MEMORY OPTIMIZATIONS OF EMBEDDED APPLICATIONS FOR ENERGY EFFICIENCY

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Jongsoo Park

May 2011

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

William Dally, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Monica Lam, Co-Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christoforos Kozyrakis

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost Graduate Education

Abstract

The current embedded processors often do not satisfy increasingly demanding computation requirements of embedded applications within acceptable energy efficiency, whereas application-specific integrated circuits require excessive design costs. In the Stanford Elm project, it was identified that instruction and data delivery, not computation, dominate the energy consumption of embedded processors. Consequently, the energy efficiency of delivering instructions and data must be sufficiently improved to close the efficiency gap between application-specific integrated circuits and programmable embedded processors.

This dissertation demonstrates that the compiler and run-time system can play a crucial role in improving the energy efficiency of delivering instructions and data. Regarding instruction delivery, I present a compiler algorithm that manages L0 instruction scratch-pad memories that reside between processor cores and L1 caches. Despite the lack of tags, the scratch-pad memories with our algorithm can achieve lower miss rates than caches with the same capacities, saving significant instruction delivery energy.

Regarding data delivery, I present methods that minimize memory-space requirements for parallelizing stream applications, applications that are commonly found in the embedded domain. When stream applications are parallelized in pipelining, large enough buffers are required between pipeline stages to sustain the throughput (e.g., double buffering). For static stream applications where production and consumption rates of stages are close to compile-time constants, a compiler analysis is presented, which computes the minimum buffer capacity that maximizes the throughput. Based on this analysis, a new static stream-scheduling algorithm is developed, which yields considerable speed-up and data delivery energy saving compared to a previous algorithm. For dynamic stream applications, I present a dynamically-sized array-based queue design that achieves speed-up and data delivery energy saving compared to a linked-list based queue design.

Acknowledgement

First and foremost, I would like to thank my advisor, Professor William Dally. Bill is an amazing person at many moments — at ski slopes or when presenting a deep insight in front of a large audience. Among his many amazing respects, what I appreciate most is his consistent optimism and belief in students' ability to solve important problems. When we do research, things do not work as we expected most of the time. I realized that an important virtue to overcome such challenging times is not being disappointed too easily and believing in solving the problem eventually. He is also a very dedicated teacher and research advisor. He had held weekly one-to-one meetings even during his leave of absence for the chief scientist position at NVIDIA. During six years (from the very first my school week at Stanford when I first met with Bill to the final meetings for comments on this dissertation), he has been amazingly consistent at carefully listening students' concern. His deep, broad insights and hard working has raised the bar for assessing my own work, which I believe one of the most valuable lessons I have learned during my PhD.

I would also like to thank my committee members, Professor Monica Lam and Professor Christos Kozyrakis. I was fortunate enough to had an opportunity to be a TA of one of Professor Lam's classes, where I have learned valuable lessons from interacting with her, particularly from her passion on teaching and research. I am also very glad to had a chance to take Professor Kozyrakis's architecture classes, which provided solid background on my research.

The CVA group is a unique environment, where students are working on vertically-diverse fields (i.e., from circuits to software). Not only have my peer students taught me valuable technical insights, but also they have been considerate while I was learning how to communicate well with colleagues in Western cultures. I found that the CVA group is one of the best research groups in Stanford EE in terms of friendly interactions including ski trips. It was also a precious experience to work in a project with many people. I would like to thank James Balfour and David Black-Schaffer, who are the founding members of the Elm project and have helped me in many respects throughout the project. Christine Avanessians, Clinton Buie, Jooseong Kim, and Youngsik Kim helped me on the programming system of the Elk project. I would also like to thank James Chen, who has shared the office with me for several years. Although he had many struggles with companies for taping out his chips, he has maintained an incredibly optimistic attitude, which has helped me to stay positive as well. There were occasional lunches at Ike's place with James Chen, Curtis Harting, and Ted Jiang. Although Ike's place is notorious for long lines, I really enjoyed joyful lunches with delicious sandwiches. I would like to thank former CVA members — Jung-Ho Ahn, Patrick Chiang, Abhishek Das, Mattan Erez, Paul Hartke, John Kim, Tim Knight, Jiyoung Park, and Manman Ren — who had kindly guided me through my earlier years. I also would like to thank other current CVA members — Daniel Becker, George Michelogiannakis, Milad Mohammadi, and Vishal Parikh — who have made my life as a CVA member enjoyable and made myself proud of being a CVA-er.

When people think of the Bay Area, the first thing coming up in their mind is usually nice weather and beautiful scenary. I was fortunate enough to enjoy not only the nice environment but also valuable interaction with many good friends. I cannot appreciate enough that I can talk face-to-face anytime with my friends whom I have known for more than 10 years even in a place far from my home country. Particularly, I thank Yeul and Eunsun for their help on settling down when I first came to Stanford, and Taehoon and Wanki for making a great year while we were roommates.

I also deeply appreciate the friendship with Marianne. We started knowing each other through the English-in-Action program but became friends, and she gave me many precious advices. Doing a PhD is a long-haul journey like marathon, and a person told me that everybody develops at least one hobby during the course. Occasional excursions to beautiful hiking trails in the Sierra Nevada has helped me to be periodically refreshed and maintain a healthy life as a graduate student. I thank Marianne for introducing those places so that I can enjoy the beauty of nature. I also thank Seokchang for teaching me photography and going to campings with me so that I can experience the nature closely. I have enjoyed running the Campus Drive with Kahye, and I hope that we can successfully finish a hiking to the Mountain Whitney and a half marathon scheduled on this July. Woongki Baek was my first roommate with whom I took many classes and studied for Quals together. With Jiwon Seo, I continued our friendship from our college robotics club and discussed many research ideas.

I would like to thank financial support from the Samsung Scholarship, the Semiconductor Research Corporation under Grant 2007-HJ-1591, and the National Science Foundation under Grant CNS-0719844. Evelin Sullivan and Claude Reichard have provided me valuable feedback on my writings including this dissertation. Professor Jaejin Lee at Seoul National University gave me an opportunity to experience compiler research before I came to Stanford.

Last but not least, I would like to thank my family. Without them, I would not be able to come to Stanford. I cannot give enough thanks to my wife, Hyejun. My time at Stanford is full of happy memories, and many of them are from Hyejun. While writing this acknowledgement, I realized multiple times that how lucky I have been at Stanford to meet such great people, and I hope this can express at least a part of my gratitude to them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Energy Efficiency Challenge in Embedded Computing

Improving energy efficiency is the one of the most critical challenges we face in contemporary computer architectures. In many domains, energy consumption is the primary constraint that prevents us from solving increasingly advanced problems that require ever higher computation power. The high performance computing community recognizes energy efficiency as the biggest challenge to achieve exa-scale computing that facilitates important scientific understandings such as global climate change, safe nuclear reactor design, and drug discovery [89]. Data centers consumed 1.5% of the total US energy use in 2006 [150]. Their electric bills for three years are comparable to hardware costs [106], and the proportion of electric bills in the total cost of ownership is expected to grow [17].

Energy efficiency plays a similarly, if not more, crucial role in embedded computing [34]. The energy efficiency of embedded devices directly affects their usability since many of them are operated by batteries. The capacity of batteries is not significantly improving [11], and they are limited by embedded devices' tight constraints with respect to form factor and manufacturing cost. This energy constraint conflicts with increasing computation demands of embedded devices as they adopt more sophisticated algorithms [137,152] and target wider application areas, sometimes substituting for personal computers.

In order to satisfy the demanding computation requirement within acceptable energy efficiency, application-specific integrated circuits (ASICs) are often required [59, 73]. Unfortunately, as semiconductor processes advance, non-recurring engineering costs of designing, implementing, and testing a new ASIC become extremely high [109]. This is particularly problematic in domains such as medical equipment, which cannot afford the excessive cost of ASIC due to their limited market size [12].

The Elm project, in which the work described in this dissertation was conducted, aims to design a programmable system that avoids excessive non-recurring engineering cost while achieving energy efficiency comparable to that of ASICs. Although this is an ambitious goal which has yet to be met, this dissertation presents a few important steps toward the goal with respect to software mechanisms that efficiently manage the memory subsystem.

The Role of Software in Improving Energy Efficiency of Instruction and Data Delivery

In order to close the efficiency gap between ASICs and programmable processors, the inefficiencies in conventional programmable processors should be identified first. Balfour et al. [13] show that 42% and 28% of the energy is consumed for instruction and data delivery, respectively, in an embedded processor based on 32-bit SPARC architecture [58]. Balfour's thesis [12] also shows that the energy consumed for executing an `add` instruction is at least 10.2× the energy expended in a 32-bit adder. In other words, it is instruction and data delivery, not computation, that contributes to most of the energy consumption in embedded processors. Consequently, the energy efficiency of the memory subsystem through which instructions and data are delivered must be improved to close the efficiency gap.

A central theme of this dissertation is that the compiler and run-time system can significantly improve the energy efficiency of delivering instructions and data by efficiently managing resources whose controls are deliberately exposed to software.

This energy efficiency improvement relies on the compiler or run-time system's ability to reason about the locality of instruction and data access, which allows optimizations that save energy. Examples include avoiding tag checks by proving that the target instruction is already cached and using statically-allocated arrays by proving a tight bound on buffer capacities to avoid dynamic allocations. Of course, this software-based approach is not a silver bullet and should be complemented by other mechanisms such as dynamic voltage scaling [162], gating [129, 143], and hardware customization [48, 59]. It is also important to know what kind of facts the compiler and run-time system can prove for energy efficiency and which of them yields the largest gain. This dissertation focuses on two optimization opportunities found in the Elm project that lead to large energy savings, are not satisfactorily handled by existing approaches, and are algorithmically interesting. Brief descriptions on other optimizations investigated in the Elm project are provided in Section 2.1.

Improving Energy-efficiency of Instruction Delivery by Compiler-managed Instruction Scratch-pad Memory

Instruction delivery accounts for the largest fraction of the energy consumed by embedded processors according to Balfour et al. [12, 13]. Similarly, the instruction cache contributes to 27% and 29% of the total energy consumed in StrongARM [115] and TM3270 TriMedia VLIW processors [151], respectively. A typical instruction cache has high energy consumption because it occupies a significant fraction of the processor die area and is accessed every cycle.

Researchers have observed that the current cache hierarchy is primarily optimized for performance, and that extending the hierarchy by adding small caches (typically 1KB or smaller) between the L1 instruction cache and the processor saves energy [84]. Energy efficiency can be further improved by replacing the small caches with scratch-pad memories (SPMs), which we call L0 instruction SPMs. SPMs are compiler-managed stores in which no tags are used to associate locations in SPMs and therefore consume less energy per access than caches with the same capacity. Small caches between the L1 instruction cache and the processor are natural targets to be replaced with SPMs since the typical locality captured by such small stores comes from loops, which are analyzable by the compiler. Chapter 3 describes a compiler algorithm that manages L0 instruction SPMs and evaluates their energy savings compared to cache-based designs and previous SPM management algorithms.

Improving Energy-efficiency of Data Delivery in Stream Applications

Data delivery energy accounts for the second largest fraction of the energy consumed by embedded processors [12,13]. Since data accesses typically have higher miss rates than instructions, data locality optimizations have a larger impact on the performance. In addition, data locality optimizations have intimate relation to efficient parallelization. Therefore, data locality optimizations have been extensively studied [44,45,94,104,105,135,164], even though they pose challenges not found in analyzing locality of instructions: in contrast to instructions of a read-only nature, dependencies must be preserved with respect to data writes, which often involves inter-procedural pointer aliasing analyses. This dissertation focuses on a relatively understudied yet an important data locality optimization problem in the embedded domain — minimizing buffer space while sustaining the throughput of parallelized stream applications. Stream applications are commonly found in the embedded domain, whose typical examples are digital signal processing algorithms.

In embedded stream applications, pipeline parallelism plays a critical role in achieving satisfactory parallelization speed-ups. Embedded applications tend to process data sets smaller than those processed by high-performance computing counterparts. Consequently, solely exploiting data-level parallelism is not enough to fill increasing number of cores [51]. In addition, components such as the arithmetic encoder in H.264 or the Huffman encoder in JPEG or MP3 are not data parallel, and pipelining is typically the only type of parallelism that can be exploited. Even a small number of non data-parallel components can significantly limit the scalability of approaches that solely exploit data-level parallelism [51]. When pipeline stages contain large local states (e.g., texture data of pixel shaders in graphics pipelines), accessing the states from a few dedicated cores using pipeline parallelism can yield better locality than swapping in and out the states using only data-level parallelism [90]. Moreover, pipelining is also an effective form of parallelization for heterogeneous architectures.

We can map a stage to the component where it can be efficiently executed. When there is a pipeline stage that is a prominent bottleneck, hardware that is highly optimized for that particular stage can be incorporated (e.g., a rasterizer in a graphics pipeline).

Efficient pipeline parallelization however poses several challenges. Whereas challenges such as load imbalance can be resolved by existing load balancing scheduling algorithms such as work stealing [29], appropriately sizing queues between stages has been relatively understudied. Arbitrary "large enough" numbers are often chosen for inter-stage queue capacities. The queue capacity computation has particularly important consequence on energy efficiency, since queues too large will incur considerable amount of costly accesses to remote or off-chip memories. On the other hand, queues should have large enough capacities (e.g., double buffering) so that producers and consumers can work in parallel or the run-time variation of pipeline stages can be hidden. This is an example of traditional trade-off between storage and parallelism [8]: just as using more registers allows for more instruction-level parallelism, using more memory in inter-stage queues allows for more pipeline parallelism. Chapter 4 describes a compiler analysis that computes the minimum queue capacity that maximizes the throughput of static stream applications, applications in which production and consumption rates of stages are close to compile-time constants. Chapter 5 describes a dynamically-sized array-based concurrent queue design for dynamic stream applications.

