.

1.1.2 Improving ILP Performance

Coarse-grained reconfigurable architectures (CGRAs) present an appealing hardware platform by providing programmability with the potential for high computation throughput, scalability, low cost, and energy efficiency. CGRAs have been effectively used for innermost loops that contain an abundant of instruction-level parallelism. Conversely, non-loop and outer-loop code are latency constrained and do not offer significant amounts of instruction-level parallelism. In these situations, CGRAs are ineffective as the majority of the resources remain idle.

In Chapter 3, *dynamic operation fusion* is introduced to enable CGRAs to effectively accelerate latency-constrained code regions. Dynamic operation fusion is enabled through the combination of a small bypass network added between function units in a conventional CGRA and a sub-cycle modulo scheduler to automatically identify opportunities for fusion.

1.1.3 Improving Task Level Parallelism Performance

To handle complexities of media applications, composable accelerators such as the *Polymorphic Pipeline Array*, or PPA, present an appealing hardware platform by adding a degree of hardware configurability over existing CGRAs. Hardware resources can be both statically as well as dynamically partitioned among executing tasks to maximize execution efficiency. However, an effective compilation framework is essential to partition and assign resources to make intelligent use of the available hardware.

In Chapter 4, a compilation framework is introduced that maximizes application throughput with hybrid resource partitioning of a PPA system. Static partitioning handles part of the resource assignment, but this is followed up by dynamic partitioning to identify idle resources and put them to use – *resource recycling*.

1.2 Design of Future Mobile Accelerators

1.2.1 Finding the Guideline for Developing Future Tiled Architectures

Tiled multi-core architectures are an appealing hardware platform for mobile systems by providing programmability with the potential for high computational throughput, low cost, and energy efficiency. Unfortunately, current tiled architectures fail to meet future performance requirements due to their inability to scale. Simply increasing the size of the array is too expensive in terms of power and area.

In Chapter 5, we first perform a deep analysis of several mobile applications from the domains of multimedia and gaming. We then explore potential solutions in the context of these applications for scaling the array performance in an energy efficient manner: homogeneous versus heterogeneous functionality, interconnect topologies, simple versus complex processing elements, and scalar versus vector memory support.

1.2.2 Libra Accelerator

To design a mobile unified accelerator, we start from traditional SIMD accelerators because they offer the combination of high performance and low energy consumption through low control and interconnect overhead. However, SIMD accelerators are not a panacea. Many applications lack sufficient vector parallelism to effectively utilize a large number of SIMD lanes. Further, the use of symmetric hardware lanes leads to low utilization and high static power dissipation as SIMD width is scaled. To address these inefficiencies, chapter 6 focuses on breaking two traditional rules of SIMD processing: homogeneity and static configuration. The *Libra* accelerator increases SIMD utility by blurring the divide between vector and instruction parallelism to support efficient execution of a wider range of loops, and it increases hardware utilization through the use of heterogeneous hardware across the SIMD lanes.

In Libra, multiple small cores enable the SIMD execution for exploiting DLP and, when there is a high degree of ILP within a loop, a larger core can be created by merging small cores. With this flexible execution model, different levels of parallelism can be exploited with a single piece of hardware. For example, Libra can execute as a wide-SIMD datapath and also Libra behaves as a VLIW accelerator. Libra also supports mixed-mode execution where the fraction of ILP and DLP is configured. Libra consists of an array of simple processing elements (PEs) that are tightly interconnected by a scalar operand network and a shared memory similar to a CGRA [66]. Groups of four PEs form cores that are normally driven by a single instruction stream. Each core can behave as a building block for a SIMD processor (e.g., cores behave as SIMD lanes) or a CGRA (e.g., cores behave as a cluster of function units in the VLIW-style CGRA). Cores feature dense interconnection between the PEs, while sparse interconnection is available across cores to provide better cost and energy scalability. The compiler maps 1 or more loops to Libra by combining and configuring clusters of cores to efficiently exploit the available DLP and ILP.

CHAPTER 2

Efficient ILP Realization on Data-parallel Architectures

2.1 Introduction

The number of worldwide mobile phones in use exceeded five billion in 2010 and is expected to continue to grow. The computing platforms that go into these and other mobile devices must provide ever increasing performance capabilities while maintaining low energy consumption in order to support advanced multimedia and signal processing applications. Application-specific integrated circuits (ASICs) were the most common solutions for the heavy lifting, performing the most compute intensive kernels in a high performance but energy-efficient manner. However, new demands push designers toward a more flexible and programmable solution: supporting multiple applications or variations of applications, providing faster time-to-market, and enabling algorithmic changes after the hardware is constructed.

Processors that exploit instruction-level parallelism (ILP) provide the highest degree of computing flexibility. Modern smart phones employ a one GHz dual-issue superscalar

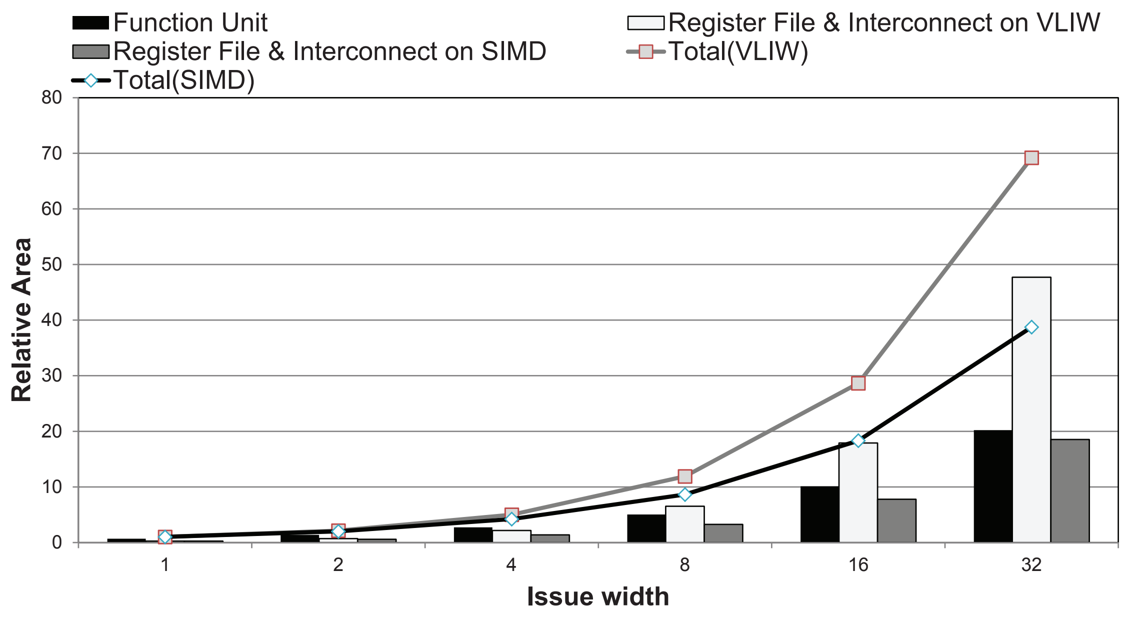


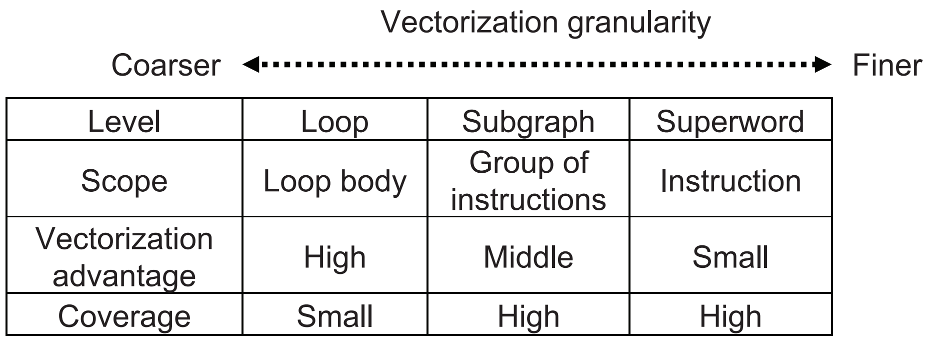
Figure 2.1: Scalability of datapaths that exploit instruction-level parallelism (VLIW) and data-level parallelism (SIMD). Plotted is the relative area as issue width increases from 1 to 32. Area is broken down into function unit and register file & interconnect.

ARM as an application processor. Higher performance digital signal processors are also available such as the 8-issue TIC6x. However, the scalability of ILP processors is inherently limited by register file (RF) and interconnect complexity as shown in Figure 2.1. Single-instruction multiple-data (SIMD) accelerators have long been used in the desktop space for high performance multimedia and graphics functionality. But, their combination of scalable performance, energy efficiency, and programmability make them ideal for mobile systems as well [17, 16, 33, 59, 89]. Figure 2.1 shows that the area of SIMD datapaths scale almost linearly with issue width. Power follows a similar trend [89]. SIMD architectures provide high efficiency because of their regular structure, ability to scale lanes, and low control overhead.

The difficult challenge with SIMD is programming. The application developer or compiler must find and extract sufficient data-level parallelism (DLP) to efficiently make use of the parallel hardware. Automatic loop vectorization is a popular approach and is available in a variety of commercial compilers including offerings from Intel, IBM, and PGI. Applications that resemble classic scientific computing (regular structure, large trip count loops, and few data dependences) perform well on most SIMD architectures.

However, mobile applications are not limited to these types of applications. High-definition video, audio, 3D graphics, and other forms of media processing are high value applications for mobile devices. These applications continue to grow in complexity and resemble scientific applications less and less. Computation is no longer dominated by simple vectorizable loops. Instead, current media processing algorithms behave more like general-purpose programs with DLP available selectively and to varying degrees in different loops. Also, significant amounts of control flow are present to handle the complexity of media coding and limits the available DLP. The overall affect is that loop-level DLP is less prevalent and less efficient to exploit in media algorithms. Due to these application-specific complexities, available SIMD resources cannot be fully utilized and a substantial portion of resources are idle at runtime. Talla [84] reports that only 1-4% performance improvement exists when scaling the SIMD components from 2-way to 16-way on the MediaBench suite [54]. Thus, an improved approach beyond simple loop level techniques is necessary in order to effectively use wide SIMD resources.

To supplement the insufficient degree of DLP from traditional vectorization, superword-level parallelism (SLP) [52] can be applied. SLP is a short SIMD parallelism between isomorphic instructions within a basic block. As shown in Figure 2.2, SLP can cover more code regions as compared to loop-level vectorization because SLP can be performed in


Vectorization granularity
Coarser
Finer

Level	Loop	Subgraph	Superword
Scope	Loop body	Group of instructions	Instruction
Vectorization advantage	High	Middle	Small
Coverage	Small	High	High

Figure 2.2: A spectrum of the vectorization at different granularities.

non-loop regions, in loops having cross-iteration dependences, and in outer loops. For vectorizable loops, traditional vectorization is preferred because SLP misses loop-specific optimization opportunities [67]. The weakness of SLP is that the vectorization scope is too fine, resulting in a high overhead of getting data into packed format that is suitable for SIMD execution. Often, this packing overhead can exceed the benefits of parallel execution on the SIMD hardware. In addition, SLP is performed with a local scope that commonly misses opportunities for vectorization when a large number of isomorphic instructions exist.

To address the limitations of SLP, we introduce a coarser level of vectorization within basic blocks, referred to as *Subgraph Level Parallelism (SGLP)*. SGLP refers to the parallelism between subgraphs (groups of instructions) having identical operators and dataflow inside a basic block: parallel subgraphs that can execute together on separate data. SGLP has two major advantages that allow it more opportunities to convert ILP to DLP: 1) data rearrangement and packing overhead can be minimized by encapsulating the data flow inside the subgraph, 2) natural functional symmetries that exist in media applications (e.g., a sliding window of data long which computation is performed) can be exposed to enable vectorization of larger groups of instructions. The net result is SGLP leads to a combination of more SIMD execution opportunities and fewer instructions dedicated to data reorganization and inter-lane data movement.

This chapter presents the design of a supplemental vectorization pass referred to as the *SIMD Defragmenter*. It automatically identifies and extracts SGLP from vectorized loops and orchestrates parallel execution of subgraphs with minimum overhead using unused resources. In the SIMD Defragmenter, a loop is first vectorized using traditional vectorization techniques. Then, vectorizable subgraphs are identified based on the availability of unused lanes in the hardware. The compiler then allocates the subgraphs to unused SIMD resources to minimize inter-lane data movement. Finally, new SIMD operations for SGLP are emitted and operations for inter-lane movements are added where necessary. Small architectural features are provided to enhance the applicability of SGLP and the configuration is statically generated during compilation.

This work offers the following three contributions:

- An analysis of the difficulties of putting SIMD resources to efficient use across three mobile media applications (MPEG4 audio decoding, MPEG4 video decoding, and 3D graphics rendering).

- The introduction of SGLP that can efficiently exploit unused SIMD resources on already vectorized code.

- A compilation framework for SGLP that identifies isomorphic subgraphs and selects a mapping strategy to minimize data reorganization overhead.

2.2 Background and Motivation

In this section, we examine the current limitations of SIMD architectures based on an analysis of the following three widely used multimedia applications:

- AAC decoder: MPEG4 audio decoding, low complexity profile
- H.264 decoder: MPEG4 video decoding, baseline profile, qcif
- 3D: 3D graphics rendering

We then analyze why the well-known solutions are not as effective as expected. Finally, we discuss several potential approaches to overcome these bottlenecks and increase the utilization of existing resources.

2.2.1 Baseline Architecture Overview

A basic SIMD architecture that is based on SODA [59] (Figure 2.3) is used as the baseline architecture. This architecture has both SIMD and two scalar datapaths. The SIMD pipeline consists of a multiple-way datapath where each way has an arithmetic unit working in parallel. Each datapath has a two read-ports, one write-port, a 16 entry register file, and one ALU with a multiplier. The number of ways in the SIMD pipeline can vary depending on the characteristics of target applications. The SIMD Shuffle Network (SSN) is implemented to support intra-processor data movement. The scalar pipeline consists of one 16-bit datapath and supports the application's control code. The AGU pipeline handles DMA (Direct Memory Access) transfers and memory address calculations for both scalar and SIMD pipelines.

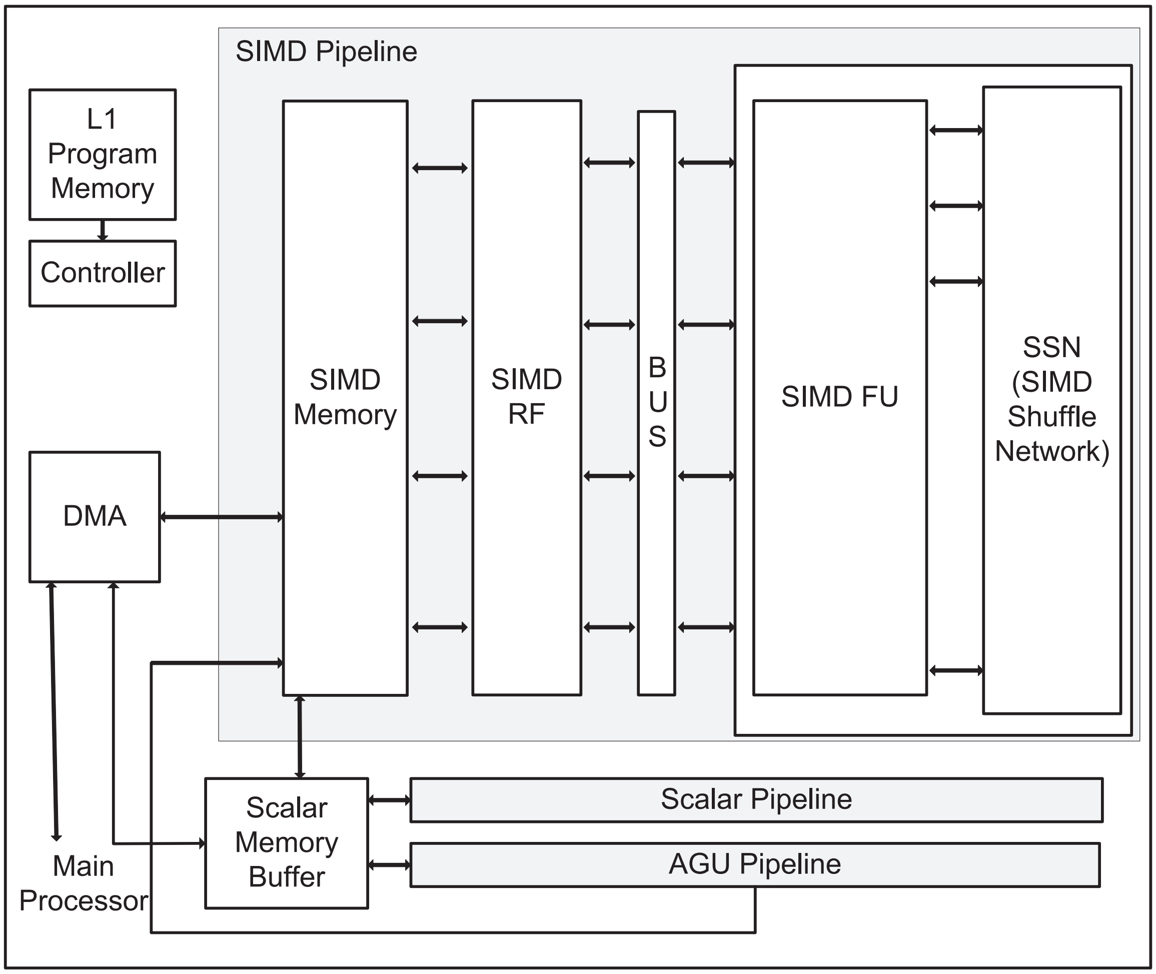


Figure 2.3: Baseline SIMD architecture.

2.2.2 Analysis of Multimedia Applications

SIMD architectures provides an energy-efficient means of executing multimedia applications. However, it is difficult to determine the optimal number of SIMD lanes because the number depends on the algorithms that constitute the workload. In this analysis, we first categorize the innermost loops of three applications into different groups according to their vector width. Then, two types of SIMD width variance are identified and the practical difficulties of finding the optimal SIMD width and achieving high utilization are discussed.

2.2.2.1 SIMD Width Characterization

Multimedia applications typically have many compute intensive kernels that are in the form of nested loops. Among these kernels, we analyze the available DLP of the innermost loops and find the maximum natural vector width which is achievable. Based on the Intel Compiler [41], the rules to be selected as a vectorizable innermost loop are as follows:

- The loop must contain straight-line code. No jumps or branches, but predicated assignments, are allowed only when the performance degradation is negligible.
- The loop must be countable and there must be no data-dependent exit conditions.
- Backward loop-carried dependencies are not allowed.
- All memory transfers must have same strides over iteration.

If a loop satisfies the above four conditions, the minimum iteration count is set to the vector width of the loop.

2.2.2.2 SIMD Width Variance

Figure 2.4 shows how many different natural vector widths reside in the three target benchmarks. The execution time breakdown between loops having different vector widths are shown in Figure 2.4. The three pie charts show the distribution of scalar execution time spent in innermost loops at various SIMD widths for three applications. From Figure 2.4, we can see that there are many different vector widths inside each application, hence it is quite difficult to determine the optimal SIMD width even for one application. For example, to define 16 as the SIMD width for H.264 is not desirable because the maximum vector

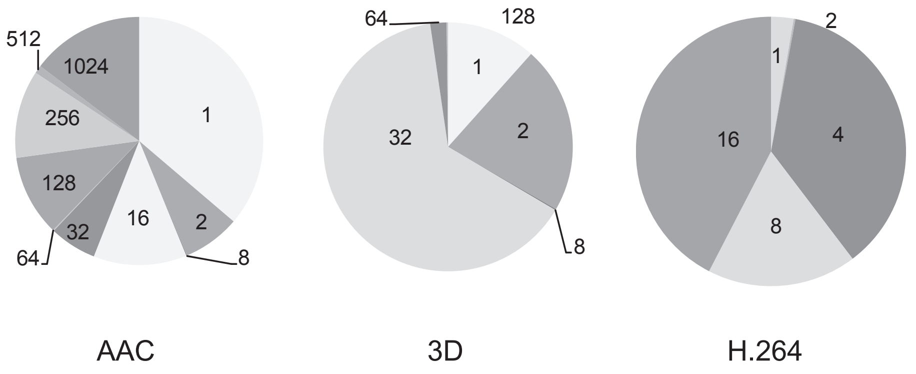


Figure 2.4: Scalar execution time distribution at different SIMD widths for three media applications: the maximum SIMD widths are 1024, 128, and 16, and the SIMD widths, which can be fully utilized for more than 50% execution time, are 16, 32, and 8 for AAC, 3D, and H.264 applications.

width is 16 but the execution time ratio of loops with vector width of 16 is just 42% and some SIMD lanes are wasted for the remaining time. On the other hand, four is also not desired because the execution time of the loops with a width of four is not dominant with substantial execution occurring in loops having larger SIMD widths. Similarly, AAC and 3D applications cannot set the number of SIMD lanes as the maximum vector width due to the waste of resources, nor dominant vector width due to the low performance. Therefore, effectively supporting multiple SIMD widths is required to take advantage of the SIMD architectures.

Dynamic power gating is one of the most successful energy saving techniques for the resource waste problem. Each SIMD lane can be selectively cut off from the power rails when the lane is not utilized using a MOSFET switch. This technique is attractive because it is effective for dynamic power saving and also has positive impact on leakage power savings. Although dynamic power gating achieves high energy savings, the relatively high

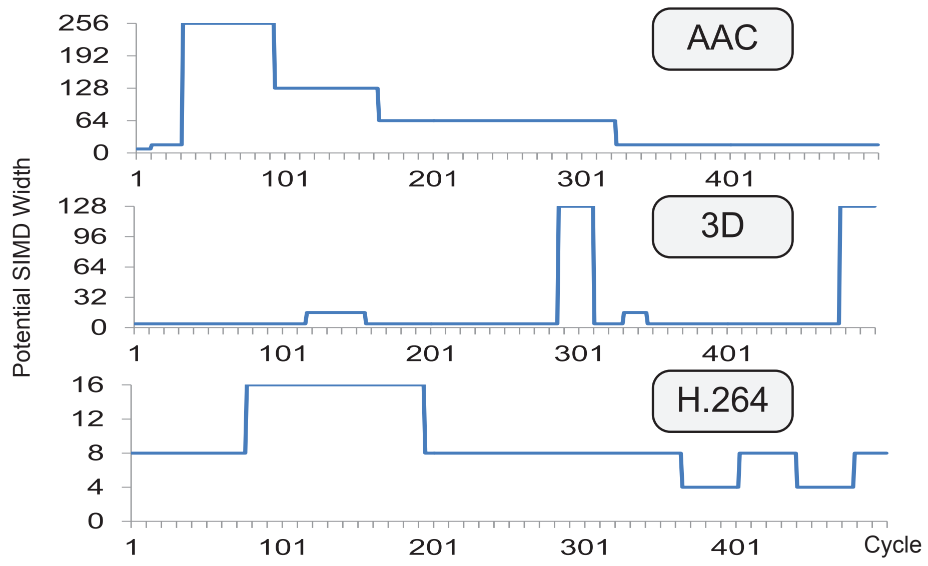


Figure 2.5: The SIMD width requirement changes at runtime: The X-axis indicates the execution clock cycle and the Y-axis is the maximum SIMD width assuming infinite resources. The minimum duration between width transition is 20 cycles from 311 to 330 for 3D application.

overhead when changing modes prevents current SIMD architectures from applying it [75]. Even applying simple dynamic power gating techniques [63, 39, 62] is not effective since at least a few microseconds are required to compensate the power on/off energy overhead in current technologies. Figure 2.5 shows the SIMD width requirement changes over the runtime for three applications. The x-axis is the time stamp for 500 cycles when the SIMD architecture supports infinite DLP and the y-axis is the natural SIMD width that achieves the best performance. As shown, power gating cannot even compensate the transition energy overhead because of frequent power mode transitions within less than 200 cycles (1 μs at 200 Mhz) based on the different SIMD width requirements. Moreover, power gating comes with about 8% area overhead due to the header/footer power gate switch implementation. Therefore, power gating is hard to integrate into current SIMD architectures.

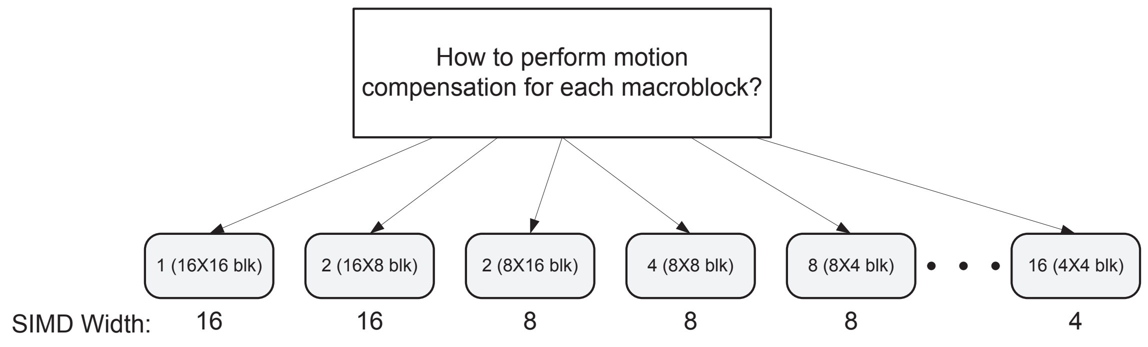


Figure 2.6: Different SIMD width requirements for each macroblock in the motion compensation process in H.264 decoder. The information is provided at runtime.

Thread-level Parallelism (TLP) for SIMD architectures has also been proposed to solve the temporal resource waste due to the small amount of DLP [90]. TLP supports running multiple threads that work on separate data on a wide SIMD machine when the SIMD width is small. By exploiting two kinds of parallelism, the SIMD lanes can be maximally utilized but the realization of TLP's potential in SIMD architectures has some critical limitations. First, TLP might not be fully exploited if parallel threads have different instruction flow. The motion compensation process for the H.264 decoder is a well-known example of this case. Figure 2.6 shows the various configurations of the motion compensation process for one macroblock. In this figure, the configuration of each macroblock is different so that SIMD specific restriction, which needs to execute the same instruction stream across the lanes, prohibits executing multiple processes in parallel even though the process has high TLP. Second, TLP cannot handle input-dependent control flow. For example, conditions to choose the macro block configuration in Figure 2.6 are decided from input header data hence TLP cannot be considered in the compilation phase. Finally, TLP generally requires more memory pressure. As a result, TLP looks appealing but the actual implementation of

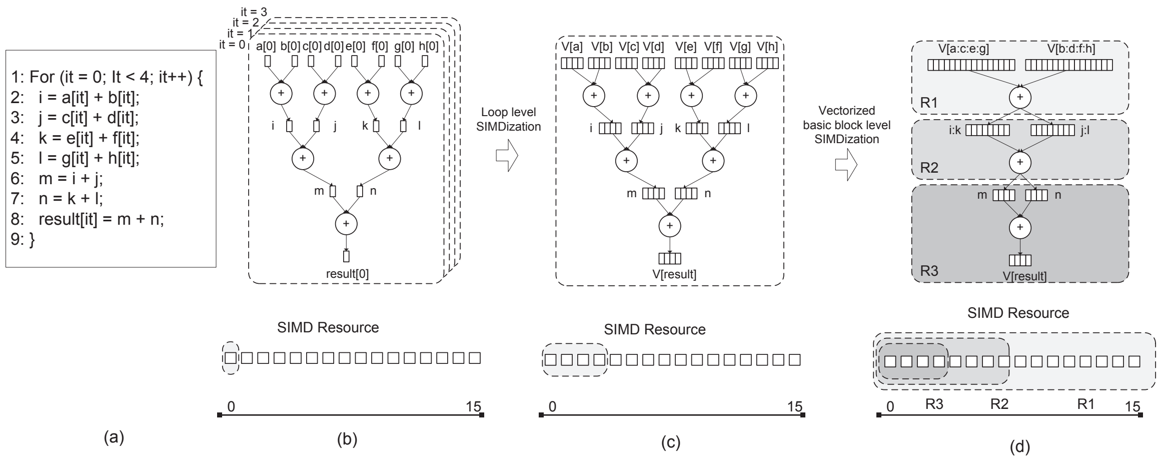


Figure 2.7: Different levels of parallelism: (a) an example loop's source code, (b) original multiple scalar subgraphs utilizing a single SIMD lane, (c) a vectorized subgraph using four SIMD lanes, and (d) the opportunity of partial SIMD parallelism inside the vectorized basic block (SIMD lane utilization: (R1: 16), (R2: 8), (R3: 4))

it is complicated.

The analysis reveals the difficulty of implementing common solutions in the real world. To further improve resource utilization, it is necessary to find a way to exploit other forms of parallelism.

2.2.3 Beyond Loop-level SIMD Parallelism

Most kernels have some degree of DLP, which can be easily vectorized using loop unrolling. An interesting question is how to find extra parallelism when the degree of DLP is smaller than the degree supported in the architecture. For this question, the next opportunity can be found inside the vectorized basic block. Even if the basic block is not fully vectorizable, some parts inside the block can be vectorized as a restricted form of ILP. Compared to ILP, DLP requires two more conditions: 1) the instructions should perform the same work and 2) data flow should also be in the same form. Therefore, parallel instructions with the same opcode can be executed together in a SIMD architecture. Figure 2.7 shows examples of additional SIMD parallelism inside the vectorized basic block for our three applications. Figure 2.7 (a) is a vectorizable loop to generate the sum of eight input data arrays. (b) shows the unrolled dataflow graph (DFG) that can be executed in only one lane when assuming a 16-way SIMD datapath. This loop can then be vectorized as shown in (c) and four lanes can be assigned as the trip count of the loop but still 12 lanes are idle. In this case, another opportunity for partial SIMD parallelism can be found inside the basic block as illustrated in (d). Four ADD instructions in the 'R1' region are able to execute together with 16 degrees of parallelism, two ADD instructions in the 'R2' region can also execute together using eight lanes. Based on the application analysis, more than 50% of total instructions have at least one parallel identical instruction.

2.2.4 Summary and Insights

The analysis of these three applications provides several insights. First, resource utilization of a wide SIMD architecture is low because multimedia applications have various degrees of SIMD parallelism, and current solutions are not effective due to the high dynamic variance and the unpredictability. Second, ILP inside the vectorized basic block can be converted to DLP in many cases. Therefore, additional partial SIMD parallelism can be added when the DLP is insufficient.

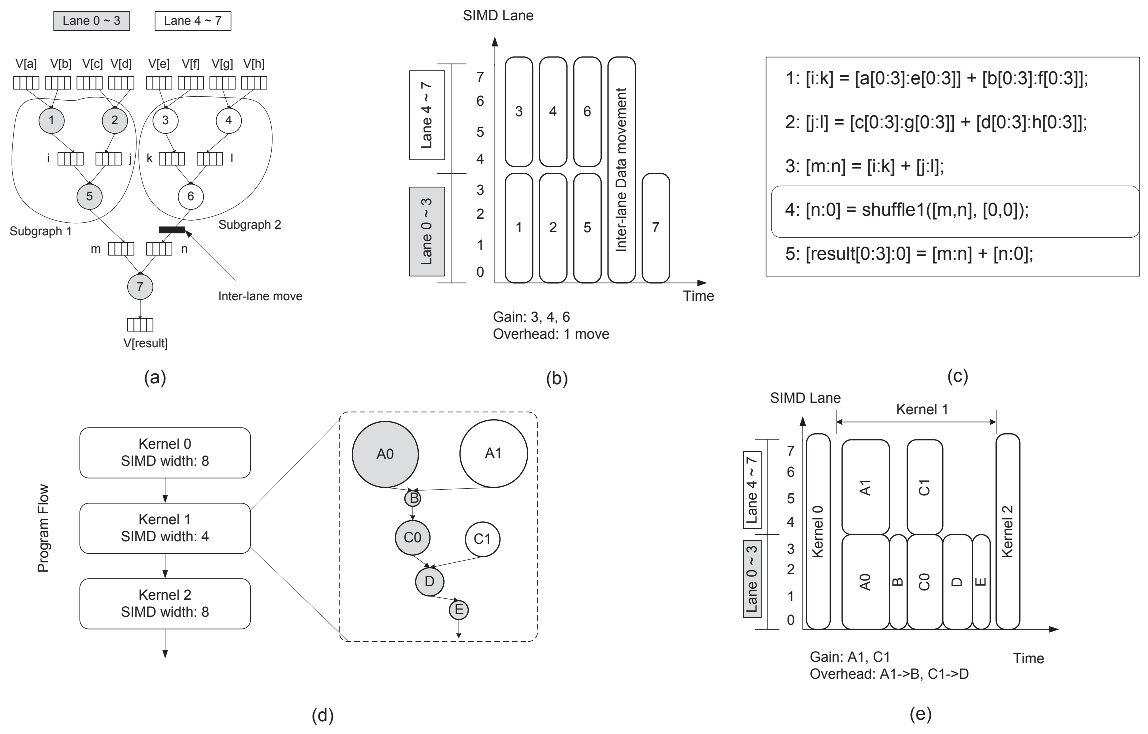
A major challenge is how to minimize the data movement across the different SIMD lanes. For loop-level DLP, inter-lane data movement does not happen, whereas partial DLP has a large number of such movements due to each part having different levels of DLP, causing the amount of the occupied SIMD lanes to change at runtime in such a manner that the data packing/unpacking/reorganizing process happens frequently. For example, two data movements across the lane need to be done when exploiting partial SIMD parallelism in Figure 2.7 (d): 1) ’R1’ to ’R2’: After the 16-wide instruction, half of the data in lanes 9 to 15 should move to 0 to 8, and 2) ’R2’ to ’R3’: After the 8-wide instruction, half of the data in lanes 4 to 7 should move to 0 to 3. Therefore, we can save just two total instructions due to the data movement even if we save four instructions on partial SIMD parallelism. The conclusion is that minimizing inter-lane data movements is the key challenge in getting benefits from partial SIMD parallelism.

2.3 Subgraph Level Parallelism

This section describes a new vectorization technique. We first introduce some new terminologies and discuss its effectiveness in contrast to other related techniques. An execution model using SGLP is then proposed on the conventional wide SIMD architecture. Finally, we list practical challenges to exploit this parallelism and suggest proper solutions.

2.3.1 Overview

Subgraph level parallelism is defined as SIMD parallelism between *identical* subgraphs which 1) have an isomorphic form of dataflow with SIMDizable operations and 2) have no dependencies on each other inside the basic block. This parallelism is detected through the identical subgraph search inside the whole dataflow graph extracted from a basic block.



```
1: [i:k] = [a[0:3]:e[0:3]] + [b[0:3]:f[0:3]];
2: [j:l] = [c[0:3]:g[0:3]] + [d[0:3]:h[0:3]];
3: [m:n] = [i:k] + [j:l];
4: [n:0] = shuffle1([m,n], [0,0]);
5: [result[0:3]:0] = [m:n] + [n:0];
```

Figure 2.8: Subgraph level parallelism: (a) identical subgraphs are identified, and (1, 2, 5, 7) and (3, 4, 6) are executed in parallel with one overhead, (b) execution of the graph on two SIMD lane groups, (c) SGLP exploited output source code, (d) high level program flow with three sequential kernels and kernel 1 can exploit SGLP, and (e) execution of three kernels with SGLP exploration on kernel 1.

These identical subgraphs can be executed in parallel in the form of a sequence of SIMD instructions inside the subgraph. There are two major advantages when searching packing opportunities at the subgraph level:

- **Packing steering:** SGLP minimizes the overall data reorganization overhead because the data movements between instructions inside a subgraph are automatically captured and assigned to one SIMD lane, and the alignment analysis between subgraphs is performed over a global scope. This benefit becomes more apparent when converting ILP to DLP in the low-DLP region such as loop-level vectorized or scalar code because the subgraph guides the instruction packing in directions that reduce or keep constant the amount of conversion overheads when the packing opportunities are not restricted by the memory alignment so that the number of possible packing combinations increase.

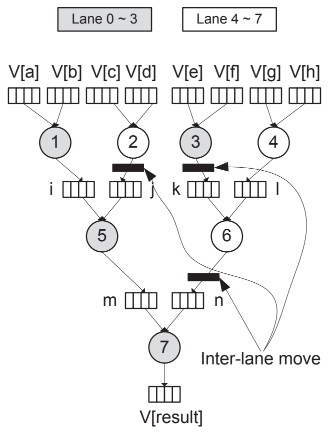
- **High packing gain:** Converting ILP to DLP is not common because it is hard to expect that the data reorganization process will provide enough gain to compensate for its performance loss due to the expensive nature. However, the considerable instruction savings of subgraph packing gives more chances to guarantee a positive net performance gain in spite of the substantial amount of overheads.

Figure 2.8 illustrates an example of SGLP realization. Using the vectorized basic block from Figure 2.7, Figure 2.8(a) identifies two identical subgraphs of (1, 2, 5) and (3, 4, 6) due to the same dataflow and same operations with no dependencies. Each corresponding instruction of two subgraphs is packed together and executed in parallel. Figure 2.8(b) shows the actual execution model using an 8-way SIMD datapath. Due to the insufficient degree of DLP for the original innermost loop from Figure 2.7(a), SGLP is applied and two isomorphic subgraphs are identified from the 4-wide vectorized basic block (Figure 2.8(a)). From these two subgraphs, (3, 4, 6) is chosen to be executed in the unused lanes. As a result, instructions (1, 2, 5, 7) and (3, 4, 6) are executed in lane 0-3 and 4-7 as shown in Figure 2.8(b). In addition to this, one cycle of overhead is incurred to move the output data of instruction 6 to lane 0-3. Figure 2.8(c) is the pseudo code exploiting both SIMD parallelism and SGLP. In this program, parallel instructions in the isomorphic subgraphs are packed together and data movement is enabled by the *shuffle* instruction, which moves data using the shuffle network in Figure 2.3. Shuffle0 extracts the left column data of two input vectors and Shuffle1 extracts the right column data of two input vectors.

Figure 2.8(d) and (e) illustrate the high-level execution model of this paradigm. The example scenario is three consecutive kernels having different natural SIMD widths (kernel 0:8, kernel 1:4, kernel 2:8) are executed on an 8-way SIMD architecture. Kernel 0 and 2 are executed only using SIMD parallelism by loop unrolling without any inter-lane overhead. However, the natural SIMD width of kernel 1 is smaller than the architecture allows, so SGLP is exploited as shown in (d). The SGLP compiler finds two groups of two isomorphic subgraphs as (A0, A1) and (C0, C1) and offloads two subgraphs of A1 and C1 onto lanes 4-7. As a result, the whole program can improve the total performance by the execution time of A1 and C1 as shown in (e) with some overhead. Inspired by this scenario, the total speedup achieved by SGLP over the current execution model is derived as the following equation when executing n different kernels with iv invocations, which have t normal execution time, t_{sglp} execution time can be saved by subgraph offloading and ov inter-lane movement overhead.

$$Speedup = \frac{\sum_{k=0}^{n-1}(t(k) \times iv(k))}{\sum_{k=0}^{n-1}((t(k) - t_{sglp}(k) + ov(k)) \times iv(k))} \tag{2.1}$$

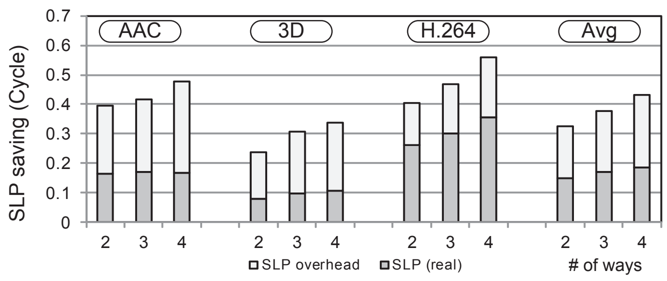
Based on Equation (2.1), the performance gain can be maximized when a program has a high number of invocations on kernels with a small degree of DLP, a high degree of SGLP and a small overhead. Therefore, an SGLP compiler needs to increase the number of instructions covered by identical subgraphs with minimum inter-lane overhead. The key



(a)

```
1: [i:j] = [a[0:3]:c[0:3]] + [b[0:3]:d[0:3]];
2: [k:l] = [e[0:3]:g[0:3]] + [f[0:3]:h[0:3]];
3: [i:k] = shuffle0([l,j], [k,l]);
4: [j,l] = shuffle1([l,j], [k,l]);
5: [m:n] = [i:k] + [j:l];
6: [n:0] = shuffle1([m,n], [0,0]);
7: [result[0:3]:0] = [m:n] + [n:0];
```

(b)



(c)

Figure 2.9: Superword level parallelism difficulty: (a) (1, 3, 5, 7) and (2, 4, 6) are chosen to execute in parallel and three overheads occur, (b) superword level parallelism exploited output source code, and (c) average cycle savings of SLP: Y-bar means ideal savings and it is broken down as overheads and real savings.

algorithm to achieve this goal is explained in section 2.4.