1.2 Contributions

This dissertation makes six contributions: The first is a mechanism to reduce instruction delivery energy; the subsequent three are mechanisms to reduce data delivery energy in stream applications. The last two are with respect to infrastructure built for the Elm project.

Instruction Scratch-pad Memory Management (Chapter 3)

This dissertation presents a compiler algorithm that manages instruction scratch-pad memories (SPMs), whose instruction delivery energy saving is estimated to be 87%. In contrast to the previous SPM management algorithms, instruction SPMs with our algorithm achieve not only a lower access energy but also lower miss rates compared to caches with the same capacity. This ensures consistently higher energy efficiency than a competing design called filter caches over diverse memory hierarchy configurations found in embedded processors. In addition, I discuss reasons for the failure of previous algorithms to consistently achieve energy savings over filter caches, namely, because they neglect the conservative nature of compilers, unjustifiably rely on integer linear programming formulations, or do not take rigorous care in evaluation designs.

Queue Capacity Computation for Static Stream Applications (Chapter 4)

This dissertation presents a static analysis that computes minimum inter-actor queue capacities that sustain the maximum throughput of static stream applications. This analysis allows to reduce data delivery energy by minimizing costly remote or off-chip memory accesses while avoiding throughput degradation and deadlocks.

Buffer-space Efficient Static Stream Scheduling (Chapter 4)

This dissertation presents a static stream-scheduling algorithm called *team scheduling* that minimizes communication and synchronization overhead while avoiding costly remote or off-chip memory accesses, which in turn saves considerable data delivery energy in static stream applications. It is estimated that the team scheduling can achieve 27% higher throughput while requiring 33% less energy for data delivery than a previous stream-scheduling algorithm called SGMS [90] in a 16-core Elm processor. The team scheduling is based on the queue capacity computation analysis mentioned above, which allows to flexibly apply transformations that minimize communication and synchronization without incurring too large queues, throughput degradation, or deadlocks.

Dynamically-sized Queues with Minimal Memory Footprint (Chapter 5)

This dissertation presents a concurrent queue data structure called QED (Queue Enhanced with Dynamic sizing) that adjusts its capacity at run-time to maximize the throughput while minimizing the memory footprint of inter-actor queues, which in turn reduces the data delivery energy in dynamic stream applications. Whereas the queue capacity computation analysis mentioned above minimizes the memory footprint in parallel execution of static stream applications, QED does the same for dynamic stream applications. It is estimated that QED can achieve an average of 18% speed-ups and 15% savings in data delivery energy compared to a linked-list based queue design.

The Elk Stream Programming Language (Chapter 2)

This dissertation presents a stream programming language called Elk that is extended from StreamIt [145]. Compared to StreamIt, Elk supports multiple input and output streams per actor, which allows representations with less clutter in applications such as Data Encryption Standard. In addition, Elk uses array notations to access input and output stream tokens instead of `push`, `pop`, and `peek` primitives in StreamIt. Our experience tells us that the array notation leads to more concise programs and easier compiler implementation.

Programming System for the Elm Architecture (Chapter 2)

This dissertation presents a programming system infrastructure for the Elm architecture. The programming systems facilitates architecture evaluations [12] and demonstrates that a primary goal of the Elk project — efficient but *programmable* architecture — can be met with reasonable effort (i.e., primarily one Ph.D. student plus one to two M.S. students).

1.3 Collaboration

This dissertation describes work that was performed as part of the Elm project. James Balfour was the main architect of the Elm architecture, and I collaborated with him on the Elm cycle-accurate simulator that he designed. The Elk stream language is based on the initial design by him, whose features include array notations for accessing input/output streams.

David Black-Schaffer proposed a stream programming model called "Block Parallel Programming" [26, 28] that is an alternative to existing stream programming models such as StreamIt [145] and StreamC/KernelC [110]. His work inspired parts of the Elk programming system, such as supports for multiple input/output streams per actor, real-time constraints propagation, and physical core mappings with simulated annealing.

James Balfour, Curt Harting, and James Chen provided the numbers for the energy expended in circuit components, on which most of the estimated energy consumption presented in this dissertation is based. David Black-Schaffer did an initial investigation on instruction SPMs [27], and Clinton Buie participated in initial discussions on our instruction SPM algorithm. Youngsik Kim helped build a converter from LLVM [95] intermediate representation to our intermediate representation. Jooseong Kim implemented the front-end of the Elk compiler and helped on debugging our Elm C++ compiler back-end. Keiji Matsumoto, a visitor from Renesas Technology, examined the partitioning step of SGMS [90] that is formulated as integer linear programming. The QED presented in Chapter 5 was discussed with Jiwon Seo.

1.4 Organization

The remainder of this dissertation is organized as follows:

Chapter 2 describes the Elm project, in which the work presented in this dissertation was performed. The programming system for the Elm architecture consists of two layers. The lower layer is a C++ compiler that effectively targets the micro-architectural features of Elm. The upper layer is a stream programming system that efficiently utilizes system-level features of Elm such as software-managed memories with DMA operations. The upper layer compiles a language called Elk that extends StreamIt [145]. Although the algorithms presented in the subsequent chapters are not Elm-specific, this chapter provides useful context to facilitate understanding them.

Chapter 3 presents a compiler algorithm that manages instruction scratch-pad memories (SPMs). Energy efficiency, miss-rates, execution times, and code size are compared with SPMs managed by previous algorithms, filter caches, and loop caches. We use MiBench [55] applications and the Elm cycle-accurate simulator for the comparison.

Chapter 4 presents a compiler analysis that computes the minimum inter-actor queue capacities that sustain the maximum throughput of static stream applications. Chapter 4 also presents a static stream-scheduling algorithm, called team scheduling, based on the analysis. Energy efficiency and execution times are compared with those of a previous scheduling algorithm called SGMS. We use StreamIt [145] benchmark applications and a simulated 16-core Elm processor for the comparison.

Chapter 5 presents a dynamically-sized array-based concurrent queue design called QED (Queue Enhanced with Dynamic sizing) that minimizes the memory footprint of inter-actor queues in dynamic stream applications, which is complementary to the mechanism described in Chapter 4. Energy efficiency and execution times are compared with those of a popular linked-list based design called Michael and Scott's queue [112] and array-based statically-sized queues with "arbitrary large" capacities. We use applications selected from GRAMPS [141], MiBench [55], and SPEC benchmarks for the comparison.

Chapter 6 summarizes the dissertation, focusing on insights obtained with respect to software and hardware coordination for energy efficiency. This dissertation concludes with discussion on future directions that can improve the work presented.

Chapter 2

The Elm Project and Its Programming System

The work described in this dissertation has been done in the context of the Elm (Embedded Low-power Microprocessor) project, whose main goal is closing the energy-efficiency gap between programmable embedded processors and application-specific integrated circuits (ASICs). As presented in the previous chapter, the Elm project identified the main energy consumption source to be instruction and data delivery, not computation itself. This motivates using the memory subsystem in a more energy efficient manner by software, which is the main objective of this dissertation. However, software-only approaches are often insufficient to target the energy efficiency of ASICs, and synergistic efforts between hardware software are often desired. For example, the energy reduction that is achieved by the instruction scratch-pad memory management algorithm presented in Chapter 3 partially stems from features provided by instruction scratch-pad memories in the Elm architecture. Similarly, the static scheduling algorithm that minimizes buffer space overhead presented in Chapter 4 relies on efficient direct memory access (DMA) operations provided by the Elm architecture.

This chapter first briefly overviews the Elm architecture (Section 2.1), focusing on its impact on, and interaction with, the programming system. For comprehensive introduction to the Elm architecture, the readers are referred to Balfour's thesis [12].

Section 2.2 describes the programming system for the Elm architecture, which consists of two layers. The lower layer is a C++ compiler called `elmcc`, which leverages the LLVM compiler infrastructure [95]. The instruction scratch-pad memory management algorithm presented in Chapter 3 is implemented as a part of `elmcc`. The upper layer is a stream programming system called Elk that is extended from StreamIt [145]. The static stream-scheduling algorithm presented in Chapter 4 is implemented in the Elk compiler.

2.1 Elm Architecture Overview

The Elm architecture is designed for exploiting parallelism and locality abundant in embedded applications to achieve high energy efficiency. The fine-grain control of architectural resources is exposed to software so that the complexity of embedded systems design can be amortized in the compilers and programming tools, avoiding the design and verification cost of special-purpose hardware incurred for each new system. It was estimated that an Elm system implemented in a 45nm CMOS processor would deliver 500 GOPS at less than 5W in about 10mm $\times$ 10mm of die area (this estimation is based on the register-transfer level (RTL) model presented in Appendix A.2 of Balfour's thesis [12]). In comparison, it was estimated that an embedded processor based on 32-bit SPARC architecture [58] delivers about 380 MOPS at 36mW in 0.5mm $\times$ 0.3mm of die area, which means that an Elm processor achieves about 7$\times$ the energy efficiency per performance and 2$\times$ the area efficiency per performance [12] compared to the SPARC-based embedded processor.

2.1.1 System-level Architecture

Figure 2.1 illustrates the system level architecture of Elm. An Elm system consists of tiled cores (i.e., processors) that are grouped into ensembles and connected with a mesh on-chip interconnection network. In addition to core tiles, an Elm system also has distributed memory tiles.

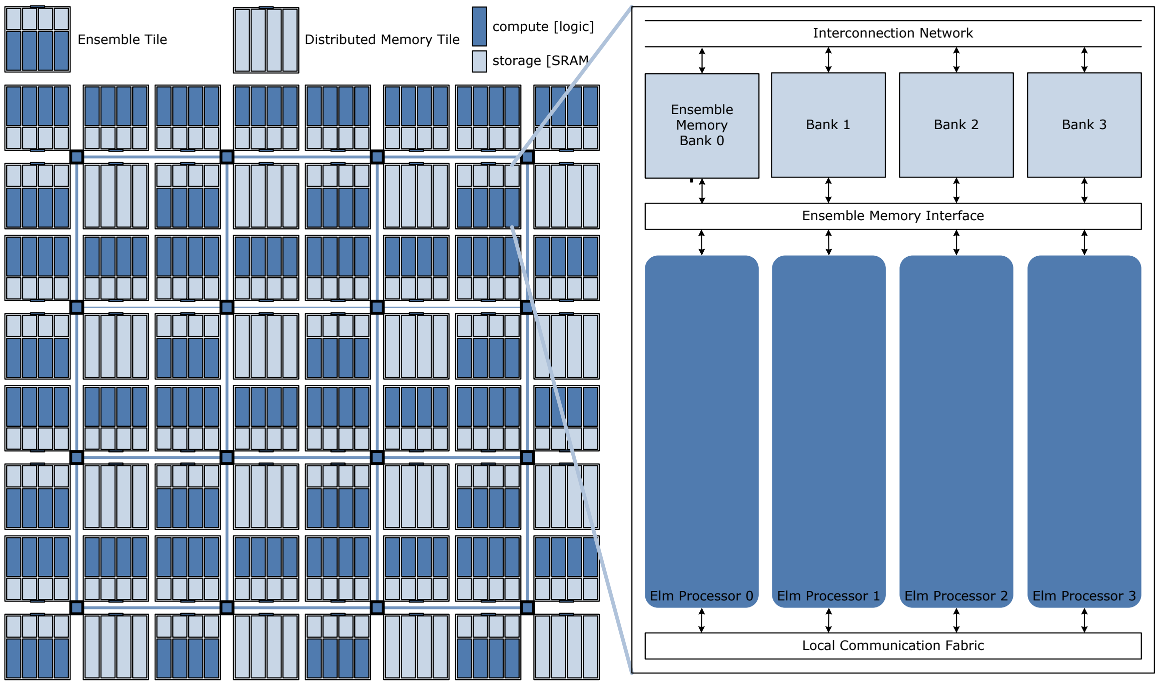


Figure 2.1: The system-level Elm architecture [12].

Distributed Software-managed Memory and Cache Memory

The processors communicate over an on-chip interconnection network with a mesh topology. Elm allows software to control the placement and transfer data directly (i.e., via DMA) between ensemble memories. Multiple outstanding DMA transfers can be issued so that a sufficient amount of communication time can be overlapped with computation. Scatter-gather DMA transfers are also provided to minimize the overhead of communication. Software-managed placement and non-blocking scatter-gather DMAs are particularly useful for stream applications whose communication patterns are analyzable by the compiler. These features are exploited by several stages of the Elk stream compiler, including the team scheduling phase presented in Chapter 4.

These software-managed ensemble memories are backed by a collection of distributed memory tiles, which can be configured as caches. These cache memories are useful for capturing the type of locality that is not easily statically analyzable.

Ensemble Structure

As illustrated in Figure 2.1, four processors are grouped into *ensemble*s, the primary building block of the Elm architecture. The motivation behind this grouping is to share expensive resources such as a local memory and an interface to the interconnection network. In addition, low-overhead communication and synchronization mechanisms are provided between the processors within an ensemble so that the compiler or the run-time system can exploit the locality between tightly coupled tasks mapped to the same ensemble.

Processors in the same ensemble share an ensemble memory, which corresponds to the L1 cache in conventional embedded processors. The ensemble memory provides a low latency access due to its proximity, while allowing concurrent access to the local processors and the network interface by through its bank structure. Processors within an ensemble are also connected through a local communication fabric with high-bandwidth point-to-point links, which can be used to capture predictable communications which demand low latency and high bandwidth.