2.3.2 Comparison with Superword Level Parallelism

Superword level parallelism [52] is the most similar paradigm to our work with respect to searching potential parallelism inside the basic block. Because SLP focuses on short SIMD instructions, isomorphic instructions are only considered and thus they cannot handle inter-lane data movement. This problem is often ignored because the overhead of data movement inside the vector is fairly small in a narrow SIMD component, however, it usually induces high performance degradation in a wider SIMD component [52]. In addition to this, the local scope of superword level parallelism may be fooled into selecting packing instructions when a large number of isomorphic instructions exists.

Figure 2.9 shows the result of exploiting superword parallelism from Figure 2.7 (a). For a fair comparison, we relax the memory alignment constraint of [52] so that all memory instructions can be packed. As the compiler searches the isomorphic instructions in program order with local scope, instructions are packed as (1, 2), (3, 4), (5, 6). Then lanes 0-3 execute (1, 3, 5, 7) and lanes 4-7 execute (2, 4, 6) as shown in Figure 2.9 (a). Even though total instruction savings are the same as SGLP , the overhead also increases to three instructions (Figure 2.9 (b)). Therefore, there is no performance gain even in this small basic block, and when the block becomes more complex the algorithm cannot ensure a good result.

Based on the above consideration, we analyze the cost of these overheads for the vectorized kernels of three media applications. Figure 2.9(c) shows average cycle savings when applying SLP at different SIMD ways from two to four compared to the original schedule on the baseline processor. The Y-bar shows the ideal savings assuming the SIMD overhead is free, and each bar is broken down by SLP overhead and real savings. The SLP overhead is calculated assuming all the data rearrangement instructions take one cycle. The results give two major insights: 1) SLP, the well-known SIMDization technique used inside the basic block, can ideally deliver a fair amount of performance enhancement and is also scalable as the number of ways increases, and 2) large SIMD overheads of more than 50% of ideal savings hinder the effectiveness of SLP and make SLP barely scalable as the over-

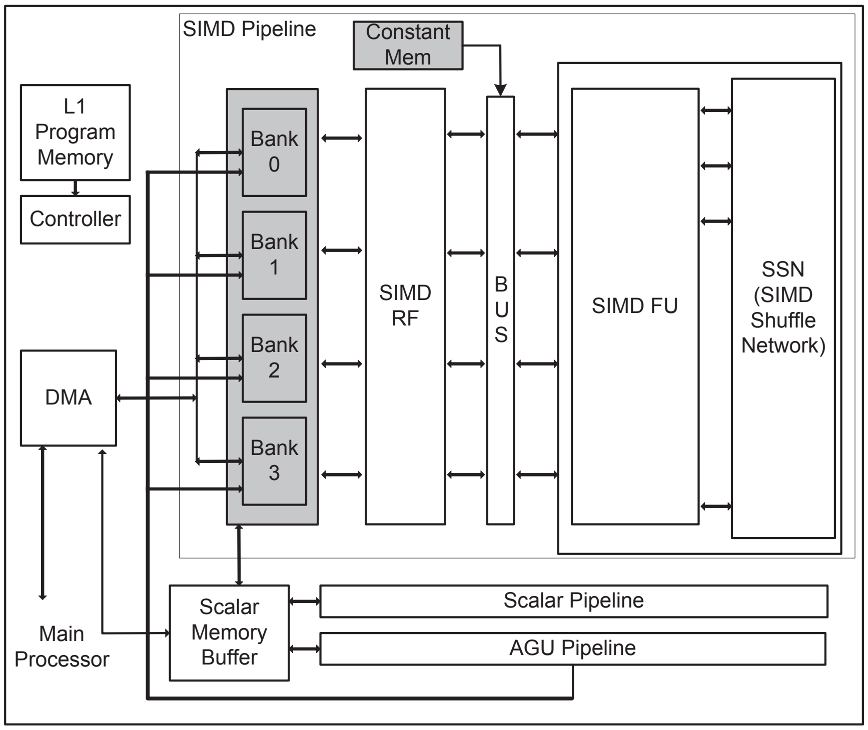


Figure 2.10: Architectural modifications: (1) multi-bank memory and (2) wide SIMD constant memory is supported.

heads also grow dramatically at wider ways. The actual performance gain will be worse in a real situation because many SIMD overhead instructions take more than an single cycle with current technology. Section 2.5 shows how much SGLP improves performance by both increasing the ideal savings and decreasing the overheads when compared to SLP. In addition to this, we also show how much ILP can be converted into SGLP.

2.3.3 Challenges and Solutions

As discussed, SGLP introduces more potential parallelism but has many principal challenges to make this paradigm feasible. We list the four major architectural challenges and suggest possible solutions with architectural or compiler modifications. Simple architectural changes are proposed as shown in Figure 2.10 and compilation challenges are addressed in Section 2.4.

Control flow: Because SGLP is basically exploited within the basic block, control flow is not a big issue. Furthermore, as scalar pipelines are primarily responsible for handling control flow, SGLP generally does not need to consider control flow. However, basic blocks are sometimes merged using if-conversion with predication. Even in this case, SGLP is not affected because predication also can be detected in the identical subgraph identification process.

Instruction flow: When multiple SIMD lane groups execute some task in parallel, all the instructions are not covered as subgraphs, and some SIMD lane groups may not be enabled because the number of identical subgraphs is smaller than the number of SIMD lane groups. Therefore, the main SIMD lane group is necessary to cover all the instructions.

Register flow: First, data movement across or inside the SIMD lane groups can be supported by single-cycle shuffle instructions using a shuffle network. Second, although multiple SIMD lane groups execute the same instructions, their actual register names are different. Therefore, the compiler must handle register renaming, which packs multiple parallel short registers into a wide register. In addition to this, some instructions covered by multiple identical subgraphs may have different immediate values, and therefore the architecture must provide a way to support wide constant values in a cycle because it is impossible to supply multiple values in a cycle. Therefore, a small constant value memory can be added. The compiler then automatically generates the wide constants from multiple immediate values. The application study shows that these cases rarely exist, and thus the overhead incurred is trivial.

Memory flow: If identical subgraphs have memory instructions, the references of the instructions may be different, and thus the architecture must provide a smart memory packing mechanism such as gather-scatter operation.

The possible architectural modification is to replace the SIMD scratchpad memory from one wide memory to a short width multiple bank memory. This change is required to relax the memory alignment constraint. The most critical reason why the basic block typically has high ILP but low DLP is that the architecture does not support unaligned memory access [52]. By supporting unaligned memory packing/unpacking from DMA using the multi-bank memory, more memory instructions can be executed in parallel. One key point is that multi-addressing is only allowed for Memory-DMA communications, while the SIMD pipeline views the memory as a single bank. Another key point is that the number of banks depends on the ratio of the number of memory instructions to normal instructions because the address calculations are the responsibility of the AGU pipeline and they are not scalable, thus the performance of the AGU may be the limiting factor.

2.4 Compiler Support

2.4.1 Overview

In this section, we describe the compiler support for SGLP. Taking the concept of subgraph identification [29], we developed a SGLP scheduler that can support both simple

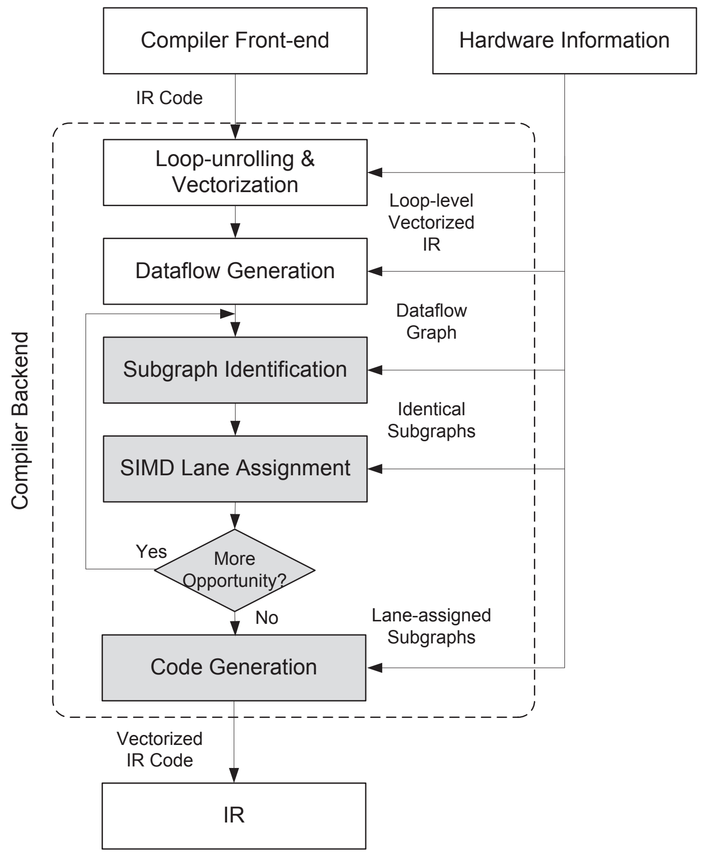


Figure 2.11: Compilation flow of the SIMD defragmenter: shaded regions exploit subgraph level parallelism.

loop-level DLP and SGLP for wide SIMD components. The system flow is shown in Figure 2.11. Applications are run through a front-end compiler, producing generic Intermediate Representation (IR), which is unscheduled and uses virtual registers. The compiler also gets high-level machine specific information, including the number of SIMD lanes, and supported inter-lane movement instructions. Given the IR and hardware information, the compiler performs loop-level vectorization on the selected SIMDizable loops. The compiler then exploits SGLP if the SIMD parallelism is insufficient. After generating the DFG, the compiler iteratively discovers identical subgraphs in the DFG and assigns the subgraphs to unused SIMD lanes until no more SGLP opportunities exist. Finally, the compiler generates the final vectorized IR.

2.4.2 Subgraph Identification

First, identical subgraphs are extracted from the given DFG. The compiler sets the maximum number of identical subgraphs as the available degree of SGLP. The compiler then iteratively searches the groups of identical subgraphs having some number of instances from maximum number down to two (the minimum degree). Heuristic discovery [29], which picks the seed node and grows the nodes, is used for DFG exploration. Exploration starts by examining each node in the DFG and using it as the seed for a candidate identical subgraph. The algorithm attempts to find the largest candidate subgraphs with n identical instances within the given DFG, where n is the degree of SGLP. If, however, the algorithm identifies m identical instances of a candidate subgraph, where $m > n$, only n instances are saved and the nodes from the remaining $m - n$ instances are "discarded" and "re-used" in the next exploration phase. This of course assumes that the current candidate subgraph could not be grown further while still ensuring that n instances could still be identified. If all the identical subgraphs with the target number of instances, n, are found, the compiler decreases the target number by one and starts the subgraph search again.

Additional conditions for the general subgraph search are that 1) the corresponding operations from each subgraph should be identical, 2) live values and immediate values should also be taken into consideration, and 3) inter-subgraph dependencies should not exist. Condition 1) enables the corresponding instructions inside the subgraphs to be packed into one opcode, and condition 2) enables packing whole operands of the instructions. Live values and immediate values are not generally considered in common subgraph pattern matching, but the SGLP compiler must take them into account because only same type of operands can be packed for SGLP. The last condition ensures that the subgraphs are parallelizable.

2.4.3 SIMD Lane Assignment

Once all possible groups of identical subgraphs are identified, the compiler selects the subgraphs to be packed and assigns them to SIMD lane groups. Instructions included in remaining subgraph groups lose the subgraph information and are reused in the next subgraph identification process. The objectives of SIMD lane assignment process are two-fold: 1) pack maximum number of instructions with minimum inter-lane data movement, and 2) ensure packed groups of instructions can be executed safely in parallel without any dependence violation. To achieve these goals, the compiler considers three kinds of criteria: gain, partial order, and affinity.

The gain of the subgraph is the most critical criteria and is largely calculated by the size of the subgraph. Larger subgraphs can provide higher performance with less overheads as more dataflow can be covered. The memory packing overhead is also accounted for in the gain if it incurs performance degradation. The compiler tries to assign subgraphs to specific SIMD lanes based on decreasing order of the gain.

The partial order between subgraphs inside the SIMD lane group is the next most critical issue. When assigning new identical subgraphs to different SIMD lane groups, the partial order of the subgraphs inside the SIMD lanes may be different across the SIMD lanes because identical subgraphs are only parallel with each other and the relations with

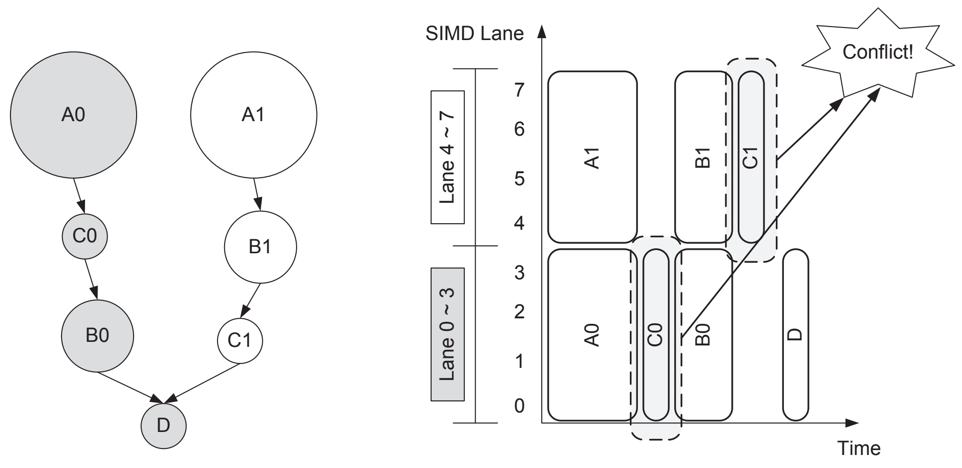


Figure 2.12: Subgraph partial order mismatch: when (B0, B1) is chosen to execute in different SIMD lanes, (C0, C1) cannot be chosen due to the partial order mismatch between lanes.

other subgraph groups are not considered. If the relation between different subgraphs in some lane groups is different from the relations in other lane groups, the corresponding subgraphs cannot be executed in parallel. Figure 2.12 shows a simple example case of this kind of conflict. From a vectorized basic block having 3 groups of identical subgraphs with (A0, A1), (B0, B1), and (C0, C1), (A0, A1) and (B0, B1) are chosen to be parallelized using the two SIMD lane groups. After this assignment, C0 and C1 cannot execute in parallel through two SIMD lane groups because C0 must execute before B0 in the lane group 0-3 but C1 must execute after B1 in the lane group 4-7.

As the inter-lane data movement overheads inside the subgraphs are already solved by subgraph identification, the next objective is to minimize the overheads between different subgraphs. Typically, a subgraph is related to multiple other subgraphs, so the compiler must consider which combination of subgraphs can minimize the overall overhead. To address this issue, an *affinity cost* was introduced inspired by previous works [72, 73]. The affinity value for a pair of subgraphs reflects their proximity in the DFG. When a group of identical subgraphs is chosen to be parallelized, each lane group is assigned an affinity cost depending on how close the subgraph candidate is to any already placed subgraphs that have high affinity with the candidate. This gives preference for assigning a subgraph in the same lane group as other subgraphs it is likely to communicate with thus reducing inter-lane data movements.

$$affinity(A,B) = \sum_{a \in nodes\_A} \left(\sum_{d=1}^{max\_dist} 2^{max\_dist-d} \times ((N_{cons}(a,B,d) \right. \tag{2.2}$$

$$\left. + N_{prods}(a,B,d)) \times C_0 + (N_{com\_cons}(a,B,d) + N_{com\_prods}(a,B,d)) \times C_1)) \right)$$

$$, where \quad C_0 >> C_1$$

Equation (2.2) calculates the affinity between two subgraphs A and B. The value is determined by four different relations between nodes inside A and B: producer, consumer, common consumer, and common producer relations. Producer/consumer relation means that nodes in A have direct/indirect producer-consumer/consumer-producer relations to nodes in B. Common producer/consumer relations mean that nodes in A and nodes in B have common producer/consumer relations. The former two relations have explicit data movement between subgraphs but the latter relations just imply that they may have some data movements when merging or diverging. Therefore, we put more weight on the former two relations ($C_0 >> C_1$). Nodes within max_{dist} are used, where N refers to the number of nodes in subgraph A that have a relationship with a node in subgraph B at a distance d. The distance is the number of nodes to reach the target node.

Algorithm 1 shows how the SIMD lane assignment works. The inputs are the list of identical subgraph groups (*IdSubGroups*), dataflow graph (*G*) and the current list of SIMD

Algorithm 1 SIMD Lane Assignment

Input: *IdSubGroups, G, SIMDGroups*

Output: *SIMDGroups*

```
    { Assign subgraphs into the appropriate SIMD lane group.}
 1: SortSubGraphGroupsByGain(IdSubGroups);
 2: while HasGroup(IdSubGroups) do
 3:     curSubGroup ← Pop(IdSubGroups);
 4:     while HasGroup(curSubGroup) do
 5:         curSubGraph ← Pop(curSubGroup);
 6:         curSIMDGroup ←
            findSIMDGroupByMaxAffinity(SIMDGroups, curSubGraph);
 7:         curSIMDGroup → addSubGraph(curSubGraph);
 8:     end while
 9:     if (!PartialOrderCheck(SIMDGroups)) then
10:         Restore(SIMDGroups);
11:     end if
12: end while

    { If no more updates, find the main lane group and assign remaining nodes.}
13: if (!IsUpdated(SIMDGroups)) then
14:     curSIMDGroup ←
        findSIMDGroupByMaxOverhead(SIMDGroups);
15:     curSIMDGroup → addRemainingNodes(G);
16:     SetMainSIMDGroup(curSIMDGroup);
17: end if
```

lane groups (*SIMDGroups*). The output is the list of SIMD lane groups with new subgraph assignment (*SIMDGroups*). The algorithm starts by sorting the *IdSubGroups* by subgraph gain because we place the top priority on the gain of subgraph. Based on the sorted order of the list, the while loop assigns the subgraphs on the appropriate SIMD lane group. Lines 3-8 take one identical subgraph group and assign each of the subgraphs onto the SIMD lane group having the maximum affinity. Lines 9-11 perform the partial order check for all the SIMD lane groups and, if some conflicts occur, the latest update is cancelled. When no more subgraphs are assigned to the initial *SIMDGroups*, the compiler decides not to try the subgraph identification process again using the remaining nodes, sets the SIMD lane group with the maximum overhead as the main lane group, and assigns uncovered nodes of DFG to the main lane group in order to minimize the total overhead.

2.4.4 Code Generation

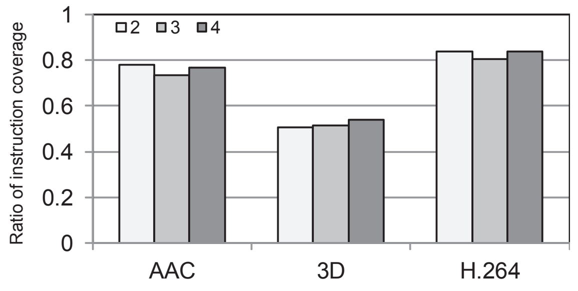
The compiler generates new vectorized IR from the lane assignment and inter-lane movement information from the previous process. When the compiler meets instructions covered by the identical subgraphs, the compiler gathers each parallel operand and converts them into one long register by remapping, a short immediate, or a wide constant. When a wide constant exists, the compiler generates the data and saves it to the constant memory. Shuffle instructions are also added if the compiler detects inter-lane data movement.

2.5 Experimental Results

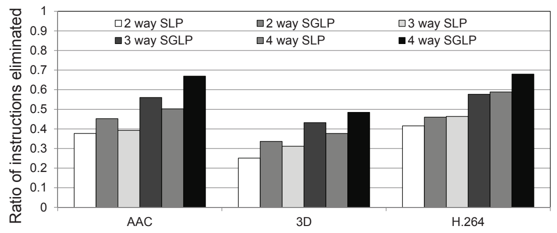
2.5.1 Experimental Setup

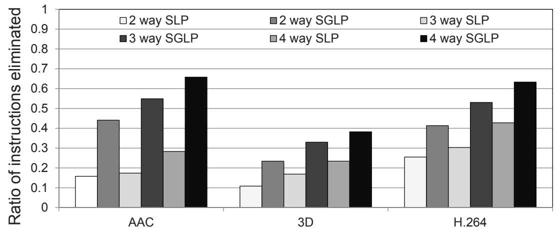
To evaluate the availability and performance of SGLP, 144 loop kernels, varying in size from 4 to 142 operations, are extracted from three media applications in the embedded domain (AAC decoder, 3D graphics, and H.264 decoder). The iteration count per invocation of the kernels varies from 1 to 1024, and the natural SIMD widths are decided by the conditions discussed in Section 2.2.2.1 and memory dependence checks are performed using profile information. The IMPACT compiler [71] is used as the frontend compiler and both SGLP and SLP [52] are implemented in the backend compiler using a SODA-style [60] wide vector instruction set. The inter-lane move is performed using a single-cycle delay shuffle instruction, supporting data rearrangement in the SIMD RF as indicated by the permutation pattern similar to vperm (VMX) or vec_perm (AltiVec [82]). We also allow some similar instructions (e.g. add/sub) to be packed as common vector architectures allow this. The vectorizable kernels are automatically vectorized by loop unrolling and the evaluation is performed using the loop-level vectorized basic block. The wide SIMD architecture as discussed in Section 2.2.1 is used as the baseline architecture. The number of SIMD resources can vary from 16 to 64, while the number of memory banks are limited to four.

Our experiments do not apply SGLP more than 4-way. Two main reasons for this are: 1) the degree of ILP, the theoretical maximum gain of SGLP, is mostly smaller than four, and 2) only computation instructions can be SIMDized, and therefore the decreased ratio of computation to memory instructions causes the performance to be constrained by the AGU pipeline.



(a)



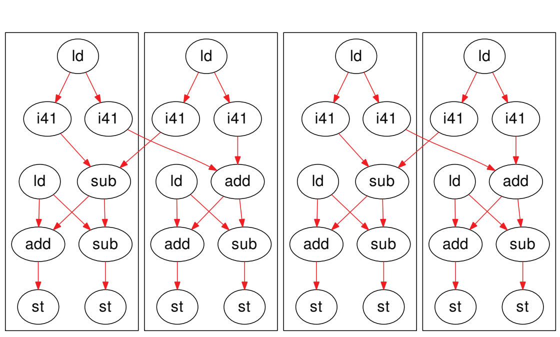


(b) (c)

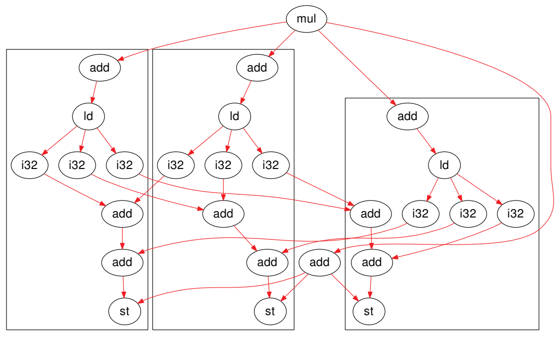
Figure 2.13: Ratio of instructions covered by the subgraph level parallelism and static instructions eliminated for three media applications: (a) instruction coverage, (b) static instruction elimination without inter-lane overheads, and (c) static instruction elimination with inter-lane overheads.

2.5.2 Subgraph Level Parallelism Coverage

We first calculate the percentage of instructions covered by identical subgraphs in order to gauge the availability of subgraph level parallelism. From the vectorized basic blocks of kernels, we found identical subgraphs ranging from 2-way to 4-way. The coverage is calculated as the number of instructions covered by the identical subgraphs. The results for three applications are shown in Figure 2.13(a). For H.264 and AAC, a large percentage of instructions is covered by identical subgraphs because high degrees of parallelism still exists even inside the vectorized basic block. Even though SGLP covers relatively small



(a)



(b)

Figure 2.14: Example dataflow graphs: (a) FFT: two identical subgraphs ((1) ld, i41, i41, (2) ld, (sub/add), add, sub, st, st), (b) MatMul3x3: two identical subgraphs ((1) add, ld, i32 , i32, i32 (2) add, add, st). i41 and i32 are intrinsic instructions.

amount of instructions, more than 50% of instructions in the 3D application are still covered. Compared to other applications, the 3D application has a smaller degree of SIMD opportunity due to each instruction having a small number of parallel instructions with the same operation.

The interesting point here is that the coverage of the 3-way for AAC and H.264 applications is smaller than the 2- and 4-way. This is because most dataflow graphs have a tree structure and therefore 2 and 4 way are well matched but 3-way frequently misses some instructions when dataflow merges. For example, a dataflow graph of the FFT kernel is likely parallelizable in a 4-way, and thus 3-way exploration cannot find the profitable identical subgraphs in the one remaining flow as shown in Figure 2.14(a).

Figure 2.13(b) and Figure 2.13(c) show the ratio of static instructions eliminated from the vectorized basic block when applying SGLP and SLP. The configuration is expressed as: (number_of_simdization_ways) way (technique). Figure 2.13(b) shows the result without overhead (number of shuffle instructions) and the percentage of savings has the trend similar to that of the SGLP coverage. An interesting question is how the SGLP can eliminate more instructions than the SLP even though both techniques have a fair amount of gains. This is because 1) SLP frequently makes the wrong decision among various packing opportunities and 2) SLP cannot vectorize pure scalar codes [15]. When considering the inter-lane data movement overhead as shown in Figure 2.13(c), SLP performs dramatically worse than the ideal condition due to many shuffle instructions. On the other hand, SGLP was found to still deliver consistent amounts of instruction eliminations by smart data-movement control. Based on the application complexity, H.264 and 3D have a notable degradation of savings, whereas AAC is rarely affected by the overhead.

2.5.3 Performance

Inspired by the promising result of finding abundant opportunities for SGLP in the vectorized basic block, we compared the performance of SGLP to both SLP and ILP. Performances of SGLP and SLP are measured as the schedule length when the kernel is mapped a (a degree of loop-level vectorization $\times$ the number of ways)-wide SIMD architecture. As SGLP is the restricted form of ILP, the ILP result can be thought of as the theoretical upper bound. The performance of ILP is measured as the schedule length when the kernel is scheduled in the same sized fully-connected VLIW machine having a central register file. For example, if an example kernel is loop-level vectorized by 16 and 2-way SGLP is applied, the corresponding ILP performance is calculated when an ideal 32-wide VLIW machine executes unrolled scalar code.

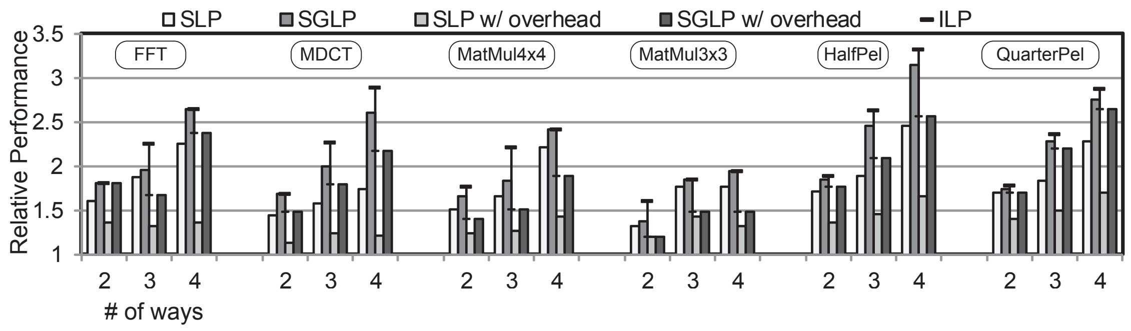


Figure 2.15: Performance comparison of SLP/SGLP without overhead, SLP/SGLP with overhead, and ILP for key kernels: FFT, MDCT for AAC, MatMul4x4, MatMul3x3 for 3D, and HalfPel, QuarterPel for H.264.

Figure 2.15 and 2.16 show the individual performance enhancement results of six well-known kernels and geometric mean of gains for each application. The target ways are shown on the X-axis, relative performance normalized to the original vectorized kernel on the Y-axis. The following techniques are examined and shown as a bar form: SLP and SGLP with zero-cycle data-movement latency (SLP and SGLP) and SLP and SGLP with single-cycle data-movement latency (SLP and SGLP w/ overhead). The ILP results are also shown as a short horizontal form and the vertical line indicates the performance difference between ideal ILP and loop-level vectorization combined with practical SGLP. From these two graphs, substantial amounts of speedups exist in both ideal cases and are similarly scalable as ILP. In addition to this, gains from SGLP in real situations are also mostly prominent and scalable without large overhead increases on wider ways. In contrast, SLP with overhead has a large performance degradation due to the immense inter-lane data-movements, and increasing overheads on wider ways make it barely scalable.

Unlike most cases, a few kernels showed negligible performance improvements while

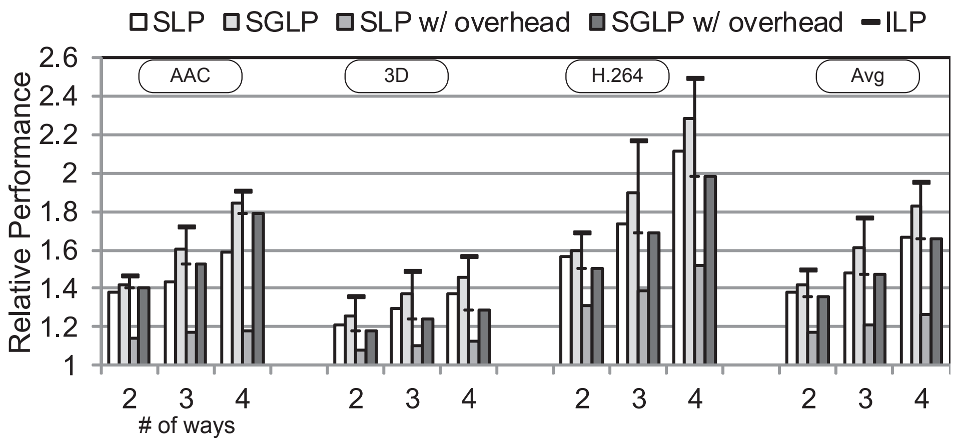


Figure 2.16: Average kernel performance comparison of SLP/SGLP without overhead, SLP/SGLP with overhead, and ILP for three application domains.

applying SGLP, namely FFT in AAC and 3x3 matrix multiplication in 3D. These are due to the specific characteristics of each dataflow graph. First, as shown in Figure 2.14(a), the FFT kernel can have two subgraphs without inter-lane data movements in the 2-way case. In the 4-way case, each subgraph for the 2-way case is split once more with only two data movements such as (i41 $\rightarrow$ add) and (i41 $\rightarrow$ sub). In the 3-way case, three of the subgraphs for the 4-way case are identified and a remaining subgraph cannot be effectively executed in multi-lane, which has a high data-movement overhead. As a result, the gain of 3-way SGLP is worse than that of 2-way SGLP including overheads. Second, the 3x3 matrix multiplication can be split into three subgraphs as shown in Figure 2.14(b), and therefore a considerable increasing in overheads when applying 4-way SGLP hinder it from fully exploiting the benefits.

As shown in Figure 2.16, on average, SGLP without and with overheads achieve relative performance improvements of 1.42x, 1.36x at 2-way, 1.61x, 1.47x at 3-way, and 1.84x, 1.66x at 4-way. In addition to this, SGLP with overheads also provides 18-40% more per-

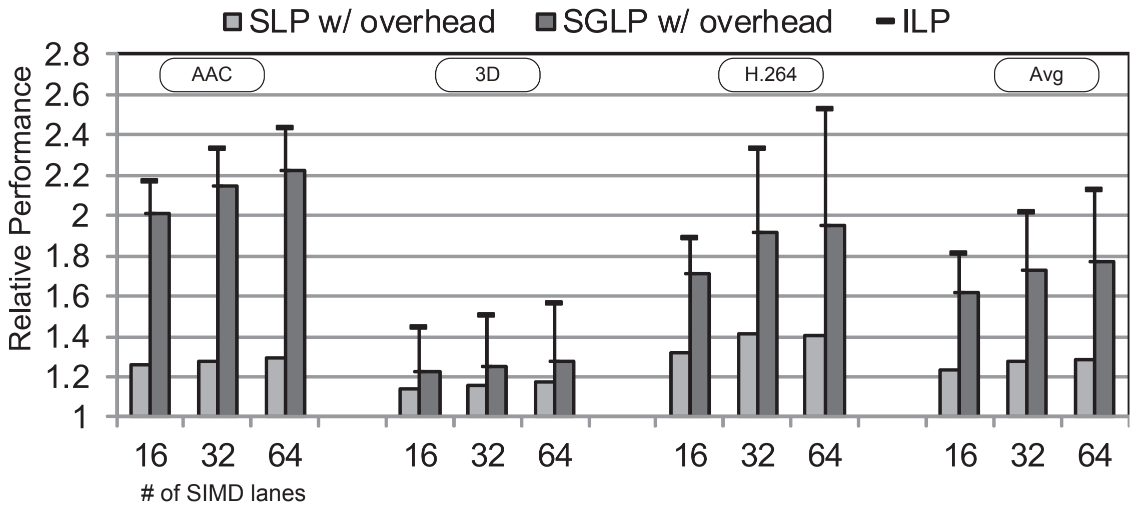


Figure 2.17: Overall performance comparison of SLP/SGLP with overhead and ILP for three domains on SIMD architectures.

formance improvement over baseline compared to SLP with the same resources. The performance difference between SGLP and SLP increases as applied on wider ways. Finally, a comparison with ILP suggests that SGLP is a cheap and powerful solution to accelerate performance, considering that SGLP only requires minimum additional hardware while a wide fully-connected VLIW architecture is impractical.

Based on the schedule results for kernels, we execute three applications on three wide SIMD architectures having 16, 32, and 64 lanes. When the original SIMD width of the kernel is equal or larger than the width of the architecture, SGLP or SLP is not exploited. Only when the current SIMD width of the kernel is insufficient to fully use the architecture, the available amount of SGLP, up to 4-way, is exploited. For example, 4-way SGLP is exploited if a kernel is 4-wide vectorized on the 16 way architecture and 2-way when 8-wide vectorized. The final performance is also compared to traditional ILP in the equivalent VLIW architecture and SLP. The results are provided in Figure 2.17 with considerable performance gains. The X-axis shows the number of SIMD lanes on the wide SIMD architecture and the Y-axis shows speedup relative to the simple SIMD execution time on the baseline architecture. The two bars of each application represent the runtime speedup of real SLP and SGLP with overheads. In a similar ways from previous Figures, ILP results are also provided. For all the applications, real SGLP still shows notable performance gain by utilizing more SIMD resources with smart overhead control. As we discussed in Section 2.2.2.2, kernels having smaller than 16 SIMD width are accelerated by SGLP when using 16 wide architecture, and AAC and H.264 have high gains due to the high execution time ratio of such kernels, which are more than 50% of their total execution time. As the architecture size becomes larger, the performance is saturated at some point because SGLP is constrained by maximum degree of 4. The key observation is that the real performance gain of SGLP is also fairly scalable due to the fact that the performance gain successfully compensates for the increased overheads different from SLP. Finally, on average, SGLP with overhead can have 1.61x, 1.73x, and 1.76x speedups at 16, 32, and 64 wide SIMD architectures while SLP only achieves 1.24x, 1.28x, and 1.29x speedups.

2.5.4 Energy Measurement

To evaluate the energy savings of SGLP in the real world, we measured total energy consumption for running H.264 to determine the effectiveness of SGLP. We used a 32-wide SIMD architecture for SGLP, and a practical 4-way VLIW, in which each datapath supports 8-wide SIMD instructions for ILP and an 8 read-ports, 4 write-port 8-wide SIMD RF. Both architectures are generated in RTL Verilog for a 200 MHz target frequency, then

	SGLP @ 32-wide SIMD	ILP @ 4 way 8-wide VLIW	ratio
power (mW)	54.40	93.17	58.39%
cycle(million)	13.07	10.77	121.36%
energy (mJ)	3.55	5.02	70.86%

Figure 2.18: Energy comparison for the SGLP on the 32-wide SIMD architecture and ILP on the 4 way 8-wide VLIW architecture.

synthesized with the Synopsys Design Compiler and Physical Compiler using IBM 65nm standard cell library with typical operating conditions. PrimeTime PX is used to measure power consumption. Instead of measuring power for every cycle, average activity of each component was monitored. Figure 2.18 shows that the SGLP is 30% more energy efficient than ILP. Even though the performance of SGLP is slightly lower, the high power overheads of VLIW implementation, such as those introduced by a multi-port register file and a complex interconnect, dominate the results. The power number for constant memory is also considered based on the standby power and read power extracted from the SRAM compiler. The constant memory power overhead is trivial because the standby power is roughly 1/250 of read power and the wide constant values are rarely read. The access timing is also smaller than 5 ns (i.e., 200 MHz), hence data can be read in one cycle.

2.6 Related Works

Most prior work in automatic vectorization are performed on the loop-level [68, 10] and some of the techniques have already been implemented on commercial compilers such as the Intel Compiler [41]. These types of vectorization are usually exploited by loop unrolling. Our SGLP vectorization starts after simple loop-level vectorization, and thus it is an orthogonal approach and can be helpful to enhance the overall performance of our compiler framework by finding more loop-level DLP. SGLP tries to identify opportunities for parallelism within the vectorized basic block.

Superword-level parallelism [52] is the closest related work but this work is hard to apply to long vector architectures as discussed in Section 2.3. To improve this technique, some research [83, 53] focuses on smart memory control such as increasing contiguous memory instructions and decreasing memory accesses. There are two key differences between SGLP and SLP: 1) SGLP tries to minimize the SIMD overheads in the scope of dataflow graph analysis, whereas most approaches do in the scope of memory management, and 2) we focus on finding groups of instructions to guarantee sufficient gain over the overheads but others usually focus on decreasing the overheads. Unroll-and-jam with SLP [68] is the most similar work and we can get 30% higher performance on average due to SLP being less effective when applied to scalar code.

Another key contribution of this work is the ability to minimize the interaction between the SIMD lanes. This scheme is highly related to the research in the area of clustering [24, 37, 23, 9]. However, general clustering techniques for VLIW machines focus on load balancing and critical path search, and thus cannot handle dataflow and instruction mismatches between clusters.

Subgraph exploration for finding identical subgraphs is also a well-known research area [29, 25, 26, 28] but the goal of these works is mostly to generate custom accelerators for the subgraphs. We introduce a new algorithm for orchestrating sets of subgraphs at a high-level for SIMDization on existing architectures.

AnySp [90] or SCALE [49], which exploit multiple forms of parallelism, are also similar to this work. AnySp integrated DLP and TLP, and SCALE exploits both vector parallelism and TLP. However, they need substantial architectural changes like multiple AGUs, flexible functional units, and swizzle networks in AnySp, or additional multiple fetch unit, special inter-cluster network, and Atomic Instruction Block (AIB) cache in SCALE. However, we can provide SGLP with two minimal hardware modifications (a small wide-constant memory and banked memory access) that incur very little overhead.

2.7 Summary

The popularity of mobile computing platforms has led to the development of feature-packed devices that support a wide range of software applications with high single-thread performance and power efficiency requirements. To efficiently achieve both objectives, embedding SIMD components is an attractive solution, However, utilization of SIMD resources is a major limiting factor for adopting such a scheme. In response, we propose an efficient vectorization framework, called the *SIMD defragmenter*, to enhance the throughput by maximizing SIMD utilization. The *SIMD defragmenter* framework first performs simple loop-level vectorization, then tries to find more DLP within the vectorized basic block using subgraph level parallelism (SGLP). To achieve this, partially parallelizable subgraphs are identified inside the basic block, which are offloaded to the unused SIMD lanes while minimizing the number of inter-lane data movements. We introduce a new way to orchestrate the partially parallel subgraphs, which is implemented in our SGLP compiler. The SGLP compiler is able to effectively assign the SIMD lanes for each subgraph based on the relations between subgraphs. On a 16-wide SIMD processor, SGLP obtains an average 62% speedup over traditional vectorization techniques, with a maximum gain of 2x. In comparison to superword-level parallelism, the well-known basic block level vectorization technique, SGLP achieves an average 30% speedup. We believe as SIMD, or more general data-parallel, accelerators become more commonplace, new techniques to put these resources to work across a wide spectrum of applications will be essential.

CHAPTER 3

Accelerating Execution using Dynamic Operation Fusion

3.1 Introduction

The embedded computing systems that power today's mobile devices demand both high performance and energy efficiency to support various high-end applications such as audio and video decoding, 3D graphics, and signal processing. Traditionally, application-specific hardware in the form of ASICs is used on the compute-intensive kernels to meet these demands. However, the increasing convergence of different functionalities combined with high non-recurring costs involved in designing ASICs have pushed designers towards more flexible solutions that are post-programmable. Coarse-grained reconfigurable architectures (CGRA) are becoming attractive alternatives because they offer large raw computation capabilities with low cost/energy implementations [61, 85, 65]. Example CGRA systems that target wireless signal processing and multimedia are ADRES [66], MorphoSys [61], and Silicon Hive [78].

CGRAs generally consist of an array of a large number of function units (FUs) inter-

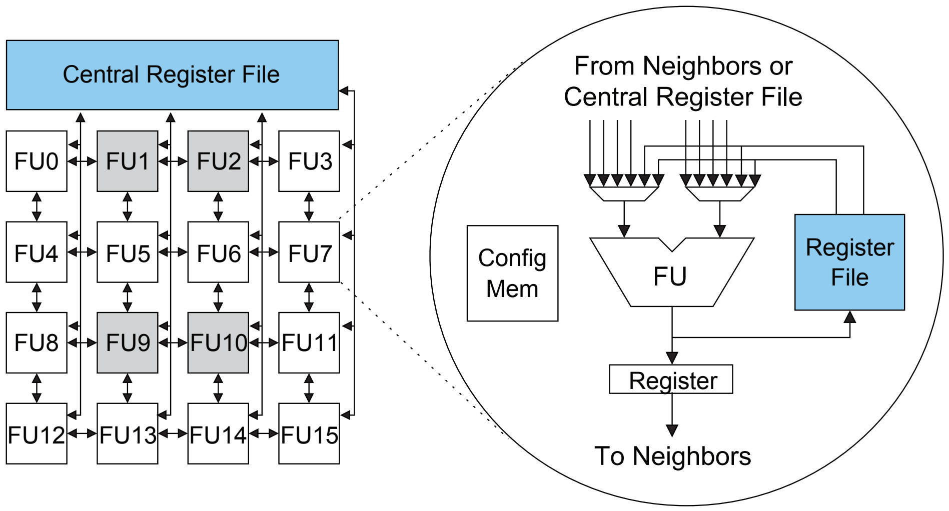


Figure 3.1: Overview of a 4x4 CGRA.

connected by a mesh style network, as shown in Figure 3.1. Register files are distributed throughout the CGRA to hold temporary values and are accessible only by a small subset of the FUs. The FUs can execute common integer operations, including addition, subtraction, and multiplication. In contrast to FPGAs, CGRAs sacrifice gate-level reconfigurability to achieve hardware efficiency. Thus, CGRAs have short reconfiguration time, low delay characteristics, and low power consumption.

While CGRAs are fully programmable, an effective compiler is essential for achieving efficient execution. The primary challenge is instruction scheduling wherein applications are mapped in time and space across the array. However, scheduling is challenging due to the sparse connectivity and distributed register files. On CGRAs, dedicated routing resources are not provided. Rather, FUs serve as either compute or routing resources at a given time. Therefore, the scheduler must manage the computation, flow, and storage of operands across the array to effectively map applications onto CGRAs. Compilers generally focus on mapping compute-intensive innermost loops onto the array. Early work focused on exploiting instruction-level parallelism [57, 21]. However, these approaches could not make efficient use of the available resources due to limited ILP, thus more recent research focuses on exploiting loop-level parallelism through modulo scheduling [65, 72, 73].

CGRA research has generally focused exclusively on efficiency for throughput-constrained innermost loops. However, real-world media applications consist of more than highly parallel inner loops. Specifically, substantial fractions of time are spent in non-loop or outer loop code, as well as recurrence dominated innermost loops. Traditional CGRAs do not handle such *latency-constrained* code segments in an effective manner as they have no mechanisms to accelerate dataflow graphs that are narrow and sequential. In fact, the majority of the resources sit idle in such situations.

This chapter proposes a new technique referred to as *dynamic operation fusion* to accelerate latency-constrained code segments on CGRAs. The core idea is to dynamically configure the existing processing elements of a CGRA into small acyclic subgraph accelerators. Each cycle, any FU can be fused with multiple of its neighbors to create an accelerator capable of executing a small computation subgraph in a single cycle. In essence, small configurable compute accelerators are realized on the array to accelerate sequential code [26]. The necessary hardware extensions for a conventional CGRA are quite simple – an inter-FU bypass network is added between neighboring FUs in the array using a few multiplexors. The compiler scheduler automatically identifies opportunities to accelerate subgraphs by managing the scheduling process at the sub-cycle granularity. The net result is that the usefulness of CGRAs is extended beyond highly parallel loops to effectively operate in latency-constrained code regions.

The contributions of this chapter are as follows:

- An analysis of common media applications to understand the limitations presented by latency constraints.
- CGRA design that supports dynamic operation fusing.
- A compiler scheduler that automatically identifies opportunities for dynamic fusion.
- An evaluation of dynamic operation fusion across a set of media applications.

3.2 Motivation

3.2.1 Analysis of Multimedia Applications

To understand the effectiveness and limitations of traditional CGRAs, we examine the characteristics of commonly used multimedia applications. In mobile environments, three of the most widely used multimedia applications are: audio decoding, video decoding and 3D graphics acceleration. We first identify the characteristics of each application, and verify the importance of enhancing performance in latency-constrained code.

3.2.1.1 Baseline Architecture

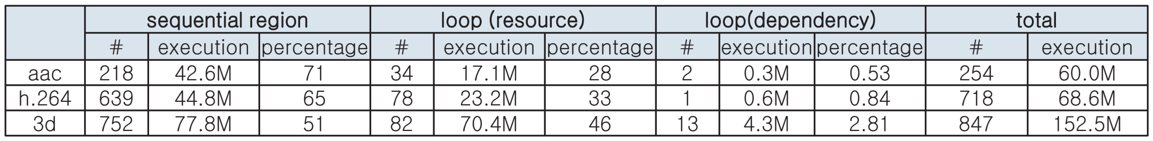
In this work, ADRES[66] is used for the baseline CGRA architecture. This architecture consists of 16 FUs interconnected by a mesh style network. Register files are associated with each FU to store temporary values. The FUs can execute common integer operations. The architecture has two operation modes: one is CGRA array mode and the other is VLIW processor mode. In CGRA array mode, all 16 computing resources are available and loop-level parallelism is exploited by software pipelining compute-intensive innermost loops.

The baseline architecture is also able to function as a VLIW processor to execute sequential and outer loop code. The four FUs in the first row and the central register file support VLIW functionality, while the other components are de-activated. This type of architecture provides high performance by eliminating huge communication overhead to transfer live values between host processor and the array as well as a multi-issue VLIW for non-loop code that is more powerful than a traditional general-purpose processor used as the host (e.g., an ARM-9).

3.2.1.2 Application analysis

Code of general applications can be categorized into sequential and loop regions. Sequential regions often perform control flow for decision making and handle setup for the compute-intensive loops by transferring live values between loops. Loop regions execute iterative work like calculating pixel data on graphic application. Multimedia applications typically have many compute intensive kernels that are in the form of nested loops. Software pipelining, which can increase the throughput of the innermost nest by overlapping the executions of different iterations, can decrease run time of this type of loops tremendously. In this section, we first decompose applications into various region types. The applications consist of :

- AAC decoder: MPEG4 audio decoding
- H.264 decoder: MPEG4 video decoding
- 3D: 3D graphics rendering accelerator



	sequential region			loop (resource)			loop(dependency)			total	
	#	execution	percentage	#	execution	percentage	#	execution	percentage	#	execution
aac	218	42.6M	71	34	17.1M	28	2	0.3M	0.53	254	60.0M
h.264	639	44.8M	65	78	23.2M	33	1	0.6M	0.84	718	68.6M
3d	752	77.8M	51	82	70.4M	46	13	4.3M	2.81	847	152.5M

Figure 3.2: Execution time breakdown for three multimedia applications (#: number of basic blocks, execution: number of cycles, percentage: percent of execution cycles). Execution time is broken down into three categories: sequential are all non-innermost loop regions, loop (resource) are inner-most loops whose performance is constrained by the availability of resources, and loop (dependency) are inner-most loops whose performance is constrained by cross-iteration dependences.

For our benchmarks, we analyzed the relative importance of sequential and loop regions by analyzing the execution time spent in each. Loops were also categorized loops as their performance was most constrained by resources or cross-iteration data dependences. This grouping provides more precise insights because the characteristics of dependence-constrained loops are more similar to sequential code rather than resource-constrained loops. Performance of the sequential regions was determined by scheduling those onto the VLIW subset of the ADRES CGRA (a 4-wide VLIW) [66]. Modulo scheduling, an efficient software pipelining technique that exploits loop level parallelism by overlapping the execution of different iterations [79], was used to compute the run time of loop regions executing on the 4x4 ADRES CGRA.

Figure 3.2 presents the execution time breakdown for each benchmark. Software pipelining can successfully reduce the execution time of loop regions, making it less than 50% of the total execution time. To further improve the overall performance, it is clear that improving the performance of sequential code regions is critical since they are taking more

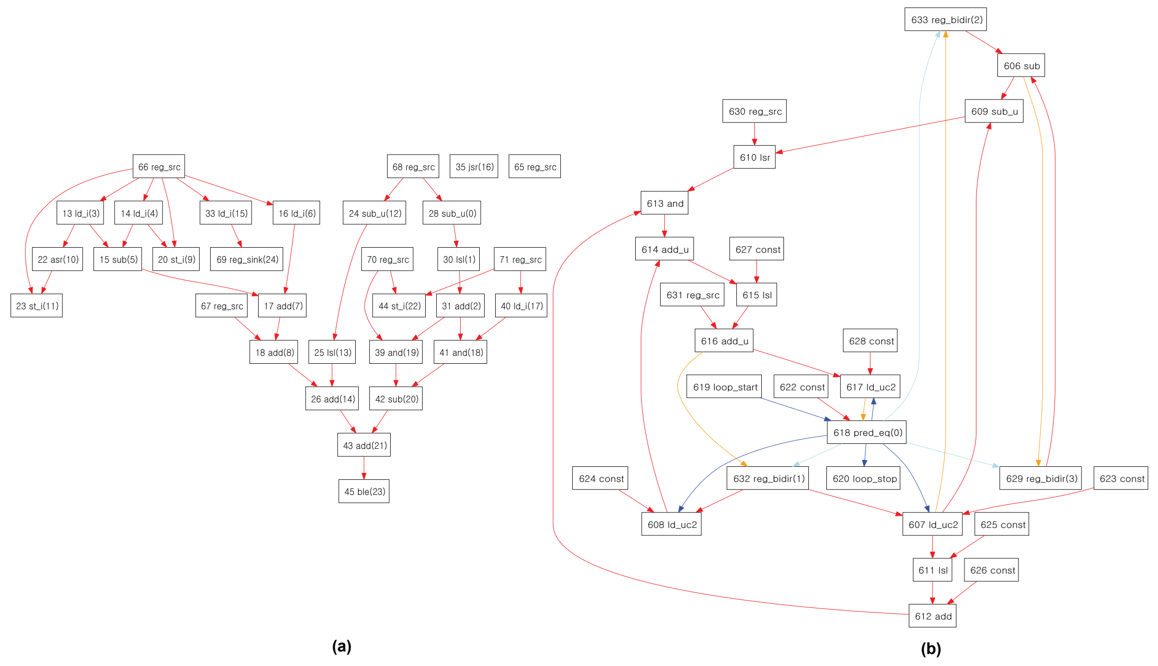


Figure 3.3: Example dataflow graphs in AAC: (a) Sequential code, (b) Loop code

than 60% of the total execution time.

To get a better understanding of the structure of the code in both the acyclic and loop regions, consider the dataflow graphs in Figure 3.3 from the AAC benchmark. Figure 3.3(a) is a data flow graph of a sequential region that performs some control flow between compute-intensive loops and has many data dependences between instructions. Generally, this type of sequential code doesn't have a large number of instructions so providing abundant compute resource does not improve performance. Decreasing the dependence length through a chain of instructions is the only solution to accelerate such code. Figure 3.3(b) is an example of dependence-constrained loop. This loop also has a small number of instructions with long chains of sequential dependences. This type of code is also hard to overlap iterations by software pipelining because last instruction on each iteration has data dependence with the first instruction of the next iteration, and the next loop cannot start execution before finishing the execution of the prior loop.

3.2.2 Accelerating Sequential Code

Most prior research in CGRA has focused on improving the performance of innermost loops through intelligent parallelization or software pipelining techniques. However, none are effective at enhancing the performance of sequential code regions, which occupy a significant fraction of total execution time as demonstrated in Figure 3.2. In this work, we take a circuit-level approach to attack the problem of improving the performance of sequential and dependence-constrained loops on CGRAs.

One obvious approach to improve performance of all region types is to increase the clock frequency of the CGRA. However, this approach increases power consumption a large amount due to additional pipeline registers and higher voltage needed to operate the CGRA. Rather, our approach is to exploit the slack cycle-time to accomplish more work in a single clock cycle when the critical timing paths are not exercised through the CGRA. In this manner, multiple arithmetic operations can be “chained” together when the critical timing paths are not exercised to accomplish more work in a single cycle.

Configurable compute accelerator (CCA) [26] is one related research based on this concept. CCA is also designed to execute a number of sequential instructions on fixed clock period in a general purpose processor. The clock period of a general purpose processor is larger than that of CGRA and the depth of maximum sequentialized instruction is quite large. However, this type of accelerator cannot cover all the subgraphs because of fixed numbers of input/output ports and limitations of subgraph depth. Expression-grained reconfigurable architectures [19] are proposed to solve these problems but they still cannot cover all the cases. In addition to coverage problem, low utilization of FUs is another critical drawback on this type of research. They put abundant resources to obtain high subgraph coverage on fixed hardware hence utilization of each individual FU becomes low. Thus, a more efficient strategy is required to enable the acceleration of sequential subgraphs without adding significant cost or power to a baseline CGRA.

3.3 Dynamic Operation Fusion

In this section, we propose dynamic operation fusion that can accelerate the execution of sequential code regions by executing multiple operations in a single cycle. The basic idea is explained first and the opportunities for dynamic operation fusion in multimedia applications is shown. Lastly, the hardware support is discussed.

The basic idea of operation fusion stems from the observation that the clock period of a CGRA is determined by the worst case delay (critical path delay) in the architecture. When the clock period is not fully utilized, the slack can be used to execute the successive operation if the delay fits into the slack.

The critical path of a CGRA usually consists of: register file read, longest execution in a FU, and write back to register file, as shown in Figure 3.4(a). While register file access is required for every operation in conventional architectures, CGRAs have distributed interconnect across the array that can directly transfer operands between FUs. When an operation is executed without a register file access through the interconnect, it does not fully utilize the clock period and there is significant slack left. For example, the ADD op-

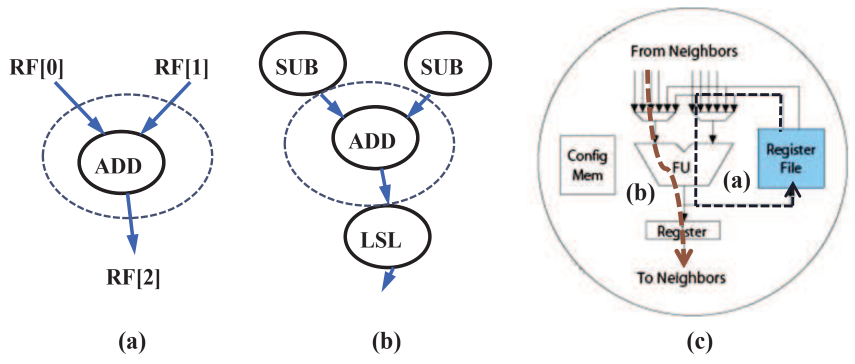


Figure 3.4: Comparison of flow of data through a processing element in a CGRA: (a) Operation with register file access, (b) Operation without register file access, (c) Flow of data for (a) and (b)

eration in Figure 3.4(b) reads the operands from its neighboring FUs and transfers its result directly to another FU. If the time slack is bigger than the delay of the successive operation LSL, both ADD and LSL can be executed in the same clock period. As previously mentioned, vertical collapsing of dependent operations is similar to the CCA [26]. In CCA, the subgraphs with simple operations (i.e., arithmetic, logical) are identified either at compile time [25] or at run-time [26]. The execution of the subgraphs are offloaded to a specially designed accelerator that can collapse the execution of multiple operations into a single cycle.

Instead of using dedicated hardware as in CCA, we propose dynamic operation fusion that utilizes existing resources in a CGRA to collapse the dependent operations into a single cycle. Since there are a large number of FUs in a CGRA, a subset of them can be combined dynamically at run-time and execute dependent operations in a single cycle. A simple modification to the hardware can allow dynamic merging of FUs for operation fusion; providing an interconnect between FUs that bypasses the output registers. Figure 3.5

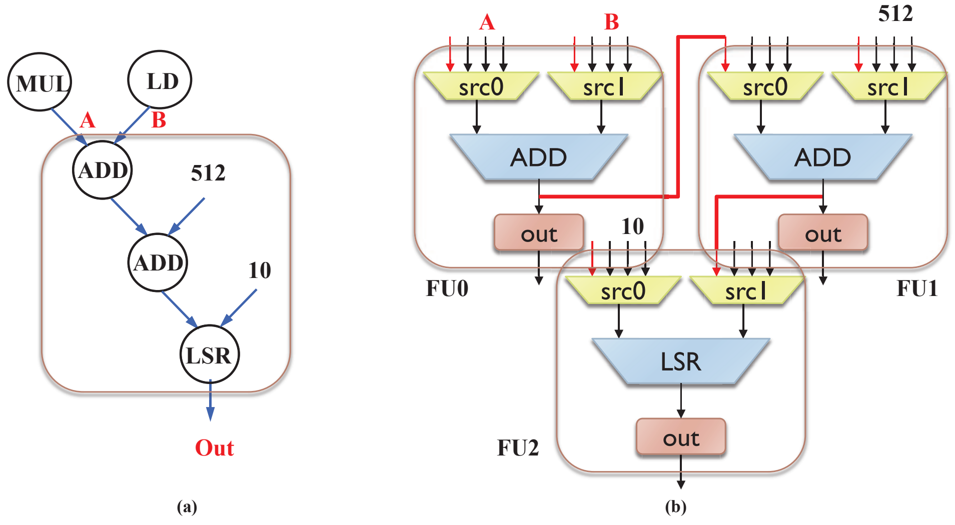


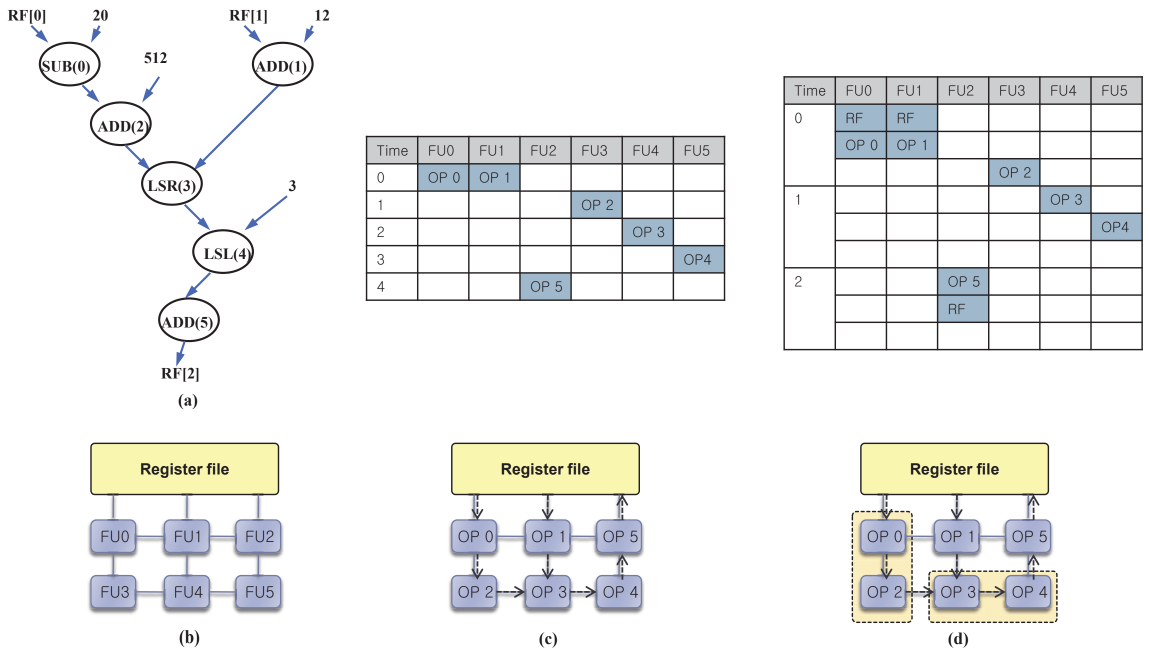
Figure 3.5: Combining of FUs for dynamic operation fusion: (a) Target subgraph, (b) 3 FUs combined.

shows the additional interconnect from the combinational output of an FU to the input of its neighboring FUs. Here, three FUs on the right are serially merged together to execute the three dependent operations on the left (ADD - ADD - LSR) in a single cycle. So, the execution time of the sequential code region can potentially be reduced with dynamic operation fusion, while the hardware overhead is minimal.

Dynamic operation fusion has the following benefits over the CCA approach with a dedicated accelerator:

- Minimal hardware overhead utilizing the existing resources.
- Multiple subgraphs can be executed simultaneously when resources are available.
- Dynamic merging of FUs allow exploiting various shapes of the subgraphs.

We will compare the schedule results using dynamic operation fusion with traditional scheduling for a CGRA with the example shown in Figure 3.6. The dataflow graph on the



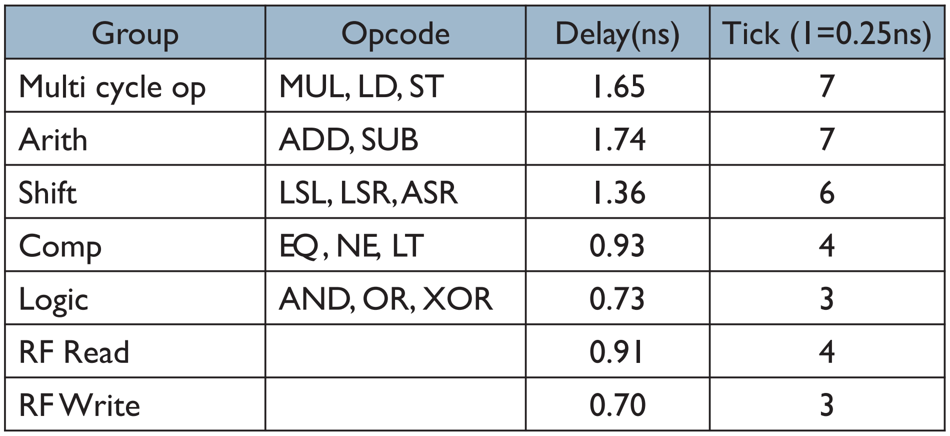
Time	FU0	FU1	FU2	FU3	FU4	FU5
0	OP 0	OP 1				
1				OP 2		
2					OP 3	
3						OP4
4			OP 5			

Time	FU0	FU1	FU2	FU3	FU4	FU5
0	RF	RF				
	OP 0	OP 1				
				OP 2		
1					OP 3	
						OP4
2			OP 5			
			RF			

Figure 3.6: Dynamic operation fusion example: (a) dataflow graph under consideration, (b) target 2x3 CGRA, (c) conventional scheduling that requires 5 cycles, and (d) scheduling with dynamic operation fusion that requires 3 cycles.

left contains a series of dependent operations that read operands from register files and store the result back into them. It is mapped onto a hypothetical 2x3 CGRA in Figure 3.6(b). The conventional approach will generate a schedule shown in Figure 3.6(c), where the total execution time is 5 cycles. Because of the serial data dependences, the utilization of the FUs is quite low.

Figure 3.6(d) shows how the execution of the dataflow graph can be accelerated with dynamic operation fusion. Here, we assume that one register file access and two arithmetic operations can fit into the clock period. More detailed studies on the comparison between the clock period and operation latencies are provided in the following section. With the bypass network, two sets of back-to-back operations are collapsed into the same cycle as



Group	Opcode	Delay(ns)	Tick (1=0.25ns)
Multi cycle op	MUL, LD, ST	1.65	7
Arith	ADD, SUB	1.74	7
Shift	LSL, LSR, ASR	1.36	6
Comp	EQ, NE, LT	0.93	4
Logic	AND, OR, XOR	0.73	3
RF Read		0.91	4
RF Write		0.70	3

Figure 3.7: Delay and tick breakdown for common opcodes.

shown in the schedule. At cycle 0, FU 0 and FU 3 are merged together to execute back-to-back operations 0 and 2 in an single cycle. In the same fashion, operations 3 and 4 are collapsed into cycle 1 on FU 4 and FU 5. Operation 5 cannot be scheduled at cycle 1 since it stores the result into the register file. By applying dynamic operation fusion, the total execution time is reduced by 2 cycles over the conventional approach.

3.3.1 Delay Statistics and Tick Time Unit

As shown in the previous section, dynamic operation fusion is an effective approach to accelerate the execution of sequential code region. However, the feasibility of dynamic operation fusion depends on the hardware characteristics of the underlying architecture. Dynamic operation fusion can be applied only if there is enough slack in a clock period to execute multiple operations. So, we investigated the delay characteristics of our CGRA design in a real implementation. Figure 3.7 shows the delay information when the clock period is 3.5 ns. The delays are computed with Synopsis Design Compiler and Physical Compiler using the IBM 90nm standard cell library in typical condition. The delay here

Tick	aac (%)	3d (%)	h.264 (%)
Multi cycle	2419 (31)	17077 (34.5)	11579 (30.7)
Arith	2018 (26)	12339 (25)	11075 (29.3)
Shift	370 (4.7)	1165 (2.3)	2086 (5.5)
Comp	506 (6.5)	2788 (5.6)	1923 (5.1)
Logic	2492 (32)	15919 (32.2)	11024 (29.2)

Figure 3.8: Breakdown of opcodes for three target applications.

includes the delay of input MUXes for each unit. In this table, single cycle operations are categorized based on their execution time. For multi-cycle operations, the delays of the last stage is shown in the table. The execution time of all instructions are smaller than half of a clock period. Logical operations show the minimal delay and four of them can be fused together into a single cycle. On average, two sequential operations can be collapsed. The opportunities for dynamic operation fusion maximizes when there are a large number of operations with a small delay. As in Figure 3.8, there are a large portion of comparison and logic operations, which suggests that dynamic operation fusion can potentially improve the sequential code performance in multimedia applications.

Since multiple operations can be mapped into a single cycle, we need a smaller time unit than the traditional clock cycle used by compiler schedulers. We propose a new time unit called a *tick*, a small time unit based on the actual hardware delay information. The unit delay of one tick is set by the actual latency of the smallest logic component, normally a small MUX. With the tick unit, the clock period and the delays of other hardware components can be converted into tick numbers. Every logic component on CGRAs has their own tick information and the information is used for dynamic operation fusion scheduling.

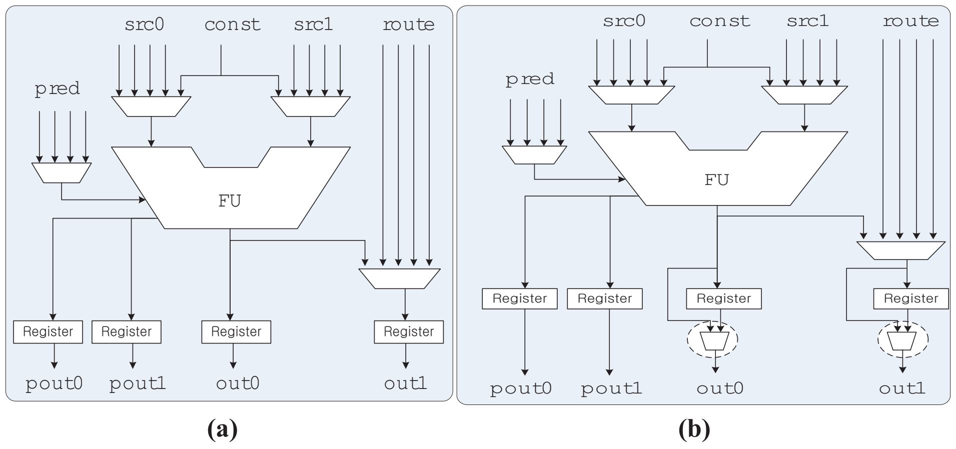
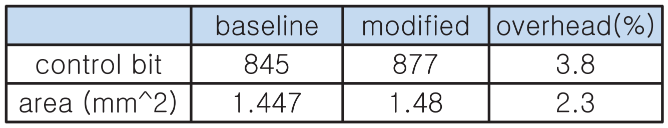


Figure 3.9: Comparison of bypass network implementation details: (a) baseline network and (b) network that supports dynamic operation fusion.

Tick information based on IBM 90nm library is shown in the last column of Figure 3.7.

3.3.2 Bypass Network

Figure 3.9 shows the real implementation of the bypass network with some practical considerations. Figure 3.9(a) is the original FU on the baseline architecture. Each FU has three source MUXes for predicate and data inputs. In addition to this, each FU has one additional MUX to increase the routing bandwidth of the array. Four predicate, compute, and routing outputs are generated from the FU and connected to other FUs through registers. Bypass connections between FUs are implemented by adding a small two-input MUX to two data outputs (Figure 3.9 (b)). The MUX has both an FU output and register output as inputs and one of these signals is chosen by the select signal of the MUX every cycle. This type of MUX is selected to minimize the additional area and delay cost to the baseline architecture. As FU and register outputs are shared, the bandwidth is restricted but the hardware overhead can be reduced by minimizing change of the baseline architecture. An



	baseline	modified	overhead(%)
control bit	845	877	3.8
area (mm^2)	1.447	1.48	2.3

Figure 3.10: Hardware overhead of the bypass network. Two forms of overhead are specified: control bits to control the bypass MUXes and area of the bypass network.

additional 32 control bits and 32 MUXes with 33644 um^2 area are required and the costs are 3.8% and 2.3% overhead (Figure 3.10).

3.4 Compiler Support

In this section, we describe the compiler support for dynamic operation fusion using the bypass network in CGRA Express. Taking the concept of edge-centric modulo scheduling (EMS) [73], we developed a scheduler that can support both sequential and loop code regions for CGRAs. We enhanced the original algorithm with the ability to place multiple operations in a single cycle without incurring the structural hazard of the resources. The concept of tick slot in Section 3.3.1 is introduced into the scheduler and scheduling is performed on a tick basis rather than a conventional cycle-based manner.

First, we will briefly introduce the EMS framework and then describe the basic concepts of tick-based scheduling. Finally, we will provide the added features to attack the problems specific to tick-based scheduling.

3.4.1 Edge-centric Modulo Scheduling

The most distinctive feature of the EMS is that it takes routing of values as the first-class objective. The routing of operands is often ignored in traditional schedulers since it can be guaranteed by the centralized resources (i.e., central register file) of a traditional VLIW processor. Any value generated by a producer can be routed to its consumers by putting the operand into the central register file. However, the distributed interconnect and register files in CGRAs require the compiler to orchestrate the communications between producers and consumers explicitly. The modulo constraint that must be observed to create a correct modulo schedule allows only a limited available slots for each resource, making the routing of operands on the array even harder.

For this reason, EMS constructs the schedule by routing the edges in a dataflow graph, rather than placing the nodes. This approach allows both performance gain and compilation time reduction over the traditional node-centric approach. The following are the major features of the EMS that differentiate it from conventional schedulers.

- **No explicit backtracking.** With the distributed interconnect and abundant computation resources, the scheduling space for CGRAs can get quite large and the compilation time can be a critical issue. To reduce the compilation time, EMS does not have a backtracking mechanism. Especially for CGRAs, it is hard to make forward progress with backtracking since placing and unplacing of operations usually involves multiple resources for routing. Therefore, routing decisions are made just once.

- **Proactive prevention of routing failures.** To compensate for the lack of backtracking, EMS proactively avoids routing failures using probabilistic cost metrics. Before routing an edge, the probabilities of the future usages of scheduling slots are calculated. By avoiding the slots with high probabilities, routing failures can be effectively prevented.

- **Recursive routing calls for critical components.** Some components in a dataflow graph require more cautious scheduling since they can easily make the scheduling fail. One good example is a recurrence cycle. To meet the timing constraints of the recurrence cycles, traditional schedulers usually treat them with highest priority. Additionally, EMS schedules the edges in a critical component altogether by routing them recursively. When an edge in a recurrence cycle is routed, it only finalizes the routing only if all other edges in the component are successfully routed in recursive calls. This recursive routing provides an implicit form of backtracking for scheduling critical components.

3.4.2 Tick-based Scheduling

To enable the scheduler to place back-to-back operations in the same cycle, it needs to keep track of where the operations are placed at the precision of ticks. Figure 3.11(a) shows the scheduling space for tick-based scheduling where each cycle is divided into multiple ticks. For illustration purposes, register file access time is ignored. The number of ticks in a cycle is determined by the frequency of the target architecture and is given as input to the scheduler. Here, operations are placed into tick slots, and the resource management is still done on a cycle basis; only one operation is allowed to be placed in a cycle for each resource.

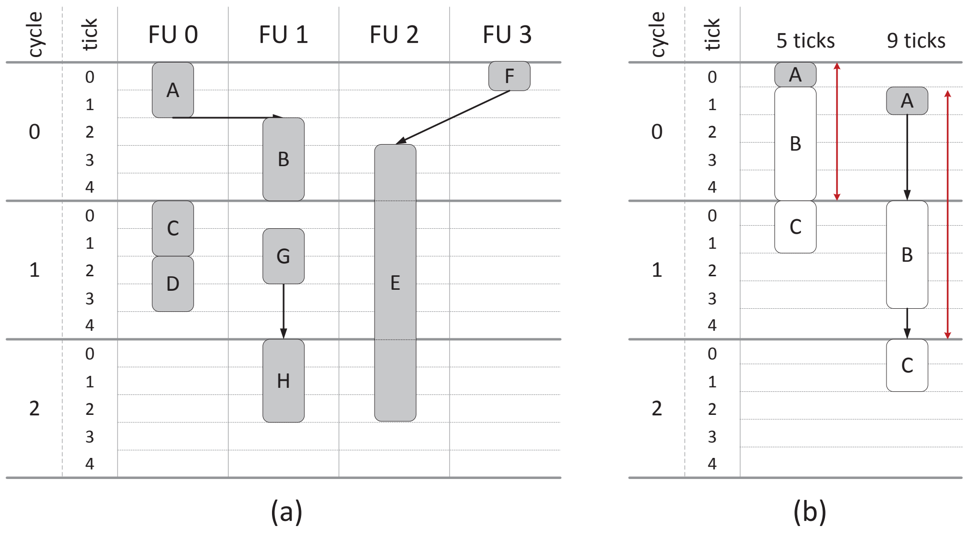


Figure 3.11: Tick-based scheduling example: (a) possible placements in the tick scheduling space and (b) different longest path delays per tick slots.

To manage the cycle and tick times together, we defined *STime* which is a pair, *(cycle, tick)*. *STime* is used for two purposes: schedule time unit , and delay of resources and operations. For example, the input time of operation A in Figure 3.11(a) is scheduled at (0, 0) and its delay is (0, 2). For multi-cycle delays of pipelined operations, *STime* has an additional field of *init_tick* making it a tuple of *(cycle, tick, init_tick)*. *init_tick* indicates the number of ticks required to process the operation at the first pipeline stage. The load operation E shown in Figure 3.11(a) has a delay of (2, 3, 2). While the load operation will have a delay of 3 cycles in a traditional approach, it requires 2 ticks and 3 ticks for the first and last stages, respectively. Therefore, the pipelined operations can also participate in dynamic operation fusion.

Figure 3.11(a) shows some possible placements of operations in tick-based scheduling. Operations A and B are scheduled in the same cycle using the bypass network. However, since the resources are managed in cycles, only one operation can be mapped on a resource in a single cycle. So, it is illegal to place back-to-back operations C and D in the same resource/cycle. Also, an operation cannot be mapped across the clock boundary unless it has a multi-cycle delay. When there is not enough tick slots in a given cycle, the scheduler delays the operation to the next cycle as shown with operations G and H.

Operator Overloading We replaced all the time/delay units in the EMS with our *STime* unit, while keeping the basic structure of the scheduler. So, the changes applied to the original scheduler are minimized. The basic arithmetic operators such as +, -, *, / were overloaded in a way that the *cycle* field increases/decreases as the *tick* field crosses the cycle boundary. Often times, a delay is added or subtracted to a schedule time to create another schedule time. For example, the output time of operation B in Figure 3.11(a) can be calculated by adding the delay (0, 3) to the output time of operation A (0, 1).

However, there are two things to consider when a delay is applied to a schedule time. First, the clock boundary constraint should be checked so that the operation is not placed across the boundary. Also, when adding a multi-cycle delay to a schedule time, the resulting time should be adjusted along the clock boundary since multi-cycle operations should be aligned with the clock boundaries. Basically, the time gap between the output time of the producer and the consumer needs to be added to get the output time of the consumer. The equation below shows how the addition is performed between a schedule time and a delay. *num_ticks* denotes the number of tick slots in a single cycle. *T* is the schedule time and *D* is the delay. When adding a delay to a schedule time, the timing constraint is checked by looking at *init_tick* of the delay (Equation 3.1). When it passes the timing constraint, the delay is added using the overloaded operator '+'. For multi-cycle delays, the time is converted to its floor to align the resulting time along the clock boundary (Equation 3.2).

After performing the addition, Equation 3.3 checks if the performed addition violates the clock boundary constraint.

$$if(D.cycle > 0)\ num\_ticks - T.tick >= D.init\_tick \tag{3.1}$$

$$add(T, D) = (D.cycle > 0)?(T.cycle, 0) + D : T + D \tag{3.2}$$

$$check(T, D) = (add(T, D).cycle - T.cycle == D.cycle) \tag{3.3}$$

3.4.3 Tick Specific Features.

By introducing the new *STime* unit, we could minimize the modifications applied to the original EMS. However, there are some features that need to be adapted to efficiently perform tick-based scheduling. Three major features are explained in this section.

ASAP/ALAP time calculations. In schedulers, ASAP and ALAP times are used to estimate how early/late an operation can be placed without destroying timing dependences between operations. The ASAP time of an operation C can be calculated by Equation 3.4. *p* denotes an placed predecessor of *C* and *d(x, y)* is the longest path delay between *x* and *y*.

$$ASAP(C) = MAX(time(p) + d(p, C)) \tag{3.4}$$

Basically, the scheduler looks at all the already-placed predecessors in the dataflow graph and adds the longest delay between the predecessor and the current operation, and picks the maximum time. In cycle-based scheduling, the longest delay stays constant no matter which cycle the predecessor is placed. However, in tick-based scheduling, the longest delay changes depending on which tick slot the predecessor is placed. Figure 3.11(b) shows an example of the different delays between operation A and C. Here, we assume that A is already placed and B and C are not. Since the operations cannot be scheduled across the clock boundaries, the delays are different between the two cases. Therefore, the tick-based scheduler calculates the longest delay of two operations for each producer's tick slot in a cycle.

Identifying Subgraphs. To find the opportunities for dynamic operation fusion, the scheduler takes a greedy approach for finding the target subgraphs. When an operation is placed, the scheduler looks at its neighboring operations in the dataflow graph and checks the timing constraints to see if they can fit into the same cycle using the bypass network. If there is an opportunity for fusion, the scheduler recurses on the routing of an edge between the two back-to-back operations. The use of the bypass network is encouraged in routing by giving a penalty when the cycle is increased during the routing. The router will visit the available slots in the same cycle first using the bypass network. However, this can result in wasting FU slots just for routing since the bypass network connects neighboring FUs. For this reason, we only allow the use of the bypass network when back-to-back operations can be placed in neighboring FUs.

Register Access Region. Even though the register access time was ignored in Figure 3.11, the register read and write times need to be considered in reality. The shaded regions in the scheduling space in Figure 3.12 display the register access region. Here, we assume the register read and write time is 1 tick. For each cycle, the first tick slot is called the register read region and the last tick slot is called the register write region. When

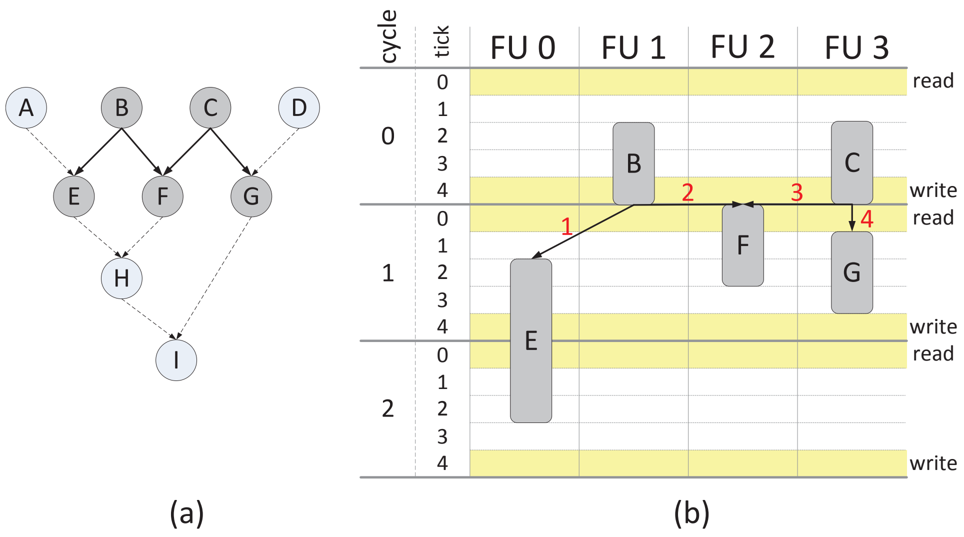


Figure 3.12: Register access regions in a tick schedule: (a) dataflow graph, (b) register read/write regions (shaded) within each cycle.

operations are placed in these regions, they cannot access register files due to timing constraints. For example, operation B's output is placed at (0, 4) slot and it can only route its value to neighboring FUs through the FU's output register. Therefore, routing flexibility is greatly limited for operation B. When all the neighboring FUs are occupied, the scheduling will fail since there is no backtracking mechanism. To avoid this situation, our scheduler performs recursive calls for routing edges when an operation is placed in the register access region. Figure 3.12(a) shows an example dataflow graph. When operation B is placed at cycle 0 as shown in the figure, its output is placed in the register write region. Therefore, the scheduler makes sure that all the edges coming out from operation B are successfully routed before finalizing the placement. Therefore, it recurses on the routing of two edges (E and F). When operation F is placed in cycle 1, the scheduler also recurses on the edge to operation C since F is placed in the register read region. The numbers shown in the figure denote the order of routing call of each edge. Since the operations E, F, and G are not placed in the register write region, they can store values into the register files. So, the scheduler does not proceed with routing the outgoing edges from them. When all the edges with solid lines in Figure 3.12(a) are successfully routed, the scheduler finalizes the placement of operation B.