These low-overhead communication mechanisms allow the programming system to exploit the parallelism of tightly coupled tasks even if their communication-to-computation ratio is so high that the benefits of parallelization are typically more than offset by the overhead of conventional cache coherent memories. For example, pipeline stages that communicate a considerable amount of data (e.g., back-end pipeline stages of the graphics rendering pipeline) can be mapped to the same ensemble so that the low-overhead local communication fabric can be used. When a task has enough instruction-level parallelism (ILP), four processors in an ensemble can be used as an ILP core with four coupled instruction streams, as the RAW processors are used as ILP machines [101]. An ensemble can also be used as a SIMD processor using the concept of *processor corps*. Processors operating as a processor corps execute the same instruction, amortizing the cost of instruction fetch.

Unfortunately, the ensemble structure is not fully exploited by the current Elm programming system, and, therefore, we used hand-optimized code to evaluate features associated with the ensemble structure. This is mostly because of limited time

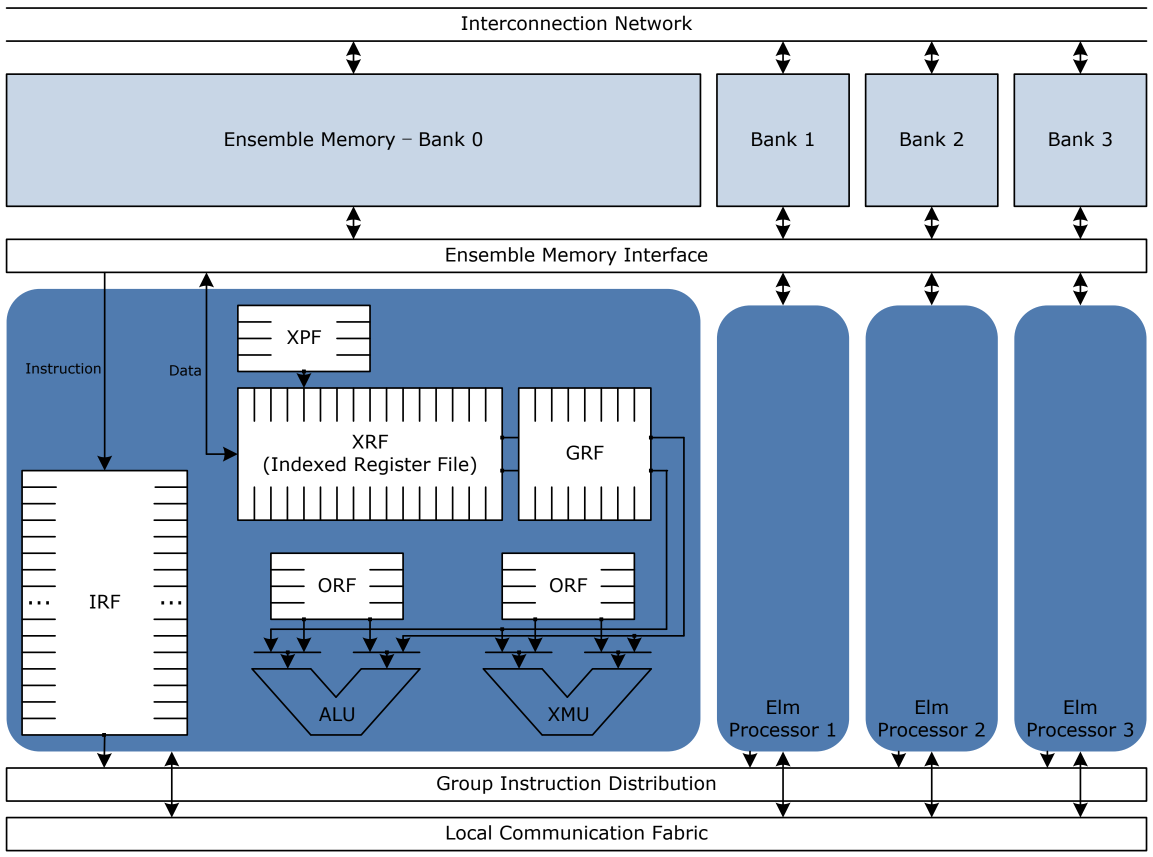


Figure 2.2: The core-level Elm architecture [12].

and resources, which we decided to use for targeting architectural features that result in a higher impact. Although ensemble level optimizations can result in a large energy-efficiency gain, their scalability is limited to four processors, and most of the optimizations are only a matter of implementing existing algorithms: e.g., we can use an ensemble as an ILP core using the algorithm described by Lee et al. [101], and the SIMD mode can be used by employing affine partitioning algorithms for data-parallel loops as described by Feautrier, Lim, and Lam [44, 45, 105]. Instead, we focus on architectural features that are not satisfactorily handled by existing algorithms (as in Chapter 3) or that support scalable parallelism (as in Chapter 4).

2.1.2 Micro-architecture

Figure 2.2 illustrates the core-level architecture of an Elm system. The Elm cores are statically scheduled dual-issue 32-bit processors. Each of them has two functional units, ALU and XMU. While the ALU executes generic computation instructions, the XMU is specialized for memory, control, communication, synchronization, and a few simple arithmetic (e.g., `add` and `sub`) instructions.

Efficient Instruction Delivery: Instruction Scratch-pad Memory

One of the most distinguished feature of the Elm processor is that instructions are issued from compiler-managed scratch-pad memories (SPMs). In Elm, we call the scratch-pad memories *instruction register file*s (IRFs) to emphasize their specialization for instruction fetches and small capacity (typically 64-256 instructions). The compiler is responsible for making sure that instructions are transferred to the instruction scratch-pad memory before they need to be issued, thereby eliminating the need for tags. Eliminating tag checks significantly reduces the energy per access, particularly for small caches where long tags are needed. For example, Section 3.5.1 shows that eliminating tag checks of a 256-instruction direct-mapped cache saves 53% of the read access energy.

Instructions are transferred from the ensemble memory in blocks whose length, source location, and target location are specified by the compiler. The processor continues to execute during instruction transfers, decoupling issuing and transferring instructions. This allows the compiler to pre-transfer instructions to hide the latency of instruction transfers. Instruction scratch-pad memories filter out a significant fraction of instruction fetches, enabling a single read and write port design of the ensemble memory, which is another major energy efficiency improvement.

Chapter 3 presents a compiler algorithm that places instructions at appropriate SPM locations to minimize conflict misses and inserts instructions that dynamically transfer instructions from backing memories (the ensemble memory in Elm) to SPMs. Even though there have been numerous compiler algorithms for managing instruction SPMs, Chapter 3 shows that SPMs managed by these previous algorithms cannot achieve clearly higher energy efficiency than caches with the same capacity. Chapter 3 elaborates on the reasons behind the unsatisfactory results of previous SPM placement algorithms and shows how our algorithm realizes noticeable energy savings compared to caches.

Instruction SPMs in Elm provide useful features that further improve the effectiveness of the SPM placement algorithm presented in Chapter 3. As mentioned above, decoupling issuing and transferring instructions allows the algorithm to hide the latency of instruction transfers, thereby improving the execution time. In addition, the instruction SPMs in Elm provide wrap-around access: when the target SPM location plus the transfer block length exceeds the capacity of the SPM, the target location wraps around to the beginning of the SPM. This considerably reduces the fragmentation of instruction placement in SPMs.

On the other hand, we find that not all features provided by the instruction SPMs in Elm are clearly beneficial with respect to energy efficiency. The instruction SPMs in Elm do not allow bypassing. In other words, all instructions must be issued from the SPMs, and there is no bypass that can be used for issuing instructions directly from the backing memory. In Section 3.6, we show that adding bypass support reduces the number of accesses to the backing memory by 40%, and the energy consumed for accessing the SPM and the backing memory cells by 31%. It is true that removing bypass support has several advantages such as simpler connectivity and a shorter program counter. However, it is unclear that these advantages will more than offset the aforementioned 31% energy saving of bypass.

Efficient Data Delivery

Operand Register File As shown in Figure 2.2, Elm processors have distributed and hierarchical register organization. Each function unit has a small local register file called an *operand register file* (ORF), which is backed by the *general-purpose register file* (GRF). ORFs capture short-term data locality so that a considerable number of operand accesses can be provided by inexpensive registers that are small and close to the functional units. ORFs also filter out a significant fraction of bandwidth demand for larger and more expensive GRFs. This allows to keep the number of read and write ports of the GRF small (note that this is similar to the idea behind reducing the number of read and write ports of the ensemble memory by filtering out instruction accesses by the instruction SPMs). The ORF concept is extended to explicitly forward temporary values through result registers to avoid writing dead values back to ORFs or GRFs.

We allocate registers and schedule instructions using conventional algorithms. We first schedule instructions assuming an infinite number of registers using the list scheduling algorithm. The purpose of this initial schedule is to balance the load of two functional units and to estimate the pressure of ORFs. Then, we allocate registers using the conventional graph coloring register allocation algorithm [33] for each register file, starting from smaller ones. The initial schedule imposes constraints on register allocation with respect to limited connectivity between register files and functional units. For example, the XMU cannot read the ALU's ORF, and, therefore, variables that are read by instructions that execute on different functional units must be assigned to the GRF or duplicated to multiple ORFs. To effectively utilize the ORFs, we use the spill cost function that prefers variables with a high access frequency and short life time. Since register allocation and instruction scheduling for distributed and hierarchical register organization in Elm is a straightforward combination of existing approaches, this dissertation does not go into the further details. Those who are interested in the details are referred to Balfour et al. [12, 14].

In our initial design of the Elm architecture presented in Balfour et al. [13] and Dally et al. [38], we took a radical approach to the distributed and hierarchical register organization, where the connectivity is more limited. For example, functional units are not allowed to directly access the GRF and every operand must go through ORFs. This limited connectivity combined with explicit pipelining of instructions [13, 38] makes the instruction scheduling and register allocation considerably more tightly coupled than conventional architectures. In order to avoid a severe phase ordering problem between register allocation and instruction scheduling, a unified allocation and scheduling approach was desired. Therefore, we developed *path finding scheduling* (an algorithm presented in a separate technical report [125]), which simultaneously solves the allocation and scheduling problem by finding paths through a graph

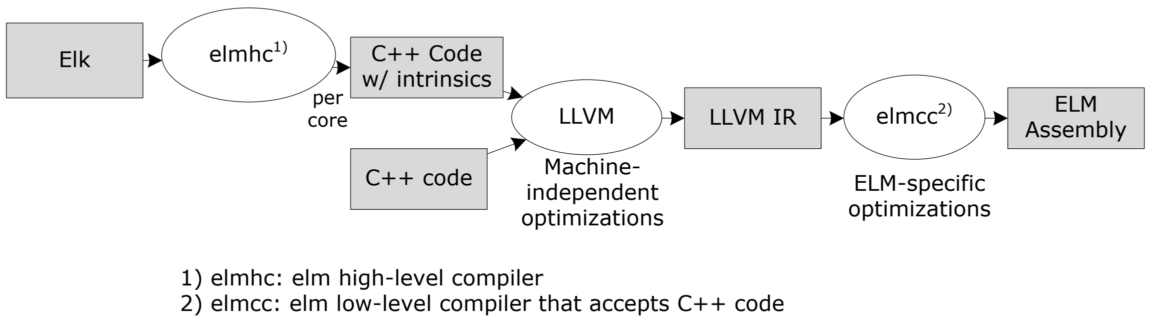

Elk
elmhc1)
per core
C++ Code w/ intrinsics
C++ code
LLVM
Machine-independent optimizations
LLVM IR
elmcc2)
ELM-specific optimizations
ELM Assembly
1) elmhc: elm high-level compiler
2) elmcc: elm low-level compiler that accepts C++ code

Figure 2.3: The architecture of Elm programming system.

that describes the connectivity of resources. However, in the aim to avoid potential scheduling deadlocks resulting from simultaneous allocation and scheduling, the algorithm was made to be too complicated and slow (even through it is a polynomial-time algorithm). This motivated the current simpler distributed and hierarchical register organization in Elm.

Indexed Register File In additional to its distributed and hierarchical nature, register organization in Elm provides indirect access [14, 126]. The *indexed register file* (XRF) is indirectly accessed through the *index pointer registers* (XPs). The XRF allows to block data into registers, as we block data into caches [94]. Although it is possible to block data into registers without indirect access support, doing so involves loop unrolling, which degrades the instruction locality. The XRF also supports bulk transfers to/from the ensemble memory, which can be viewed as DMAs between the ensemble memory and the XRF.

2.2 Elm Programming System

Figure 2.3 illustrates the architecture of Elm's programming system. There are primarily two ways of using the programming system. When the target application follows the stream programming model, one can use a stream programming language called Elk. The high-level Elk compiler called `elmhc` compiles Elk code and generates C++ code per core. Architecture-dependent features such as DMA operations are exposed as intrinsic function calls, which are generated by `elmhc` during its code generation phase. When direct control of the architecture-dependent features is desired, or when the target application does not follow the stream programming model, the programmer can write C++ code per core. In this case, the programmer must manually use intrinsic functions to access architecture-dependent features.

Section 2.2.1 describes the low-level C++ compiler called `elmcc`, and Section 2.2.2 describes the high-level Elk compiler called `elmhc`.