3.5 Experimental Results

3.5.1 Experimental Setup

Target Architecture Two CGRA architectures are used to evaluate the performance of dynamic operation fusion. The baseline architecture is the 4×4 heterogeneous CGRA shown in Figure 3.1. Four FUs are able to perform load/store instructions to access the data memory and 6 FUs support 2-cycle pipelined multiply. A 64-entry central register file with 6 read and 3 write ports and sixteen 8-entry local register files exist in the array. Only four FUs on the first row have direct access to the central register file and other FUs must use data buses to access the central register file. Local register files with one read and one write port are placed similar to the FUs and each register file can be written by FUs in diagonal directions. There is also one 64-entry predicate register file with four read and four write ports. The CGRA Express architecture has the same architectural shape except the addition of the bypass network.

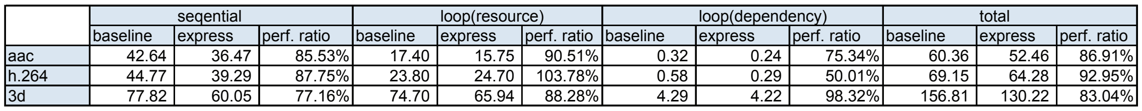
Target Applications All the sequential and loop code are taken from three application domains: audio decoding (aac), video decoding (h.264) and 3D graphics (3d). The sequential code regions are mapped using VLIW mode of the array and loop code regions are mapped using CGRA mode of the array. Performance is evaluated by the overall execution time.

Power/Area Measurements Both the baseline and CGRA Express architectures are generated in RTL Verilog and synthesized with the Synopsys design compiler and Physical compiler using IBM 90nm standard cell library in typical operation conditions. Synopsys PrimeTime PX is used to measure power consumption. The SRAM memory power was calculated using SRAM model generated by the Artisan Memory Compiler. The target frequency of both baseline and the CGRA Express architectures are 200MHz.

3.5.2 Performance Measurement

In order to illustrate the effectiveness of dynamic operation fusion, performance of the three benchmarks is compared on the baseline CGRA and CGRA Express. In sequential code regions, run-time is measured by the schedule length multiplied by the frequency of execution. The run-time of the loop code regions is calculated by multiplying the Initiation Interval (II) achieved by EMS and the loop trip count. II means the interval between successive iterations thus II is the indicator of throughput in modulo scheduling. The results of this experiment are shown in Figure 3.13. The numbers in the table show the execution time in millions of cycles and perf.ratio is the ratio of execution time on CGRA express over the baseline.

Overall, dynamic operation fusion achieves 7-17% reduction in execution time over the baseline. This is a promising result because the hardware overhead is about 3% as discussed in Section 3. More specifically, most of the performance improvements are due to the



	seqential			loop(resource)			loop(dependency)			total		
	baseline	express	perf. ratio	baseline	express	perf. ratio	baseline	express	perf. ratio	baseline	express	perf. ratio
aac	42.64	36.47	85.53%	17.40	15.75	90.51%	0.32	0.24	75.34%	60.36	52.46	86.91%
h.264	44.77	39.29	87.75%	23.80	24.70	103.78%	0.58	0.29	50.01%	69.15	64.28	92.95%
3d	77.82	60.05	77.16%	74.70	65.94	88.28%	4.29	4.22	98.32%	156.81	130.22	83.04%

Figure 3.13: Performance evaluation of the baseline and CGRA Express architectures for three multimedia applications. Performance is broken down into non-innermost loop regions (sequential), inner-most loops whose performance is constrained by the availability of resources (loop (resource)) and inner-most loops whose performance is constrained by cross-iteration dependences (loop (dependency)).

schedule length reduction in sequential code regions, which was expected since dynamic operation fusion collapses the series of operations into a single cycle.

However, we could also observe a good amount of reduction in resource-constrained loops. This is primarily due to the additional bypass network. The additional connection doubles the number of reachable slots from an FU. With the bypass network, an FU can access its neighboring FUs results in the same cycle as well as in the next cycle. This gives the scheduler more flexibility and improves the throughput of the resource constrained loops. Also, when a loop has small trip count, schedule length will be more dominant than the II for run time, hence dynamic operation fusion can improve performance. The dependence-constrained loops show up to 50% reduction in execution time. This was expected since the throughput of these loops was mainly limited by the critical path of a single iteration, which can be efficiently reduced by dynamic operation fusion.

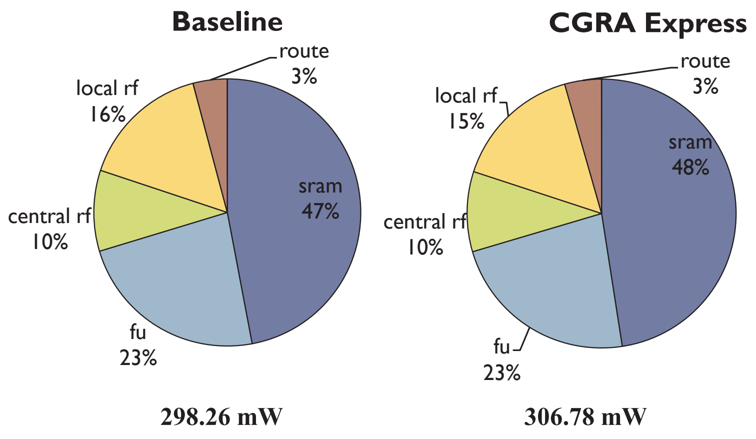
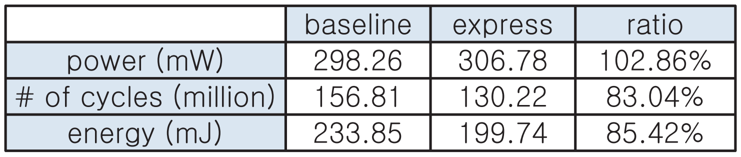


Figure 3.14: Power breakdown comparison for the baseline and CGRA Express architectures.

3.5.3 Power and Energy Measurement

The instantaneous power consumption of CGRA Express architecture is seemingly higher than that of baseline architecture due to additional hardware overhead. However, the bypass network implementation can also decrease the total run time. Since there is such trade off between power and run time, we measured total energy consumption for running complete applications to determine the effectiveness of dynamic operation fusion.

Overall power consumption and the breakdowns of both architectures for 3D are shown in Figure 3.14. Overall, average power consumption on the CGRA Express architecture is 3.1% higher than the baseline architecture. Compared to the baseline architecture, the power increase observed for the datapath is smaller than the increase in the SRAM for control signals. The bypass network adds just a small amount of combinational logic (MUXes) on the baseline architecture, hence the overall effect is quite small. On other hand, adding control signals is more critical for power consumption on CGRAs because all the control signals must be read every cycle. Therefore, overall power overhead for adding bypass network is trivial but careful consideration is necessary due to the additional number of



	baseline	express	ratio
power (mW)	298.26	306.78	102.86%
# of cycles (million)	156.81	130.22	83.04%
energy (mJ)	233.85	199.74	85.42%

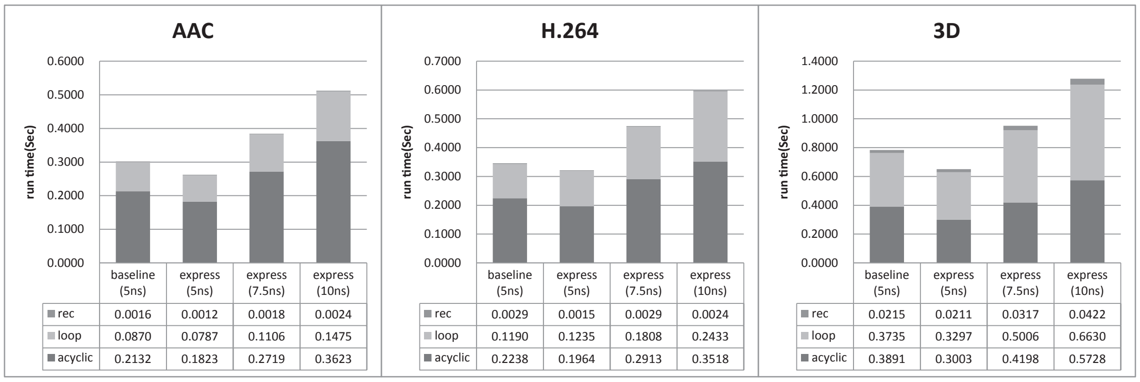
Figure 3.15: Energy comparison for the baseline and CGRA Express architectures.

control signals.

An interesting result can be found on total energy consumption comparison between both architectures. Figure 3.15 shows that the CGRA Express architecture is 15% more energy efficient than the baseline architecture. Even though average power consumption of the new architecture is slightly higher, the decrease in application run time dominates the results.

3.5.4 Operating Frequency Optimization

As discussed in prior experiments, dynamic operation fusion can decrease total run time by decreasing number of cycles in fixed clock period. However, measuring total run time on various clock periods will be another interesting question with dynamic operation fusion. With different clock periods, total run time is calculated by multiplying the number of cycles and the clock period. If clock period is large, more operations can be chained into a single cycle. But, these gains must offset the losses in performance due to a reduced clock rate. We can expect some optimal smallest run time exists as the clock period is swept that represents the sweetspot of a fast clock rate while permitting some degree of chaining. Figure 3.16 shows the total run time of the three applications with various clock periods in nanoseconds.



AAC	baseline (5ns)	express (5ns)	express (7.5ns)	express (10ns)
▨ rec	0.0016	0.0012	0.0018	0.0024
▨ loop	0.0870	0.0787	0.1106	0.1475
▨ acyclic	0.2132	0.1823	0.2719	0.3623

H.264	baseline (5ns)	express (5ns)	express (7.5ns)	express (10ns)
▨ rec	0.0029	0.0015	0.0029	0.0024
▨ loop	0.1190	0.1235	0.1808	0.2433
▨ acyclic	0.2238	0.1964	0.2913	0.3518

3D	baseline (5ns)	express (5ns)	express (7.5ns)	express (10ns)
▨ rec	0.0215	0.0211	0.0317	0.0422
▨ loop	0.3735	0.3297	0.5006	0.6630
▨ acyclic	0.3891	0.3003	0.4198	0.5728

Figure 3.16: Performance comparison of the baseline and CGRA express architectures for different clock periods. Performance is broken down into dependence-constrained loops (rec), resource-constrained loops (loop) and non-innermost loops (acyclic) regions.

Dynamic operation fusion works efficiently at 5ns compared to traditional scheduling but expanding the clock period to more than 5ns achieves no additional performance improvement. As the clock period becomes longer, sequential code regions require fewer cycles to execute and their characteristics start to resemble loop code regions. This behavior occurs because just 4 FUs are used for executing sequential code regions. With the most aggressive fusion, the dependences of 4 successive instructions are collapsed which basically eliminates all dependences that can constrain performance and converts the code region into a resource constrained one. Moreover, the number in sequentially dependent instructions before a memory instruction is encountered is typically smaller than 4, thus there are limited opportunities for fusion. As a result, using a clock period of 7.5ns results in 50% increase of total run time because there is no additional reduction of the number of clock cycles due to dynamic operation fusion (beyond those saw at 5ns), but the clock period is 50% larger.

3.6 Related Work

3.6.1 Architecture

Many CGRA-based systems have been proposed in various papers and some of the models have been implemented. Each design has different scalability, performance, and compilability. ADRES [64] is the most well-known CGRA system with an 8x8 mesh of processing elements with central and local register file. As we mentioned prior sections, ADRES also supports CGRA array mode as well as VLIW mode using central register file and FUs on the top row. MorphoSys [61] is another famous example of 8x8 grid with a more sophisticated interconnect network. In MorphoSys, each node has an ALU and a small local register file. RAW architecture is more general system which node is small MIPS processor with memory, registers, and a processor pipeline. PipeRench [34] and RaPid [31] are also 1-D architectures have similar concept to CGRAs. In PipeRench, each processing elements are arranged in stripes to support pipelining. RaPid has a lot of heterogeneous elements (ALUs and registers), which can be connected by reconfigurable interconnection.

The results of recent research about general architecture exploration on CGRAs are also promising. Kim [48] focussed on the power consumption for configuration memory and proposed spatial and temporal mapping with pipelining. Moreover, Kim [47] proposed different approach based on data flow graph of applications.

Research on instruction set customization with configurable compute accelerator (CCA) is also closely related to this research. Clark [25] studied how to create efficient CCA based on sub graph modulation and improved the idea to virtualized execution accelerator [27].

Hormati [38] also studied CCA to be more faster and smaller. Lastly, Bonzini[19] adopt the CGRA idea to CCA and diminish disadvantages of CCA, such as logic depth limitation and low coverage.

3.6.2 Compilation Techniques

As dealing with sparse connectivity and distributed register file is huge challenge on compiler, many techniques have been proposed for compiling CGRAs. Lee [56] proposed a schedule approach for a generic CGRA, which generates pipeline schedules for innermost loop. Park [72] also worked on innermost loop, but they focussed on loop level parallelism while Lee worked on instruction level parallelism. Park's work is more similar to Mei at al [65]'s work on modulo scheduling.

Research on CGRA scheduling is partially similar to the research on VLIW machine scheduling. As clustered VLIW machines are also spatial architecture, many compilation techniques on VLIW can be adopted to CGRAs. However, VLIW machine does not have routing issues related to sparse interconnection network hence some modification is necessary to support CGRA.

On this chapter, we introduce some cost function about actual delay of synthesized hardware (MUX, Adder, Shifter). This concept is similar to the research about module mapping and placement on FPGA area. Callahan [21] performed datapath module placement simultaneously with the mapping using area and delay cost. They used the area and delay cost to minimize both area and delay on FPGA. We also adopt the delay cost to increase utilization of FUs on pre-defined clock period.

3.7 Summary

This work proposes dynamic operation fusion, an effective approach to accelerate sequential code regions on CGRAs. As scheduling techniques for loops have been developed, the run-time for loops decreases by large factors as the compiler is able to make effective use of the abundance of resources available in a CGRA. However, the side effect is that sequential code region become more and more of the overall performance bottleneck as these regions have limited instruction-level parallelism. We introduce two key concepts to execute sequential code region faster. First, a bypass network is implemented to support dynamic operation fusing wherein existing function units on a CGRA are configured to execute back-to-back operations in a single cycle using any available slack in the cycle time. A simple hardware extension in the form of an additional connection between neighboring function units and a bypass MUX are required. Second, the compiler scheduler automatically identifies opportunities for dynamic fusion based on sub-units of clock cycles, called ticks. Overall, dynamic operation fusion reduces total application run-time by 7-17% and total energy by 15% on a 4x4 CGRA.

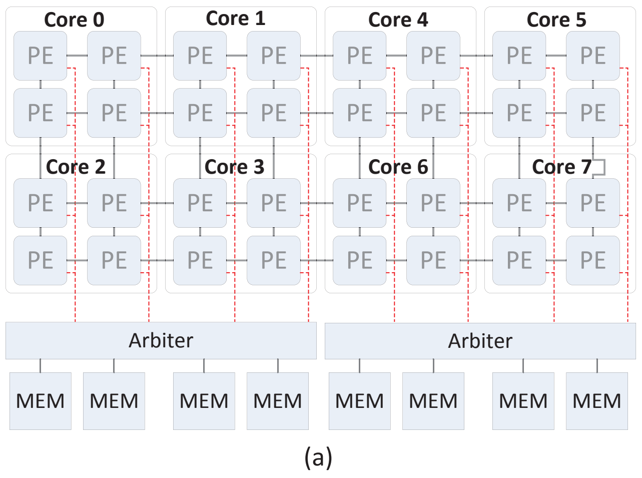
CHAPTER 4

Putting Idle Resources to Work on a Composable Accelerator

4.1 Introduction

The mobile devices market, including cell phones, netbooks, and personal digital assistants, is one of the most highly competitive businesses. The computing platforms that go into these devices must support ever increasing performance capabilities while maintaining low energy consumption. Advanced multimedia and signal processing applications are key drivers. Traditionally, application-specific integrated circuits (ASICs) were used for the heavy lifting to perform the most compute intensive kernels in a high performance but energy-efficient manner. However, several features push designers to a more flexible and programmable solution: supporting multiple applications or variations of applications, providing faster time-to-market, and enabling algorithmic changes after the hardware is constructed.

For wireless signal processing, programmable designs that exploit high degrees of



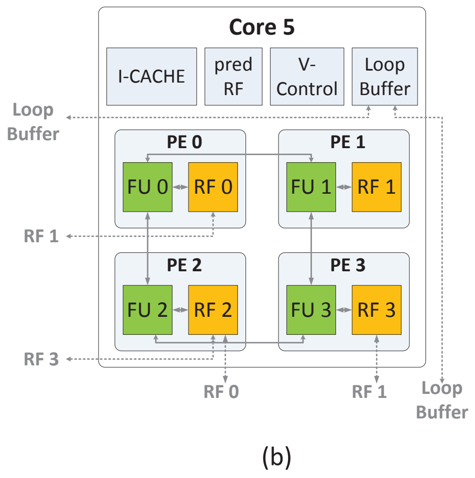


Figure 4.1: PPA Overview: (a) PPA with 8 cores, (b) Inside a single PPA core

single-instruction multiple-data (SIMD) parallelism have emerged to challenge ASICs [17, 16, 33, 59, 89]. While these solutions suffice for wireless signal processing, multimedia applications contain more complex data dependence patterns and frequent control flow for which wide-SIMD is inefficient. Thus, a different approach is necessary.

Polymorphic pipeline arrays (PPAs) are attractive alternatives for accelerating multimedia applications because the hardware is more flexible and can accelerate the code in multiple ways [74]. Coarse-grain pipeline parallelism is exploited by concurrently executing filters in streaming applications [35, 36, 50], as well as fine-grain instruction level parallelism is also found by modulo scheduling innermost loops [79]. A PPA is a generalization of a coarse-grain reconfigurable architecture (CGRA) shown in Figure 4.1 [66]. It consists of an array of simple processing elements (PEs) that are tightly interconnected by a scalar operand network and a shared memory. Groups of four PEs form cores that are driven by a single instruction stream. These cores can execute tasks (filters in a streaming application) independently or neighboring cores can be coalesced to execute loops with high degrees of fine-grain parallelism. The use of a regular interconnection fabric allows the core boundaries to be blurred, thereby allowing the hardware to be customized differently for each application.

While PPAs provide the opportunity for hardware customization, an effective compiler is necessary to configure the hardware to maximize application performance. In this work, we adopt the stream programming paradigm. Stream programming is generally based on synchronous dataflow wherein the application is represented as a directed graph (stream graph) where each node represents an actor and each arc represents the flow of data [55]. The number of data samples produced and consumed by each node are specified a priori. For this work, we focus on stream-style C code where a program is represented as a set of autonomous actors (also called filters) that operate on data and communicate through first-in first-out data channels [87]. During program execution, actors fire repeatedly in a periodic schedule [36]. Each actor has a separate instruction stream and an independent address space, thus all dependences between actors are made explicit through the communication channels. Compilers can leverage these characteristics to plan and orchestrate parallel execution.

Given a streaming application, the primary challenge is to perform resource allocation and assignment so as to achieve maximum throughput. More specifically, a PPA compilation framework must not only partition filters across the available cores, but also aggregate cores together into core-groups to jointly execute the assigned filters. Larger core-groups are effective for long-running filters because higher levels of fine-grain parallelism can be exploited. By modulo scheduling across more resources, higher performance is achieved. However, selecting large core-groups reduces the overall number of groups and hence the amount of coarse-grain pipeline parallelism that can be exploited. Greedily speeding up a small portion of the application often results in poor overall performance. Thus, an intelligent compiler must achieve a balance.

In this chapter, the goal is to solve the joint filter assignment and core aggregation problem for mapping streaming applications onto a PPA. We start by defining the main scheduling constraints on PPA architectures, and propose a new compilation process to solve the difficulties. In this framework, we adapt the key concept from the stream graph modulo scheduling algorithm for coarse-grain parallelism [50]. The main difference is that parallel composition of the each filter is not performed with split-joins, but by modulo scheduling across larger core-groups. With this change, the PPA compiler can be used for more generic code by removing the restrictions of static data rates on stream programming languages like StreamIt [87]. Edge-centric modulo scheduling (EMS) [73], which focuses on routing of values between functional units, is used as the modulo scheduling technique for exploiting fine-grain parallelism.

The compilation process consists of three steps. First, filters are assigned to virtual cores using static partitioning and an approximate load balancing algorithm. Next, core allocation is performed to map the virtual cores to the physical cores considering core locations and the inter-filter communication patterns. Finally, fine-grain dynamic partitioning is performed to identify and recycle under-utilized resources.

This work offers the following three contributions:

- An analysis of the scheduling difficulties for composable accelerators such as the PPA.

- A compilation framework for jointly partitioning streaming applications across hardware resources and selecting resource aggregations that jointly exploit coarse-grain parallelism between filters and fine-grain parallelism within filters.

- An efficient resource borrowing technique is proposed to reduce the execution time of the largest coarse-grain pipeline stage by borrowing resources from underutilized stages.

4.2 Background and Motivation

4.2.1 Composable Accelerators

As chip multiprocessors (CMPs) have become commonplace in today's desktop environment, their importance is growing rapidly in the mobile environment. The disparity between the granularity of parallelism in workloads and the granularity of processing cores inspired a flexible execution model that allows the aggregation of small cores to create larger logical cores [44],[42].

Composable accelerators are multi-core accelerator designs that incorporate this flexible execution model in embedded systems. Multiple small cores enable the parallel execution of individual tasks, exploiting task level parallelism. Additionally, when there is a high degree of parallelism within a task, such as loop level parallelism or instruction level parallelism, a larger core can be created by merging small cores. With this flexible execution model, different levels of parallelism can be exploited with a single piece of hardware.

Our specific compilation target is the *Polymorphic Pipeline Array* (PPA) shown in Figure 4.1. A PPA is a composable accelerator for embedded systems that can exploit both the fine-grain parallelism found in innermost loops and the pipeline parallelism found in streaming applications. A PPA consists of multiple simple cores that are tightly coupled to neighboring cores in a mesh-style interconnect. A PPA with 8 cores is shown in Figure 4.1(a). There are a total of 32 processing elements (PEs) in this PPA, each containing one function unit (FU) and a register file (RF). Four PEs are combined to create a core that can execute its own instruction stream. Each core has its own scratch pad memory and column buses connect 4 PEs to a memory access arbiter that provides sharing of scratch pad memories among the cores.

The detailed diagram of a single PPA core is shown in Figure 4.1(b). Each PE contains a 32-bit FU and a 16 entry register file. PEs are connected to a mesh-style interconnect. The distributed nature of PPA provides low power consumption and hardware cost making it an attractive solution for embedded systems. The mesh interconnect also connects the neighboring PEs in different PPA cores. This allows fast inter-core communication for mapping compute intensive loop nests across multiple cores. A detailed description of PPA cores can be found in [74].

4.2.1.1 Supporting Different Levels of Parallelism

The major feature of the PPA is its ability to exploit both fine-grain and coarse-grain pipeline parallelism. Since each PPA core can process its own instruction stream, coarse-grain parallelism can be exploited for streaming applications. The communication between pipeline stages can be efficiently supported with DMA connections between cores. Abundant fine-grain parallelism within a pipeline stage can also be exploited by aggregating multiple cores to form a larger logical core allowing for maximized resource utilization. This is efficient since the PPA provides fast inter-core communication using a mesh-style interconnect.

4.2.1.2 Virtualization

One of the major characteristics of a PPA is virtualized execution of software pipelined loops [74]. Virtualized modulo scheduling generates a unified schedule that can be mapped onto different target sub-arrays of the PPA. At runtime, the PPA cores are dynamically merged to create larger logical cores based on the resource availability. With virtualization support, tasks can execute on different sized cores without rescheduling, improving the overall performance when the resource requirement in the workloads varies dynamically during execution [74]. However, there are some limitations of virtualization on a PPA, such as sub-optimality of the unified schedules and runtime overhead for virtualization.

4.2.1.3 Partitioning Schemes

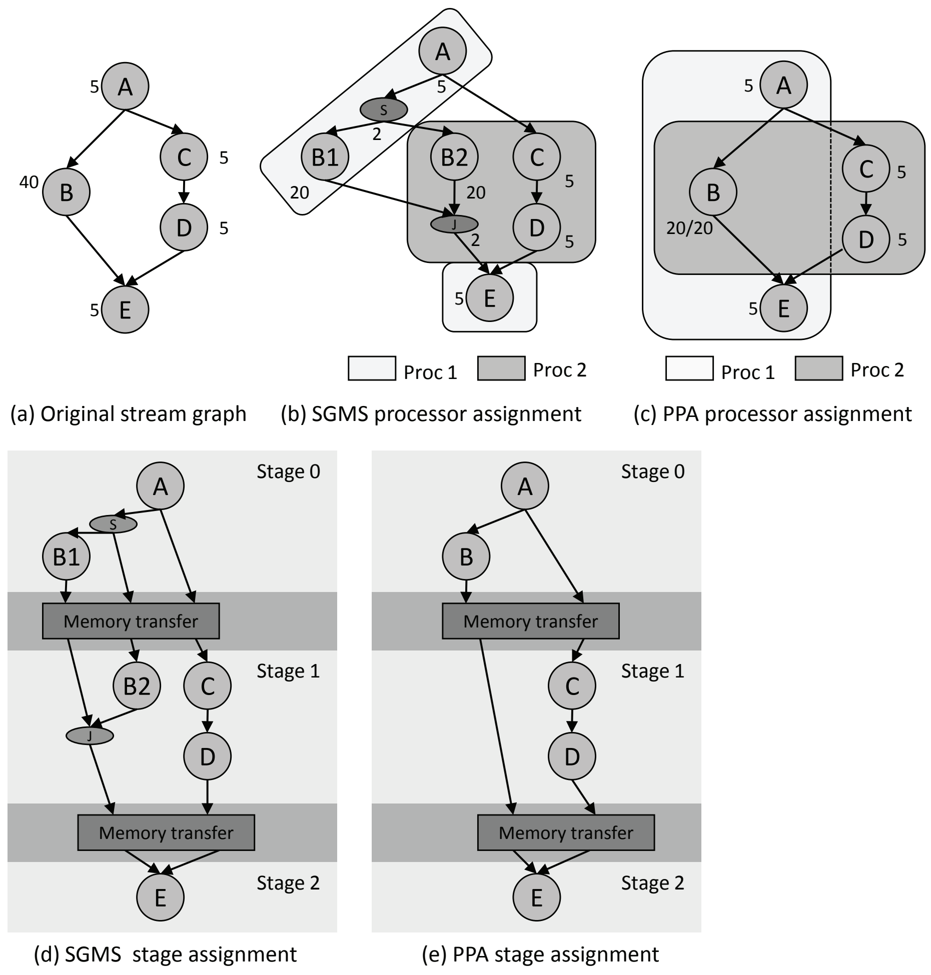
Static Partitioning. The PPA array can be partitioned statically based on the resource requirement of each coarse-grain pipeline stage. Static partitioning has its benefit in achieving high quality schedules, but it cannot adapt to dynamically changing resource availability. When an application has a large variation in execution pattern, static partitioning can either result in low utilization of resources, or may not be able to fully accelerate the application when there are not enough resources available.

Dynamic Partitioning. Coarse-grain pipeline stages in multimedia applications have different execution patterns, resulting in fluctuating resource requirements. Dynamic partitioning can come in handy with the presence of dynamic variation of resource requirements. The partitioning of the PPA array can change during runtime on demand. For a single pipeline stage, a single core can be assigned to an acyclic region of code, but more resources can be assigned to the compute intensive loop kernels to exploit fine-grain parallelism. Dynamic partitioning assumes the sharing of resources between neighboring pipeline stages. The resources sitting idle in one stage can be utilized by neighboring stages through resource borrowing. So, it is not guaranteed that the required resource is available at all times in dynamic partitioning. When the required resource is not available, the stage stalls and waits for the resource. Virtualization can avoid stalls due to resource contention by generating a schedule that can be modified easily at runtime to run on different number of resources.

4.2.2 Stream Graph Modulo Scheduling

This work presents a compiler technique specifically for composable accelerators based on *stream graph modulo scheduling*, or SGMS [50]. SGMS is a modulo scheduling algorithm for mapping streaming applications onto multicore systems. Modulo scheduling is traditionally a form of software pipelining applied at the instruction level to find a valid schedule for a loop such that the interval between successive iterations (initiation interval, or II) is minimized [79]. SGMS is the same technique on a coarse-grain stream graph to pipeline the actors across multiple cores. The objective is to maximize concurrent execution of actors while hiding communication overhead to minimize stalls.

SGMS consists of two steps: 1) integrated fission and processor assignment and 2) stage



A
B
C
D
E
5
40
5
5
5
(a) Original stream graph
S
J
B1
B2
2
20
20
2
Proc 1
Proc 2
(b) SGMS processor assignment
20/20
(c) PPA processor assignment
Stage 0
Stage 1
Stage 2
Memory transfer
(d) SGMS stage assignment
(e) PPA stage assignment

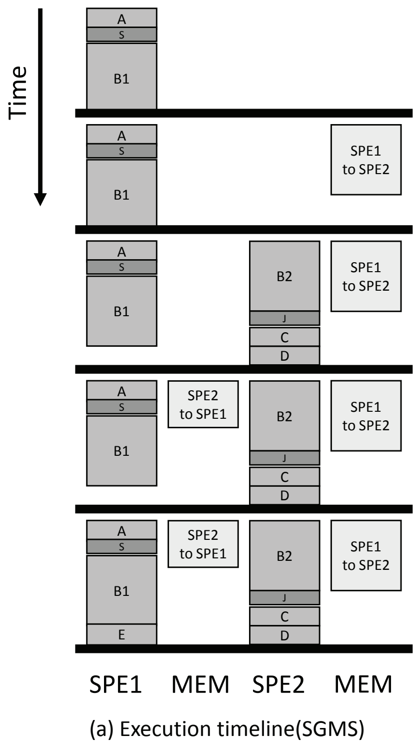
Figure 4.2: Example of processor and stage assignment for SGMS and PPA scheduling.

assignment. The first step is to assign actors to each processor with maximum load balance using an integer linear program formulation. Stateless data actors are replicated and fissed to achieve even work distribution. In stage assignment, the compiler decides a pipeline stage for each actor at runtime. The optimization process in this stage is to maximally hide inter-processor communication latency and not to violate data dependences.

Even though this work adapts the basic concept of the SGMS, task scheduling in PPAs is different in several aspects. First, the PPA scheduler is proposed using legacy C code, hence it has less restrictions than SGMS using streaming languages such as StreamIt. For example, SGMS can exploit parallelism for only stateless actors, but modulo scheduling also can be applied to stateful actors. In addition, PPAs do not incur fission overhead (split, join) to assign multiple cores due to the tightly coupled inter-core scalar network for aggregation.

Figure 4.2 shows the differences between SGMS and PPA scheduling. Given an example stream graph (Figure 4.2(a)), all actors are assumed data parallel. When SGMS schedules the graph on 2 processors (Figure 4.2(b)), the resultant II is 32 because the slowest node B is fissed once and corresponding split-join overhead is incurred. Figure 4.2(c) is the resultant schedule for the PPA, enabling the processor assignment to achieve an II of 30 as node B is accelerated by core aggregation without overhead. Finally, Figure 4.2(e) shows the stage assignments for PPA schedule in which the entire node B is executed in stage 0 within 20 time units by using both cores.

Figure 4.3 shows the execution timeline of both SGMS and PPA schedules. The main difference between the two schedules is the locations of node B: it is split into two independent pieces using SGMS on a multicore and with the PPA it is executed as a whole by aggregating the resources of both core 1 and 2. Note that with the PPA, node B must be scheduled at the same time on both cores in order to exploit resource aggregation. Another interesting point is that since tightly-coupled memory system in the PPA provides lightweight memory synchronization mechanism, scheduling is more tolerable to high memory transfer.



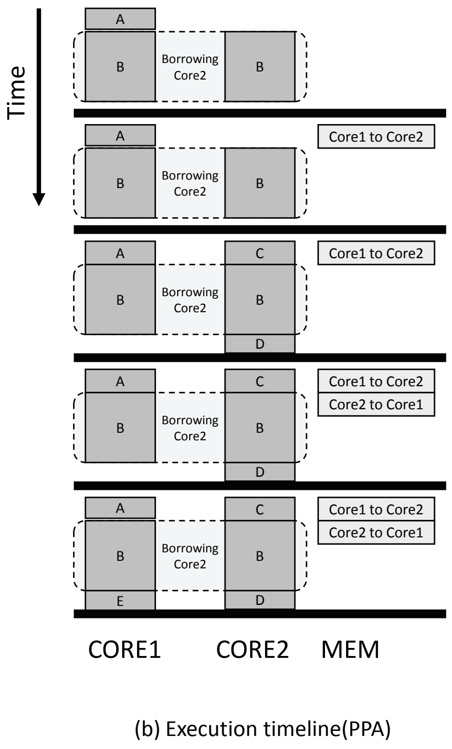
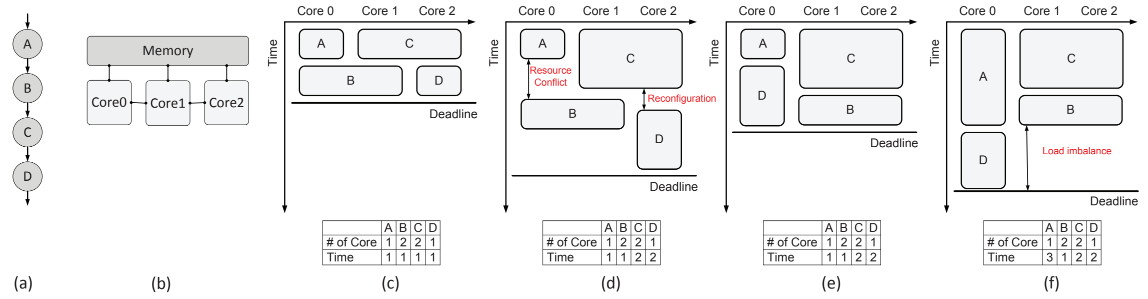


Figure 4.3: Example of running a SGMS on multi-core and a modulo scheduling on PPA.

4.2.3 Compilation Challenges

Efficient scheduling for composable accelerators is now emerging as an interesting, and challenging problem due to the high degree of freedom in both the hardware and software. Some factors that make scheduling difficult are:

Resource Requirement Variance: The optimal resource requirement for efficient parallelism depends on the task-specific characteristics. For example, cyclic code regions can be accelerated efficiently by appropriating more resources, but the performance of acyclic code with sequential dependences cannot be improved by supplying additional resources [77]. Assuming worst-case requirements for all code segments leads to either over-provisioned designs to achieve a desired performance or under-performance for a fixed



(c)

	A	B	C	D
# of Core	1	2	2	1
Time	1	1	1	1

(d)

	A	B	C	D
# of Core	1	2	2	1
Time	1	1	2	2

(e)

	A	B	C	D
# of Core	1	2	2	1
Time	1	1	2	2

(f)

	A	B	C	D
# of Core	1	2	2	1
Time	3	1	2	2

Figure 4.4: Examples of the runtime overhead: (a) original task graph, (b) simple 1x3 PPA, (c) expected ideal schedule with high resource utilization, (d) runtime overhead: stall, reconfiguration time, (e) static partitioning with low runtime overhead, (f) a possible problem of the static partitioning: workload imbalance.

design.

Execution Time Variance: Composable accelerators typically have multiple tasks running in parallel, and they usually have complex dependences. Thus, it is hard to predict the resource usage pattern and accommodate the optimal execution of multiple instruction streams.

Geometry: In CMPs, full connectivity between processors is often provided. However, in a low-cost accelerator, the interconnect is much more sparse and merging cores should be performed in a connectivity-aware manner.

To illustrate these difficulties, Figure 4.4 shows some simple, but frequently occurring examples that result in resources being wasted. The simple dataflow graph (DFG) in Figure 4.4(a) is being scheduled on a simple composable architecture (Figure 4.4(b)). Assuming the optimal resource requirements of each node(A, B, C, D) is 1, 2, 2 and 1 cores with the same execution time, the expected schedule is similar to Figure 4.4(c). However, even though the optimal number of cores is assigned, the different amounts of work in each node results in different execution times. On top of that, if C and D have long execution time, node B cannot start execution at the completion of task A, but must wait until the execution of node C is finished because of resource conflicts (Figure 4.4(d)). Another potential source of resource waste occurs when changing the core assignment. In Figure 4.4(d), task D is delayed by the reconfiguration time even though enough resources are available.

Static partitioning of the cores can potentially eliminate these problems, such as stalls and reconfiguration overhead (Figure 4.4(e)). Static partitioning means the core aggregation is not changed at runtime and each task is assigned to a suitable merged core. In this scheme, task A is not preferred to be executed in core group (1, 2) because the best resource requirement for A is one core. If A is assigned to 2 cores, resources cannot be utilized sufficiently. However, the workload of each core may not be balanced well because we categorized all the tasks based on optimum resource requirements (Figure 4.4(f)). To minimize this side effect, a final performance tuning phase is performed using dynamic partitioning of cores. For example, task D can be changed to run using 3 cores after final tuning because all the other resources remain idle. Additionally, we also propose a core reallocation mechanism to avoid geometry-based runtime overhead.

In this work, our work is focused on finding the optimal partitioning of cores for a given task graph rather than changing the task graph itself. Although modifying the task graph is also a common load balancing strategy, it usually cannot be applied well to the graph itself without changing the source code due to the memory and control dependences.

4.3 Compiler Framework

In this section, we describe our new compilation framework based on the insights discussed in the previous section. The purpose of this framework is to achieve the highest throughput by minimizing stalls due to resource contention and reconfiguration processes. The compilation process consists of three different stages: prepass static partitioning, core allocation, and postpass dynamic partitioning. Prepass heuristically fuses virtual (no geometry information) PPA cores to accommodate larger pipeline stages based on the profile workload information with static partitioning. Core allocation maps the virtual cores onto physical cores, avoiding failures that occur when cores in same group are not connected together. Postpass performs final performance tuning to reduce the completion time of bottleneck pipeline stages by exploiting resource borrowing.

All compilation steps are performed at compile time. Virtualization is not considered in this framework because of performance overheads, both on the hardware and compiler sides. For the hardware, a virtualization controller has execution time overhead for checking the resource availability of the neighbor cores. In addition to this, virtualized modulo schedule also has some performance degradation as it generates only one schedule to support various core configurations [74]. Despite these performance overheads, virtualization can improve the overall performance in specific situations, such as when running an application on a small number of resources or running an application with huge dynamic variance [74]. However, we just generate one schedule per stage and disable virtualization even when using dynamic partitioning to evaluate the real effectiveness of this strategy.

4.3.1 Prepass: Static Partitioning

As we discussed in Section 4.2.3, the goal of this compilation stage is to minimize idle and reconfiguration time between tasks and to create high quality schedules that maximize resource utilization in order to minimize execution time of assigned work. To achieve this goal, we propose resource grouping using static partitioning. This section describes our method for effectively grouping tasks requiring similar number of cores. The performance improvement achieved by this stage mainly comes from recognizing the huge variance between the optimal resource requirements and execution times of each task. The key idea is to categorize all tasks into some number of available resource combinations, enabling high utilization and assigning the different portions of composable cores based on this information. This method basically enables all the tasks to use the resources efficiently, achieving high throughput. This stage also performs coarse load balancing because the throughput of the program depends on the slowest pipeline stage. Therefore, imbalance between core groups leads to performance degradation even if all the tasks execute efficiently. Load balancing is also performed in the postpass step after identifying the optimal static partition with maximum resource utilization.