2.2.1 Low-level Compiler Back-end

The Elm C++ compiler (`elmcc`) is implemented as a back-end of the LLVM compiler framework [95]. The LLVM compiler parses C++ code, applies machine-independent optimizations, and emits optimized LLVM intermediate representation (IR). Several Elm-specific optimizations are implemented in `elmcc`, and its total line count is about 110K. Elm-specific optimizations include XRF bulk transfer generation, allocation of distributed and hierarchical register files, explicit forwarding generation, and instruction scratch-pad memory management. Chapter 3 describes the instruction scratch-pad memory management in detail; other Elm-specific optimizations are described briefly in Section 2.1.2. The quality of code generated by `elmcc` is high enough that the efficiency of Elm architecture is mostly evaluated using compiled code in Balfour's thesis [12]. MiBench [55] and StreamIt [145] benchmarks whose line counts are up to tens of thousands are successfully compiled to produce correct results.

2.2.2 High-level Parallelizing Stream Programming System

Elk Stream Programming Language

Among different kinds of parallelism that can be exploited by stream programming, a class of stream programming languages focus on data-level parallelism, which can be found from computations whose successive iterations are independent of each other. In domains where data-parallel computations are common and the data set is large enough, solely exploiting data-level parallelism often results in a near perfect utilization of all cores. Scientific computation is an example of such a domain and

Sequoia [43] is an example of a stream programming language that focuses on data-level parallelism.

Each computation chunk of common embedded applications operates on data sets that are smaller than those of scientific applications, and, therefore, data-level parallelism is sometimes not enough to achieve a degree of parallelism required to fully utilize the computing power of emerging multi-core processors. In addition to data-level parallelism, task-level and pipeline parallelism also should be exploited [51]. To the best of our knowledge, StreamIt [145] is the current state-of-the-art language that exposes data-level, task-level, and pipeline parallelism of streaming applications together to the compiler.

Comparison with StreamIt The Elk stream programming language extends StreamIt to address the issues discussed below. For the further details on constructs of StreamIt, refer to Thies et al. [145]. Black-Schaffer's thesis [26] provides an excellent literature review on stream programming systems other than StreamIt.

Single Input and Output Stream per Actor: StreamIt allows at most one input and output stream per actor. This constraint introduces unnecessary clutter in many applications. For example, Figure 2.4(a) shows a stage of DES (Data Encryption Standard) implemented in StreamIt, which includes several splitters, joiners, and identity actors that would have been unnecessary if actors with multiple input and output streams were supported as shown in Figure 2.4(b). Note that, in Figure 2.4(a), a spurious splitter (the producer of `KeySchedule`) is used to work around the lack of no-input-stream actors. Also, note that joiners are used before `Xor` actors to interleave two streams. While implemented the `Xor` actor, the programmer must keep in mind that even input tokens are from the first input stream to the joiner and odd input tokens are from the second one.

The rationale for restricting the number of stream inputs and outputs per actor in StreamIt was to restrict the methods of actor composition to `pipeline`, `splitjoin`, and `feedback`. In Thies et al. [145], the authors make an analogy to structured programs (i.e., programs without `goto`s), and argue that the restricted composition

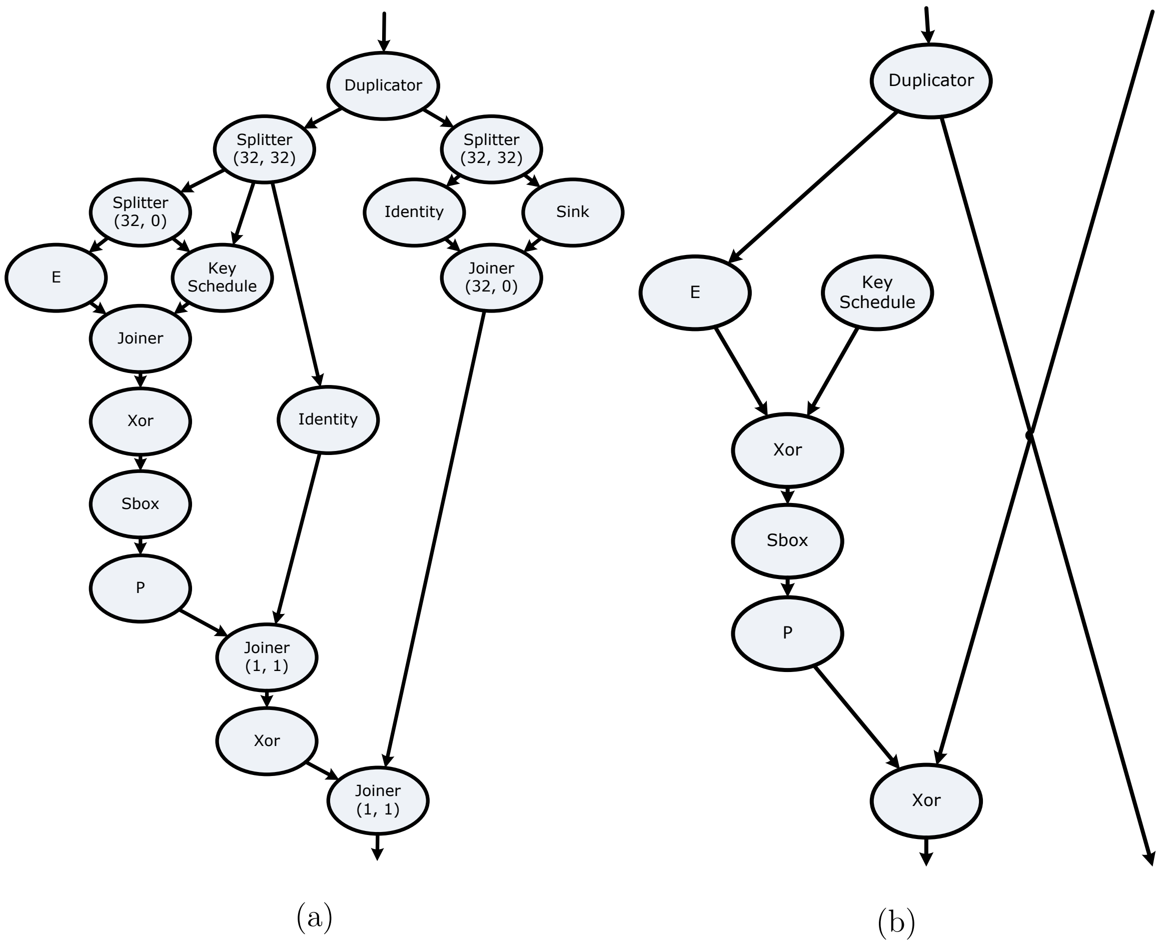


Figure 2.4: A DES stage with (a) single input and output actors as in StreamIt and (b) multiple input and output actors as in Elk.

methods facilitate well-structured programs and simplify compiler analyses. However, experience from programming more than 20 representative applications has shown me that allowing arbitrary actor connections does not lead to undisciplined structuring of stream programs and that the limited expressiveness of StreamIt on actor connectivity does more harm than good. A single generic composition construct, such as `bundle` in Elk, which will be described shortly, is enough for writing well-structured stream programs. My experience from the Elk compiler implementation has also shown me that the restricted compositions do not necessarily simplify compilation. For most compiler analyses and transformations, the composition hierarchy should be flattened anyway to data structures such as *stream graph*s [90] that represent the connectivity of actors in the entire application. Compiler analyses occasionally need to find topological structures such as `splitjoin`s or `feedback`s, but such structures can be easily identified from flattened graphs by simple graph algorithms that have been commonly used by traditional compiler analyses without relying on explicit specification in source code.

Push, Pop, and Peek Primitives: In StreamIt, an actor produces and consumes stream tokens through `push` and `pop` primitives. StreamIt also provides a `peek` operation that allows an actor to read more input tokens than it consumes, which implies buffering on input stream tokens. For example, the permutation block of DES can be implemented as follows:

```
work pop 32 push 32 {
  for (int i = 31; i >= 0; i--) {
    push(peek(32 - P[i])); // P is the permutation array
  }
  for (int i = 0; i < 32; i++) {
    pop();
  }
}
```

First, the primitive names may confuse programmers since `push` and `pop` are commonly used for stacks, whereas each stream in stream programs has queue semantic. In addition, `push` and `pop` impose a specific ordering on reading input tokens and writing output tokens. This is why the first loop uses the reverse iteration order and why we need the second loop for book keeping.

In Elk, we use array notations to represent the current window of input and output tokens as follows:

```
(int[32] in) -> (int[32] out) {
  for (int i = 0; i < 32; i++) {
    out[31 - i] = in[32 - P[i]];
  }
}
```

This description is more concise and closer to code that will be generated by an optimizing stream compiler; due to efficiency reasons or in order to preserve the atomic nature of each actor firing, the stream compiler will transfer tokens in bulk. In fact, an intermediate representation of a StreamIt compiler uses a form that is more similar to the Elk code shown above than to the StreamIt code. In short, StreamIt provides unnecessarily low-level primitives that help neither the programmer nor the compiler.

Basic Constructs of Elk:

Actor *Actor* is the building block of Elk. When an actor is fired, it invokes its *map* function, whose syntax is as follows:

```
actor Fir<int N> {
  const int[N] coeffs; // member variable

  Fir(const T[N] coeffs) { // constructor
    this.coeffs= coeffs;
  }

  (int[N] in strides=1) -> (int[1] out) { // map function
    int sum = 0;
    for (int i = 0; i < N; i++) {
      sum += coeffs[i]*in[i];
    }
    out[0] = sum;
  }
}
```

The syntax of actor definition is similar to that of class definition in C++. An actor can have template parameters, member variables, constructors, and methods. One difference is that each actor actively executes its own code instead of being passively executed by threads as in C++ classes. In addition, each actor has one map function with special semantics: i.e., is invoked whenever an actor instance is fired.

The definition of map function starts with input and output streams that are denoted as arrays. The array notation associated with a stream represents the window of stream tokens that is accessed by the current actor firing. An input stream can have a stride attribute. Strides denote the amount of the window shift between consecutive actor firings. Within the map function, the syntax is identical to that of C++ except that an array for input stream cannot be written and an array for output stream cannot be read.

The number of tokens that are consumed or produced by an actor can be a variable. The code below shows an RLE (run length encoder) actor as an example.

```
actor Rle {
  int count = 1;
  (int[2] in stride=1) -> (int[] out) {
    if (in[0] == in[1]) {
      count++;
    }
    else {
      out[0] = in[0];
      out[1] = count; // Streams advance per actor invocation.
      count = 1;
    }
  }
}
```

There are a few pre-defined actors such as `RoundRobinSplitter`, `RoundRobinJoiner`, and `Duplicator`, which are similar to the filters with the same names in StreamIt.

Several standard actors are also implemented as a library, which includes `FileReader`, `FileWriter`, `Transpose`, `Buffer`, `Decimator`, `Adder`, `Summer`, `Subtractor`, and `Multiplier`.

Bundle Elk provides a general composition construct, *bundle*, which allows arbitrary connections within. The syntax of bundle definition is identical to that of actor except that its map function describes actor or bundle connections instead of computation. The code below shows how we can build a band-pass filter using bundles:

```
bundle BandPassFilter<int N> {
  int[N] hpfCoeffs, lpfCoeffs;

  BandPassFilter(int lowFreq, int highFreq) {
    computeHpfCoeffs(hpfCoeffs, lowFreq);
    computeLpfCoeffs(lpfCoeffs, highFreq);
  }

  static void computeHpfCoeffs(int[N] coeffs, int cutoff) {
    ...
  }

  static void computeLpfCoeffs(int[N] coeffs, int cutoff) {
    ...
  }

  (int[] in) -> (int[] out) { // map function
    in >> Fir<N>(hpfCoeffs) >> Fir<N>(lpfCoeffs) >> out;
  }
}

bundle Main {
  () -> () {
    FileReader("in.dat") >>
      BandPassFilter<32>(1000, 3000) >> FileWriter("out.dat");
  }
}
```

In map functions of bundles, we connect actors or bundles through `>>` operator, which is adopted from the C++ I/O stream operator. Using `>>` operator, we can describe an arbitrary actor/bundle connectivity. For map functions in bundles, we can omit the size of arrays associated with input/output streams, which will be inferred by the compiler. The `Main` is a pre-defined bundle which is the entry point of the program.

We can instantiate multiple identical actors as follows:

```
// from dct.elk
in >> RoundRobinSplitter(8) >> IDCT8x8_1D_row_fast()*8 >>
  RoundRobinJoiner(8) >> ...
```

We can also use `if` and `for` statements provided their conditional statements use compile-time constants as follows:

```
// from bitonic_sort.elk
for (int i = 0; i < numSeqP; i++) {
  // numSeqP should be constant at the bundle invocation time.
  if (l > 2) { // l should be constant
    tempIn[i] >> PartitionBitonicSequence(l, sortDir) >> tempOut[i];
      // sortDir should be constant
  }
  else {
    tempIn[i] >> CompareAndExchange(sortDir) >> tempOut[i];
  }
}
tempOut >> RoundRobinJoiner(1) >> out;
```

Stream A *stream* represents data communication between actors. We can annotate a stream with a rate as shown below. The compiler exploits parallelism just enough to satisfy the real-time constraint associated with the rate instead of performing a best-effort parallelization, if a rate is specified as follows:

```
bundle Main {
  () -> () {
    stream<int> in rate=1MHz;
    FileReader("in.dat") >> in >>
      BandPassFilter<32>(1000, 3000) >> FileWriter("out.dat");
  }
}
```

We can define an array of streams, which is useful in conjunction with `for` statements in a bundle:

```
// from bitonic_sort.elk
stream<int>[numSeqP] tempIn, tempOut;
for (int i = 0; i < numSeqP; i++) {
  for (l > 2) {
    tempIn[i] >> PartitionBitonicSequence(l, sortDir) >> tempOut[i];
  }
  else {
    tempIn[i] >> CompareAndExchange(sortDir) >> tempOut[i];
  }
}
tempOut >> RoundRobinJoiner(1) >> out;
```

Elk High-level Compiler

Output C++ Code The Elk compiler (`elmhc`) is a source-to-source compiler. It generates C++ code that is designed to be architecture-independent. For each core, a separate C++ code is generated, and each of these separated codes has its own main function. In the generated C++ code, architecture dependencies mainly come from how to express streaming (i.e., DMA) operations. We abstract these architecture-dependent parts into two levels: concurrent queue library and intrinsic functions. Most of architecture-dependent parts can be expressed by our concurrent queue library interface, which is the higher level of the two abstractions.