Algorithm 2 shows how the optimal core groupings (to support the assigned tasks) are identified to exploit fine-grain parallelism effectively. The general idea is to heuristically assign more cores to larger tasks based on the execution time estimate. However, assigning too many resources to larger cores may not be the best solution because performance enhancement depends on the task-specific characteristics and may result in missed opportunities to accelerate other tasks, given a limited number of cores. Therefore, a *quality factor*

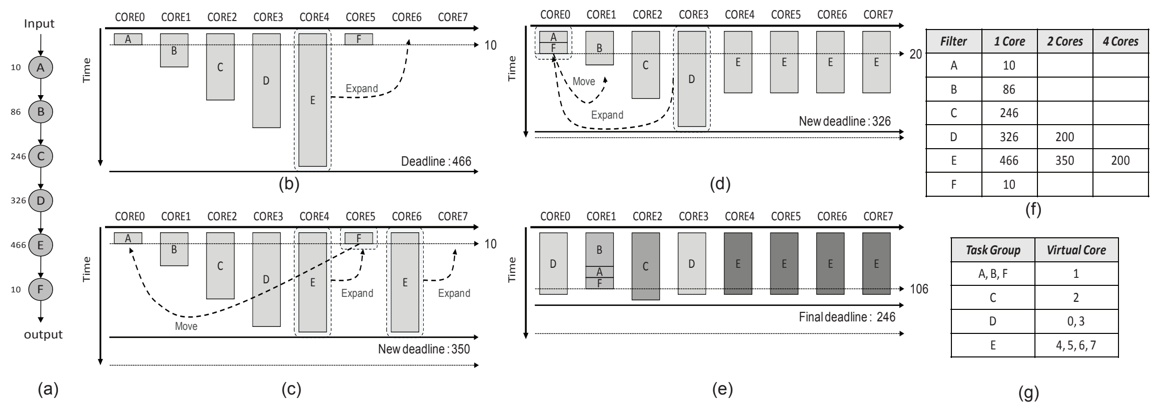
Algorithm 2 Prepass: Static Partitioning Algorithm

Input: *G:(V, E)*, *#virtualCores*, *balance*, *quality*

```
 1: groups ← PartitionGraph(G, #virtualCores);
 2: while true do
 3:     SortGroupsByExecTime(groups);

        { Find task groups with max and min execution time estimate. }
 4:     maxTaskGroup ← MaxExecTimeTaskGroup(groups);
 5:     numCores ← NumRequiredCoresToExpand(maxTaskGroup);
 6:     minTaskGroups ← FindContractTaskGroups(groups, numCores);

        { Generate candidate for new task groups. }
 7:     maxTaskGroupCand ← ExpandGroup(maxTaskGroup);
 8:     minTaskGroupCand ← ContractGroup(minTaskGroups);

        { Test the availability of the new task groups. }
 9:     if (ExecTime(maxTaskGroupCand) > ExecTime(maxTaskGroup) ∗ quality || ExecTime(maxTaskGroupCand) <
        ExecTime(minTaskGroupCand)) then
10:         Finish;
11:     end if

        { Update task groups.}
12:     Remove(maxTaskGroup, minTaskGroups);
13:     Add(maxTaskGroupCand, minTaskGroupCand);

14:     if (ExecTime(maxTaskGroupCand) < ExecTime(minTaskGroupCand) ∗ balance || timeOut) then
15:         Finish;
16:     end if
17: end while
```

is introduced to define the minimum performance gain that must be achieved to justify the assignment of additional cores.

Algorithm 2 starts from assigning one core to each task (Line 1). If the number of tasks is larger than the number of cores, tasks are grouped by the total execution time estimate(`ExecTime`). Based on this initial assignment of one core to each task group, the while loop in Algorithm 2 identifies the optimal number of cores per task group. Line 3-6 finds the task groups with the maximum ExecTime(*maxTaskGroup*), and minimum ExecTime(*minTaskGroups*). *maxTaskGroup* is the candidate for receiving more cores to enable faster execution while *minTaskGroups* will potentially lose cores. The number of task groups in *minTaskGroups* varies because number of additional cores, for *maxTaskGroup* to be the larger fused core, are set by the current assigned core topology of *maxTaskGroup* (Line 5) and the minimum ExecTime task group may not have enough number of cores to give. In this case, an additional second minimum ExecTime task group is required. If current *maxTaskGroup* has 1 core with 1x1 configuration, just 1 more core is required to be 1x2 or 2x1 array-style fused core. However, if current configuration of *maxTaskGroup* is 1x2 with 2 cores, 2 more cores are required to expand because 1x3 or 3x1 array-style core group is not allowed and next available core configuration is 2x2, 1x4, or 4x1 with 4 cores on current PPA. Moreover, an additional task group may be required to subsume the tasks from the minimum task group if the minimum workload group loses all its assigned cores. Then, line 7-8 creates the candidates of new maximum and minimum task groups given the new core assignments. `ExpandGroup` is the function for *maxTaskGroup* to get more cores to accelerate execution and `ContractGroup` is to take cores from *minTaskGroups*. Line 9-11 checks the benefit of these new resource assignments and determines whether



Filter	1 Core	2 Cores	4 Cores
A	10		
B	86		
C	246		
D	326	200	
E	466	350	200
F	10		

Task Group	Virtual Core
A, B, F	1
C	2
D	0, 3
E	4, 5, 6, 7

Figure 4.5: Static Partitioning example: (a) example data flow graph, (b) phase 0: each task is assigned to one core, (c) phase 1: the slowest task E gets one more core to accelerate, (d) phase 2: task E is still the slowest and gets two more cores(5, 7), thus task F loses own core(5), (e) phase 3: new slowest task D is accelerated as getting more core(0) and finally task C with one core(2) takes the maximum execution time, (f) execution time estimate table , (g) final core assignment: D has 2, E has 4 cores.

new combinations are updated. First, the new ExecTime estimate of *maxTaskGroupCand* should be less than some relative ratio of the original ExecTime(*example quality factor = 0.9*), meaning that the performance gain should be at least 10%. Also, the ExecTime estimates of the *minTaskGroupsCand* should not become a new bottleneck. Line 12-13 updates the changes to the core assignment and this process is repeated until the load imbalance is less than the balance factor or the task group combination does not change within the defined timeout period.

Figure 4.5 shows an example of the prepass static partitioning algorithm. An original task graph (Figure 4.5(a)) with 6 nodes is mapped onto a PPA with 8 cores. The original graph only has 6 nodes and each node is initially scheduled using 1 core. The annotated numbers show the ExecTime estimate for each node. The prepass algorithm performs

ExecTime estimation of the partitions then tries to appropriate more cores to the heavier workloads to balance the task groups. More specifically, node E is *maxTaskGroup* and gets 1 additional core because 2 cores are idle (Figure 4.5 (c) Phase 1). Then, node E is selected again as *maxTaskGroup* because the reduced ExecTime is still the highest at 350. In this case, an idle core and another core is selected to accelerate node E. As a result, node E is scheduled with 4 cores. Since node F lost all its assigned cores, it is merged into another task group with minimum ExecTime estimate, node A(Figure 4.5 (d) Phase 2). *maxTaskGroup* then becomes the task group with node D and is accelerated by taking one more core from nodes A and F. Again, nodes A and F lost all the cores and are merged into node B(Phase 3). At Figure 4.5 (c) Phase 3, the process is finished since it meets the balance condition (example balance factor 2.5) and 8 cores are divided as 4 task groups with different core numbers (Figure 4.5 (e)).

4.3.2 Core Allocation

After static partitioning, the number of PPA cores assigned to each task group is known, but their relative positions on the PPA array is not determined yet. Core allocation maps virtual PPA cores assigned to task groups onto the physical structure of the PPA. As discussed in Section 4.2.3, most composable accelerators, including PPA, provide limited interconnects. The fast scalar network connecting adjacent cores in PPA can be utilized to exploit fine-grain parallelism. So, cores assigned to the same task group are placed next to each other. Core allocation also attempts to place cores assigned to task groups with maximum ExecTime next to the cores with minimum ExecTime. This is to increase the opportunities for dynamic partitioning in postpass. With dynamic partitioning, idling resources can also be loaned to the neighboring task groups, further increasing the resource utilization. Algorithm 3 shows the process for core allocation. First, all the task groups are sorted by ExecTime estimates. In each attempt, the *maxTaskGroup* and the *minTaskGroups* are identified(lines 3 - 5) with Prepass-similar process, and they are placed closely on the PPA array to enable sharing cores at runtime(lines 6 and 7). Continuing the example from the prior section, Figure 4.6 shows the core allocation results and the slowest task group (C) is assigned the core next to the core reserved for the fastest task group (A, B, F).

Algorithm 3 Core Allocation: Physical Core Mapping

Input: *groups*, *#physicalCores*

Output: *phyTaskGroups*

```
 1: SortGroupsByExecTime(groups);
 2: while HasGroup(groups) do
 3:     maxTaskGroup ← MaxExecTimeTaskGroup(groups);
 4:     numCores ← NumRequiredCoresToExpand(maxTaskGroup);
 5:     minTaskGroups ← MinExecTimeTaskGroups(groups, numCores);

        { Assign physical cores.}
 6:     SetPhysicalCores(maxTaskGroup);
 7:     SetPhysicalCores(minTaskGroups);

        { Update task groups.}
 8:     Remove(maxTaskGroup, minTaskGroups, groups);
 9:     AddTo(maxTaskGroup, minTaskGroups, phyTaskGroups);
10: end while
```

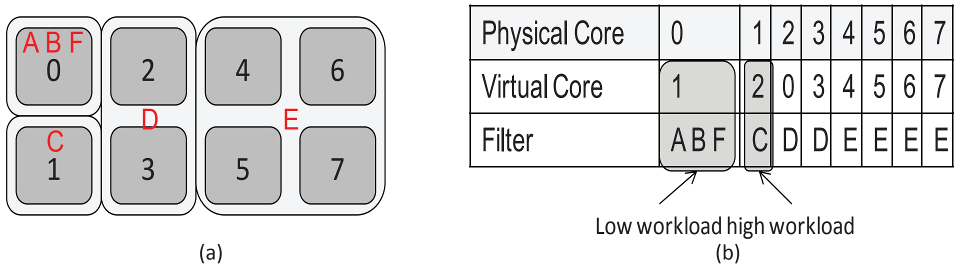


Figure 4.6: Core Allocation example: (a) physical placement of cores, (b) the slowest group is placed next to the fastest group.

4.3.3 Postpass: Dynamic Partitioning

In this section, we propose the final performance acceleration process: dynamically adjusting the resource assignment of the bottleneck task groups. The basic concept is to accelerate the slowest stage by dynamically acquiring the idle resources of neighboring cores at runtime. While the static partitioning achieves a good load balancing of PPA cores, workload variation still exists leaving some time slack for cores assigned to lightly loaded task groups. The idle time of cores can be exploited by neighboring cores using dynamic partitioning proposed in this section.

Algorithm 4 begins the optimization process by constructing the group adjacency information map (Line 1). The compiler automatically identifies which group is physically adjacent based on the PPA core connection information. Then, it identifies the slowest task groups and tries to find physically connected task groups. Among these task groups, the task group with the lowest ExecTime estimates is selected (Line 4-7). Line 8 calculates the performance estimate when dynamic partitioning is enabled between these groups. In this process, only tasks from the maximum ExecTime task group are allowed to execute with dynamically varying resources. The other task groups are restricted to their initial static

Algorithm 4 Postpass: Dynamic Partition Algorithm

Input: *phyTaskGroups, sharing*

```
1: adjMap ← ConstructAdjacentMap(phyTaskGroups);
2: while true do
3:     SortGroupsByExecTime(phyTaskGroups);
       { Find task groups with max and min execution time estimate. }
4:     maxTaskGroup ← MaxExecTimeTaskGroup(phyTaskGroups);
5:     nextMaxTaskGroup ← NextMaxExecTimeTaskGroup(nextMaxTaskGroup);
6:     numCores ← NumRequiredCoresToExpand(maxTaskGroup);
7:     minTaskGroups ← MinExecTimeAdjacentGroups(phyTaskGroups, numCores, adjMap);

       { Test the availability of dynamic partitioning of shared execution.}
8:     newMaxTaskGroupExecTime, newMinTaskGroupsExecTime
       ← EstimateExecTimeSharing(maxTaskGroup, minTaskGroups,
       sharing);
9:     if (newMaxTaskGroupExecTime < ExecTime(maxTaskGroup)
       && newMinTaskGroupsExecTIme < ExecTime(maxTaskGroup)) then
10:        UpdateSharing(maxTaskGroup, minTaskGroups, groups);
11:    end if

12:    if (newMaxTaskGroupExecTime > ExecTime(nextMaxTaskGroup)) then
13:        Finish;
14:    end if
15: end while
```

resource assignments. This is to limit dynamic resource assignment only to the performance limiting groups to minimize the reconfiguration overhead. The compiler identifies resource-constrained loop nests in the *maxTaskGroup* that can further exploit fine-grain parallelism with the extra resources. Then, the compiler gradually changes the resource assignment for the loop nests, until the ExecTime estimate of the *minTaskGroups* reaches a performance threshold. This threshold is set to the relative ExecTime of the second lim-

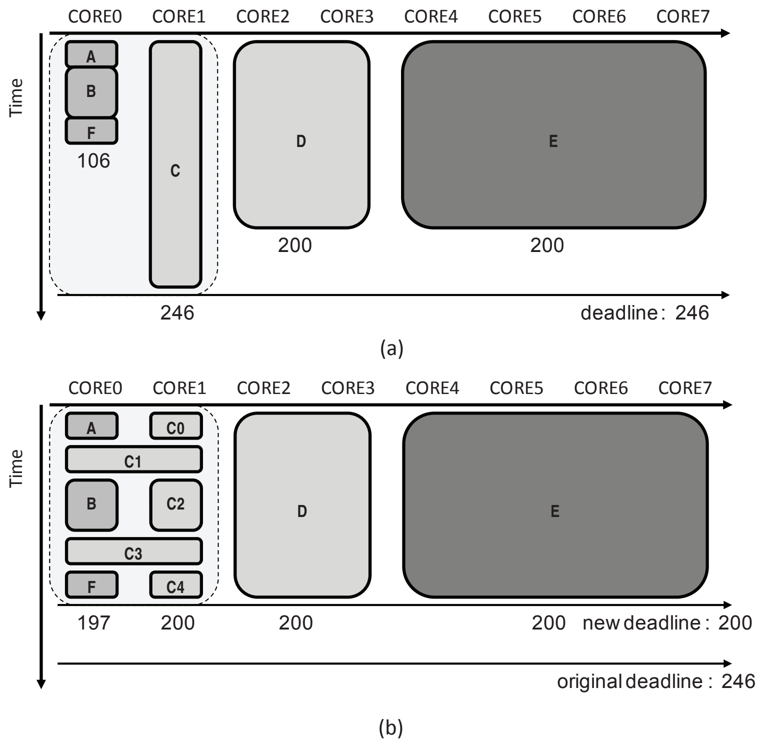


Figure 4.7: Dynamic Partitioning example: (a) coarse-grain pipeline using static partitioning, (b) coarse-grain pipeline with final performance tuning process

iting group (*nextMaxTaskGroup*). The *sharing* coefficient is introduced to determine the threshold and it depends on the application characteristics (dynamic variance) for each task at runtime. For example, a stage execution time of AAC fluctuates between 150k and 200k cycles [74], and the coefficient will be smaller than 0.75 considering dynamic overhead. Line 9-11 updates the new assignment if there is any performance gain with the resource sharing. This process will finish if the new ExecTime is still larger than the ExecTime of the *nextMaxTaskGroup*. Another key point of this process is that the *quality factor* is not considered in this phase because the objective of this process is to accelerate the pipeline limiting stage using marginal resources.

An example of the postpass optimization is shown in Figure 4.7. In this example, the slowest task group(C) and the fastest task groups (A, B, F) are placed next to each other after the core allocation step in Figure 4.7 (a). The compiler identifies five candidate loop nests in task group C, and two of them are rescheduled using the additional resources(cores 0 and 1). The final result in Figure 4.7(b) shows that the pipeline deadline decreases from 246 to 200 cycles, achieving 20% performance gain for this stage. The overall resource utilization is improved by recycling the wasted resources of core 0 between cycle 106 to 197.

4.4 Experimental Results

This section presents the results of the experimental evaluation of proposed high-level compilation techniques. We first present a brief explanation of the target architecture and benchmark applications. Performance measurement for prepass and postpass processes is explained based on the experimental environment described below.

4.4.1 Experimental Setup

Target Architecture PPAs are used to evaluate the performance of the compilation techniques. The PPA has 8 cores in the form of a 2×4 array as shown in Figure 4.1. Virtualization controller is disabled to evaluate the real performance of the compilation strategy. For the experiments using less than 8 cores, PPA is partitioned into two parts and the unused partition is disabled.

Target Applications and fine-grain parallelism To evaluate the performance, we used three application domains: audio decoding (aac), video decoding (h.264) and 3D graphics

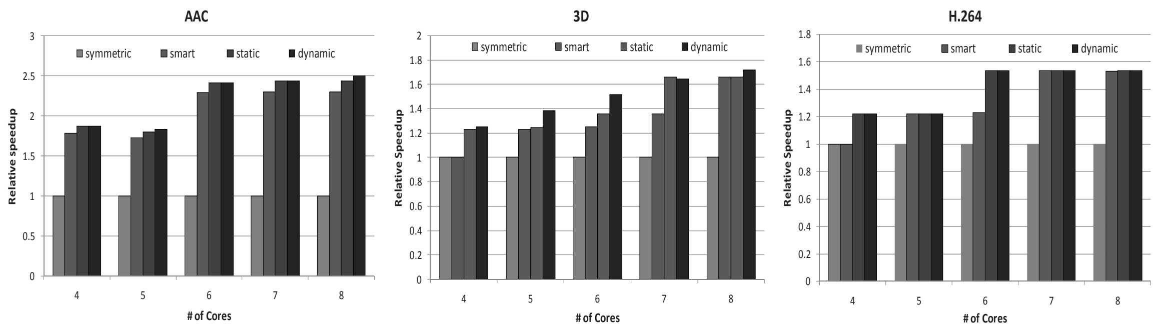


Figure 4.8: Relative speedup normalized to simple symmetric partitioning

(3d). All software-pipelineable loops from these applications are taken and scheduled using edge-centric modulo scheduling with all available partitions. Topology of the core groups are also considered. For example, 2x1 and 1x2 core groups with 2 cores are individually scheduled. Performance is evaluated using the overall execution time.

For coarse-grain pipelining, three applications are split into multiple tasks that communicate in a feed-forward fashion and without any inter-iteration dependencies contained within a single task. Each task is able to have both loops and acyclic blocks of code. Based on the control and data dependency restrictions, aac, 3D, h.264 have 10, 5, and 3 tasks on experiments.

4.4.2 Performance Evaluation

Figure 4.8 shows the relative speedup obtained by various partitioning algorithms on 4 to 8 cores. Symmetric partition means that each task is scheduled using the same number of cores. If the number of tasks is smaller than the number of cores, the cores are divided by the number of tasks and each task has its own partition. If the tasks are more than the cores, the overall application is split by the number of cores and each task group is executed using one core. Smart partitioning means manually divided static partition based on the application characteristics. For example, tasks containing substantial portion of loops are executed on a large core group to exploit fine-grain parallelism and the others are run on only one core. Static partitioning represents the execution result when the program runs on an automatically divided partition with prepass. In dynamic partitioning, the program executes on the same partition with static partitioning and dynamic reconfiguration is applied as well.

4.4.3 Static Partition

As shown in Figure 4.8, smart partitioning always outperforms symmetric partitioning by a significant amount because most of the loop-intensive task groups are accelerated using fine-grained pipelining. The promising point is that manual partitioning cannot achieve better throughput than our static partitioning algorithm, and the speedup of static partitioning on aac benchmark is always better than smart partitioning. As other benchmarks have small number of tasks, 3 and 5, manually partitioning with traditional load balancing algorithm can achieve the same speedup as with using the same partitioning with the result of prepass. However, if the application can be split into multiple subsets of tasks, our prepass optimization is able to minimize the performance degradation induced by low quality schedule, stall, and reconfiguration overhead. Note that tasks cannot be divided for perfect load balancing because memory and control dependences on the program prevent tasks from being partitioned from the middle. Despite these inherent difficulties, our algorithm successfully finds the throughput limiting tasks and accelerates them. On an 8-core

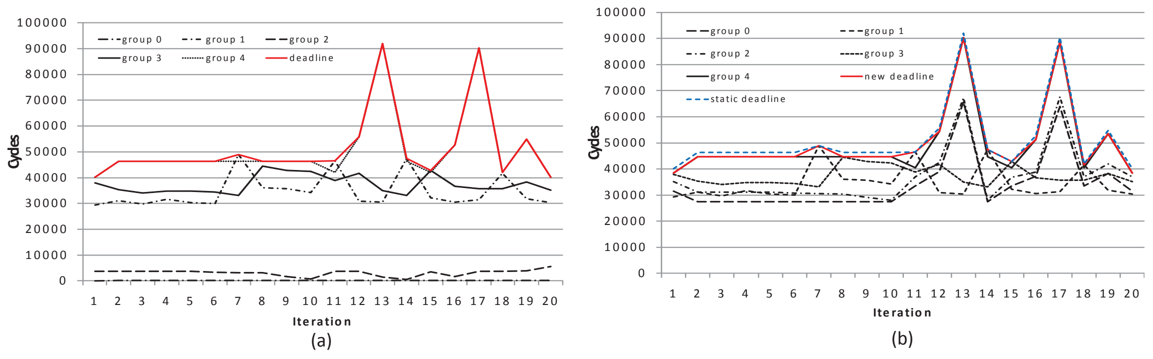


Figure 4.9: stage execution time for aac benchmark: (a) dynamic computation variance on static partitioning, (b) pipeline deadline reduction with dynamic partitioning

PPA, static partitioning allows 2.44x, 1.66x and 1.66x speedup over symmetric partitioning.

4.4.4 Dynamic Partition

AAC Figure 4.8 shows that postpass with dynamic partitioning is effective when the number of cores are 5 and 8 but the gain is small, 1.7% and 2.8%, respectively. This is because the task group with the largest execution time on AAC application consists of a large amount of sequential code and a small portion of the software-pipelineable code. In prepass, this huge sequential task cannot reserve enough cores because of the low quality schedule and remains the performance bottleneck. This task is then accelerated by sharing its neighbors' resources during postpass since it doesn't need to meet the quality factor any more, hence the final performance is slightly enhanced by using the neighbor's idle resource.

Runtime observations of the real execution on both static partitioning and dynamic partitioning are shown in Figure 4.9. Figure 4.9 (a) shows that task group 4 is the performance bottleneck over time and execution times of task group 0 and 2 are small. Core allocation

Cores	**Perf (smart)**	**Perf (static)**	**Perf (dyn)**	**Overall**
4	1.79	1.05	1	1.87
5	1.73	1.05	1.02	1.83
6	2.29	1.05	1	2.41
7	2.30	1.06	1	2.44
8	2.30	1.06	1.03	2.50

Table 4.1: Relative speedup for AAC benchmark (normalized to the preceding column).

process places the cores, assigned to these three task groups, next to each other and group 4 gets some performance gain as shown in Figure 4.9 (b). Despite the small performance gain of group 4, 0 and 2 have substantial runtime overhead because these groups should share the low quality schedule.

3D Rendering 3D rendering application has 5 tasks, two with small acyclic code and three with big software-pipelineable code. Dynamic partitioning increases the throughput by a large amount for all the cases because three huge tasks, which are easy to accelerate by fine-grain parallelism, have similar workload and quality of the schedule is still high when sharing the resources at runtime. The performance gain is up to 11.5% compared to static partitioning, just with reusing idle resources. Figure 4.10 shows how dynamic reconfiguration efficiently decreases the execution time of the slowest task group. On iteration 19-23, task 4 takes up to 60000 cycles to render 3D images and this work is finished in 50000 cycles by resource borrowing from task 0 and 1. After dynamic performance tuning, execution time on task 0 and 1 increases a large amount to help task 4 finish early on iteration 19-23.

H.264 For H.264 benchmark, dynamic reconfiguration is not enabled because execution

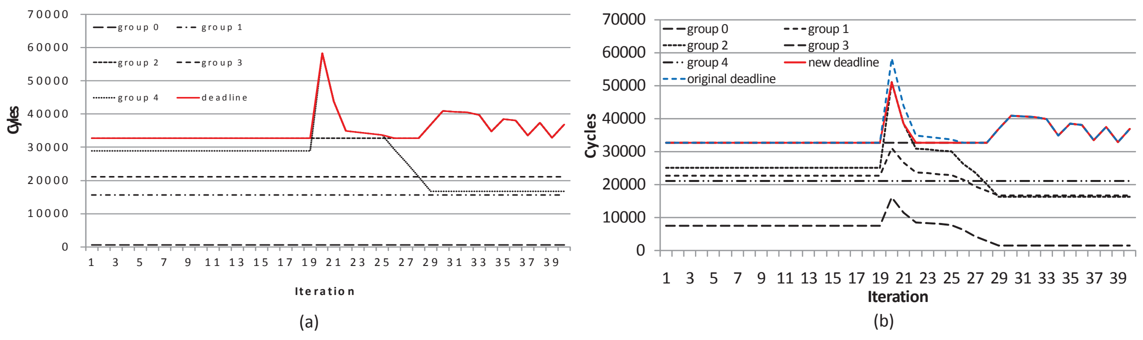


Figure 4.10: Stage execution time for 3D benchmark: (a) dynamic computation variance on static partitioning, (b) pipeline deadline reduction with dynamic partitioning

Cores	**Perf (smart)**	**Perf (static)**	**Perf (dyn)**	**Overall**
4	1	1.23	1.02	1.25
5	1.23	1.01	1.11	1.38
6	1.25	1.09	1.11	1.52
7	1.35	1.22	0.99	1.65
8	1.66	1	1.03	1.72

Table 4.2: Relative speedup for 3D benchmark (normalized to the preceding column).

time of the performance limiting task group fluctuates too widely and is sometimes even smaller than the fastest task group. Therefore, the compiler decides not to adapt dynamic partitioning because runtime overheads of the fastest stage are much bigger than the gains of the limiting task and the overheads may adversely affect the final performance as the fastest task becomes the slowest. Figure 4.11 shows that execution time changes by a huge amount and sometimes is even lower than the fastest task. In this case, the compiler does not allow dynamic reuse of the neighbor resources because adopting dynamic partitioning is optional.

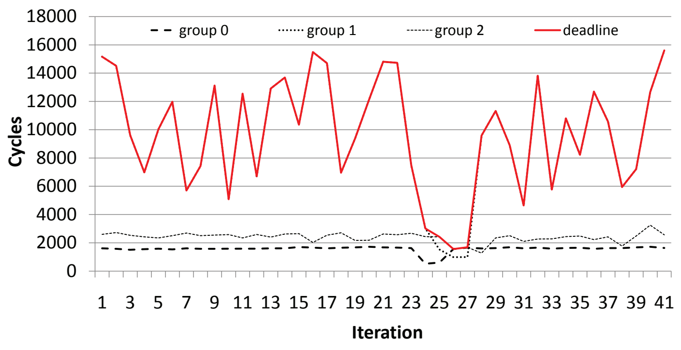


Figure 4.11: Stage execution time for H.264 benchmark: dynamic partitioning is not applied due to huge dynamic variance.

Cores	**Perf (smart)**	**Perf (static)**	**Perf (dyn)**	**Overall**
4	1	1.22	1	1.22
5	1.22	1	1	1.22
6	1.23	1.25	1	1.54
7	1.54	1	1	1.54
8	1.53	1.08	1	1.66

Table 4.3: Relative speedup for H.264 benchmark (normalized to the preceding column).

4.5 Related Work

Architectures: Combining cores to create a bigger logical core is relatively a new technique, recently proposed by Core fusion [42] and Composable Lightweight Processors [44]. Core Fusion is a CMP architecture that can dynamically allocate independent cores together for a single thread execution maintaining ISA compatibility. CLPs also allows dynamic allocation of cores to form a larger and powerful single-threaded processors. It also keeps the binary compatibility for the special EDGE ISA. The major difference between [42] and [44] is the target environment. PPA is designed to exploit single thread performance in mobile environments where power consumption and hardware cost is a first-class constraint. The building blocks of PPA are simple in-order cores similar to clustered VLIW processors [91]. Also, the statically controlled point-to-point interconnect provides a fast inter-core communication, allowing PPA to exploit fine grain pipeline parallelism efficiently for multimedia applications.

The PE level view of PPA is similar to Coarse-Grained Reconfigurable Architectures. ADRES [65] is a reconfigurable architecture where PEs are connected to a mesh-style interconnect. Modulo scheduling using simulated annealing is employed to exploit fine grain pipeline parallelism of nested loops. The top row in the array behaves as a VLIW processor with a multi-ported central register file. However, the non software pipelineable region of the application can only utilize the VLIW part of the array. So, it cannot pipeline the application in a coarser granularity as PPA. With identical resources, PPA outperforms our best approximation of ADRES by 1.43x. PipeRench [34] is a 1-D architecture in which processing elements are arranged in stripes to facilitate pipelining, but it has a fixed configuration of resource partitioning for pipelining while PPA can partition the array differently as to the characteristics of the target application. RaPiD [31] is another CGRA that consists of heterogeneous elements (ALUs and registers) in a 1-D layout, connected by a reconfigurable interconnection network.

Exploiting Parallelism: Exploiting coarse-grained pipeline parallelism is one of the most attractive approaches to accelerate single thread performance as multicore architectures enter the mainstream. Even this type of parallelism has many advantages compared to other types of parallelism, adapting in real situation is difficult because of program-inherent data dependences [86]. To overcome this difficulty, [86] has proposed a dynamic analysis tool to extract a stream graph from legacy C code in order to give a programmer hints for manual parallelization. [86] also tries load balancing by changing a program but this work's focus is more on compile time optimization for given program. [36] and [50] are similar to this work to exploit coarse-grained pipeline parallelism but the parallelization mechanism is limited only to stateless components as using StreamIt language. Our work also considers composable architecture specific features such as resource conflict and reconfiguration overhead whereas these works targeted fixed multi-core solutions(RAW architectures [57] and Cell processors [40]). Resource borrowing on dynamic partition is a similar concept to Work stealing [18] but our approach is performed in more fine-grained level, not thread level.

4.6 Summary

The popularity of mobile computing platforms has led to the development of feature packed devices that support a wide range of software applications, ranging from high-definition audio and video to high-end 3D graphics. However, the variable resource requirements and complex data/control flow of these workloads limit the applicability of traditional acceleration techniques. In response, this work proposes a novel, efficient compilation framework to enhance the throughput by maximizing resource utilization of a composable accelerator called a polymorphic pipeline array. The compilation consists of three phases: static partitioning into task groups, physical core allocation, and dynamic partitioning to reclaim idle resources to accelerate performance bottlenecks. The experimental results show that static partitioning achieves up to 2.44x speedup, with dynamic partitioning achieving even greater success in certain benchmarks.

CHAPTER 5

Efficient Performance Scaling of Future CGRAs for Mobile Applications

5.1 Introduction

The embedded systems that power today's mobile devices demand both high performance and energy efficiency in order to support the various applications, such as audio and video decoding, 3D graphics, and signal processing. Traditionally, application-specific hardware in the form of ASICs is used on the compute-intensive kernels to simultaneously meet tight performance/energy requirements. However, the increasing convergence of different functionalities combined with high non-recurring costs involved in designing ASICs have pushed designers towards more flexible solutions that are post-programmable. Coarse-grained reconfigurable architectures (CGRAs) are becoming attractive alternatives because they offer large raw computation capabilities with low cost/energy implementations [61, 85, 65]. Example CGRA systems that target wireless signal processing and multimedia are ADRES [66], MorphoSys [61], and Silicon Hive [78].

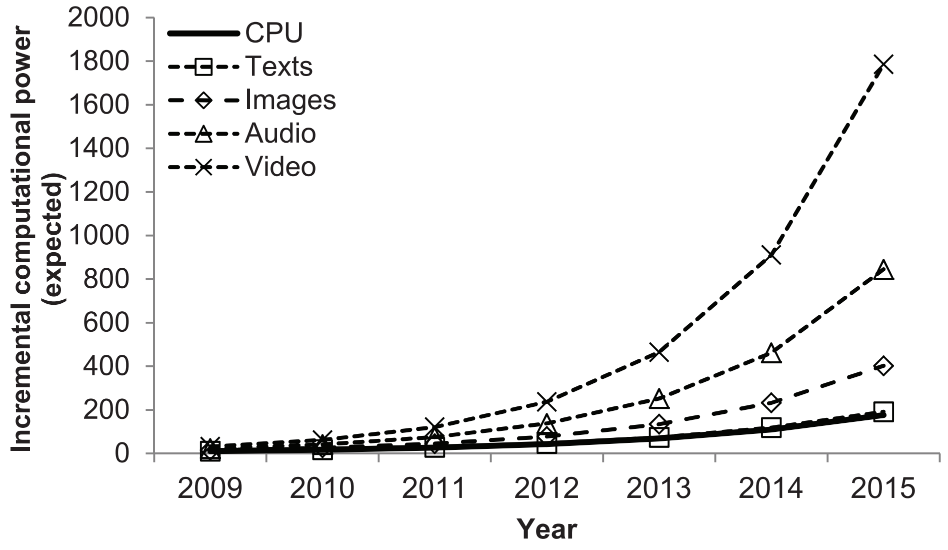


Figure 5.1: The computational power trends for social sites in each resource type:texts, images, audio, video, and CPUs.

CGRAs generally consist of an array of a large number of function units (FUs) interconnected by a mesh style network, as shown in Chapter 3.2.1.1. Register files are distributed throughout the CGRA to hold temporary values and are accessible only by a small subset of the FUs. The FUs can execute common integer operations, including addition, subtraction, and multiplication. CGRA resources are fully managed in software to maintain high energy efficiency. In contrast to FPGAs, CGRAs sacrifice gate-level reconfigurability to achieve hardware efficiency. Thus, CGRAs have short reconfiguration time, low delay characteristics, and low power consumption.

Even though CGRAs can meet the performance requirements of many of today's applications, future computational demands of mobile applications are predicted to increase exponentially [22]. Figure 5.1 depicts the trends in computational requirements for several media processing domains (text, image, audio and video) along with the projected performance gains of CPUs based on technology scaling based on data from [22]. This projection shows clearly that hardware scaling alone will be quickly out distanced by the performance requirements of all these domains. Further, simple hardware replication will not solve this problem as the power budgets for mobile devices are not increasing at a fast rate.

Previous works on CGRAs show that considerable performance improvements are possible by applying various techniques such as exploiting multiple types of parallelism [74, 45] or generating complex processing elements (PEs) [13]. However, these only consider features in isolation and fail to consider other issues including the topology and memory subsystem.

In this chapter, we perform a deep study to help the engineers design future CGRAs to meet future computation requirements while maintaining a tight power budget. We consider the following four key questions for scaling the performance of CGRAs:

1. How effective is heterogeneous functionality at increasing efficiency?
2. For the same number of processing elements (PEs), what are efficient interconnection topologies?
3. For power efficiency, can a complex PE be helpful compared to a simple PE?
4. For the memory interface, how useful is the introduction of vector memory operation support?

This work does not propose the best optimized CGRAs or new features. The goal of this work is to investigate these factors and their feasibility in the view of performance and power efficiency. We consequently place emphasis on finding the potential for architectural features and CGRA organization. For the first question, we show that heterogeneous FUs are indeed effective at reducing area and power at a small loss of performance. Second, we demonstrate that recent fixed multi-core solutions are often restricted by the application characteristics and a flexible solution with an advanced compilation technique is required. Third, we investigate whether complex PEs are indeed energy efficient. We show that CGRAs with complex PEs can improve performance with small additional energy consumption. Lastly, we examine the effect of vector memory operation support and conclude that it is helpful due to the high degrees of spatial locality found in media and gaming applications.

This chapter is organized as follows. Section 5.2 provides the background information on CGRAs, target applications, and simulation tool-chain. Section 5.3 presents the experimental methodologies, results, and discussions on four considerations. Section 5.4 concludes this chapter.

5.2 Analysis Infrastructure

This section introduces target benchmarks and the analysis infrastructure. ADRES [66] is used for the baseline CGRA accelerator as introduced in Chapter 3.2.1.1.

5.2.1 Benchmarks Overview

Two major classes of mobile benchmarks are used for this application analysis. The benchmarks consist of:

- Media benchmark: Three key mobile media applications are selected: AAC decoder (MPEG4 audio decoding, low complexity profile), H.264 decoder (MPEG4 video decoding, baseline profile, qcif) [43], and 3D (3D graphics rendering) [3]. These benchmarks are optimized for DSPs in the production-quality level and a large portion of the loops have a high potential degree of ILP and are software pipelinable.

- Game physics benchmark: Three common kernels are extracted from mobile gaming applications [2]. First, lineOfSight plays an important role of separating visible objects and non-visible objects. Sound effects, collision detection and other functions involving linear equations often exploit convolution and the conjugate gradient method. The three kernels mostly consist of high DLP loops.

5.2.1.1 Loop Characterization

Applications typically have many compute intensive kernels that are in the form of nested loops. Among these kernels, we analyze the available ILP and DLP of the innermost loops and find the maximum natural vector width which is achievable. To extract maximum degree of ILP, we found the *Software pipelinable* innermost loops to which modulo scheduling can be applied: 1) counted loop, 2) no subroutine call, and 3) no multiple exits/backedges. Control flows inside the innermost loops are solved by the if-conversion compiler technique. Among the software pipelinable (SWPable) innermost loops, we also identify the *SIMDizable* innermost loops which can utilize DLP. Based on the Intel Compiler [41], the rules to be selected as a SIMDizable innermost loop are as follows:

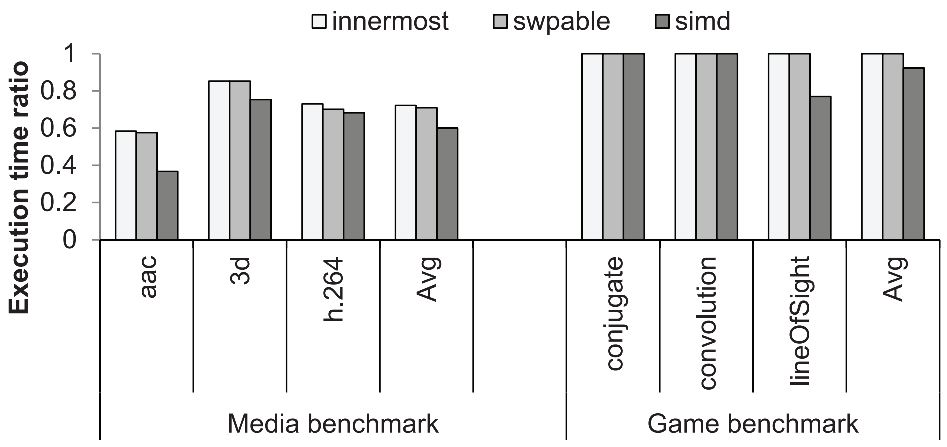


Figure 5.2: Loop categorization of various benchmarks: The three bars indicate ratio of execution time in innermost loops, SWPable loops, and SIMDizable loops.

- The loop must contain straight-line code. No jumps or branches, but predicated assignments, are allowed only when the performance degradation is ignorable.
- The loop must be countable and there must be no data-dependent exit conditions.
- Backward loop-carried dependencies are not allowed.
- All memory transfers must have same strides over iteration.

If a loop satisfies the above four conditions, the minimum iteration count is set to the maximum available SIMD width.

Figure 5.2 shows relative execution time of innermost loops, SWPable loops, and SIMDizable loops to total execution time on a simple 1-issue ARM processor. On average, there is a substantial amount of time spent on either or both SWPable and SIMDizable loops. More specifically, the media benchmark is originally optimized to maximize the portion of SWPable loops, but it also has high ratio of SIMDizable loops. The gaming physics benchmarks have higher levels of data parallelism. Results in Figure 5.2 confirm that not only different applications have different characteristics, but also different innermost loops in a single application can have different characteristics. In addition to this, we can have another opportunity to improve the overall performance if we have additional mechanism to support DLP.

5.2.2 Experimental Setup

Target Applications As discussed in Section 5.2.1, the evaluation is conducted for subsets of two domains. The top 10 loops having higher execution time are selected for gaming benchmark, and 144 loop kernels, varying in size from 4 to 142 operations, are extracted from the media benchmark because ratio of total execution time of top 10 loops is too small.

Compilation and Simulation The IMPACT compiler [71] is used as the frontend compiler. Edge-centric modulo scheduling (EMS) [73]-based modulo scheduler is implemented in the backend compiler on the ADRES [66] framework.

Power/Area Measurements Various CGRA templates are generated in RTL Verilog, synthesized with the Synopsys design compiler, and place-and-routed with the Cadence Encounter using IBM 65nm standard cell library in typical operating conditions with 1.0 operating voltage. Synopsys PrimeTime PX is used to measure power consumption. The Artisan Memory Compiler and RF Compiler are used to determine the power of memory operation using a 1.2 operating voltage. The target frequencies of the systems are 200MHz.