Compiler Structure and Important Phases

Front-end We use the ANTLR parser generator [1] to parse `.elk` files and generate an abstract syntax tree (AST) of the whole program. Then, we generate a three address code (TAC) and a control flow graph for each actor as an intermediate representation.

Important Phases

- Bundle Flattening: We flatten the actor hierarchy that is expressed by the `bundle` construct in Elk. The `bundle` construct facilitates code reuse and modularization, but, during compilation, it is often more convenient to flatten the hierarchy.

- Constant Propagation: During the bundle flattening, we propagate constants through the hierarchy and within actor computation codes. In many occasions, we use Janino (an embedded Java compiler) [3] to fold a complex expression into a constant.

- Rate Analysis: We propagate real-time constraints associated with streams or actors by the `rate` construct. For example, if there is a low-pass filter actor followed by a high-pass filter actor, and the real-time constraint of the low-pass filter's input stream is 1KHz, we propagate the constraint to the high-pass filter and also set the real-time constraint of the high-pass filter to 1KHz (here, we assume that both low-pass and high-pass filter consume and produce 1 stream token per firing).

- Actor Computation Rate Estimation: We estimate the number of instructions per actor firing using a static analysis. We aggressively apply compiler optimizations such as conditional constant propagation [159], dead code elimination, and partial redundancy elimination [86] to our intermediate representation in order to consider optimizations that will be performed by the C++ compiler. When we encounter an `if-else` branch, we select the longest path. When we encounter a loop whose iteration count cannot be determined at compile-time, we mark the actor as a "dynamic" actor (to override this, the programmer can manually specify the computation rates of an actor). By the combination of results from the previous rate analysis phase and this phase, we can associate each actor with the computation requirement that is used in a later phase to satisfy the real-time constraints. For example, if the propagated rate for a DCT actor is 1KHz, and if the estimated actor computation rate is 400Hz, the computation requirement for the DCT actor is 2.5 cores.

- Actor Fission/Fusion: We fuse and fission actors so that we use the minimum number of cores to satisfy the real-time constraints while maintaining a good load balance. As a pre-processing step, we selectively fuse adjacent stateless actors to coarsen the granularity as described in Gordon et al. [51]. If real-time constraints are specified by the programmer, we fission stateless (i.e., data-parallel) actors so that computation requirements are satisfied. For example, if a DCT actor requires 2.5 cores, we replicate the DCT actor three times. If real-time constraints are not satisfied, we fission stateless actors just enough to fill the cores as in the judicious fission described in Gordon et al. [51]. Although `elmhc` presently fissions only stateless actors, more data-level parallelism can be exploited by applying the technique for actors with sliding windows that is presented by Gordon [50] or more general affine partitioning [44, 45, 105]. After actor fission, we input the modified stream graph with fissioned actors to a graph partitioning software called METIS [82] to fuse actors with small computation requirements to the same core in order to achieve a good load balance. Greedy heuristics are used for actor fission/fusion in a StreamIt compiler for the RAW architecture [51, 52] and in the block parallel programming presented by Black-Schaffer [26, 28]. In a StreamIt compiler for the Cell architecture, integer linear programming is used for actor fission/fusion [90]. However, we did not find a compelling reason for using the greedy heuristics or integer linear programming formulations instead of leveraging the widely-used METIS framework.

- Team Scheduling (Actor Aggregation and Amortization): We aggregate certain actors that are mapped to the same core as a single actor so that synchronization overhead between them can be eliminated. We also multiply the number of executions of actor per synchronization (queue empty or full checks) in appropriate cases to amortize the overhead associated with synchronization and DMA initiation. This optimization significantly improves the throughput while satisfying the buffer-space constraints from limited local memory space. The key novelty of this phase is a generic buffer-space computation method that finds the minimum buffer capacities that avoid serialization or deadlock. This allows to flexibly apply actor aggregation and amortization in an arbitrary order. The details of this phase are presented in Chapter 4.

- Physical Core Assignment: In the actors-to-cores mapping resulting from the actor fission/fusion phase, the core numbers are virtual core numbers. In this phase, we find a virtual-cores to physical-cores mapping that minimizes the aggregated communication distances. We use a simulated annealing method [85]. For more details, refer to Appendix A of Black-Schaffer's thesis [26].

2.3 Chapter Summary

This chapter overviews the Elm architecture and its programming system to provide the context for the subsequent chapters. The Elm architecture is designed to efficiently exploit the abundant locality and parallelism present in embedded applications. Its distinct features at the system level include software-managed memory with efficient DMA support and ensemble structure; in the core level, distinct features include instruction SPM, operand register files, and indexed register files.

In the subsequent chapters, we present compiler algorithms for efficiently utilizing some of these features. Chapter 3 presents a compiler algorithm that manages instruction SPM and saves instruction delivery energy. Chapter 4 presents a static scheduling algorithm that utilizes software-managed memories and DMA operations to efficiently transfer data between tasks in stream applications.

The programming system of Elm consists of two parts: the low-level C++ compiler (`elmcc`) and the high-level Elk stream compiler (`elmhc`). The low-level `elmcc` implements the instruction SPM management algorithm presented in Chapter 3, The Elk stream programming language targeted by `elmhc` is based on StreamIt [145], and the differences between Elk and StreamIt are described in Section 2.2.2. The high-level `elmhc` implements the static stream-scheduling algorithm presented in Chapter 4. Chapter 5 presents a data structure useful for dynamic stream applications that is complementary to the static scheduling algorithm (but, this has not yet been incorporated in `elmhc`).

Even though it is useful to understand the subsequent chapters in the context of Elm project particularly with respect to their motivations and relations to architectures, the methods described in these chapters are not Elm-specific. Architectural features targeted by each method are mostly modular, and, therefore, each can be incorporated separately. For example, instruction SPMs have been extensively studied for embedded processors, as will be shown in Section 3.2. Our SPM management algorithm can be applied to instruction SPM designs other than the one in Elm. Software-managed memories with efficient DMA support are not unique in Elm: e.g., Cell Broadband Engine [72] provides a similar feature. The static stream-scheduling algorithm can be applied to other architectures that provide software-managed memories with DMA support. Although features provided by Elm such as scatter-gather DMAs improve the efficiency, utilizing them is an optimization, not a necessity for using our static scheduling algorithm. In fact, the stream-scheduling algorithm can be used even for architectures without software managed memories with DMA operations, along a line similar to that used by Gummaraju and Rosenblum [54].

Chapter 3

Instruction Scratch-pad Memory

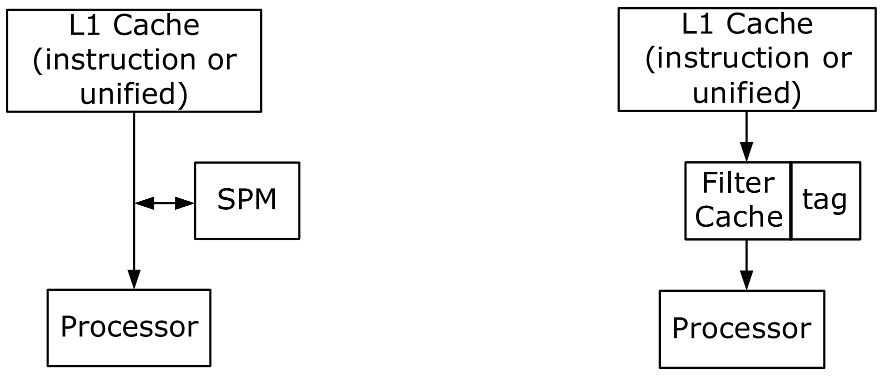
This chapter presents a compiler algorithm that manages small L0 instruction scratch-pad memories (SPMs) that achieves significant savings in instruction delivery energy. Our algorithm is fine-grain: the length of transfer blocks can be adjusted in increments of one instruction. Our algorithm captures a large fraction of instruction reuse missed by coarse-grain placement algorithm whose unit of transfer is restricted to loops or functions within the capacity of SPMs. Evaluation of L0 SPMs with our fine-grain algorithm in 17 applications show that the energy consumed by instruction storage hierarchy is reduced by 87% compared to that of the configuration where all instructions are fetched from the L1 instruction cache.

3.1 Overview

In order to reduce instruction delivery energy, researchers have proposed extending the hierarchy by adding small instruction stores (typically 1KB or smaller) between the L1 instruction cache and the processor [27,53,67–70,79,84,99,100]; in this chapter, we call these *L0 instruction stores*.

Scratch-pad memories [15, 121] (SPMs), shown in Figure 3.1(a), are compiler-managed stores in which no tags are used to associate locations in SPMs with memory addresses. Therefore, SPMs consume less energy per access than caches with the same

A shorter version of this chapter is presented in [123].


L1 Cache (instruction or unified)
SPM
Processor
L1 Cache (instruction or unified)
Filter Cache
tag
Processor

(a) L0 Scratch-Pad Memory (SPM) (b) Filter cache (FC)

Figure 3.1: L0 instruction stores.

capacity, lending themselves to being a natural choice for L0 instruction stores. There are primarily two types of instruction placement algorithms that target SPMs: static and dynamic instruction placement. In static instruction placement algorithms [9,10, 15,53,79,118,140,154,155,160], the most frequently executed instructions are identified by profiling and preloaded prior to starting an application. Throughout the application execution, the set of instructions that reside in the SPM does not change. Due to their static nature, static placement algorithms cannot efficiently utilize the SPM when an application has multiple hotspots that do not fit in the SPM altogether. In dynamic instruction placement algorithms [27,41,42,78,120,133,139,147,153,156], instructions are dynamically transferred into the SPM as needed, thereby utilizing SPM space more efficiently. For example, Udayakumaran et al. [147] show that their dynamic placement technique achieves an average of 31% energy reduction over static placement.

Alternatively, a tag-based design for L0 called *filter cache* [84] (FC), shown in Figure 3.1(b), can be used. FCs do not require any compiler modification and preserve instruction set compatibility. Given this relative simplicity of FCs, SPMs must have a sufficiently large energy efficiency advantage over FCs to be the preferred choice. Although energy efficiency has been the main motivation for using SPMs [15,139,140], they have yet to show a notably higher energy efficiency over FCs'. Section 3.2 details reasons behind this, one of which is the coarse-grain placement of instructions in SPMs

— the smallest unit of transfer is a loop or function.

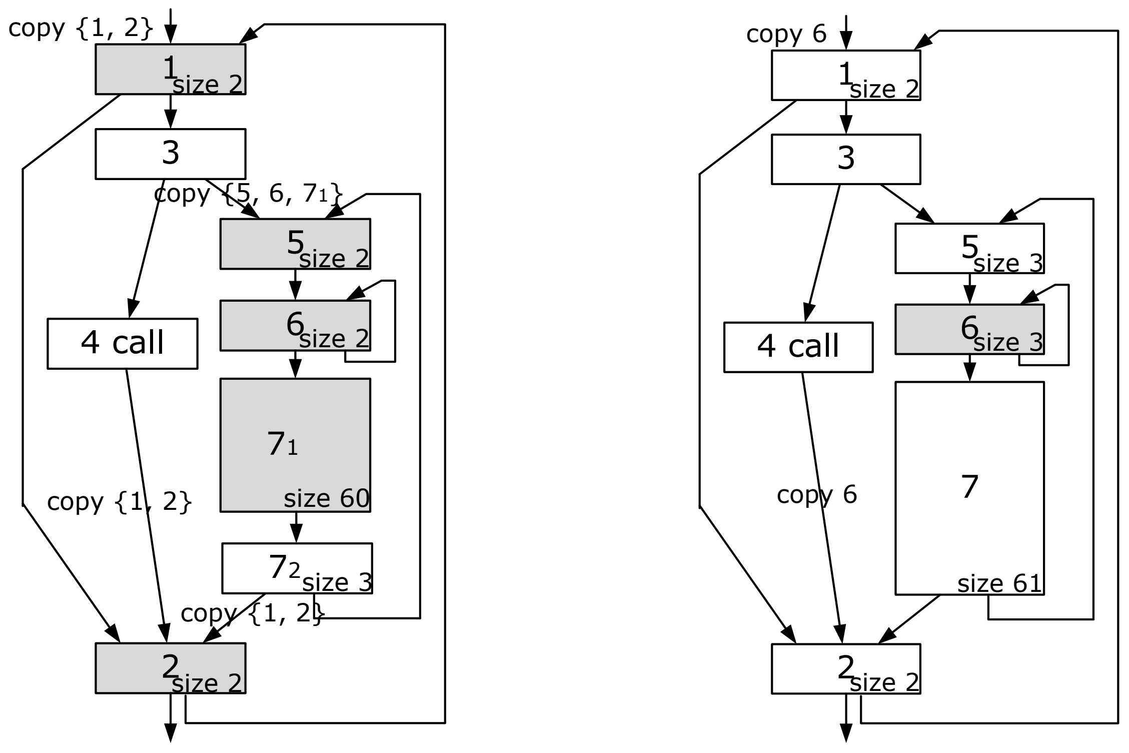
This chapter presents a fine-grain dynamic instruction placement algorithm for SPMs that achieves an average of 38% instruction delivery energy savings over FCs. We evaluate 17 representative and non-trivial embedded applications from MiBench [55] and rigorously compare SPMs to FC configurations with the best energy efficiency. We also show that our fine-grain algorithm achieves 31% instruction delivery energy savings over even an ideal coarse-grain dynamic placement algorithm which achieves zero miss rate for instructions in loops or functions that fit the SPM. The fine granularity of our algorithm does not cause proliferation of copy instructions, therefore maintaining execution time and static code size similar to those of the coarse-grain algorithm.

The remainder of this chapter is organized as follows. Section 3.2 reviews related work. Section 3.3 discusses the advantages and disadvantages of SPMs and FCs with respect to miss rates. Section 3.4 describes our fine-grain placement algorithm. Section 3.5 presents the results of our evaluation. Section 3.6 discusses the interaction of instruction SPMs with other architectural features, and Section 3.7 concludes.

3.2 Related Work

3.2.1 Dynamic Instruction Placement of SPMs

Udayakumaran et al. [147], Egger et al. [41], and Pabalkar et al. [120] restrict the smallest unit of instruction transfer to a loop or function, which results in a large fraction of instruction reuse being missed. For example, Figure 3.2 shows that, while a fine-grain algorithm can place blocks 1, 2, 5, 6, and 7_1 in the SPM, a coarse-grain algorithm can only place block 6 in the SPM since the other blocks belong to loops that exceed the SPM size. Figure 3.3 shows that even an ideal SPM placement algorithm cannot achieve more than 10% energy reduction over FCs, with coarse-grain instruction transfers. In Figure 3.3, we assume that the SPM placement algorithm achieves zero miss rate (including compulsory misses) for instructions in loops or functions that fit the SPM.



(a) fine (b) coarse

Figure 3.2: An example of (a) a fine-grain dynamic instruction placement and (b) its coarse-grain counterpart when the capacity of the SPM is 64. Blocks placed in the SPM are shaded. Section 3.4 describes the placement process in detail using the same example.

Steinke et al. [139], Verma et al. [156], and Egger et al. [41] use integer linear programming (ILP) to select the best set of instructions to be placed in SPMs, which does not scale well for large applications. During the survey of related work for this dissertation, I have encountered several occasions where ILP is used without compelling justification. ILP typically requires exponential running time in the worst case and commercial ILP solvers are quite expensive. Therefore, ILP should be used only when the user wants “super” optimization and should not be used as the default algorithm. We do not consider a compiler optimization problem to be solved by formulating it as ILP, since such formulation is a mechanical procedure for most combinatorial optimization problems. In addition, such a mechanical procedure seldom provides any

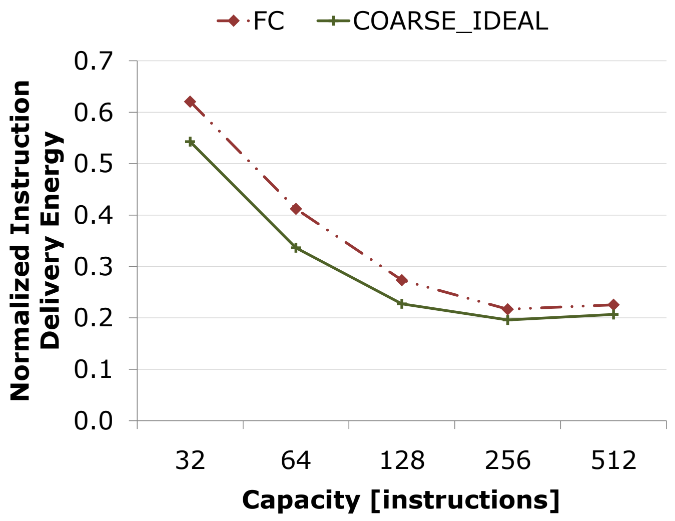


Figure 3.3: Instruction delivery energy of filter caches (FC) and SPMs with an ideal coarse-grain instruction placement (COARSE_IDEAL). The instruction delivery energy is normalized to the case when every instruction is fetched from the L1 instruction cache. Details on the evaluation setup are presented in Section 3.5.

insight on the problem. We believe that ILP is useful for compiler optimization only if 1) there is the need for super optimization, 2) one wants to compare the performance of heuristics with the optimal, 3) it can be shown that ILP solver finishes in a polynomial time for the most of cases due to a certain structure of the problem, or 4) the ILP formulation gives additional insight on the problem (e.g., through linear programming (LP) relaxation or its LP dual form).

Ravindran et al. [133] and Janapsatya et al. [78] use neither coarse-grain placement nor an exponential time algorithm. Ravindran et al. use traces and Janapsatya et al. use basic blocks as their unit of instruction transfer, both of which are less flexible than our algorithm where the length of transfer blocks can be adjusted in increments of one instruction. Ravindran et al. use *temporal relation* [47] to measure the cost of placing multiple traces in the same SPM location, and Janapsatya et al. use a similar metric called *concomitance*. Although using temporal relation or concomitance can minimize conflict misses of *caches*, we show in Section 3.3 that applying these metrics to *SPMs* overlooks the tagless and compiler-managed properties of SPMs. Using these metrics not only unnecessarily complicates profiling and compilation but can also misguide

Table 3.1: Comparison of dynamic instruction placement algorithms. In the second column, "Instruction" denotes that the length of transfer blocks can be adjusted in increments of one instruction, not instruction by instruction transfers. The algorithm without check symbols in the fourth column incorrectly rely on metrics that are meaningful only in the presence of tags (e.g., temporal relation).

	Unit of transfer	Polynomial time?	Tagless property considered?
Ours	Instruction	$\surd$	$\surd$
Udayakumaran et al. [147]	Loop and function	$\surd$	$\surd$
Egger et al. [41]	Loop and function		$\surd$
Pabalkar et al. [120]	Loop and function	$\surd$	
Verma et al. [153, 156]	Trace		$\surd$
Ravindran et al. [133]	Trace	$\surd$	
Janapsatya et al. [78]	Basic block	$\surd$	
Steinke et al. [139]	Basic block		$\surd$

placement algorithms to make decisions that are beneficial only in the presence of tags.

We also point out that the literature has not rigorously compared SPMs against the best filter cache configurations. Several papers use 4-way [41, 120, 139, 154] or 2-way associative filter caches [133]. In Section 3.3 and 3.5.2, we show that higher associativity does not necessarily result in lower miss rates when the cache is small, which limits its instruction reuse mainly to loops. Egger et al. [42] confirm this by showing that direct-mapped filter caches outperform 4-way associative filter caches with respect to execution time and energy.

Several papers evaluate only a few applications with small diversity [78, 139, 156], which can lead to an inaccurate conclusion. For example, if we only evaluate applications whose performance is dominated by regular loops such as SHA and CRC, SPMs' performance relative to that of FCs can be exaggerated. In some papers [78, 133], the majority of evaluated applications are encoder/decoder pairs, which are redundant since typical encoder/decoder pairs in embedded domain (especially symmetric encryption algorithms) have similar behavior.

Verma et al. [153] compare energy consumption of SPMs and filter caches averaged over multiple capacities (128 to 1024 bytes). They conclude that SPMs with their algorithm are more energy efficient than filter caches since their energy consumption averaged over capacities is smaller although their minimum energy consumption is larger than that of filter caches. A more meaningful comparison however would be between minimums rather than average energy reductions. There is no reason for using SPMs or filter caches with suboptimal capacities, and the average over multiple capacities has no meaning other than its weak relationship to the sensitivity of energy consumption to the capacity.

Janapsatya et al. [78] measure performance by accumulating access times of SPMs and caches (e.g., the execution time is 1μs if a cache with access time 1ns is accessed 1000 times). However, since the access times of all SPMs and caches they evaluated are faster than 1.7ns in a 0.18μm process, SPMs' faster access time will not convert into fewer cycle counts in most contemporary embedded processors.

Udayakumaran et al. [147] compare the execution time of SPMs and caches occupying the same area. However, in our evaluation, the energy optimal capacity of L0 SPMs and caches are 1KB, which does not make area of L0 SPMs or caches critical unless the processor design is extremely area constrained.

A large body of SPM-related work [41, 42, 78, 133, 139, 147, 153, 156] focuses on *substituting* L1 instruction caches. We instead focus on *extending* the memory hierarchy by adding another level (L0) with SPMs and comparing this to FCs. This is because in L0 the majority of instruction reuse comes from loops, for which the compiler has a proven ability to optimize [8, 36, 93, 132].

As SPM size increases, it is harder for compiler to achieve a hit rate similar to that of comparably-sized caches due to the lack of tags in SPMs. Nevertheless, we can use L0 SPMs on top of L1 SPMs, and the approach presented by Egger et al. [42] looks particularly promising as an L1 SPM management scheme. Since their μTLB effectively acts as a storage for tags with a coarse granularity, we believe that their design can achieve a similar hit rate to that of caches even if we increase the capacity more than a few KBs. However, their design is not suitable for L0 SPMs because L0 SPMs' small capacity will enforce small page size (e.g., 16 instructions per page), resulting in a significant internal fragmentation.

3.2.2 Loop Caches

As an alternative to software-managed SPMs, loop caches [99, 100] can be used as L0 instruction stores. Loop caches dynamically identify loops by observing backward jumps and store the identified loops to filter out costly access to larger stores. Loop caches serve well for applications in which straight-line loops dominate the performance. However, loop caches cannot store loops with if-else branches, and Gordon-Ross et al. [53] demonstrate that this is too inflexible in dealing with diverse embedded applications. Gordon-Ross et al. [53] address this inflexibility by pre-loading performance critical loops with arbitrary shapes. However, the pre-loaded loop caches [53] cannot overlay loops in different program phases, and thus cannot efficiently use the loop cache capacity as Ravindran et al. [133] show.

The loop stream detector in Intel's Core and Nehalem micro-architecture resembles loop caches: it dynamically detects loops that fit a small buffer (18 instructions in Core and 28 instructions in Nehalem) and fetches instructions in the loops from the small buffer instead of from the L1 instruction cache. In contrast to loop caches [99,100], the loop stream detector is able to capture the locality from loops with branches. However, the loop stream detector cannot capture the locality from a part of a loop that does not fit the buffer capacity. In addition to avoiding costly accesses to the L1 instruction cache, the loop stream detector allows bypassing the front-end of processor pipelines including branch prediction and decoding from x86 instructions to micro operations (the latter is supported only in the Nehalem micro-architecture). Similar optimizations on the front-end of CISC pipelines with branch prediction can be applied in the conjunction with instruction SPMs. Bypassing branch predictions can be conducted by a structure separated from the SPM that detects loops. The instruction SPM can store micro operations decoded from CISC instructions so that expensive decodings can be performed when instructions are copied into the SPM instead of when they are fetched from the SPM.

3.2.3 Tagless Hit Caches

Hines et al. [70] propose an L0 instruction store design called tagless hit instruction cache (TH-IC). TH-ICs determine if an instruction fetch will be a hit by looking up its metadata, which consumes less energy than checking tags. The metadata include *Next Target* bit (NT), one of which is associated with each instruction present in the TH-IC. When a branch is taken and when the associated NT bit is set, it is guaranteed that the branch target resides in the TH-IC. The metadata also include *Next Sequential* bit (NS), one of which is associated with each cache line. When the control falls through from the last instruction of a cache line and when the associated NS bit is set, it is guaranteed that the subsequent instructions reside in the next cache line. However, the authors do not report how much energy is spent on maintaining the metadata and the control logic. Even if we completely ignore this energy and use the best policy reported in [70] (TL policy), our evaluation shows that their best energy reduction is 83%, which is smaller than that of SPMs with our fine-grain dynamic instruction placement algorithm.

3.3 Analytic Comparison of Miss Rates of SPMs and Caches

This section discusses advantages and disadvantages of SPMs compared to FCs with respect to miss rates.

We can evaluate L0 instruction stores using the following three metrics: (A) the L0 access energy, (B) the L0 miss rate, and (C) the L0 performance penalty typically due to stall cycles that occur when there is a miss in the L0 store. Ideally, we want an L0 store that simultaneously optimizes all three metrics. SPMs achieve low access energy (metric A) because they do not require tags, and low performance overhead (metric C) because the compiler can proactively load instructions from the L1 cache. However, previous SPM placement algorithms and loop caches have failed to achieve miss rates (metric B) that are competitive with filter caches.

FCs have a clear advantage to achieve lower miss rates by having tags since the

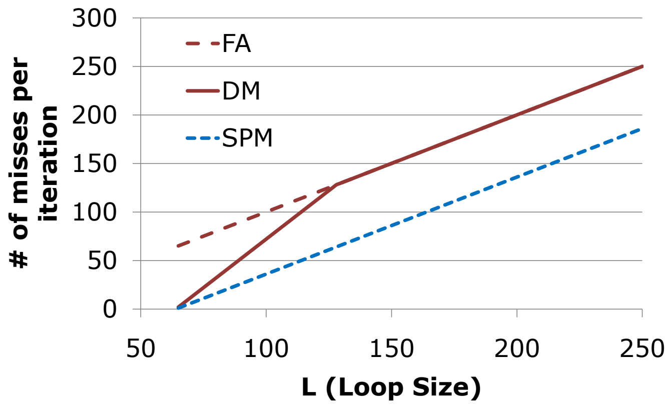


Figure 3.4: The number of misses per iteration for straight-line loops where $\boldsymbol{C}$ (the capacity of stores) is 64.

tags allow avoiding unnecessary L1 misses for instructions that are already cached. On the other hand, SPMs are more flexible in mapping from memory addresses to SPM locations since the mapping is completed controlled by the compiler, thereby reducing conflict misses. Although FCs' mappings can also be indirectly controlled by several code layout techniques [47, 60, 74, 111, 127], SPMs have more flexibility by decoupling of memory addresses and SPM locations. In addition, SPMs can avoid pollution from less-frequent instructions by bypassing them. This bypassing can be achieved by assigning a portion of address space to the SPM and fetching instructions from it if the program counter points to an address assigned to the SPM. Although the same bypassing can be also achieved with FCs, it hampers the advantages of FCs with respect to instruction set compatibility and non-compulsory compiler modification.