5.3 Analysis

In this section, we describe the key issues on scaling CGRAs, then set up the methodology in order to collect meaningful results for each factor. Finally, we analyze the experimental results and suggest several recommendations for the factors.

5.3.1 Question 1: Heterogeneity vs. Homogeneity

5.3.1.1 Overview

In common CGRAs, the use of heterogeneous FUs (mix of simple integer FUs and complex FUs) is considered as an apparent architectural choice since complex functionality such as multiply and divide operations requires high area and static power overhead but the utilization of them is often disproportionally lower than simple integer operations. For example, only 2.2% and 1.3% of the total dynamic instructions are multiplications and divisions in the H.264 video decoding application [11]. However, most architectural exploration on CGRAs has been focused on the interconnect topology and the array size [20, 51]. In this section, we examine the performance effect of heterogeneous FUs over homogeneous FUs.

5.3.1.2 Methodology

Based on the 16-PE homogeneous baseline CGRA (Section **??**), we decrease the number of FUs supporting whole functionalities. In the baseline CGRA, all FUs support all the functionalities: simple integer operations, complex operations (multiply, divide), and memory operations. Then we decrease the total number of some major functionalities.

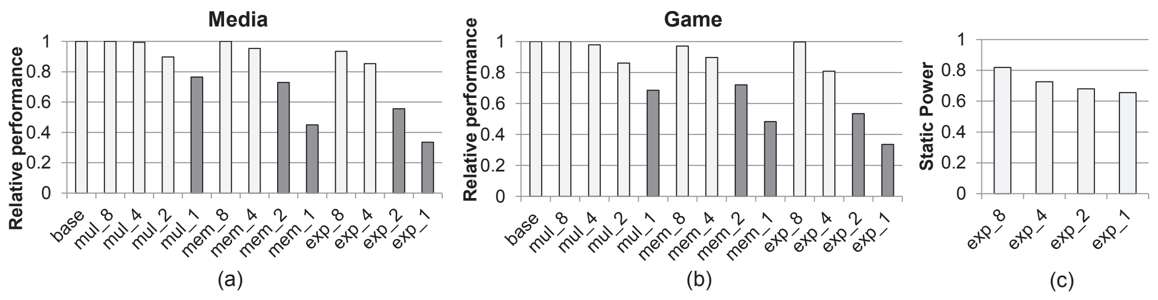


Figure 5.3: Performance degradation and static power consumption on a CGRA at different FU organizations.

First, we limit the number of FUs supporting complex operations from 8 to 1 (mul_N): only a subset of all 16 FUs supports complex operations and all FUs support all other operations. Second, we also limit the number of memory operations (mem_N). Lastly, we limit the number of FUs that supports both complex and memory operations (exp_N). For these architectures, the total execution time is used as a metric.

5.3.1.3 Result and Discussion

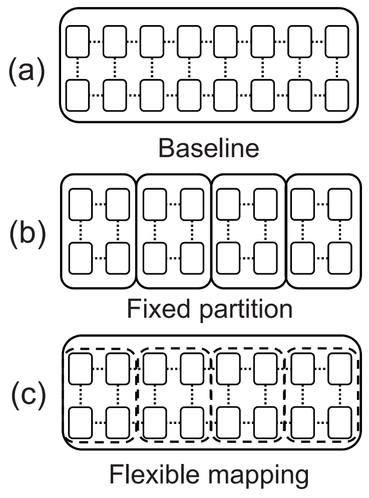
Figure 5.3 illustrates the performance degradation as the number of expensive units decrease on a 16-PE CGRA accelerator. Each bar shows the relative performance normalized to that of the homogeneous baseline CGRA. From this graph, the amounts of performance degradation are not as substantial as the area/static power benefits when reducing expensive units in both benchmarks. This is because the performance is normally constrained not by the expensive operations but by the simple integer instructions. Among complex and memory operations, the performance degradation depends much more on memory operations. If we set 80% of the baseline performance as the minimum performance target, we can decrease the number of both complex and memory units by up to 75% with high area/power benefits.

5.3.2 Question 2: Interconnection Topology

5.3.2.1 Overview

To enhance the overall performance, increasing total number of PEs is the simplest method to use. However, the key problem is the utilization of the PEs. As discussed in PPA [74], the performance saturates at some point if we simply increase the size of the CGRA due to the routing overhead and the lack of enough number of instructions inside the loopbody. The routing overhead is more critical because CGRAs do not provide a multi-ported, centralized register file and the operands must be explicitly routed using decentralized resources, often PEs. The number of instructions inside the loopbody can be increased by loop unrolling, but it will be also limited with increasing routing overhead.

Clustering is the common interconnection topology for the performance saturation problem [6, 58]. A large number of PEs are split into smaller partitions and each subset of PEs works separately. In this system, loops are scheduled targeting one partition (cluster) and executed in multiple partitions, where iteration counts are divided by the number of partitions. An interesting question at this point is how to find the optimal number of partitions and PEs inside each partition. In this section, we examine various types of interconnection topologies, including clustering, and map media and gaming benchmarks on CGRAs. We then introduce a reasonable strategy for scaling performance.



(d)

	DLP loop	Non-DLP loop
Baseline	Schedule on all the PEs Execute on all the PEs	Schedule on all the PEs Execute on all the PEs
Fixed partition (M x L)	Schedule on one partition Execute on M partition	Schedule on one partition Execute on one partition
Flexible mapping	Schedule on one partition Execute on M partition (M can vary)	Schedule on all the PEs Execute on all the PEs

Figure 5.4: Various interconnection topologies of CGRAs: (a) baseline, (b) fixed partition, (c) flexible partition, and (d) a table for execution model of loops on different topologies.

5.3.2.2 Methodology

To assess the impact of clustering as the size increases, we took all the SWPable loops in media and gaming benchmarks. Three different styles of CGRA architectures are implemented for design space exploration. Each style of architecture also has six variations of PE number: 4, 8, 16, 32, 64, and 128. The detailed explanation of the architecture styles is as follows:

- **N**: Baseline architecture (Figure 5.4(a)). The architecture consists of all the PEs, and the structure is the same as the architecture explained in Section **??**. As shown in Figure 5.4(d), both DLP and non-DLP loops are scheduled targeting whole PEs.

- **MxL**: Fixed partition (Figure 5.4(b)). N PEs are physically split into M partitions($2 \leq M \leq 8$), then L (N/M) PEs consist of each partition. Both kinds of loops are scheduled targeting one partition. Non-DLP loops are executed in one partition due to the inter-iteration dependencies, and DLP loops are executed in M partitions and each iteration count is divided by M (Figure 5.4(d)).

- **N_flex**: Flexible partition (Figure 5.4(c)). Based on a baseline architecture, the number of partitions can be dynamically changed from 1 to 8. Therefore, non-DLP loops are scheduled targeting whole PEs and executed on whole PEs. For DLP loops, the schedule of each loop is generated targeting the best partition and executed in parallel on each partition for smaller iteration counts (divided by the number of partitions).

To determine the effects of differing architectural features, the measurements of performance and the performance saturation point distribution of loops were obtained.

5.3.2.3 Result and Discussion

Figure 5.5 shows the performance results of above architecture types as the CGRA size increases. The X-axis on these graphs shows the architecture templates, and the Y-axis shows the average performance of media and gaming applications. Each performance result is normalized to when each application is mapped onto the 4-PE baseline architecture. Here, we can notice that the throughput saturates as we increase the size of the baseline architecture. For media and gaming benchmarks, the performance does not increase that much beyond the size of 32 PEs and 16 PEs, respectively. This is because the average size of innermost loops on gaming benchmarks is smaller than that on media benchmarks.

For fixed partition, the performance is often worse than the corresponding size baseline architecture on small sizes, but it scales well on large sizes. For media benchmarks, a high number of partitions does not always show the best performance among various same size

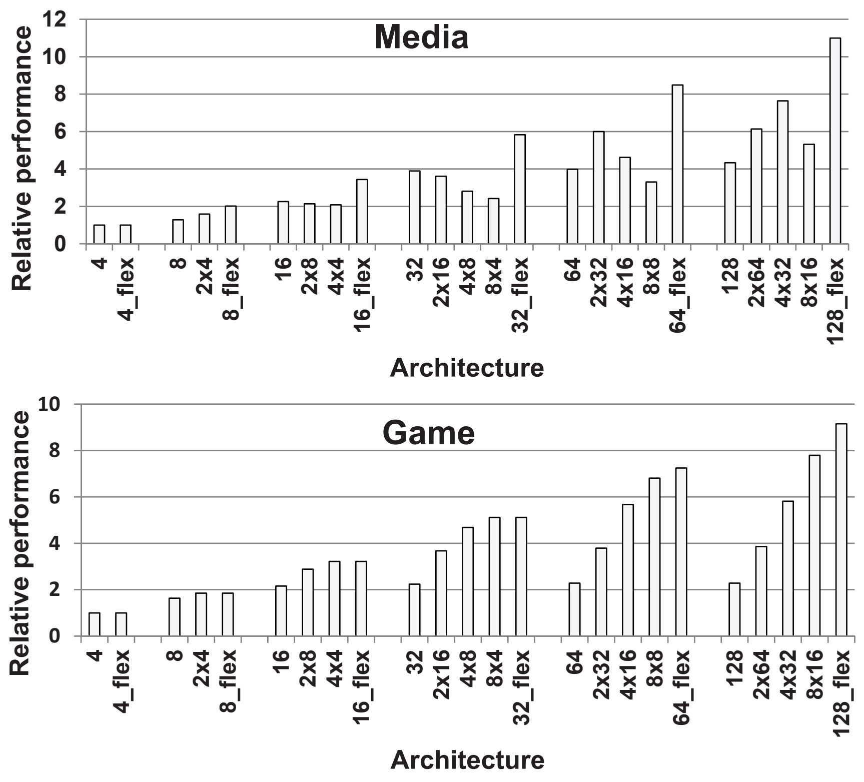


Figure 5.5: Performance comparison of various architectures for media and gaming benchmarks.

architectures because the degree of DLP is not high for DLP loops and the performance of non-DLP loops is higher on larger partition size. Different from media, gaming benchmarks always show the best performance on the highest number of partitions. This is because most of the loops are small data-parallel loops with high iteration counts. Figure 5.6 explains this difference well. Two pie charts in Figure 5.6 show loop distribution at different saturation points for two domain benchmarks. From this figure, we can see that high portion of loops in media benchmarks needs more than 32 PEs for full acceleration, hence the performance is often limited by the small size of a partition. Conversely, more partitions are much helpful for performance improvement on gaming benchmarks as most of the loops have the

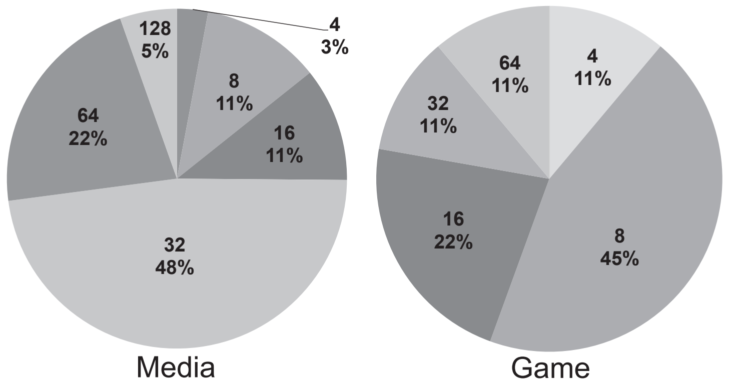


Figure 5.6: Performance saturation point distribution at different PE sizes for media and gaming benchmarks: media benchmarks need relatively high number of PEs to be sufficiently accelerated but gaming benchmarks need small number of PEs.

small saturation points less than 16.

Though fixed partitioning shows decent performance gain, it is hard to say that the application is fully accelerated. This is because the best structure highly varies over loops inside a a benchmark and also across multiple benchmarks. Therefore, we also test a unified architecture to support flexible mapping (n_flex). As shown in Figure 5.5, the flexible architecture always shows the best performance and retains scalability even in large size as all the loops can be executed on the best partition guided by the results on Figure 5.6.

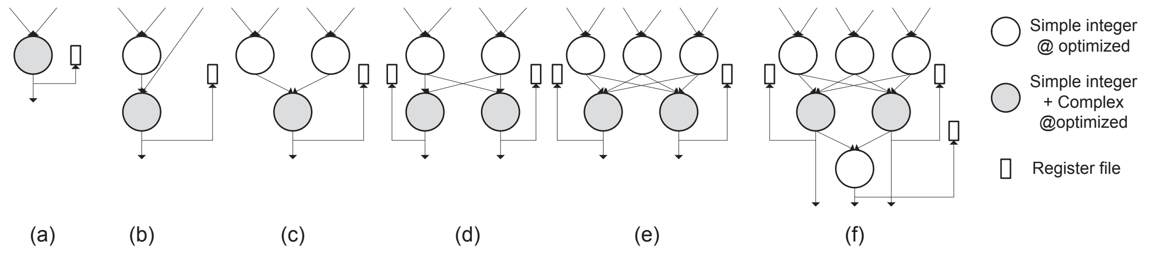
These results reveal the difficulty of performance scaling with common solutions in the real world. To further improve the single threaded performance, it is necessary to find a mechanism to flexibly change the partition adaptive to the loop characteristics. The flexible mapping without physical array partitioning will also be highly favorable to other research for improving the multi-threaded performance such as PPA [74] and MT-ADRES [6], while our flexible partitioning scheme is completely orthogonal to multi-threading of CGRAs.

5.3.3 Question 3: Complex PEs vs. Simple PEs

5.3.3.1 Overview

Interconnection topology has been a primary consideration for scaling CGRAs because most CGRAs consist of multiple simple PEs, which include one FU and one RF. Recently, CGRAs with more complex PEs, consisting of multiple FUs and RFs, are also introduced in order to improve performance [13, 14, 12]. Construction of CGRAs with complex PEs has several key advantages over conventional CGRAs. First, sparse interconnection between PEs provides better cost and energy scalability with minimum performance loss due to the dense interconnection inside PEs. Second, the number of RFs can decrease as mapping multiple instructions inside a PE can reduce RF accesses by directly consuming temporary values generated inside a PE. Third, back-to-back instructions can be chained without pipeline registers, hence execution can be faster. Lastly, heterogeneity inside PEs can be implemented while retaining PE-level homogeneity.

Despite these advantages, adopting complex PE scheme is still questionable because it is hard to attain full utilization of resources inside the PEs. In this section, we focus on the energy consumption instead of resource utilization. We investigate whether complex-PE based CGRAs can consume less or comparable energy, then show that the energy overhead is not critical in some cases. We believe that this evaluation will help developers consider complex PE based design as one of possible options.



Simple integer @ optimized
Simple integer + Complex @optimized
Register file
(a)
(b)
(c)
(d)
(e)
(f)

Figure 5.7: PE designs with different number of FUs: the number of RFs is the same as the number of output ports and only shaded FUs support all instructions in optimized PEs.

5.3.3.2 Methodology

Figure 5.7 demonstrates the structure of complex PEs varying the number of FUs from one to six. The number of RFs depends on the number of output ports. For all the PE structures, two kinds of designs are considered: uniform and optimized. In a uniform PE, all the FUs support all the functionalities including both simple integer operations (add, sub, and logic) and complex operations (mul, div), while only shaded FUs support complex operations for an optimized PE.

To estimate the energy consumption on different PE styles, we map all the loops on to those PEs by taking the concept of subgraph identification [25, 26]. Briefly, the compiler generates the dataflow graph (DFG) of each loopbody, and discovers all the subgraphs (groups of instructions) which can be mapped onto the target PE. Each remaining node is regarded as a subgraph with one instruction.

Based on the above data, estimated energy consumption of a loop is calculated as $P_{active} \times N_{subgraph}$. P_{active} and $N_{subgraph}$ refer to the power consumption when a PE is active and the number of subgraphs, and inactive PEs are assumed to be dynamically power-gated.

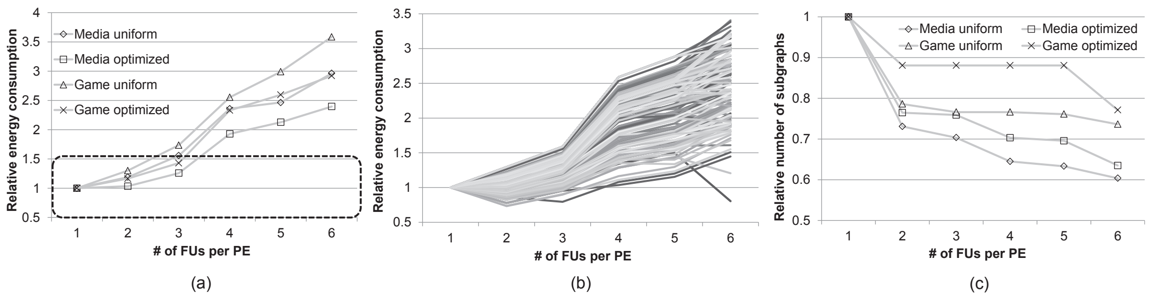


Figure 5.8: Experimental results on various PEs: (a) relative average energy consumption, (b) relative energy consumption of every loop, and (c) the number of subgraphs. All the FUs support full functionality on uniform PEs, and only a subset of FUs supports full functionality on optimized PEs.

5.3.3.3 Result and Discussion

The average energy consumption of loops on media and gaming benchmarks are shown in Figure 5.8(a). The target PEs are shown on the X-axis, and relative energy consumption normalized to the one-FU PE (Figure 5.7(a)) on the Y-axis. The following results are examined and shown as a line form: averages of energy consumptions of all loops included in media and gaming benchmarks targeting uniform PEs (Media uniform and Game uniform), and those targeting optimized PEs(Media optimized and Game optimized). Figure 5.8(b) shows the energy consumption of all loops on both benchmarks targeting only optimized PEs. Figure 5.8(c) shows the relative number of mapped subgraphs, and each line shows the average of relative numbers of subgraphs normalized to the one-FU PE.

From Figure 5.8(a), even though the utilization is always lower at more complex PEs, the energy increase is not as substantial as FU number increases. This is because the power consumption of each PE is not directly proportional to the number of FUs due to smaller number of RFs and pipeline registers. As shown in Figure 5.8(b), some loops consume less energy on 2- or 3-FU PE CGRAs by high resource utilization. For media benchmarks, complex PEs are well utilized as shown in Figure 5.8(c), and energy consumption can be highly saved when using optimized PE structure because the applications have low ratio of complex operations(Figure 5.8(a)). Conversely, execution of gaming benchmarks at complex PE architectures shows more relative energy consumption than media benchmarks because the number of subgraphs does not highly decrease for more complex PE architectures (Figure 5.8(a)). Moreover, the performance degradation from a uniform PE structure to a optimized PE structure is high because game applications have a high portion of complex operations such as multiplication/division but an optimized PE structure has smaller number of these FUs (Figure 5.8(c)).

The interesting point here is that we may allow some degree of energy overhead because of several reasons: 1) at same operating frequency, complex PE structure is faster than the one-FU PE structure, and 2) routing overhead can be reduced as the number of subgraphs decreases (Figure 5.8(c)). Therefore, if we decide that 50% energy overhead can be allowed, complex PEs with 2 and 3 FUs can also be considered as the proper solution in addition to the simple PE(Figure 5.8(a)).

5.3.4 Question 4: SIMD Memory Support

5.3.4.1 Overview

In addition to the previous consideration about the size of PEs, supporting SIMD memory operation by adding a vector unit into a PE is also introduced by some recent CGRAs. For example, ADRES system supports special intrinsic instructions that allow SIMD operations [64, 6]. Similar to Section 5.3.3, supporting SIMD memory operations on PEs has several noticeable advantages such as less fetching power and less number of instructions over simple scalar memory operations.

However, current designers often hesitate to add the SIMD capability into CGRAs due to the uncertainty of high potential degree of DLP. In this section, we investigate the frequency of spatial reuse of wide vector data on the mobile benchmarks, and then show that SIMD functionality is worthwhile to adopt in some range with slight overhead due to the domain specific characteristics.

Though there are several previous research about the memory structure and scheduling algorithm on CGRAs, most of the research focuses on the performance improvement on scalar memory-based system such as reducing memory conflicts on multi-bank scratchpad local memory [46]. We further examine the availability of SIMD memory-based system for high efficiency.

5.3.4.2 Methodology

To prove the effectiveness of SIMD memory support, we consider SIMD memory units from 1 to 16 vector length in the view of the energy consumption and the performance. For the energy consumption, we first get the memory reference footprints during sixteen iterations for each loop. Based on the footprints, we find the required number of vector instructions for each SIMD memory unit(N_{access}). We also measure the power consumption of the SRAM per memory access (P_{access}) from the datasheet generated by memory compiler. We then estimate the total energy consumption of memory accesses by $P_{access} \times N_{access}$.

Additionally, the performance effect of SIMD memory units is also examined. We measure the performance effect by substituting scalar memory units into SIMD memory units while keeping the same total bandwidth. For instance, when we set the total bandwidth as 4x32 bits, we test 16-PE CGRAs with four 32-bit scalar memory units (Figure 5.9(a)), two 2x32 vector memory units (Figure 5.9(b)), and one 4x32 vector memory unit (Figure 5.9(c)).

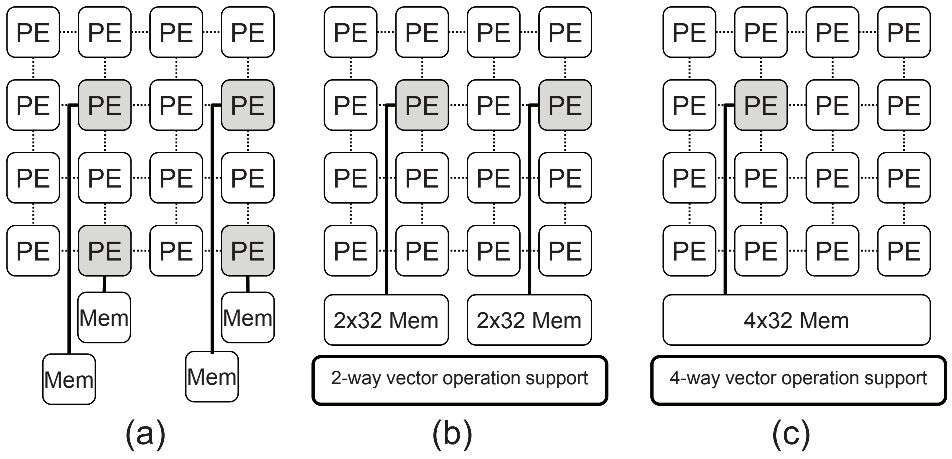

PE
Mem
2x32 Mem
2x32 Mem
2-way vector operation support
4x32 Mem
4-way vector operation support
(a)
(b)
(c)

Figure 5.9: Example CGRAs with different SIMD memory support: (a) four scalar memory support, (b) two 2x32 SIMD memory support, and (c) one 4x32 SIMD memory support.

For performance metric, we use the resource-constrained lower bound (ResMII) of memory resources: N_{access} (number of memory instructions) $/N_{Munit}$ (number of memory units). This is because the performance of a loop, which modulo scheduling is applied to, is generally determined by the initiation interval (II) when the number of iterations is large [73, 79]. The goal of the modulo scheduling is to minimize the II by MII, and therefore, if ResMII of memory resources is larger than MII of original architecture, the performance of the loop can be thought as to be affected.

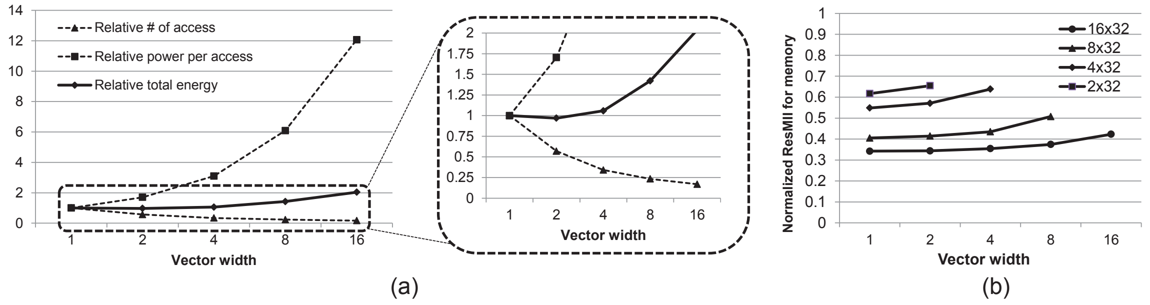


Figure 5.10: Experimental results with different vector widths: (a) relative energy consumption for total memory accesses, and (b) memory ResMII increase when using SIMD memory units with same total bandwidth.

5.3.4.3 Result and Discussion

Figure 5.10(a) shows the average energy consumption of loops over varying vector widths of memory units. X-axis shows the vector widths of memory units, the number of memory accesses, the power consumption per memory access, and the total energy consumption are shown as a line form, and these are normalized to the scalar memory unit (vector width = 1). In the left graph of Figure 5.10(a), though power consumption for one memory access highly increase at longer vector width, the total energy consumption maintains a similar level to that of a scalar memory unit by virtue of a high degree of spatial locality in memory accesses on mobile benchmarks. The enlarged graph on the right side shows that total energy consumption can be even lower than a scalar vector unit in the case of a 2-way vector unit. This is because most of loaded data are used without additional loads and the vector load consumes less power than multiple scalar loads.

The performance effect of using vector memory units is shown in Figure 5.10(b). The four lines indicate the average memory ResMII of all loops when changing the vector width while retaining same bandwidth. Each ResMII data is normalized to the MII targeting the 16-PE CGRA with scalar memory units. This graph shows the gradual growth of memory ResMII but they are always less than the actual MII, and therefore, the performance degradation does not exist.

These data show that adopting vector instructions is not as harmful as a common myth in the view of energy consumption and performance, hence developers should consider SIMD capability for designing a future mobile CGRA.

5.3.5 Summary and Insights

The analysis of these four considerations provides several insights. First, using heterogeneous FU organization is highly effective in reality and the ratio of expensive resources can be tuned by performance degradation. Second, even though the current fixed partitioning scheme is fairly effective over the baseline for performance scaling, the high variance of loops inside and across applications prevents it from further achieving the performance gain. Therefore, flexible partitioning should be supported by both architectural and compiler modifications. Third, a complex PE structure can be one of the attractive options for further improving performance because complex PE can be more energy efficient even in lower resource utilization. Lastly, the characteristics of mobile benchmarks can make the wide SIMD memory support from an aggressive solution into a realistic solution.

5.4 Summary

The mobile applications have been rapidly developed so that the future mobile devices need to provide high single-thread performance within limited power budget. CGRAs are known as one of the prominent solutions to achieve these needs, but the potential for the scalability of CGRAs are not thoroughly investigated yet. In this work, we perform a deep analysis on several key considerations when scaling: heterogeneity, interconnection topology, complexity of PEs, and SIMD memory support. The study shows us that CGRAs have high potential of performance improvement with high efficiency and some key factors, which are easy to overlook, should also be considered for designing CGRAs. We believe that these insights will be key advices for improving future applications (more DLP), compilers (support flexible mapping), and architectures (complex PEs and SIMD memory units).

CHAPTER 6

Libra: Tailoring SIMD Execution using Heterogeneous Hardware and Dynamic Configurability

6.1 Introduction

The mobile devices market, including cell phones, netbooks, and personal digital assistants, is one of the most highly competitive businesses. The computing platforms that go into these devices must provide ever increasing performance capabilities while maintaining low energy consumption in order to support advanced multimedia and signal processing applications. Application-specific integrated circuits (ASICs) are the most common solutions for meeting these requirements, performing the most compute-intensive kernels in a high performance but energy-efficient manner. However, several features push designers to a more flexible and programmable solution: supporting multiple applications or variations of applications, providing faster time-to-market, and enabling algorithmic changes after the hardware is constructed.

Processors that exploit instruction-level parallelism (ILP) provide the highest degree of computing flexibility. Modern smart phones employ a one GHz dual-issue superscalar ARM as an application processor. Higher performance digital signal processors are also available such as the 8-issue TI C6x. However, ILP processors have scalability limits including many-ported register files (RFs) and complex interconnects. Alternately, single-instruction multiple-data (SIMD) accelerators provide high efficiency because of their regular structure, ability to scale lanes, and low control logic overhead. They have long been used in the desktop space for high performance multimedia and graphics functionality. But, their combination of scalable performance, energy efficiency, and programmability make them ideal for mobile systems [80, 17, 59, 90].

In order to fully utilize the SIMD hardware, it is necessary for the programmer or compiler to extract sufficient data-level parallelism (DLP). Automatic loop vectorization is available in a variety of commercial compilers including offerings from Intel, IBM, and PGI. Classic scientific computing (regular structure, large trip count loops, and few data dependences) are naturally well-matched to SIMD accelerators. But, in many respects, the mobile terminal has become a general-purpose computer. Thus, like the desktop, only a small percentage of mobile applications look like classic scientific computing. The computation is not dominated by simple vectorizable loops, but by loops containing significant numbers of control and data dependences to handle the complexity of modern multimedia standards. As a result, applications have varying amounts of vector parallelism ranging from none to some to large amounts. The net effect is that SIMD hardware goes unused for a large fraction of application execution and thus cannot be counted on to provide significant performance gains.

A second but inter-related problem with SIMD computing is low hardware utilization even when vector loops are executed. The use of homogeneous hardware (e.g, identical lanes) is one of the best advantages of SIMD datapaths by reducing design cost and complexity. But, the utilization of the most complex components of a SIMD lane is often disproportionally lower than the simple components. For example, the H.264 video decoding application is dominated by simple integer operations (adds, subtracts, shifts) and an average of only 2.2% and 1.3% of the total dynamic instructions are multiplies and divides [11]. This is not an outlying data point, most multimedia and visual computing applications have small fractions of multiply, divide and other expensive operators. For 128-bit SIMD (4 lanes), such utilization rates may not matter, but as SIMD widths are scaled to increase performance to 1024 bits (32 lanes) or more, the problem becomes serious due to poor area utilization and high static power dissipation.

To attack these problems, we propose a customizable SIMD accelerator that is capable of tailoring its execution strategy to the running application, referred to as the *Libra*. Libra employs two key concepts, *heterogeneity* and *dynamic configurability*, to achieve broader applicability and better energy efficiency than traditional SIMD accelerators. Heterogeneity allows lanes to have different functionalities and better match functional capabilities with expected operator distributions. Dynamic configurability enables lane resource to execute as a traditional SIMD processor, be re-purposed to behave as a clustered VLIW processor, or combinations in between. Dynamic configurability also enables efficient sharing of expensive resources between lanes (e.g., multipliers) by interleaving independent instructions with each lane's expensive instruction so as to hide resource contention. Libra consists of an array of simple processing elements (PEs) that are tightly interconnected by a scalar operand network. Groups of four PEs form PE groups that are normally driven by a single instruction stream. Each group can behave as a building block for a SIMD processor (e.g., PEs behave as SIMD lanes) or a VLIW processor (e.g., PEs behave as a cluster of function units). The compiler maps 1 or more loops to the Libra accelerator by combining and configuring clusters of PE groups to efficiently exploit the available DLP and ILP.

This chapter offers the following three contributions:

- An in-depth analysis of the available ILP/DLP parallelism and its variability in three representative mobile application domains: computer vision applications, commercial media applications optimized in industry level, and game physics engine applications.

- The design of a unified loop accelerator that can effectively support future mobile applications with varying performance requirements and characteristics. To achieve this objective, we offer three key features:

    1. Scalability: Libra can meet high performance requirements by simply increasing the number of clusters, whereas most current accelerators suffer from poor scalability.

    2. Configurable performance: Libra can dynamically tune the ILP/DLP-support capability in order to successfully support ILP-intensive, DLP-intensive, and ILP/DLP-mixed applications, as well as tolerate performance degradation due to its heterogeneity.

    3. Energy efficiency: Simple hardware implementation achieves high energy-efficiency with competitive performance.

- A light-weight design and organization of a configurable processing element for supporting simple latency hiding techniques and sharing expensive resources.

6.2 Background and Motivation

In this section, we examine the limitations of traditional SIMD accelerators based on an analysis of various mobile applications. We first introduce the target benchmarks and the baseline architecture, and find two main sources of inefficiencies in SIMD accelerators. We then propose high-level solutions to overcome these challenges that facilitate designing efficient hardware and maximizing the utilization of existing resources.

6.2.1 Benchmarks Overview

Three classes of mobile benchmarks are used for this application analysis that contain varying degrees of vector parallelism. The benchmarks consist of:

- Vision benchmark: We evaluated a subset of the SD-VBS benchmark suite [88] for mobile vision applications. As these benchmarks are not originally optimized for a specific target architecture, we manually modified these benchmarks to increase the opportunities for efficient execution with function inlining and loop unrolling. All the benchmarks are functionally verified on QCIF[1] input data sizes, which is widely used on mobile devices.

[1]We used QCIF (176x144) image size for uniformity of benchmarks, and the similar trend appears on higher resolution images.

- Media benchmark: Three mobile media applications are selected: AAC decoder (MPEG4 audio decoding, low complexity profile), H.264 decoder (MPEG4 video decoding, baseline profile, qcif) [43], and 3D (3D graphics rendering) [3]. These benchmarks are optimized for DSPs in the production-quality level and a large portion of the loops have a high potential degree of ILP and are software pipelinable.

- Game physics benchmark: Three common kernels are extracted from mobile game applications [2]. First, lineOfSight plays an important role of separating visible objects and non-visible objects. Sound effects, collision detection and other functions involving linear equations often exploit convolution and the conjugate gradient method. The three kernels mostly consist of high DLP loops.

6.2.2 Baseline Architecture

A SIMD architecture that is based on SODA [59] is used as the baseline SIMD accelerator. This architecture has both SIMD and scalar datapaths. The SIMD pipeline consists of a multiple-lane datapath where each lane has an arithmetic unit working in parallel. Each datapath has two read-ports, one write-port, a 16 entry register file, and one ALU with a multiplier. The number of lanes in the SIMD pipeline can vary depending on the characteristics of the target applications. The SIMD Shuffle Network (SSN) is implemented to support intra-processor data movement. The scalar pipeline consists of one 32-bit datapath and supports the application's control code. The scalar pipeline also handles DMA (Direct Memory Access) transfers.

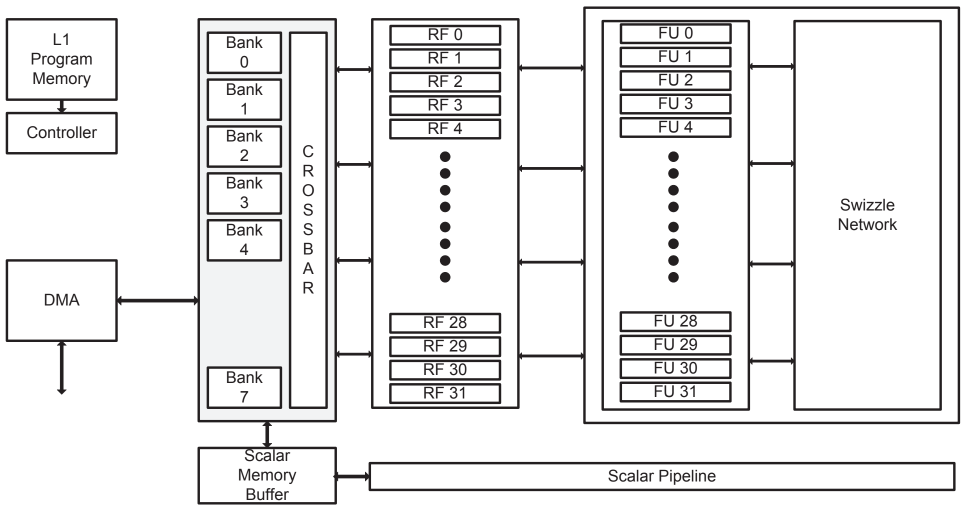


Figure 6.1: A traditional 32-lane SIMD accelerator.

6.2.3 Limitations for Current SIMD Accelerators

6.2.3.1 Loop Characterization

Applications typically have many compute intensive kernels that are in the form of nested loops. Among these kernels, we analyze the available ILP and DLP of the innermost loops and find the maximum natural vector width that is achievable. To extract the maximum degree of ILP, we found the *Software pipelinable* innermost loops: 1) counted loop, 2) no subroutine call, and 3) no multiple exits/backedges. Control flows inside the innermost loops are solved using if-conversion. Among the software pipelinable (SWPable) innermost loops, we also identify the *SIMDizable* innermost loops which can utilize DLP. We apply the conditions used by the Intel compiler [41] to determine if a loop is SIMDizable and the minimum iteration count is set to the maximum available SIMD width (natural SIMD width).

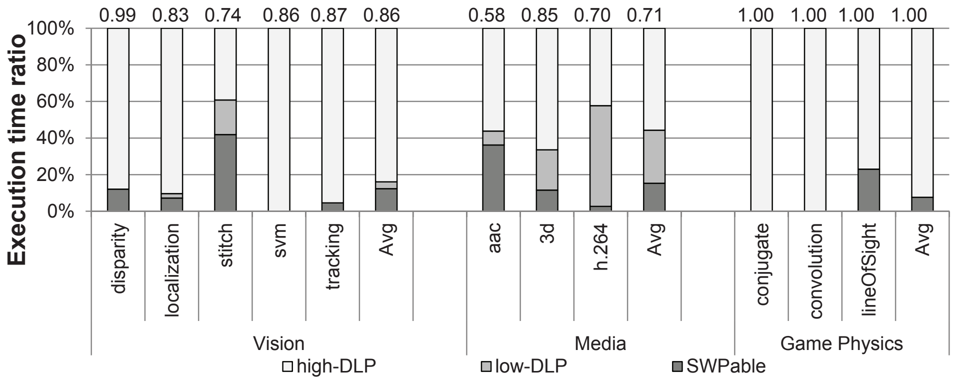


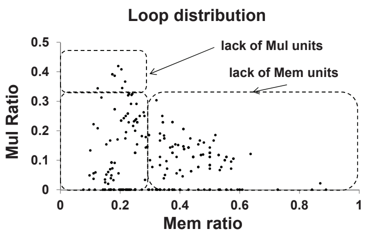
Figure 6.2: Loop categorization: The components of the bar indicate ratio of execution time in SWPable loops, low-DLP, and high-DLP SIMDizable loops. The ratio of loop execution time over total execution time is indicated as a number above each bar.

6.2.3.2 SIMD Width Variance over Loops

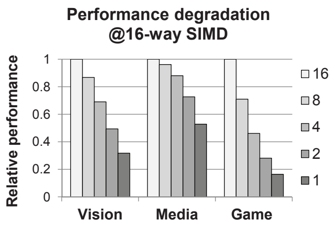
Figure 6.2 shows the relative execution time of SWPable loops and SIMDizable loops to total execution time on a simple 1-issue ARM processor. As we use a 16-lane SIMD processor for this experiment, SIMDizable loops with natural SIMD width smaller than 16 are categorized into low-DLP loops. On average, there is a substantial amount of time (87%) spent on SWPable or SIMDizable loops as expected. An interesting question here is how many applications are not well-matched to a wide SIMD accelerator. Unfortunately, 4 of 11 applications are highly dependent on SWPable and low-DLP loops, which means that not all the lanes can be utilized. For example, traditional SIMD cannot decrease the execution time of an AAC application more than 60% of the total loop execution time because around 40% of the time is spent on SWPable loops. In general, the game physics benchmarks have high levels of data parallelism, vision benchmarks have modest data parallelism, and media benchmarks have low degrees of data parallelism. Results in Figure 6.2

	Vision	Media	Game	Total
Avg ratio(MEM)	0.44	0.26	0.27	0.32
Avg ratio(MUL)	0.15	0.10	0.22	0.16
ratio of MEM loop	0.93	0.36	0.33	0.54
ratio of Mul loop	0	0.04	0	0.01

(a)



(b)



(c)

Figure 6.3: Resource utilization: (a) average ratio of dynamic instruction count of expensive instructions and ratio of Mem/Mul dominant loops, (b) loop distribution over ratio of Mem/Mul, and (c) performance degradation on a SIMD at different number of Mem/Mul resources.

confirm that a simple SIMD accelerator cannot effectively support the range of mobile applications. Even with a perfect support for DLP, SWPable and low-DLP loop execution result in low utilization of SIMD resources. Therefore, further consideration is required to fully utilize the SIMD resources on the execution of non-fully SIMDizable loops.

6.2.3.3 Resource Utilization Variance

To maximize the total utilization of computation resources, the number of each resource should be decided based on the average fraction of dynamic instructions. While current CPUs solve these challenges by out-of-order execution of parallel instructions on multiple execution units, current SIMD architectures cannot solve this problem due to its homogeneous nature: the datapath of each SIMD lane has the same functionalities, even for expensive units such as memory and multiply units. These characteristics are unfavorable in terms of efficiency because not all execution units are active every cycle, and expensive units are much less utilized (an average of only 32% for a memory unit and 16% for a multiply unit (Figure 6.3(a))). A traditional solution for this problem is to turn off the unused resources by clock/input gating, but this solution does not eliminate leakage power. Power gating is unlikely a practical solution because idle periods for expensive units tend to be relatively short.