For a straight-line loop of length L, the following equations compute the number of misses (except compulsory misses) per iteration for a fully associative FC with least recently used replacement policy (y_{FA}), a direct-mapped FC (y_{DM}), and an SPM (y_{SPM}), all with capacity C. We assume that C instructions of the loop are placed in the SPM and the other L - C instructions are directly fetched from L1.

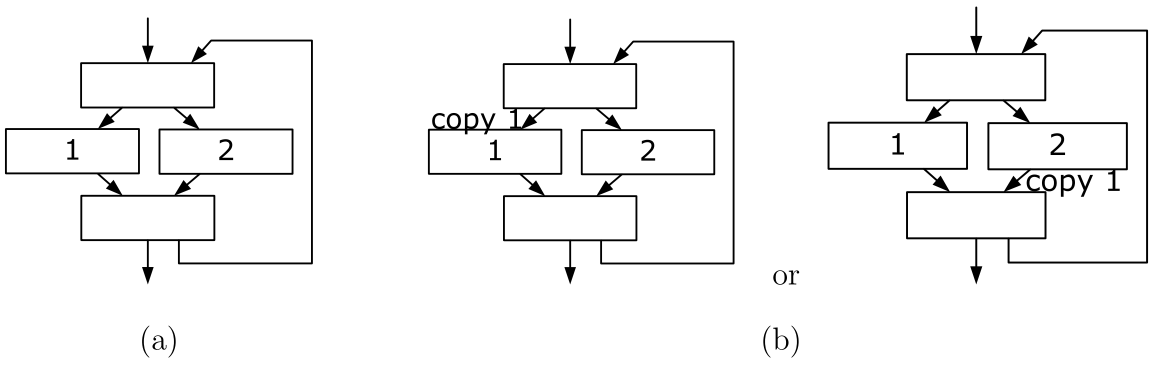


Figure 3.5: (a) A simple loop and (b) the result of placing blocks 1 and 2 of the loop in the same SPM location.

$$y_{\text{FA}} = \begin{cases} 0 & \text{if } L \le C, \\ L & \text{if } L > C \end{cases} \tag{3.1}$$

$$y_{\text{DM}} = \begin{cases} 0 & \text{if } L \le C, \\ 2(L - C) & \text{if } C < L \le 2C, \\ L & \text{if } L > 2C \end{cases} \tag{3.2}$$

$$y_{\text{SPM}} = \begin{cases} 0 & \text{if } L \le C, \\ L - C & \text{if } L > C \end{cases} \tag{3.3}$$

Figure 3.4 plots these equations when C is 64 and shows that SPMs incur fewer conflict misses in straight-line loops. The fewer conflict misses are achieved by placing only C instructions in the SPM, while the other L - C instructions bypass it through the path from L1 to the processor shown in Figure 3.1(a).

While we can easily minimize miss rates of SPMs for straight-line loops, the same optimization is not trivial for codes whose control flow is less regular. In fact, it is quite a challenge for SPMs to achieve fewer misses than FCs for less regular code.

Consider a loop shown in Figure 3.5(a) that typically executes 1 during its first half of iterations and 2 for the second half. For FC, placing 1 and 2 in memory addresses mapped to the same FC location incurs few conflict misses. In other words, instructions 1 and 2 have a weak *temporal relation* [47]. Gloy et al. [47] minimize instruction cache conflict misses by finding a layout in which instructions mapped to the same cache location have weak temporal relations.

We cannot, however, apply the same optimization scheme to SPMs due to their lack of tags. In fact, we show that it is always disadvantageous to place multiple instructions in a single-level loop (a loop without inner loops) at the same SPM location. In this light, an optimal set of instructions to be placed in SPMs with capacity C is the C most frequently executed instructions for single-level loops (proof shown in Section 3.4.4), which motivates our algorithm (described in Section 3.4).

In contrast to Ravindran et al. [133] and Janapsatya et al. [78], we show why temporal relation is not a relevant metric for SPMs as follows: Suppose that we place 1 and 2 in Figure 3.5(a) in the same SPM location. The compiler targeting the SPM must conservatively assume that 2 may be executed between consecutive executions of 1. Consequently, the compiler must copy 1 from L1 to the SPM either immediately before every execution of 1 or immediately after every execution of 2, as shown in Figure 3.5(b). Although many of these copies will unnecessarily transfer instructions that already reside in the SPM, this will not be noticed by the SPM due to the absence of tags.

In summary, while SPMs can achieve lower miss rates for a part of code with regular control flow (straight-line loops are an extreme case), miss rate optimization for SPMs is limited by the lack of tags for other part of code. When we use SPMs as L0 instruction stores for embedded applications, two facts act favorably: 1) the execution time of embedded applications is often dominated by loops and their loops have more regular control flow than those of desktop applications [64], and 2) the majority of instruction reuse that can be captured by small L0 instruction stores come from the loops. Later in this chapter, we show that, due to these two facts, L0 SPMs can achieve lower miss rates than those of FCs for embedded applications but this requires fine-grain instruction placement and careful consideration of the differences between SPMs and FCs (e.g., SPMs lack tags, therefore using metrics such as temporal relation is not useful).

```
**for each** $cfg$ of call graph in a post-order {
  $T$ = construct a loop tree of $cfg$
  re-layout($cfg$, $T$)
  **for each** loop $L$ of $T$ in a post-order {
    $B_L$ = find the longest $B_L$ subject to the constraints shown in Section 3.4.2
    insert copies and jumps for $B_L$
    remove redundant copies and jumps in inner loops
  }
}
```

Figure 3.6: Pseudo code of our fine-grain instruction placement algorithm

3.4 Algorithm

This section describes how our algorithm finds a set of instructions to be placed in SPMs in fine granularity to minimize the number of L1 accesses. We first give an overview of our algorithm, then describe the details of each step.

Our algorithm is executed as a postpass which reads an SPM-unaware assembly code and emits an SPM-aware assembly code. We process each control flow graph by traversing the call graph in a post order. The call graph handles function pointers by adding edges from a function pointer call site to all callees that may be referenced by the pointer. For each control flow graph, we construct a loop tree and re-layout the code. Then, we traverse the loop tree in a post order. For each loop visited, we select instructions to be placed in the SPM, and then insert copy and jump instructions. Figure 3.6 shows pseudo-code of our fine-grain instruction placement algorithm.

3.4.1 Pre-processing

Construct a Loop Tree

We construct a loop tree as shown in Figure 3.7(a) using Havlak's algorithm [61] which can be used for both reducible and irreducible control flow graphs [62]. In the loop tree, the root represents the entire control flow graph, other non-leaf nodes correspond to loops, and leaf nodes are basic blocks. In Figure 3.7(a), there is an edge from {1, 2, 3, 4, 5, 6, 7} to {5, 6, 7} because the latter is an inner loop of the former. There is an edge from {5, 6, 7} to {5} because the loop {5, 6, 7} includes basic block 5. By traversing the loop tree in a post-order in subsequent steps, we process loops in an inner-most loop first order.

Estimate Execution Frequencies

We estimate the execution frequency of each loop tree node as follows: For each loop L, we estimate the average iteration count of L, n_L. We build a *sub-region graph* from the subgraph of the control flow graph induced by L by removing back-edges of L and contracting inner loops to single nodes, as shown in Figure 3.7(b).

For each child x of L, we compute $p_L(x)$, the probability of executing x per execution of the sub-region graph of L. We estimate $p_L(x)$s either by profiling or static analysis. In our static analysis, we propagate $p_L(x)$s starting from the loop header, assuming that each branch direction is independently taken with 50% probability. The execution probabilities annotated in Figure 3.7(b) are estimated by this static analysis. In our static analysis, we set $n_L = \infty$. Section 3.5.3 shows that the energy consumption difference between profiling and the static analysis is less than 5%.

Relayout

After estimating $p_L(x)$s, we re-layout the code so that the C most frequently executed instructions of a loop tree node are contiguous in memory and can be copied to the SPM as a single group, where C is the SPM capacity. In Figure 3.7, blocks are numbered according to the ordering after re-layout. For example, in Figure 3.7(c), blocks 1 and 2 are assigned smaller numbers than block 3 since blocks 1 and 2 are more frequently executed, assuming the execution probability shown in Figure 3.7(b).

To keep the overhead of inserting jump instructions from the re-layout to a minimum, instead of sorting every child, we partition the children into a primary and a secondary partition so that the primary partition contains the C most frequently executed instructions. This partitioning is similar to Pettis and Hansen's *function splitting* [127] used for instruction cache miss optimization. Among the children with

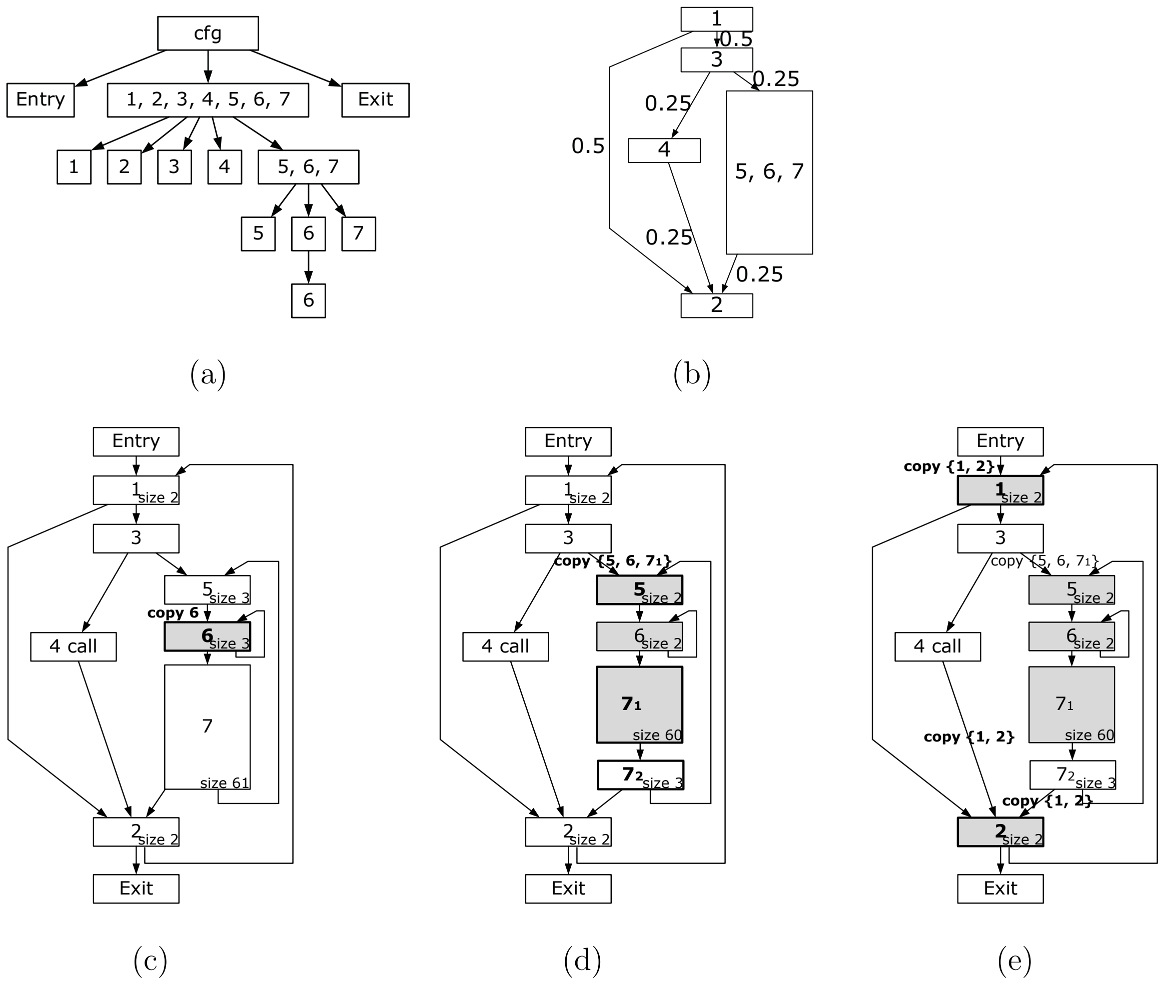


Figure 3.7: Example of placing instructions of loops in a 64-entry SPM. In (c)-(e), instructions placed in the SPM are shaded. Basic blocks are numbered in the ordering after re-layout; we generate the code following this order in our algorithm's assembly output. Block 4 has a function call whose callee uses the entire SPM.
(a) The loop tree of the control flow graph shown in (c)-(e).
(b) The sub-region graph of the outer-most loop, in which edges are annotated with the execution probability per iteration.
(c) A schedule after processing loop {6}. Since this loop fits within the SPM, the entire loop is placed in the SPM.
(d) A schedule after processing loop {5, 6, 7}. Blocks 5 and 6 are placed in the SPM, and the redundant copy of 6 at its incoming edge is eliminated. The first 59 instructions of block 7 are placed in the SPM as well and extracted as a separate block 7_1.
(e) A schedule after processing the outer-most loop. This loop has an inner loop bigger than the SPM and a function call whose callee uses the entire SPM. According to (b), the probability of executing the inner loop or the function call per iteration is 0.5. Since 1 and 2 are the only ones with execution probability higher than 0.5, we place 1 and 2 in the SPM.

the same execution probability, we prioritize for inner loops.