Another challenge is the diversity of instruction distribution across/inside applications. Even if we are somehow able to place a specific number of each execution unit based on average fraction, careful consideration is also required because the fraction varies greatly. In Figure 6.3(a), for example, the ratio of multiply instruction varies from 10% to 22% across three application domains. We also define a loop to be memory/multiplication dependent if the fraction of memory/multiplication instructions are more than 33% of the total instructions. Figure 6.3(b) shows a distribution of the loops according to the ratio of memory/multiply instructions. Based on Figure 6.3(a) and (b), more than 54% of the loops in the three benchmark sets highly depend on the memory instructions, and therefore, normal ALU functional units can be idle due to the memory operation bottleneck if only 33% of memory resources exist. On the contrary, multiplication is not the critical performance bottleneck if 33% of multiplication resources exist because only 1% of the loops are multiplication dependent. As a result, the high diversity in the instruction distribution will make most loops to not be effectively accelerated due to the lack of enough resources, or to waste resources due to the excess resources, if the SIMD accelerator simply allocate resources based on specific rules such as average fraction or one per four lanes.

6.2.4 Insights for the Traditional SIMD

Based on the application analysis, we found several fundamental sources of SIMD inefficiency. First, a traditional wide SIMD accelerator may be over-designed since the overall performance will be saturated at some point and limited by non-high-DLP loops where the SIMD accelerator is poorly utilized. Second, lane uniformity makes the SIMD datapath inefficient due to over-provisioning expensive resources. Third, the high variation in the resource requirements of loops makes the problem more difficult than simple sharing of expensive resources would accomplish. A central challenge here is how to decrease over-provided resources on traditional SIMD accelerators and to overcome the inflexibility in order to more effectively utilize the hardware.

6.3 Libra Architecture

6.3.1 Overview

The Libra accelerator presented here is a unified accelerator for mobile applications that allows flexible execution of loops by customizing the configuration adaptive to their key characteristics. The Libra accelerator is based on traditional SIMD accelerators and has several important extensions for providing both high energy-efficiency and performance improvement. First, Libra is composed of a non-uniform lane structure for power efficiency: only a subset of lanes has expensive but infrequently used execution units. Furthermore, dynamic configurability of logical lanes helps Libra in executing a target loop in an efficient manner with high utilization. In Libra, a group of logical lanes is executed

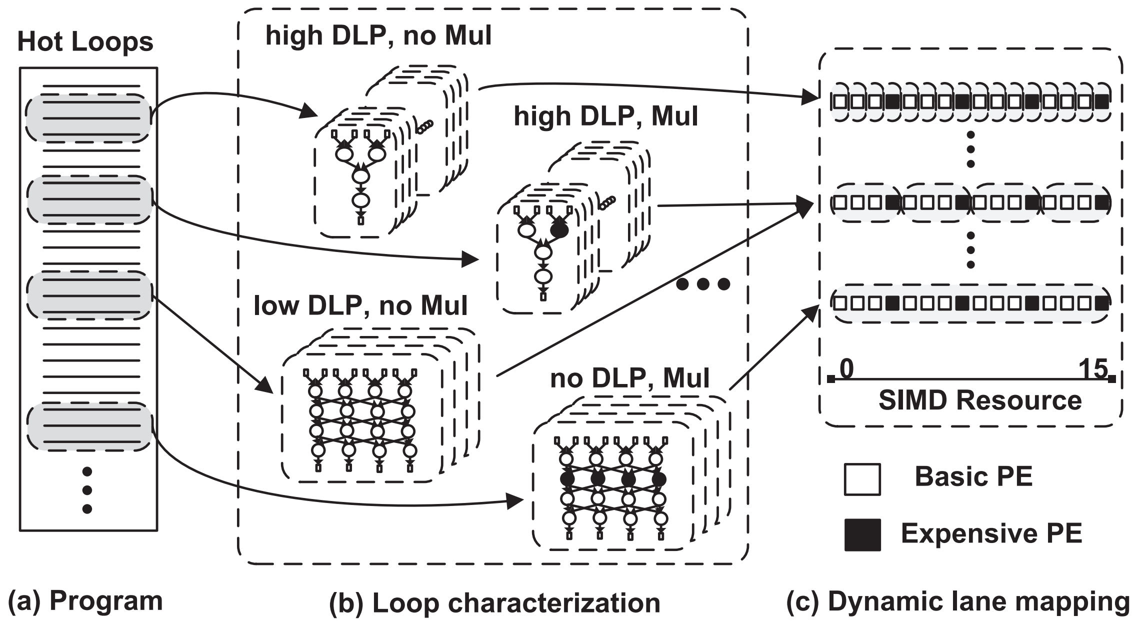


Figure 6.4: Mapping loops to Libra: (a) identify hot loops, (b) find the available DLP and resource requirement of each expensive operation, and (c) change the configuration based on the characteristics of each loop.

in a SIMD manner, where the logical lane is configured by a group of processing elements (PEs). DLP is exploited in the form of parallel execution of logical lanes, and ILP is exploited inside each logical lane in a way that each PE execute different operations. Therefore, Libra is able to flexibly tune the ILP/DLP-support capability by changing the logical lane configuration.

Figure 6.4 shows a conceptual view of the execution of Libra. First, several hot loops are identified as candidates to be accelerated utilizing the Libra architecture(Figure 6.4(a)). Second, software-pipelinable loops are selected, and the DLP availability is also determined as discussed in Section 6.2.3.1(Figure 6.4(b)). In this step, several additional key characteristics such as the amount of potential ILP in the loopbody and the ratio of expensive instructions are also considered. Finally, a best matched logical lane configuration for each loop is chosen by the compiler (Figure 6.4(c)). In Figure 6.4, we assume a 16-lane heterogeneous SIMD including 12 basic and 4 expensive PEs. Based on this, each PE constitutes one logical lane for full DLP support to execute high-DLP loops having only simple instructions, intermediate numbers of PEs form each logical lane for ILP/DLP hybrid execution to support low-DLP loops or expensive operation-intensive loops, and one large logical lane for full ILP execution is configured for non-DLP loops. Note that fully exploiting SIMD parallelism does not always outperform exploiting ILP on heterogeneous structures. Section 6.3.1.1 and 6.3.1.2 explain the core concept of Libra in detail with evidence of its effectiveness.

6.3.1.1 Heterogeneity

Heterogeneous lane organization, based on average fraction of resource utilization, is required in order to enhance power efficiency: all the lanes support simple integer operations and only a subset of the lanes support expensive operations. When an expensive instruction is fetched, the accelerator stalls until this subset of lanes generates results for all lanes, then resumes execution. This structure delivers a high level of power efficiency due to the expensive resource removal, but significant performance degradation will occur when executing expensive operation-intensive code. Figure 6.3(c) illustrates the performance degradation as the number of multiplier/memory units decreases on a 16-lane SIMD accelerator. Each bar shows the relative performance normalized to that of the homogeneous SIMD when each heterogeneous SIMD has specific number of expensive resources. From this graph, substantial amounts of performance degradation exist in vision and game benchmark because they are highly dependent on expensive operations and incur a number

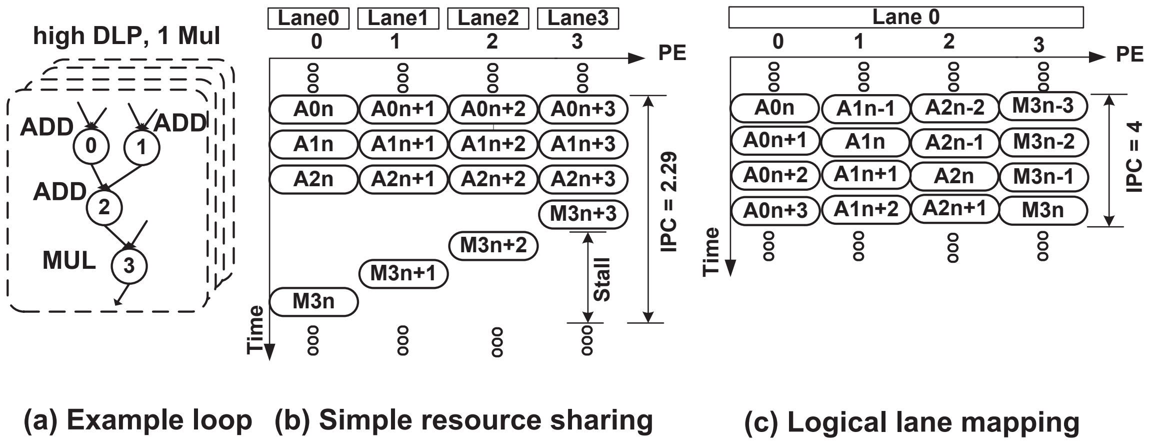


Figure 6.5: Dynamic configurability on a 4-lane heterogeneous SIMD (lane 3 has a multiplier): (a) a simple high-DLP loop with 1 multiply, (b) performance degradation due to stalls during multiply execution, (c) logical lane formation removes stalls by instruction pipelining.

of stalls to handle these operations. However, media benchmarks are not highly affected by the proportion of these expensive resources because the performance is already constrained by low DLP.

6.3.1.2 Dynamic Configurability

Dynamic configurability of lanes helps the heterogeneous SIMD accelerator in dealing with the aforementioned problems. One logical lane can consist of one PE for highly SIMDizable loops with no expensive instructions, and also consist of multiple PEs for non/low-SIMDizable loops or loops having expensive instructions. The resulting SIMD width is decided by the number of logical lanes and each logical lane executes the same instruction stream in lockstep. Inside a logical lane, ILP is exploited to use multiple lanes in parallel, and therefore it can efficiently distribute instructions between simple lanes and expensive lanes.

The effectiveness of dynamic lane mapping can be explained by the simple following performance equation. In the equation, we compare the total performance of the simple SIMD and the Libra SIMD by the metric of IPC (instruction per cycle). The IPC of SIMD can be calculated by the multiplication of IPC of one lane (IPC_{lane}) and the minimum of the number of PEs (N_{SIMD}) and the available degree of DLP (N_{DLP}) of the target loop (Equation (6.1)). Similarly, the IPC of Libra can be the multiplication of IPC of one logical lane ($IPC_{logical\_lane}$), consisting of m PEs, and the minimum of the number of logical lanes ($\frac{N_{SIMD}}{m}$) and the degree of DLP of the loop (Equation (6.2)). Therefore, when executing non/low-DLP loops, Libra can easily outperform the basic SIMD because it only requires better performance of a logical lane than that of a PE, and it is always true as a logical lane exploits ILP with multiple PEs inside(Equation (6.3)). Dynamic configurability is also able to address the performance degradation problem on the heterogeneous SIMD. When executing high-DLP loops, Libra outperforms SIMD when the IPC of a logical lane is higher than that of m PEs. Although the ILP performance is normally inferior to DLP performance because of its dependences and complexity, Libra can frequently be better due to the heterogeneity. Figure 6.5(a), (b) and (c) shows the superiority of Libra. Figure 6.5(b) and (c) show the execution of a simple high-DLP loop having a multiply instruction on both the simple SIMD and Libra which have one multiplier on the PE 3. In this example, the IPC of SIMD is less than the IPC of Libra when one large logical lane is configured due to a number of stalls.

$$IPC_{SIMD} = \min(N_{SIMD}, N_{DLP}) \times IPC_{lane} \tag{6.1}$$

$$IPC_{Libra} = \min(\frac{N_{SIMD}}{m}, N_{DLP}) \times IPC_{logical\_lane} \tag{6.2}$$

$$IPC_{Libra} > IPC_{SIMD},$$
$$when \begin{cases} IPC_{logical\_lane} > IPC_{lane}, & \text{if } \frac{N_{SIMD}}{m} > N_{DLP} \\ IPC_{logical\_lane} > m \times IPC_{lane}, & \text{if } \frac{N_{SIMD}}{m} < N_{DLP} \end{cases} \tag{6.3}$$

6.3.2 Microarchitectural Details

The Libra architecture with eight PE groups (32 PEs) is shown in Figure 6.6(a). Differently from the traditional SIMD, the Libra datapath consists of 2 groups of clusters, which can be configured to create logical SIMD lanes of 2, 4, 8, and 16 PEs based on the loop characteristics. Each of the clusters is composed of 4 PE groups. The SIMD controller performs the role of managing the logical lane status to exploit SIMD parallelism, while the thread controller manages the ILP-exploiting method inside the logical lane. Each PE group contains 4 PEs. Each of the PEs has an FU and a register file, which can be thought as one lane of the traditional SIMD. Only one of the PEs in a PE group has a multiplier while another has a memory unit. Differently from the traditional SIMD, each PE group also has two kinds of reconfigurable interconnects inside and across PE groups in order to achieve flexible configuration of logical lanes.

Key features of Libra architectures are as follows:

Scalability: The resources are fully distributed including FUs, register files, and inter-

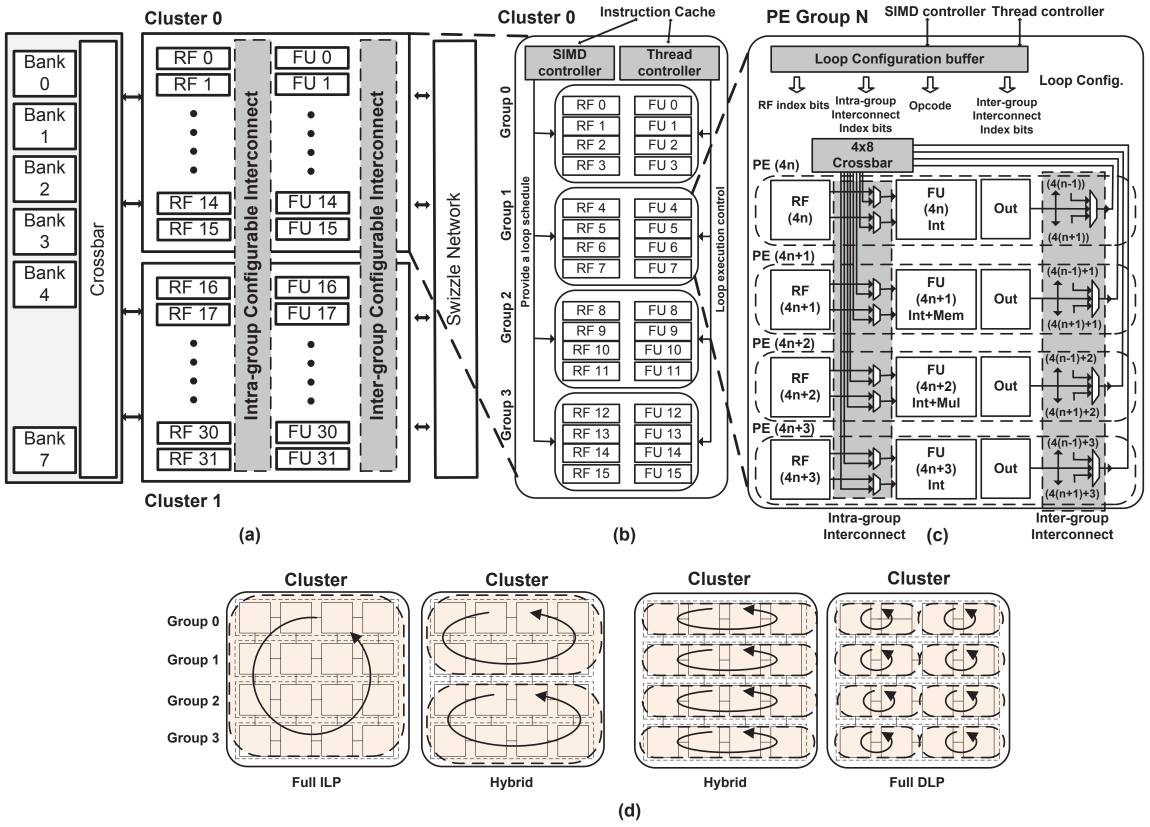


Figure 6.6: The 32-PE Libra architecture: (a) a 2-cluster Libra accelerator, (b) a cluster, (c) an example of a single PE group: PE 1 supports memory operation and PE 2 supports multiply operation, and (d) execution modes.

connections. PE groups have dense interconnections inside but each PE group is sparsely connected with neighbors. As a result, area and power costs increase approximately proportional to the number of resources, which makes Libra as scalable as a simple SIMD.

Polymorphic Lane Organization: PE groups can be aggregated to form a larger logical lane in order to exploit the existing ILP inside the loop body, or be split into multiple small logical lanes in order to exploit DLP over loop iterations.

Resource Sharing: In heterogeneity, the major challenge is how to determine the number of expensive resources and how to efficiently share them between logical lanes when necessary. To flexibly handle this, we place the expensive resources based on the average utilization and provide a sharing mechanism between them in two categories. A more detailed description is provided in Section 6.3.3.3.

Simple Multi-threading Mechanism: Even though a logical lane provides a number of parallel resources, efficient use of the available resources is limited due to the low ILP of the loopbody. Therefore, we extended the ILP into loop-level parallelism through modulo scheduling [73]. Modulo scheduling generally provides a decent performance improvement by parallelizing instructions over loop iterations and hiding long latency between back-to-back instructions. However, several Libra specific features, such as SIMD capability and fully-distributed nature, diminish the effectiveness of modulo scheduling. To compensate for this, simple static multi-threading with list scheduling is proposed in Section 6.3.4.

6.3.2.1 PE Group

A detailed illustration of a single Libra PE group is provided in Figure 6.6(c). A PE group consists of four PEs each with a 32-bit FU and a 16-entry register file with 2-read/1-write ports (write ports can be added to support threading). Integer arithmetic operations are supported in all four FUs but multiply and memory operations are available in only one FU per PE group (PE1 for memory and PE 2 for multiplication in Figure 6.6(c)). The FUs inside are modified to connect with each other with a dense 4x8 full crossbar network for passing data between the FUs without writing back to the RF. This allows the PE groups to exploit ILP in a distributed nature. In order to retain scalability, the Libra architecture has a simple and fully distributed across-PE group interconnect. Only FUs are connected between the corresponding neighbors in adjacent PE groups. In addition to these components, a loop configuration buffer is added to store instructions for modulo/list scheduled loops. The buffer is a small SRAM that saves the configuration information including instructions, register addresses and interconnect index bits of the current loop. The interconnect between the loop buffer and SIMD/Thread controllers in the cluster is used to transfer instructions for executing loops. The hardware components and execution mechanism for SIMD/ILP support is explained in detail in Sections 6.3.3 and 6.3.4.

6.3.2.2 Cluster

A cluster is a high-level basic unit that consists of four PE groups and several additional features for flexible loop execution support: the SIMD controller and the thread controller. The SIMD controller is a small controller to manage the logical lane organization inside the cluster, including the number of logical lanes and the SIMD width of memory transfer. It receives the information from the instruction cache. In addition, the SIMD controller also gets the configuration for one logical lane from the instruction cache and transfers it to each PE group. A thread controller is responsible for executing loops. It also gets the information about which mode is selected from the instruction cache and orchestrates the loop execution. When modulo scheduling is selected, it just executes the loop sequentially, and, when multi-threading is selected, it executes the loop in the order of the thread sequence table. The information is statically set during compile time and is fetched from the instruction cache. Multiple clusters can execute one large loop or can execute multiple parallel loops separately.

6.3.2.3 Configuration Process

Loop execution of Libra can be divided into two stages: configuration and execution. Configuration stage is forming logical lanes and sending configuration bits to all the loop buffers of each PE-group. For every loop, the instruction cache contains both logical lane organization information and configuration bits for one logical lane. The SIMD controller gets these information from the instruction cache and then sends the configuration bits to the loop buffers of the PE groups based on the logical lane configuration. The thread controller also gets the information about the execution mode and sequence table, if required, from the instruction cache. This process takes 3-5 cycles on average before the loop buffer receives the configuration bits for the first cycle and the time varies depending on the size of the logical lane. The thread controller starts the execution when the first cycle configuration is ready on all the loop buffers.

6.3.2.4 Memory Support

The memory operation of the Libra system needs support for both scalar and SIMD memory access. For scalar memory access, the local memory has the same number of banks as the number of total memory units. For SIMD access, the local memory also needs to support contiguous access across all logical lanes in parallel. Therefore, for the 32-PE Libra system, a 64kB local memory is used, consisting of 8 memory banks where each bank is a 2-wide SIMD containing 1024 32-bit entries. As shown in Section 6.2.3.1, all memory transfers have the same strides over iterations in SIMDizable loops. Therefore, when several logical lanes execute the same instructions for SIMDized loops, a single address calculation followed by a wide memory operation is performed. The data is then distributed to different logical lanes. Multiple memory units inside a logical lane need to generate their own memory addresses. The SIMD width of each access and the number of different addresses are determined by the logical lane configuration, which is saved in the SIMD controller.

6.3.2.5 Communication with a Host Processor

The Libra architecture is a co-processor similar to a GPU and interfaces with a host processor such as ARM using memory. The data transfer is performed through a standard AMBA bus along with a DMA.

6.3.3 Execution Model

This section describes the three different execution modes of the Libra architecture, which are full ILP, hybrid, and full DLP modes. We first explain how each mode operates and then provide proof of how the three modes can effectively support different kinds of loops. The example provided assumes a four-PE group cluster as shown in Figure 6.6(d).

6.3.3.1 Full ILP Mode

In this mode, the Libra architecture decides to use all the PEs as one large logical lane. The SIMD controller spreads different configuration informations into the loop buffer of each PE group. The execution mechanism is the same as the loop acceleration technique of common VLIW solutions but the performance might be slightly worse than previous solutions because the Libra architecture sacrifices both centralized resources and dense across-

PE group interconnects. Applications which have a high proportion of non-SIMDizable loops mostly utilize this mode for acceleration.

6.3.3.2 Hybrid Mode

When a loop is SIMDizable, a cluster has the possibility of either having several small logical lanes or forming a large logical lane. In this case, the Libra architecture may choose to use a hybrid mode with a cluster having at least two logical lanes, each having at least one PE group. With smaller logical lanes, the performance usually increases since SIMDization provides an opportunity to increase performance by the same amount as the degree of DLP. Also the routing overhead decreases with small logical lanes, further boosting performance. Figure 6.6(d) also has two examples of hybrid mode execution. The SIMD controller distributes the same configuration information and live values to the loop buffer and RFs of each logical lane. When a loop lacks sufficient level of DLP or has a moderate proportion of expensive resources, hybrid mode can achieve the best performance.

6.3.3.3 Full DLP Mode

When a loop is highly data-parallel but has a low degree of ILP, the resources (PEs) cannot be effectively utilized because the degree of ILP in the loop cannot meet the minimum degree of the PE group. To compensate for the lack of ILP, the Libra architecture supports separation of PE groups, forming two smaller logical lanes. As a result, SIMD parallelism can make up for insufficient ILP in the loops (also in Figure 6.6(d)). Hence, a cluster has a total of eight logical lanes executing in lockstep. Distinct from loops with a small number of instructions, loops with unbalanced resource usage can also be well matched to a full DLP mode, unlike the hybrid mode. As mentioned in Section 6.2.3.3, the hybrid mode cannot fully utilize resources in a PE group since performance of loops with a high proportion of memory operations are constrained by the memory unit.

The major challenge in full DLP mode is determining how to share expensive resources between two small logical lanes in a PE group. The first category for resource sharing is expensive but infrequently used functionalities such as the multiply operation. As shown in Figure 6.3(a), the average ratio of multiply is as low as 16% and only 1% of loops are multiply-dominant, and therefore simple sharing between two half-PE groups does not incur performance degradation. The second category is frequently used functionalities such as memory operations as shown in Figure 6.3(a). These instructions are already a performance bottleneck and simple sharing cannot enhance the overall performance. Therefore, this shared resource should lead to double the performance in a lightweight manner.

We accomplish these requirements using simple hardware modifications as shown in Figure 6.7(a). One PE group is mapped into two small logical lanes with (PE 0, PE 1) and (PE 2, PE 3). Based on the application analysis, only PE2 supports multiply operations and PE 1 supports memory operations. To ensure that both logical lanes support all functionalities, PE 0 and PE 2 share the multiplier and PE 1 and PE 3 share the memory unit. To share the multiplier, PE 0 connects input and output ports to the multiplier of PE 2. A memory controller in PE 1 is shared with PE 3 in a different manner. When the memory controller receives a memory operation command, only PE1 communicates with the memory with double bandwidth and send/receives the data of PE 3 through a bypass logic.

To execute the same instructions in both logical lanes using the above modifications, the following processes are required:

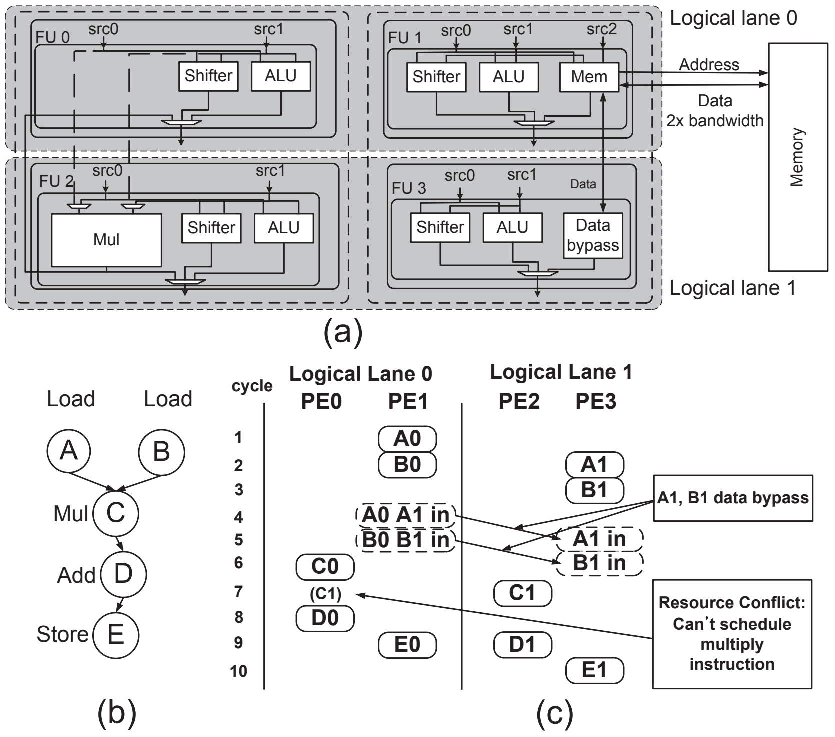


Figure 6.7: Resource sharing support: (a) hardware modification: PE 0 and 2 share the multiplier and PE 1 and 3 share the memory unit, (b) example loop body dataflow graph, and (c) actual schedule: 1-cycle difference between lanes for resource contention avoidance.

- The compiler must not schedule multiply instructions in a row, because the multiplier needs a spare cycle after the cycle in which the multiply instruction is scheduled in order to handle the operation of the other logical lane. However, other instructions can be placed since they have no resource or writeback contention. Memory instructions can be scheduled without any restrictions as the hardware supports double bandwidth.

- The SIMD controller has the instruction configuration only for one logical lane. The controller transfers the same configuration into the loop buffer of both logical lanes with one-cycle difference to avoid resource contention.

Figure 6.7(b) is an example of a full DLP mode execution. For a simple dataflow graph of the loop body, the latency of the load and multiply operations are set to 4 and 2. Due to the small size and high memory dependent characteristic of the loop body, a full DLP mode is selected and each PE group is separated into two logical lanes. Identical schedules based on two PEs are transferred into the loop buffer in the PE group with one cycle difference between logical lane 0 and logical lane 1 (see Figure 6.7(c)). Different memory operations can execute in the same cycle as shown in cycle 2 but different multiply instructions cannot be scheduled at cycle 7 because logical lane 1 needs to use the multiplier in that cycle.

6.3.4 Improving ILP Performance

Although modulo scheduling has proven to be an effective solution to exploit ILP over loops, it is not always the best solution because 1) original iteration count is divided by DLP capability, and therefore, the smaller iteration count may not compensate for the prolog and epilog overheads even in moderate DLP loops [79] and 2) sparse interconnection between PEs and no centralized RFs make the quality of the schedule worse. As a result, we suggest supporting list scheduling [7] of the loop body as another option to exploit ILP. When either there is not much total ILP in the loop, or the hardware cannot benefit from increased ILP, list scheduling can outperform modulo scheduling since it does not incur the overhead of modulo scheduling: handling modulo information such as staging predicates.

The remaining problem of adapting list scheduling to hide idle cycles comes from long latency instructions such as multiply and memory operations. To solve this problem, we propose a simple multi-threading scheme with fast context switching. Assuming the Libra architecture supports two threads, a loop with large number of iterations is divided into two threads with identical loops with half number of iterations. The two threads are then executed on the same logical lane. To make the scheme simple, a switch of running threads is allowed only when all the PEs are idle. Each thread has its own register file space divided by the number of threads, similar to what a GPU does, and therefore no context change overhead exists. The schedule with multiple threads is statically decided at compile time. The multi-threading technique is simple but highly effective and is a realistic solution because of the following two reasons: 1) low register pressure: loops with small number of instructions have a small amount of data to save in the register file and list scheduling does not require additional register overhead, and 2) a high chance of hiding latency: this technique is applied only to SIMDizable loops executing on small logical lanes, thus increasing the probability that all FUs are idle.

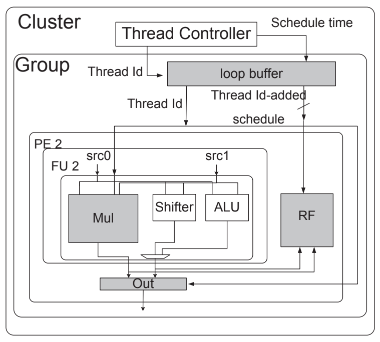
Although multi-threading looks promising, the Libra architecture faces a number of challenges in reality. There are three essential challenges and we present the lightweight solutions incorporated in the Libra architecture:

Context Saving: The fully distributed nature of Libra allows temporal data to be saved in the register files as well as the output buffer in order to directly transfer the data between FUs. As a result, the output buffer data of each thread should also be saved in addition to the register files. The register file is divided into the same number of threads. The parts are then addressed by the thread ID. However, the output buffer is originally a simple flip-flop without addressing support. Therefore, it is substituted by an *n*-entry register file addressed by thread ID(*n*: the number of threads supported). The output data can thus remain unchanged when another thread is executed.

Writeback Contention Avoidance: Handling multi-latency instructions is not a simple problem if the output data from a multi-latency instruction is generated when the other thread is executing. To solve this problem, multi-latency FUs need to save the thread ID when the input is issued and be connected to the output buffer (small register file) with an additional port addressed by the original input thread ID. Since only a single additional port is required for multiple FUs with the same latency, the overhead is negligible. For the Libra architecture, only two ports are added to the whole PE group to support a multiplier and a memory controller.

Code Bloat: Since multiple threads are scheduled at compile time, the loop buffer of each PE group needs to contain the entire schedule information of all threads for each cycle. This causes the code bloat problem, requiring an increased loop buffer size which incurs a power overhead. However, an important observation to point out is that the schedules of different threads are essentially the same, just with different execution times. We can, therefore, solve the problem by 1) saving the schedule configuration of only one thread and 2) adding a simple sequence table which contains a thread ID and the corresponding loop buffer address pointing to the actual schedule configuration. The thread controller contains the basic information for supporting multi-threading and the sequence table.

Figure 6.8 shows an illustration of the Libra architecture with an emphasis on modified features(shaded components) to support multi-threading, assuming that the architecture supports execution of two threads. The loop buffer contains configuration information for only one thread as shown in Figure 6.8(c). Therefore, its size is the same as when one thread is executed. The thread controller in the cluster has a tiny sequence table containing the actual thread ID and the address of the configuration saved in the loop buffer. Figure 6.8(b)



(a)

Cycle	Thread Id	Loop buffer address
0	0	0
1	0	1
2	1	0
3	1	1
4	0	3
5	0	4
6	0	5
7	1	3
8	1	4
9	1	5

(b)

Original Cycle	Original Configuration
0	A
1	B
2	NOP
3	C
4	D
5	E

(c)

Cycle	Thread-aware configuration
0	A0
1	B0
2	A1
3	B1
4	C0
5	D0
6	E0
7	C1
8	D1
9	E1

(d)

Figure 6.8: Multi-threading support & compiler support: (a) hardware modification: shaded components are modified, (b) sequence table in the thread controller, (c) loop buffer, and (d) final multi-threaded schedule.

depicts an example sequence table for two thread execution. Since two threads are executed in this example, the space of RF is divided by two and the output buffer is a 2-entry register file. By reading the sequence table from cycle 0 to cycle 9, the thread controller transfers the thread ID and loop buffer address for each cycle to the loop buffer. From this information, the loop buffer generates the final configuration by reading the appropriate configuration and adding a thread ID to the register file address (see Figure 6.8(d)). The multiplier gets the thread ID and has a separate data bus due to the multi-latency functionality. When the original configuration B has the multiply operation for FU 2, the result data from thread 0 and B configuration can be stored in the output buffer at cycle 2 without any writeback contention.

6.3.5 Decision Flow

In order to maximize the performance and resource utilization, the Libra architecture depends on an intelligent selection of the configuration between the number of logical lanes

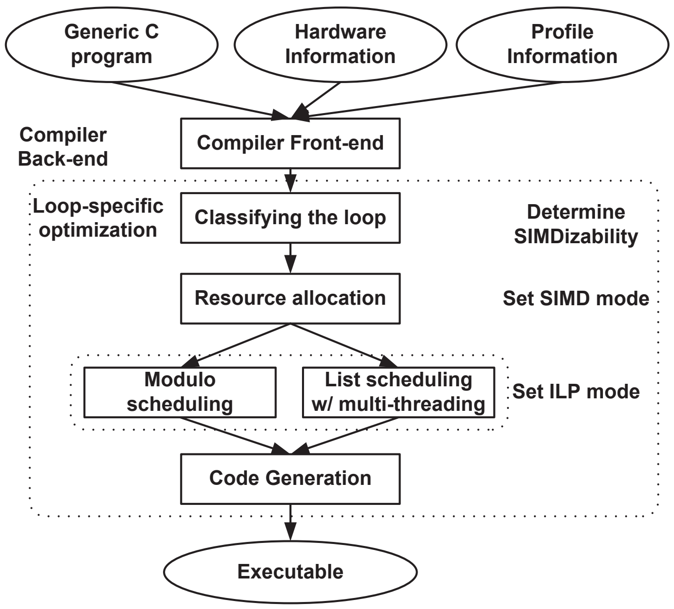


Figure 6.9: Decision flow of the Libra architecture.

and the size of each logical lane. The system flow is shown in Figure 6.9. Applications run through a front-end compiler, producing a generic Intermediate Representation (IR), which is unscheduled and uses virtual registers. The compiler also has a high-level machine specific information, including the number of resources, size of register files, the size of a cluster, and the number of supported micro-threads. In addition to this, the compiler needs to have profile information about the iteration counts of loops and memory alias information. Given the IR, hardware and profile information, the compiler categorizes loops into two basic types: SWPable and SIMDizable loops. The compiler then decides the logical lane configuration of a cluster for each loop (resource allocation). If a loop is not SIMDizable but only SWPable, the entire cluster is assigned to the loop. If a loop is proved as SIMDizable, the compiler finds the best configuration based on the provided information such as average iteration count, instruction and dependency information of the loop. Briefly speaking, the compiler tries to fully exploit SIMD parallelism by securing the maximum number of logical lanes without performance degradation due to the instruction imbalance. However, it also performs broad design space explorations by changing the number of logical lanes. This is because 1) sometimes the effectiveness of DLP is not clear when the divided trip count is small and the instruction number is not too small, and 2) the scheduler uses a heuristic way to generate the modulo schedule. After deciding the lane configuration, the compiler chooses the method to exploit ILP inside the logical lane. Finally, the compiler performs modulo scheduling or list scheduling. It then generates the final schedule and the configuration information.

6.4 Experiments

6.4.1 Experimental Setup

Target Architecture To evaluate the effectiveness of the Libra architecture, three example implementations with different sizes are used: 16 (one cluster, four PE groups), 32 (two clusters), and 64 (four clusters) PEs. Four FUs per cluster are able to perform load/store instructions to access the data memory with four-cycle latency while another four FUs support two-cycle pipelined multiply instructions. The Libra is compared against two other accelerators in our experiment. We generate 4(cluster)$\times$4(PE), 8$\times$4, and 16$\times$4 heterogeneous VLIWs having the same organization of PEs as corresponding Libra architectures. The wide SIMD architecture as discussed in Section 6.2.2 is used and the number of SIMD resources can vary from 16 to 64, having the same heterogeneous FU structure.

Target Applications As discussed in Section 6.2.1, the evaluation is conducted for subsets of three domains. Max 20 top loops having a high execution time are selected for vision and game physics benchmarks, and 144 loop kernels, varying in size from 4 to 142 operations, are extracted from the media benchmark because the ratio of execution time to the total execution time of the top 20 loops is too small. High number of loops in the media benchmarks and several major loops in the vision benchmarks have conditional statements, while the gaming benchmarks do not have them. In order to eliminate all internal branches, we applied if-conversion for these loops.

Compilation and Simulation The industrial tool chain developed by SAIT [5] is used for compilation and simulation of Libra. The IMPACT compiler [71] is used as the frontend compiler. Basic list scheduler [7], edge-centric modulo scheduling (EMS) [73]-based modulo scheduler, and simple loop-level SIMDization scheduler using a SODA-style [59] wide vector instruction set are implemented in the backend compiler. Based on the original modulo scheduler, we developed a scheduler that can support both flexible execution of Libra and list scheduling with static multi-threading technique. The performance is generated by the cycle-accurate code schedule of loops, accounting for the configuration overhead.

Performance Measurement For fair comparison, both list scheduling and modulo scheduling are applied and the better performing schedule is picked for the SIMD accelerator. For VLIW, loop unrolling is applied when a loopbody size is too small and its resources may not be fully utilized. Multi-threading technique of Libra is also not applied for a fair comparison of the performance of the three architectures. This issue is discussed in Section 6.4.6.

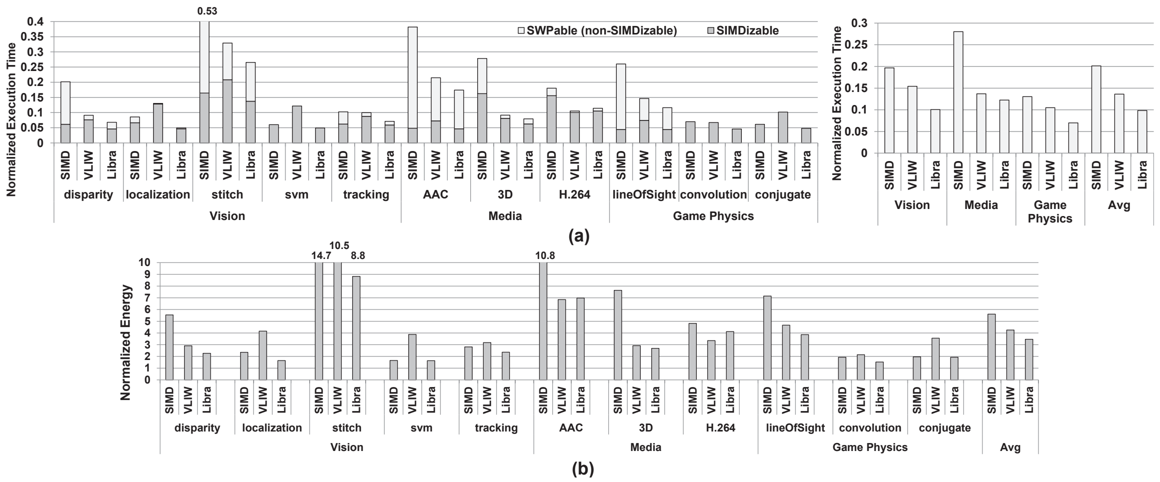


Figure 6.10: Performance/Energy comparison of 32-PE Libra/SIMD/VLIW architectures: (a) total loop execution time and (b) energy consumption. All the data are normalized to that of a simple in-order core.

Power/Area Measurements All architectures are generated in RTL Verilog, synthesized with the Synopsys design compiler, and place-and-routed with the Cadence Encounter using IBM SOI 45nm regular Vt standard cell library in slow operating conditions with a 0.81V operating voltage. Synopsys PrimeTime PX is used to measure the power consumption based on the utilization. The Artisan Memory Compiler is used to determine the area and the power of the memory operation using a 0.81 Volts operating voltage. The target frequency of Libra is 500MHz[2] similar to the latest mobile GPUs.

6.4.2 Performance/Energy Evaluation

We compared the performance of a 32-PE Libra architecture with identically sized VLIW (8×4) and SIMD(32-wide) architectures. Performance results are measured as the total loop execution time when each loop is scheduled by the method the target architecture

[2]The FO4 delay of this process is about 13ps.

supports. Figure 6.10(a) shows a plot comparing the performance of the three architectures normalized to the simple 1-issue inorder core. For individual benchmarks, the graph also indicates the fraction of two different loop categories: SIMDizable and SWPable loops.

For benchmarks with a high ratio of non-SIMDizable loops such as stitch, AAC, and lineOfSight, SIMD shows severe performance degradation, whereas VLIW and Libra show a fair performance improvement. Libra outperforms even VLIW because it can accelerate SIMDizable regions more efficiently. On the other hand, both the SIMD and Libra deliver a substantial performance improvement for benchmarks with mostly SIMDizable loops, while VLIW suffers. The Libra also shows better performance than SIMD because it effectively accelerates applications having low-SIMDizable loops (3D, H.264) and its ILP capability also helps Libra to adequately tolerate the lack of expensive resources for high-SIMDizable loops (convolution, conjugate). Overall, Libra shows the best performance in all benchmarks except H.264 benchmark. This is because of the slightly lower performance gain on SWPable regions due to its distributed nature. Among average result of each domain, performance gain of Libra is the highest on game physics. As a result, Libra shows a performance gain of 2.04x and 1.38x over SIMD and VLIW, respectively.

Despite using the same amount of computation resources, performance-only comparison may not be fair due to the different interconnection strategies among the architectures. An energy comparison may yield a better comparison considering both performance and hardware overhead. Figure 6.10(b) shows the energy consumption of three architectures and the results are also normalized to the 1-issue core. This graph shows a similar trend to Figure 6.10(a). On average, even though SIMD added extra logics for handling sharing resources (Figure 6.5(b)), VLIW shows 16% more power consumption because of bigger

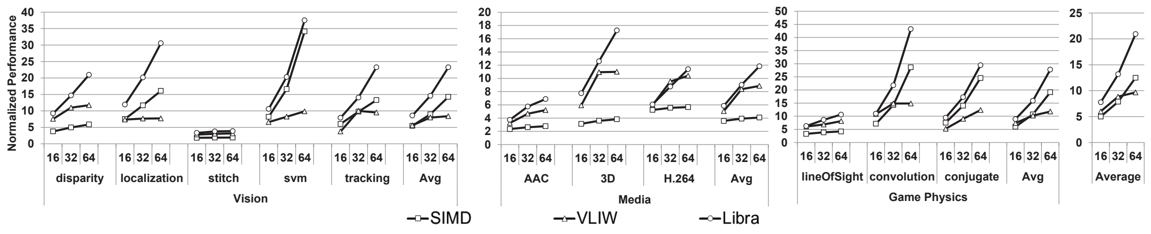


Figure 6.11: Scalability of Libra/SIMD/VLIW architectures: the Libra architecture is highly scalable for most of benchmarks, while SIMD and VLIW cannot be scalable for several benchmarks.

RFs and complex control logics, and Libra shows 20% more power consumption due to more interconnects and Libra-specific overhead such as a loop-buffer and a thread controller. Based on these power differences, the Libra saves 38% and 19% energy compared to SIMD and VLIW, respectively[3]. As a result, the Libra architecture shows a fair amount of performance improvement in addition to high energy efficiency by providing a more suitable acceleration scheme for each loop.

6.4.3 Scalability

Figure 6.11 shows the performance of each architecture normalized to a 1-issue core for different sizes across three benchmark domains. The number of PEs varying from 16 to 64 are shown on the X-axis. The results show high scalability of the Libra architecture in all benchmark domains.

In the vision and game domain benchmarks, applications are not specially optimized

[3]Figure 6.10(b) does not mean that a simple 1-issue core is 3x energy efficient than Libra because the performances are different. For a performance-equivalent comparison, Libra is much more efficient than the simple core.

to the SIMD-style architecture, but the performance is highly scalable as the number of PEs increases because most loops are simple and highly SIMDizable. Only the stitch is barely scalable because the application is mostly sequential as the dominating loop has only a small number of iterations. In the media domain, the Libra accelerator performance also fairly increases as it scales to more PEs. Compared to other architectures, VLIW performance results are frequently saturated because modulo scheduling of a big size loop-body(often unrolled) on a large number of PEs is too complex to exploit ILP, while Libra solves this problem by scheduling a small loopbody in a small logical lane and applying the same schedule to multiple logical lanes. The SIMD results are also constrained by lack of expensive resources and program complexity. To summarize, the Libra architecture can increase its performance with larger resources when the application has enough total ILP/DLP parallelism.

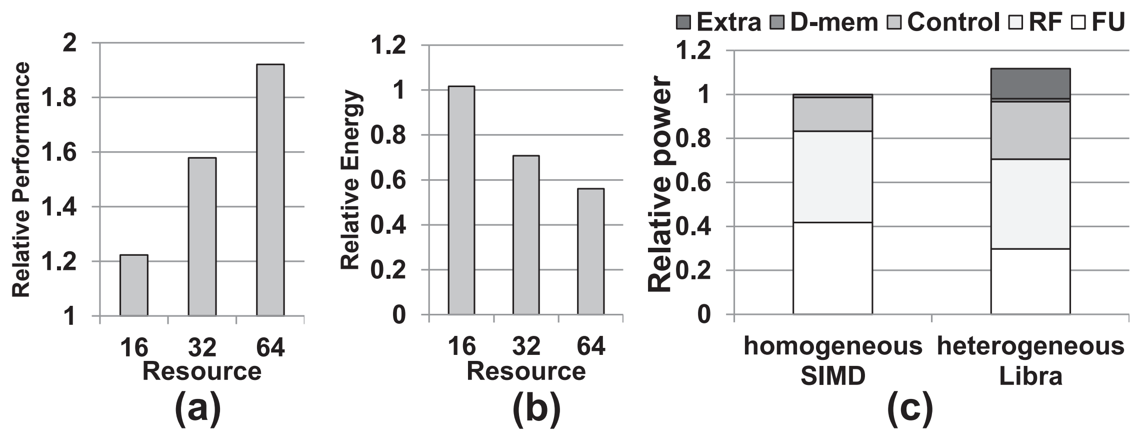


Figure 6.12: Performance/energy improvement of the heterogeneous Libra over the same sized homogeneous SIMD: (a) performance, (b) energy consumption, and (c) power breakdown with five categories: FU, RF, control logic, memory, and architecture specific additional logic.

6.4.4 From the Homogeneous SIMD to the Heterogeneous Libra

Section 6.4.2 and 6.4.3 evaluate three different architectures consisting of the same computation resources. The key question here is how much Libra surpasses the traditional SIMD architecture. To answer this question, we compared the performance and energy consumption of the heterogeneous Libra and the homogeneous SIMD. The heterogeneous Libra has a quarter of memory/multiply resources and the homogeneous SIMD has the same number of memory/multiply resources as the total number PEs. Figure 6.12 shows the average of relative performance and energy consumption of Libra over SIMD for different sizes. In terms of performance, Libra outperforms SIMD and the difference increases in proportion to the size (Figure 6.12(a)). This is because 1) the lack of expensive resources can be effectively compensated for by forming logical lanes and 2) the lane utilization of the traditional SIMD is lower for a larger size due to the program characteristics.

In terms of the energy consumption, Libra still shows similar results as its performance improvement because significantly less computational units can reduce the overall power overheads, and the result is better on larger size. For example, the 32-PE heterogeneous Libra consumes 11% more power than the same size homogeneous SIMD due to 12% power savings on FUs with 23% overheads (Figure 6.12(c)). On average, Libra shows 101%, 71%, and 56% energy consumption compared to the traditional SIMD.

6.4.5 Acceleration Mode Selection

Our experiments so far have focused on the overall performance of the Libra architecture compared to other architectures, showing considerable performance enhancement. In this section, we evaluate the effectiveness of flexible lane mapping to answer the question if Libra really needs to provide various intermediate sizes of logical lanes between SIMD and VLIW. Figure 6.13(a) shows the execution time distribution at different logical lane sizes for the three application domains on the 16, 32, and 64-PE Libra. On average, all available modes are used for considerable fraction of time and no dominating logical lane size exists, which proves the effectiveness of flexible lane mapping. Furthermore, the lane sizes are selected adaptive to the domain characteristics. For vision benchmarks, 2-PE small sized logical lane is dominant because most loops are small and memory operation dominant. In media benchmarks, large logical lanes are used for a high fraction of the execution because of lack of DLP. Game physics uses a 4-PE logical lane in substantial fraction to execute high-DLP loops with some ILP. Figure 6.13(b) compares the normalized performance of Libra to that when only one specific logical lane configuration is allowed to execute benchmarks. The results of this graph further prove the effectiveness of flexibility by showing that any fixed mode execution cannot win over the flexible execution.

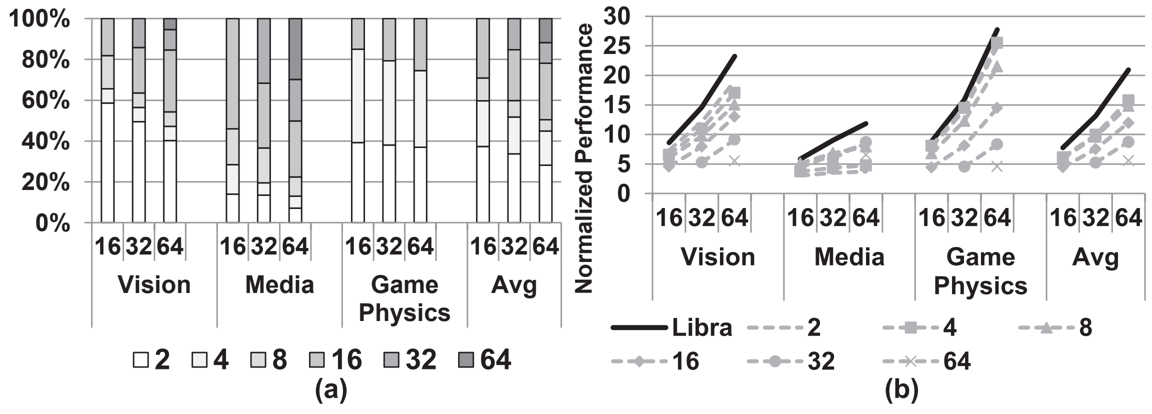


Figure 6.13: Mode selection: (a) execution time distribution at different logical lanes, (b) flexible vs. fixed execution.

6.4.6 Multi-threading Effectiveness

As discussed in Section 6.3.4, a simple multi-threading functionality is added to Libra. In this section, we evaluate the effectiveness of this functionality. Figure 6.14(a) shows the performance improvement on SIMDizable loops only, since this technique can be only applied to SIMDizable loops. On average, a performance gain of 12-16% is achieved, and this is up to 28% more effective in vision benchmarks because the majority of loops are small and multi-threading is most effective in small size logical lane mapping. Figure 6.14(b) shows the execution time distribution for different logical lane sizes when multi-threading is applied. Compared to Figure 6.13(a), a substantial amount of 2 and 4-PE logical lane execution is substituted with multi-threading. Overall, multi-threading is effective for small logical lanes when executing SIMDizable loops.

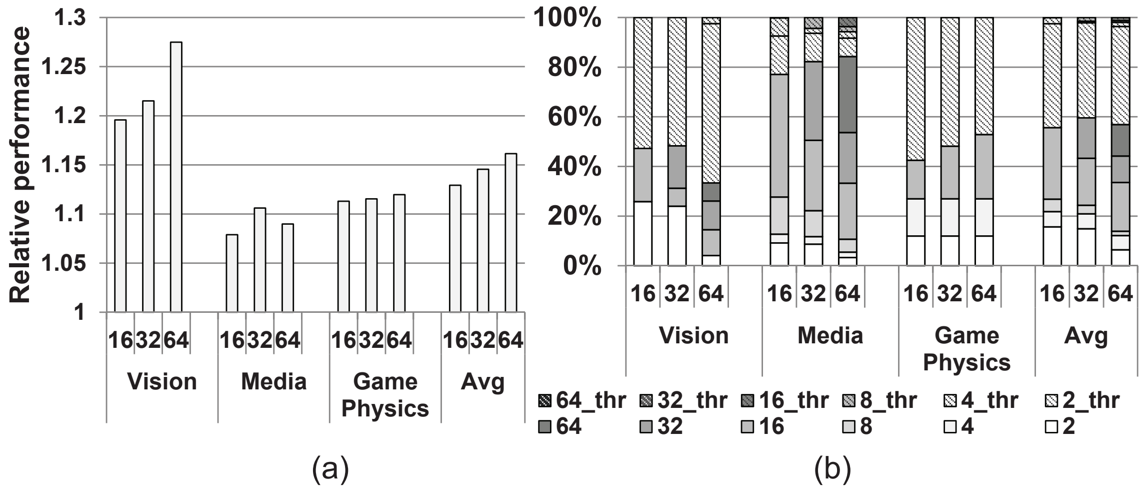
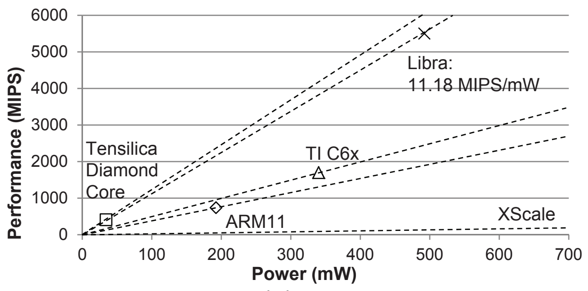
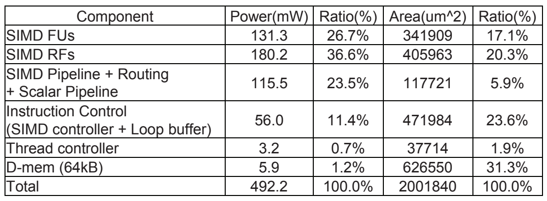


Figure 6.14: Multi-threading effectiveness: (a) performance improvement for SIMDizable loops, (b) execution time distribution at different logical lanes.



(a)



Component	Power(mW)	Ratio(%)	Area(um^2)	Ratio(%)
SIMD FUs	131.3	26.7%	341909	17.1%
SIMD RFs	180.2	36.6%	405963	20.3%
SIMD Pipeline + Routing + Scalar Pipeline	115.5	23.5%	117721	5.9%
Instruction Control (SIMD controller + Loop buffer)	56.0	11.4%	471984	23.6%
Thread controller	3.2	0.7%	37714	1.9%
D-mem (64kB)	5.9	1.2%	626550	31.3%
Total	492.2	100.0%	2001840	100.0%

(b)

Figure 6.15: (a) Power/Performance comparison, and (b) power and area breakdown of the 32-PE Libra architecture.

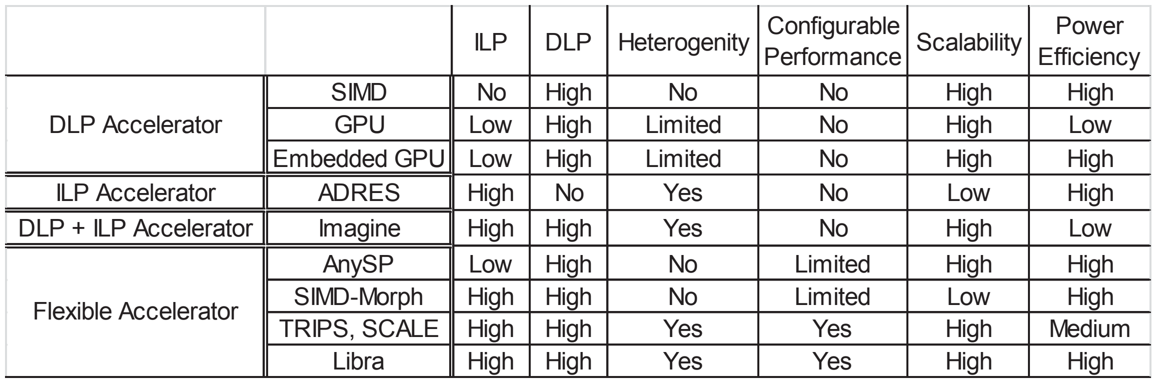
6.4.7 Power and Area Measurement

We measured the average power when the 32-PE Libra architecture executes the H.264 benchmark at 500 MHz. A power and an area consumption breakdown for various components that are part of the architecture are shown in Figure 6.15(b). Compared to the normal SIMD, the power consumption of the routing logic is larger due to its dynamic configurability, but FU power is smaller due to the smaller number of expensive units. A SIMD controller and four loop buffers, and a thread controller are added to a cluster. The power consumption of a SIMD controller and four loop buffers is substantial because the loop buffer is implemented as 64-entry wide two-port SRAM and the data is read every-cycle. In addition to this, the thread controller also consumes 0.7% of total power because the sequence table is a 256 entry 8 bit two-port SRAM. The total area of the 32-PE Libra architecture is 2.0 mm^2.

Based on the power and performance data, we compared the efficiency of Libra to other architectures using data shown in [32]. Based on Figure 6.15(a), the Libra architecture achieves 11.18 MIPs/mW and most of the other well-known solutions show lower efficiency. The Tensilica Diamond Core is slightly more efficient than the Libra architecture, but the actual performance is not enough to successfully execute compute-intensive media applications.

6.5 Related Works

Many previous works have focused on accelerators to address the challenges of improving computing efficiency. Some exploit only one type of parallelism and others introduce some flexibility to support more than one type of parallelism. Figure 6.16 compares and shows the major differences between Libra and prior works.



		ILP	DLP	Heterogenity	Configurable Performance	Scalability	Power Efficiency
DLP Accelerator	SIMD	No	High	No	No	High	High
	GPU	Low	High	Limited	No	High	Low
	Embedded GPU	Low	High	Limited	No	High	High
ILP Accelerator	ADRES	High	No	Yes	No	Low	High
DLP + ILP Accelerator	Imagine	High	High	Yes	No	High	Low
Flexible Accelerator	AnySP	Low	High	No	Limited	High	High
	SIMD-Morph	High	High	No	Limited	Low	High
	TRIPS, SCALE	High	High	Yes	Yes	High	Medium
	Libra	High	High	Yes	Yes	High	High

Figure 6.16: Comparison to prior work

Accelerators for multimedia usually focus on one type of parallelism without adaptive configuration. Conventional SIMD [17, 59] only supports DLP and misses the opportunity of improving performance with other form of parallelism. By Amdahl's law, low-DLP regions quickly become the bottleneck of applications. Conventional SIMD also wastes expensive resources due to imbalanced utilization. While the latest GPUs [70, 69] support the limited level of heterogeneity and embedded GPUs such as Qualcomm Adreno [4] and

ARM Mali [1] are power-efficient, GPUs have the same fundamental weakness as other data-parallel accelerators.

ILP accelerators, such as ADRES [64], tackle the problem in another way by exploiting ILP with the help of modulo scheduling. Even though it has high scalability by providing distributed architecture, the throughput quickly saturates as the number of resources increases due to the scheduling difficulty as shown in PPA [74]. Hybrid accelerators such as the Stanford Imagine [8] use the VLIW-SIMD scheme but the fixed configuration frequently incurs a lack or waste of resources.

Recently, several architectures have tried to embrace flexibility in a conventional SIMD accelerator in order to support multiple application domains with different characteristics. AnySP [90] targets mobile applications such as 4G wireless communication and high-definition video coding. AnySP achieves the goal efficiently by simply chaining two SIMD lanes and supporting limited thread level parallelism, but underutilization in low-DLP loops is still inevitable due to the lack of general policy to support ILP. SIMD-Morph [30] employs subgraph matching to accelerate sequential code region. Despite their fair performance gain, their simple ILP/DLP mode transition policy cannot adaptively adjust the degree of ILP and DLP inside a specific code region. For example, it is impossible to fully utilize the SIMD-Morph for a low-DLP code region since an insufficient degree of DLP cannot be supplemented by ILP exploitation, while Libra can. In addition, they are still homogeneous SIMD, and therefore, cannot improve utilization and power efficiency.

TRIPS [81] and SCALE [49] are also similar to this work. TRIPS integrates ILP, DLP and TLP, and SCALE exploits both vector parallelism and TLP. They are targeting more the desktop/server space, and therefore, need expensive architectural features such as inter-cluster networks, additional multiple fetch units, and specialized caches for generality. However, Libra focuses on more efficient execution of loops with minimal hardware modifications.

Avoiding resource contention of expensive instructions by pipelined execution is also introduced in an instruction-systolic array architecture [76]. However, systolic execution may incur severe performance degradation on high number of PEs because of the pipelining delay, while Libra limits sharing only between two logical lanes in full DLP mode.

6.6 Summary

The popularity of mobile computing platforms has led to the development of feature-packed devices that support a wide range of software applications with high single-thread performance and power efficiency requirements. To efficiently achieve both objectives, SIMD-based architectures are currently proposed. However, the SIMD is not able to efficiently support a wide range of mobile applications due to several limiting factors: limited availability of high trip count vector loops and the homogeneous nature of the hardware. To enhance the applicability of SIMD and improve its inherent energy efficiency, we break two long-standing traditions of SIMD design: identical lanes and static configuration. The *Libra* accelerator adapts the SIMD lane resources to target application. The Libra architecture customizes the lane configuration based on the loop structure from many resource-constrained logical lanes for highly data-parallel loops, to a modest number of lanes with moderate resources, up to a single resource-rich logical lane that is effectively a multicluster VLIW. A 32-PE Libra system achieves an average 1.58x speedup over the traditional

SIMD system, and the gain becomes higher as the number of PEs increases. Through a judicious mechanism to share expensive resources, Libra also achieves a 29% reduction in energy compared to the SIMD system. We believe that as industry requires higher performance with high energy efficiency, the proposed scalable architecture puts more resources to work in order to meet this demand.

CHAPTER 7

Conclusion

The Libra accelerator is a unified loop accelerator that can effectively support future mobile applications with varying performance requirements and characteristics. Libra can dynamically tune ILP/DLP-support capabilities in order to successfully support ILP-only, DLP-only, and ILP/DLP-mixed applications. Also, Libra's simple hardware implementation and its distributed nature achieve high energy-efficiency with competitive performance at a high degree of scalability which other current accelerators hardly realize.

In this work, a number of compiler optimizations are presented for execution models supported in the Libra accelerator. There are several crucial performance bottlenecks in exploiting ILP, DLP, and Task-level parallelism in current accelerator models. Thus, three compilation techniques are proposed to enhance the quality of schedules over the traditional approach.

The SIMD Defragmenter successfully increases the DLP coverage by finding potential DLP opportunities from the code written in the form of ILP. The data packing/unpacking overhead can be overcome by SIMDizing in groups of parallel compatible instructions (subgraphs) to maximize SIMD gain. On a 16-lane SIMD execution, experimental results show that SIMD defragmentation achieves a 1.6x mean speedup over traditional loop vectorization and a 31% gain over prior research approaches for converting ILP to DLP.

Dynamic operation fusion is proposed to enable a CGRA model to effectively accelerate latency-constrained code regions such as non-loop, outer-loop, and recurrence-constrained loop code. Dynamic operation fusion is enabled through the combination of a small bypass network added between functional units in a conventional CGRA and a sub-cycle modulo scheduler to automatically identify opportunities for fusion. Results show that dynamic operation fusion reduced total application run-time by up to 17% on a 4x4 CGRA execution.

Based on the previous compilation optimizations, a high level compilation framework is introduced that maximizes application throughput with hybrid resource partitioning of a dynamic multicore accelerator based on the stream graph modulo scheduling algorithm. Static partitioning handles part of the resource assignment, but this is followed up by dynamic partitioning to identify idle resources and put them to use. Experimental results show that real-time media applications can take advantage of the static and dynamic configurability of the PPA system for increased throughput.

While these optimizations attack the major performance bottlenecks of various acceleration models, using multiple solutions still incurs three critical problems: static power/area overhead, low execution efficiency due to the application complexity, and higher software development costs. In response, we decided to propose a new unified accelerator for mobile applications. To achieve this, we find four key issues for future accelerators: homogeneous versus heterogeneous functionality, interconnect topologies, simple versus complex processing elements, and scalar versus vector memory support. Then, the proper future directions for those issues are proposed based on deep application analysis.

Under the guidance of the above study, we propose the Libra accelerator, an accelerator that allows flexible execution of loops by customizing the configuration and adapting resources to the underlying characteristics of the application. Libra achieves the goal using datapath heterogeneity and dynamic configurability. First, Libra is composed of a non-uniform lane structure for power efficiency: only a subset of lanes have expensive but infrequently used execution units. Transparent sharing mechanisms provide the appearance of uniformity. Second, dynamic configurability repartitions resources to match execution patterns at run-time to maintain high utilization. In Libra, a group of logical lanes is executed as SIMD, while the lane itself is composed of a group of processing elements (PEs) similar to a CGRA. DLP is exploited in the form of parallel execution across the logical lanes, and ILP is exploited inside each logical lane. In essence, Libra provides a spectrum of resource configurations from a large number of skinny lanes for executing code with high levels of DLP to a small number of fat lanes for code with low levels of DLP. Experimental results show that 32-PE Libra outperforms the traditional SIMD system by average of 1.58x, and the performance is linearly scalable as the size increases.

To conclude, we believe that a unified accelerator substrate would eliminate major problems from which today's mobile computing platforms with multiple accelerators on a chip suffer. However, the unified accelerator must support a diverse set of applications, loops, and acyclic code regions to be performance competitive. The architectural and compiler solutions presented in this dissertation provide an important step towards the future unified mobile solution.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] ARM Mali Graphics Hardware
- http://www.arm.com/products/multimedia/mali-graphics-hardware/. 177

[2] Cuda toolkit. - http://developer.nvidia.com/cuda-toolkit. 118, 142

[3] Glbenchmark - http://www.glbenchmark.com/. 118, 142

[4] Qualcomm Adreno
- http://www.qualcomm.com/solutions/multimedia/graphics/. 176

[5] Samsung advanced institute of technology
- http://www.sait.samsung.co.kr/. 167

[6] T. V. Aa, M. Palkovic, M. Hartmann, P. Raghavan, A. Dejonghe, and L. V. der Perre. A multi-threaded coarse-grained array processor for wireless baseband. In *Proc. of the 2011 IEEE Symposium on Application Specific Processors*, pages 102–107, June 2011. 123, 127, 132

[7] T. Adam, K. Chandy, and J. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, Dec. 1974. 161, 167

[8] J. H. Ahn et al. Evaluating the Imagine stream architecture. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, pages 14–25, June 2004. 177

[9] A. Aletà, J. Codina, J. Sánchez, and A. González. Graph-partitioning based instruction scheduling for clustered processors. In *Proc. of the 34th Annual International Symposium on Microarchitecture*, pages 150–159, Dec. 2001. 47

[10] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures: A dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002. 46

[11] M. Alvarez, E. Salami, A. Ramirez, and M. Valero. A performance characterization of high definition digital video decoding using h.264/avc. In *2005 IEEE International Symposium on Workload Characterization*, pages 24–33, Oct. 2005. 121, 139

[12] G. Ansaloni, P. Bonzini, and L. Pozzi. Design and architectural exploration of expression-grained reconfigurable arrays. In *Proc. of the 2008 IEEE Symposium on Application Specific Processors*, pages 26–33, June 2008. 128

[13] G. Ansaloni, P. Bonzini, and L. Pozzi. Egra: A coarse grained reconfigurable architectural template. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 19(6):1062–1074, June 2011. 116, 128

[14] G. Ansaloni, L. Pozzi, K. Tanimura, and N. Dutt. Slack-aware scheduling on coarse grained reconfigurable arrays. In *Proc. of the 2011 Design, Automation and Test in Europe*, 2011. 128

[15] R. Barik, J. Zhao, and V. Sarkar. Efficient Selection of Vector Instructions Using Dynamic Programming. In *Proc. of the 43rd Annual International Symposium on Microarchitecture*, Dec. 2010. 41

[16] K. Berkel, F. Heinle, P. Meuwissen, K. Moerman, and M. Weiss. Vector processing as an enabler for software-defined radio in handheld devices. *EURASIP Journal Applied Signal Processing*, 2005(1):2613–2625, 2005. 1, 10, 83

[17] H. Bluethgen, C. Grassmann, W. Raab, and U. Ramacher. A programmable platform for software-defined radio. In *Intl. Symposium on System-on-a-Chip*, pages 15–20, Nov. 2003. 1, 10, 83, 138, 176

[18] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999. 113

[19] P. Bonzini, G. Ansaloni, and L. Pozzi. Compiling custom instructions onto expression-grained reconfigurable architectures. In *Proc. of the 2008 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 51–59, Oct. 2008. 58, 80

[20] F. Bouwens, M. Berekovic, B. D. Sutter, and G. Gaydadjiev. Architecture enhancements for the ADRES coarse-grained reconfigurable array. In *Proc. of the 2008 International Conference on High Performance Embedded Architectures and Compilers*, pages 66–81, Jan. 2008. 121

[21] T. Callahan, J. Hauser, and J. Wawrzynek. The Garp architecture and C compiler. *IEEE Computer*, 33(4):62–69, Apr. 2000. 52, 80

[22] C. Canali, M. Colajanni, and R. Lancellotti. Performance evolution performance evolution. *Internet Computing Magazine, IEEE*, 13(2):60–68, Mar. 2009. 115, 116

[23] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned register files for VLIWs: A preliminary analysis of tradeoffs. In *Proc. of the 25th Annual International Symposium on Microarchitecture*, pages 103–114, Dec. 1992. 47

[24] M. Chu, K. Fan, and S. Mahlke. Region-based hierarchical operation partitioning for multicluster processors. In *Proc. of the SIGPLAN ’03 Conference on Programming Language Design and Implementation*, pages 300–311, June 2003. 47

[25] N. Clark et al. Application-specific processing on a general-purpose core via transparent instruction set customization. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 30–40, Dec. 2004. 47, 59, 79, 129

[26] N. Clark et al. An architecture framework for transparent instruction set customization in embedded processors. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 272–283, June 2005. 47, 52, 57, 59, 129

[27] N. Clark, A. Hormati, and S. Mahlke. VEAL: Virtualized execution accelerator for loops. In *Proc. of the 35th Annual International Symposium on Computer Architecture*, pages 389–400, June 2008. 79

[28] N. Clark, A. Hormati, S. Mahlke, and S. Yehia. Scalable subgraph mapping for acyclic computation accelerators. In *Proc. of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 147–157, Oct. 2006. 47

[29] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. In *Proc. of the 36th Annual International Symposium on Microarchitecture*, pages 129–140, Dec. 2003. 30, 32, 47

[30] G. Dasika, M. Woh, S. Seo, N. Clark, T. Mudge, and S. Mahlke. Mighty-morphing power-simd. In *Proc. of the 2010 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Oct. 2010. 177

[31] C. Ebeling et al. Mapping applications to the RaPiD configurable architecture. In *Proc. of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 106–115, Apr. 1997. 79, 112

[32] K. Fan, M. Kudlur, G. Dasika, and S. Mahlke. Bridging the computation gap between programmable processors and hardwired accelerators. In *Proc. of the 15th International Symposium on High-Performance Computer Architecture*, pages 313–322, Feb. 2009. 175

[33] J. Glossner, E. Hokenek, and M. Moudgill. The sandbridge sandblaster communications processor. In *Proc. of the 2004 Workshop on Application Specific Processors*, pages 53–58, Sept. 2004. 1, 10, 83

[34] S. Goldstein et al. PipeRench: A coprocessor for streaming multimedia acceleration. In *Proc. of the 26th Annual International Symposium on Computer Architecture*, pages 28–39, June 1999. 79, 112

[35] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. Meli, A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, Oct. 2002. 83

[36] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 151–162, 2006. 83, 84, 113

[37] J. Hiser, S. Carr, and P. Sweany. Global register partitioning. In *Proc. of the 9th International Conference on Parallel Architectures and Compilation Techniques*, pages 13–23, Oct. 2000. 47

[38] A. Hormati et al. Exploiting narrow accelerators with data-centric subgraph mapping. In *Proc. of the 2007 International Symposium on Code Generation and Optimization*, pages 341–353, Mar. 2007. 80

[39] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose. Microarchitectural techniques for power gating of execution units. In *Proc. of the 2004 International Symposium on Low Power Electronics and Design*, pages 32–37, Aug. 2004. 18

[40] IBM. *Cell Broadband Engine Architecture*, Mar. 2006. 113

[41] Intel. Intel compiler, 2009. software.intel.com/en-us/intel-compilers/. 16, 46, 118, 143

[42] E. Ipek, M. Kirman, N. Kirman, and J. Martinez. Core fusion: Accommodating software diversity in chip multiprocessors. In *Proc. of the 34th Annual International Symposium on Computer Architecture*, pages 186–197, 2007. 86, 111

[43] H. Kalva. The H.264 video coding standard. *IEEE MultiMedia*, 13(4):86–90, 2006. 118, 142

[44] C. Kim, S. Sethumadhavan, M. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable lightweight processors. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, pages 381–393, Dec. 2007. 86, 111

[45] Y. Kim, M. Kiemb, C. Park, J. Jung, and K. Choi. Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization. In *Proc. of the 2005 Design, Automation and Test in Europe*, pages 12–17, Mar. 2005. 116

[46] Y. Kim, J. Lee, A. Shricastava, and Y. Paek. Memory access optimization in compilation for coarse-grained reconfigurable architectures. *ACM Transactions on Design Automation of Electronic Systems*, 16(4), Oct. 2011. 132

[47] Y. Kim and R. N. Mahapatra. A new array fabric for coarse-grained reconfigurable architecture. In *Proc. of the 34th Euromicro Conference*, pages 584–591, Sept. 2008. 79

[48] Y. Kim, I. Park, K. Choi, and Y. Paek. Power-conscious configuration cache structure and code mapping for coarse-grained reconfigurable architecture. In *Proc. of the 2006 International Symposium on Low Power Electronics and Design*, Oct. 2006. 79

[49] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. The vector-thread architecture. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, 2004. 47, 177

[50] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proc. of the SIGPLAN '08 Conference on Programming Language Design and Implementation*, pages 114–124, June 2008. 83, 85, 89, 113

[51] A. Lambrechts, P. Raghavan, M. Jayapala, F. Catthoor, and D. Verkest. Energy-aware interconnect optimization for a coarse grained reconfigurable processor. In *Proc. of the 2008 International Conference on VLSI Design*, pages 201–207, Jan. 2008. 121

[52] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proc. of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 145–156, June 2000. 11, 26, 27, 30, 38, 47

[53] S. Larsen and S. Amarasinghe. Increasing and detecting memory address congruence. In *Proc. of the 11th International Conference on Parallel Architectures and Compilation Techniques*, pages 18–29, Sept. 2002. 47

[54] C. Lee, M. Potkonjak, and W. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proc. of the 30th Annual International Symposium on Microarchitecture*, pages 330–335, 1997. 11

[55] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987. 84

[56] J. Lee, K. Choi, and N. Dutt. Compilation approach for coarse-grained reconfigurable architectures. *IEEE Journal of Design & Test of Computers*, 20(1):26–33, Jan. 2003. 80

[57] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-time scheduling of instruction-level parallelism on a RAW machine. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, Oct. 1998. 52, 113

[58] W.-J. Lee, S.-H. Lee, J.-H. Nah, J.-W. Kim, Y. Shin, J. Lee, and S.-Y. Jung. Sgrt: A scalable mobile gpu architecture based on ray tracing, 2012. 123

[59] Y. Lin et al. Soda: A low-power architecture for software radio. In *Proc. of the 33rd Annual International Symposium on Computer Architecture*, pages 89–101, June 2006. 1, 10, 14, 83, 138, 142, 167, 176

[60] Y. Lin et al. Soda: A high-performance dsp architecture for software-defined radio. *IEEE Micro*, 27(1):114–123, Jan. 2007. 38

[61] G. Lu et al. The MorphoSys parallel reconfigurable system. In *Proc. of the 5th International Euro-Par Conference*, pages 727–734, 1999. 2, 50, 79, 114

[62] A. Lungu, P. Bose, A. Buyuktosunoglu, and D. J. Sorin. Dynamic power gating with quality guarantees. In *Proc. of the 2009 International Symposium on Low Power Electronics and Design*, pages 377–382, Aug. 2009. 18

[63] N. Madan, A. Buyuktosunoglu, P. Bose, and M. Annavaram. A case for guarded power gating for multi-core processors. In *Proc. of the 17th International Symposium on High-Performance Computer Architecture*, Feb. 2011. 18

[64] B. Mei et al. ADRES: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In *Proc. of the 2003 International Conference on Field Programmable Logic and Applications*, pages 61–70, Aug. 2003. 79, 132, 177

[65] B. Mei et al. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *Proc. of the 2003 Design, Automation and Test in Europe*, pages 296–301, Mar. 2003. 2, 50, 52, 80, 112, 114

[66] B. Mei, A. Lambrechts, J. Y. Mignolet, D. Verkest, and R. Lauwereins. Architecure exploration for a reconfigurable architecture template. In *Proc. of the 2005 Design, Automation and Test in Europe*, pages 90–101, Mar. 2005. 2, 8, 50, 53, 55, 83, 114, 117, 120

[67] D. Nuzman et al. Vapor simd: Auto-vectorize once, run everywhere. In *Proc. of the 2011 International Symposium on Code Generation and Optimization*, pages 151–160, Apr. 2011. 12

[68] D. Nuzman and A. Zaks. Outer-loop vectorization - revisited for short simd architectures. In *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 2–11, 2008. 46, 47

[69] NVIDIA. NVIDIAs Next Generation CUDA Compute Architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. 176

[70] NVIDIA. GeForce GTX 200 GPU architectural overview, 2008. http://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf. 176

[71] OpenIMPACT. The OpenIMPACT IA-64 compiler, 2005. http://gelato.uiuc.edu/. 38, 120, 167

[72] H. Park, K. Fan, M. Kudlur, and S. Mahlke. Modulo graph embedding: Mapping applications onto coarse-grained reconfigurable architectures. In *Proc. of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 136–146, Oct. 2006. 2, 34, 52, 80

[73] H. Park, K. Fan, S. Mahlke, T. Oh, H. Kim, and H. seok Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 166–176, Oct. 2008. 2, 34, 52, 65, 85, 120, 133, 154, 167

[74] H. Park, Y. Park, and S. Mahlke. Polymorphic pipeline array: A flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *Proc. of the 42nd Annual International Symposium on Microarchitecture*, pages 370–380, Dec. 2009. 83, 87, 88, 95, 104, 116, 123, 127, 177

[75] J. Park, D. Shin, N. Chang, and M. Pedram. Accurate modeling and calculation of delay and energy overheads of dynamic voltage scaling in modern high-performance microprocessors. In *Proc. of the 2010 International Symposium on Low Power Electronics and Design*, pages 419–424, Aug. 2010. 18

[76] J. Park, H. Yang, G. Park, S. Kim, and C. C. Weems. An instruction-systolic programmable shader architecture for multi-threaded 3d graphics processing. *Journal of Parallel and Distributed Computing*, 70(11):1110–1118, 2010. 178

[77] Y. Park, H. Park, and S. Mahlke. Cgra express: Accelerating execution using dynamic operation fusion. In *Proc. of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 271–280, Oct. 2009. 92

[78] M. Quax, J. Huisken, and J. Meerbergen. A scalable implementation of a reconfigurable WCDMA RAKE receiver. In *Proc. of the 2004 Design, Automation and Test in Europe*, pages 230–235, Mar. 2004. 2, 50, 114

[79] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, Nov. 1994. 55, 83, 89, 133, 161

[80] R. M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, Jan. 1978. 138

[81] K. Sankaralingam et al. Exploiting ILP, TLP, and DLP using polymorphism in the TRIPS architecture. In *Proc. of the 30th Annual International Symposium on Computer Architecture*, pages 422–433, June 2003. 177

[82] F. Semiconductor. Altivec, 2009. www.freescale.com/altivec. 38

[83] J. Shin, J. Chame, and M. W. Hall. Compiler-controlled caching in superword register files for multimedia extension architectures. In *Proc. of the 11th International Conference on Parallel Architectures and Compilation Techniques*, pages 45–55, 2005. 47

[84] D. Talla, L. K. John, and D. Burger. Bottlenecks in multimedia processing with simd style extensions and architectural enhancements. *IEEE Transactions on Computers*, 52(8):1015–1031, 2003. 11

[85] M. B. Taylor et al. The Raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*, 22(2):25–35, 2002. 2, 50, 114

[86] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, Dec. 2007. 112, 113

[87] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. of the 2002 International Conference on Compiler Construction*, pages 179–196, 2002. 84, 85

[88] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. L. S. Garcia, S. Belongie, and M. B. Taylor. SD-VBS: The san diego vision benchmark suite. In *2009 IEEE International Symposium on Workload Characterization*, pages 55–64, Oct. 2009. 141

[89] M. Woh et al. From SODA to scotch: The evolution of a wireless baseband processor. In *Proc. of the 41st Annual International Symposium on Microarchitecture*, pages 152–163, Nov. 2008. 1, 10, 83

[90] M. Woh, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. AnySP: Anytime Anywhere Anyway Signal Processing. In *Proc. of the 36th Annual International Symposium on Computer Architecture*, pages 128–139, June 2009. 19, 47, 138, 177

[91] H. Zhong, K. Fan, S. Mahlke, and M. Schlansker. A distributed control path architecture for VLIW processors. In *Proc. of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 197–206, Sept. 2005. 112