3.4.2 Instruction Placement

For each loop L, we select a block of instructions to be placed in the SPM, denoted as B_L. We find the longest B_L subject to the following three constraints:

1. $|B_L| \leq C$, where C is the SPM capacity.
2. B_L is a contiguous block of instructions starting from the first instruction in L.
3. Let S_{Lx} be the set of inner loops and function calls of L that can overwrite the SPM location mapped to instruction x. $\forall$ instructions $x \in B_L$, $c_1 \cdot p_L(x) > c_2 \cdot (\frac{1}{n_L} + p_L(S_{Lx}))$, where c_1 is the access energy difference between the SPM and L1 cache, c_2 is the energy for copying an instruction from the L1 cache to SPM, and n_L is the average iteration count of L; $p_L(S)$ is the probability of executing any instruction of S and $p_L(\emptyset) = 0$.

The first constraint is trivial[1]. The second constraint and our re-layout method ensure that instructions copied into the SPM at incoming edges of L are the $|B_L|$ most frequently executed ones. Note that this constraint is based on the claim proven in Section 3.4.4 — placing the C most frequently executed instructions of L in the SPM minimizes the number of L1 accesses for single-level loops. We assume that the SPM supports wrapping around the control from its last entry to the first entry, which is important for reducing fragmentation. The third constraint ensures that energy saved by fetching x from the SPM during iterations of L outweighs the cost of copying x at incoming edges to L and at outgoing edges from S_{Lx} (inner loops or function calls that conflict with x). Figure 3.7(e) shows an example of applying the third constraint: 3 is

[1] However, we need to consider the code size increase from inserting copy and jump instructions when we compute $|B_L|$. Since we traverse the call graph and loop trees in post orders, $|B_L|$ can decrease as we eliminate redundant copies and jumps when we visit an outer loop or callee as described in Section 3.4.3. In other words, we conservatively compute $|B_L|$ larger than its eventual value. To reduce the gap between the conservative $|B_L|$ and its eventual value, we apply simple rules such as the following: when the outer loop fits in the SPM, we do not count copy or jump insertions between the current loop and the outer loop since they will be eliminated when we visit the outer loop.

not placed in the SPM because its execution probability does not exceed the probability of executing function call in 4 or executing inner loop {5, 6, 7}, both of which use the entire SPM (i.e., $c_1 \cdot p_{\{1,2,3,4,5,6,7\}}(3) = c_1 \cdot 0.5 < c_2 \cdot p_{\{1,2,3,4,5,6,7\}}(\{4, \{5, 6, 7\}\}) = c_2 \cdot 0.5$, because $c_1 < c_2$). Since we visit the call graph and loop trees in a post order, we can determine S_{Lx}s, except for recursive functions. For recursive functions, we conservatively assume that the callee uses the entire SPM.

Note that we keep our algorithm simple by imposing the following two restrictions: First, the location from which an instruction is fetched is constant regardless of which call graph path or which control flow path was taken. In other words, the placement of an instruction is neither context-sensitive nor flow-sensitive [8]. We can therefore denote instructions placed in SPMs as *SPM instruction*s and the others as *non-SPM instruction*s. Second, at incoming edges of loop L, we copy only one block of instructions, B_L, into the SPM. The only other places where instructions are copied are at outgoing edges from those inner loops or function calls that can overwrite a portion of B_L. As a result, the fine granularity of our algorithm does not cause proliferation of copy instructions and therefore maintains execution time and static code size similar to those of a coarse-grain algorithm, as will be shown in Section 3.5.

3.4.3 Copy and Jump Insertion

After selecting *SPM instructions* (instructions to be placed in the SPM), we adjust the control flow by inserting copy and jump instructions and then eliminate redundant copies and jumps of inner loops.

We first insert copy instructions at incoming edges to the current loop and outgoing edges from conflicting inner loops and function calls. For example, in Figure 3.7(c), we insert "copy 6" at the incoming edge of {6}. Since jumping to the first instruction right after copying a block of instructions is a common case, in addition to `copy` instructions, we support `jcopy` instructions that transfer instructions from the L1 cache to the SPM and jump to the first transferred instruction. To avoid an unnecessary copy at a basic block with outgoing edges with different copy targets, we modify jumps as shown in Figure 3.8. In Figure 3.8, symbolic addresses that start with `@` are

```
@bb_i: ...
...
jump.lt @taken
@nontaken: ...
```

(a)

```
@bb_i: ...
...
jump.lt @bb_i_t
jcopy 17 @nontaken 15
@bb_i_t: jcopy 32 @taken 7
@nontaken: ...
```

(b)

Figure 3.8: Modifying a jump (i.e., creating a launching-pad) to avoid an unnecessary copy when outgoing edges of `bb_i` have different copy targets. (a) Before and (b) after the modification.

mapped to main memory, while the first arguments of `jcopy` instructions denote absolute addresses mapped to the SPM. The third arguments of `jcopy` instructions denote the number of instructions that are transferred by the `jcopy` instructions. When we return from a function to an SPM location, we use an indirect copy instruction whose source memory address and target SPM location are stored in registers.

If either the source or target of a fall-through control flow edge becomes an SPM instruction, we insert jump instructions at the edge. For example, in Figure 3.7(c), we insert an instruction at the end of 5 that jumps to 6. We also insert an instruction that jumps from 6 to 7.

A copy for an inner loop can be redundant after copies for the current loop are inserted. For example, in Figure 3.7(d), "copy 6" at the incoming edge of loop {6} becomes redundant after "copy {5, 6, 7_1}" is inserted at the incoming edge of loop {5, 6, 7}.

Placing instructions of an outer loop in the SPM can render certain jump instructions in its inner loops unnecessary. For example, the jumps from 5 to 6 and from 6 to 7 that are added in Figure 3.7(c) become unnecessary in Figure 3.7(d) because the edges from 5 to 6 and from 6 to 7 are no longer the ones between an SPM instruction and a non-SPM instruction. Therefore, we eliminate the jump instructions as shown in Figure 3.7(d).

3.4.4 Optimality for Single-level Loops

Let G be the subgraph of the control flow graph induced by a single-level loop (a loop without any inner loops), L. We follow Havlak's definition of loop header and back-edges [61]. The header of L is the first visited node of L when we depth-first search the control flow graph to construct a loop tree. The back-edges of L are the edges whose source is in L and whose target is the header of L. When L is a natural loop [62], the header is uniquely defined regardless of the particular depth-first search order used, and it has exactly one back-edge. Let L's header be G's entry. When L is a natural loop, let the source of L's unique back-edge be G's exit. When L is not a natural loop, we add an exit node in G and connect sources of L's back-edges to the exit node.

A set S dominates a node x, denoted by S *dom* x, if every path in G from the entry to x must go through at least one element in S. S post-dominates a set T, denoted by S *pdom* T, if every path in G from an element in T to the exit must go through at least one element in S. A set S vacuously post-dominates a set T if T is empty. Let A_x be the set of program locations in L where a "`copy` x" resides. Let X_i be the set of instructions in L that are placed at the ith SPM location.

Lemma 3.4.1 *For a correct copy schedule,* $\forall x \in X_i$,
(A_x dom x) $\vee$
((A_x pdom $X_i - \{x\}$)) $\wedge$
(x resides in the ith SPM *location at incoming edges of L whose target is L's header)).*

Proof of Lemma 3.4.1. We prove the contrapositive of Lemma 3.4.1. Assume $(A_x\ \neg dom\ x) \wedge (A_x\ \neg pdom\ X_i - \{x\})$. By the definition of *dom* and *pdom*, this assumption implies that the control can follow a path from an element of $X_i - \{x\}$ to x through the loop header without executing any "`copy` x". In this case, the ith SPM location does not hold x when the processor tries to fetch it from the SPM to execute. Assume $(A_x\ \neg dom\ x) \wedge$ (x does not reside in the ith SPM location at incoming edges of L whose target is L's header). This implies that the control can flow from the outside of L to x through the loop header without executing any "`copy` x". $\square$

Let $p(x)$ be the execution frequency of x and $p(S) = \sum_{x \in S} p(x)$. Let the baseline be a schedule such that $\forall x \in L$, $p(A_x) = p(x)$; e.g., copy x right before executing it. If x satisfies the first clause of Lemma 3.4.1 (i.e. A_x *dom* x), then $p(A_x) \geq p(x)$. Therefore, the only way of reducing $p(A_x)$ from the baseline is through the second clause; but at most one instruction in X_i can satisfy the second clause since only one can reside in the ith SPM location at L's incoming edges. Hence, the implication of Lemma 3.4.1 is that, among the instructions placed in the same SPM location, at most one can have fewer L1 accesses than the baseline. Based on this, we can easily prove the following claim.

Claim 3.4.1 *Let C be the capacity of the* SPM *and L be a single-level loop. Placing the C most frequently executed instructions of L in the* SPM *achieves the minimum number of L1 accesses*[2].

3.5 Evaluation

This section describes the experimental setup for our algorithm evaluation and analyzes the results.

3.5.1 Experimental Setup

For our evaluation, we use Elm [12]. Although Elm is a multi-core architecture with an in-order dual-issue pipeline and software-managed memories, we modify the architecture model to a single-core one with a single-issue pipeline and an L1 instruction cache in order to make our evaluation less sensitive from Elm-specific features. We change the compiler and the simulator accordingly. Our algorithm is implemented

[2] It is minimum under the assumption that loop fission and code duplication are not allowed. However, loop fission can be implemented as a separate compilation phase, while code duplication incurs an exponential code size increase in the worst case.

Table 3.2: Evaluated applications.

Category	Benchmark	Description
Automotive	`bitcnts`	Counting the number of bits in an array of integers
	`qsort`	Quick sort a large array of strings
	`susan`	Recognize corners and edges in Magnetic Resonance Images
Consumer	`cjpeg`	JPEG encoding
	`mad`	MPEG audio encoder
Network	`dijkstra`	Dijkstra shortest path algorithm
	`patricia`	Trie data structure used in routing tables
Office	`ispell`	Spelling checker
	`stringsearch`	Searches for given words in phrases
Security	`blowfish`	A symmetric block cipher with a variable length key
	`pgp`	Pretty Good Privacy: a public key encryption system
	`rijndael`	Advanced Encryption Standard (AES)
	`sha`	A Secure Hash Algorithm
Tele-communication	`rawcaudio`	Adaptive Differential Pulse Code Modulation (ADPCM) encoder
	`crc`	A 32-bit Cyclic Redundancy Check
	`fft`	Fast Fourier Transform
	`gsm_encode`	Global Standard for Mobile communication encoder

in `elmcc`, a compiler back-end for Elm that reads fully-optimized LLVM intermediate representation [95].

Table 3.2 lists the applications we evaluate. We use all integer and fixed-point applications of MiBench [55]. We also use `fft` in MiBench after converting its floating point operations to fixed-point ones. Since Elm does not support floating point operations, we exclude the other applications.

Table 3.3 summarizes the configurations used in the evaluation. We compare SPMs with 32 - 512 instructions to fully associative filter caches (FA), direct-mapped filter caches (DM), and loop caches (LC) [99, 100]. For FA and DM, we use 8-instruction (32-byte) cache lines, which achieve the best energy-delay product [49] (under the assumption that the instruction cache consumes 27% of the total energy as in the StrongARM processor [115]). To control for improvements due to code re-layout, we apply the same re-layout method to DMs when it is beneficial. In the COARSE configuration, we evaluate the maximum energy savings that can be achieved by a coarse-grain instruction placement: we assume that SPMs achieve zero miss rates for instructions in loops or functions that fit the SPM. We have two configurations for our

Table 3.3: Experimental Setup

Baseline	No L0 instruction store, 4-way 16KB L1 I-cache with 8-instruction cache lines
LC	Loop cache with flexible loop size scheme [100]
FA	Fully associative FC with LRU replacement policy and 8-instruction (32-byte) cache lines
DM	Direct-mapped FC with 8-instruction cache lines
COARSE	SPM with an ideal coarse-grain placement
FINE_P	SPM with fine-grain placement
FINE	FINE_P without profiling
OPT	Fully associative FC with optimal replacement policy [19]

fine-grain dynamic instruction placement algorithm: the FINE_P configuration uses profiling information, whereas FINE uses a static method for computing execution frequency as described in Section 3.4.1. To provide a lower bound for the number of L1 cache accesses, we include fully associative caches with an optimal replacement policy [19] (OPT). Note that this optimal replacement policy requires an oracle that predicts the future, thus cannot be implemented. In our evaluation, we are able to evaluate the performance of OPT as a theoretical bound by off-line trace-based simulations. We use a 16KB L1 instruction cache with 8-instruction cache lines and 4-way set associativity. The L1 instruction cache with no L0 instruction store is the baseline of our comparison.

We measure the performance of SPMs using our cycle-accurate execution-driven simulator, which is used in Balfour et al. [12]. We measure the performance of FA and DM using the Dinero IV trace-driven cache simulator [40]. We have implemented trace-driven simulators for LC and OPT.

Table 3.4 lists the energy of each operation estimated from detailed circuit models of caches and memories realized in a commercial 45 nm low-leakage CMOS process. The models are validated against HSPICE simulations, with device and interconnect capacitances extracted after layout. Leakage current contributes a negligibly small component of the energy consumption due to the use of low-leakage devices. DMs use

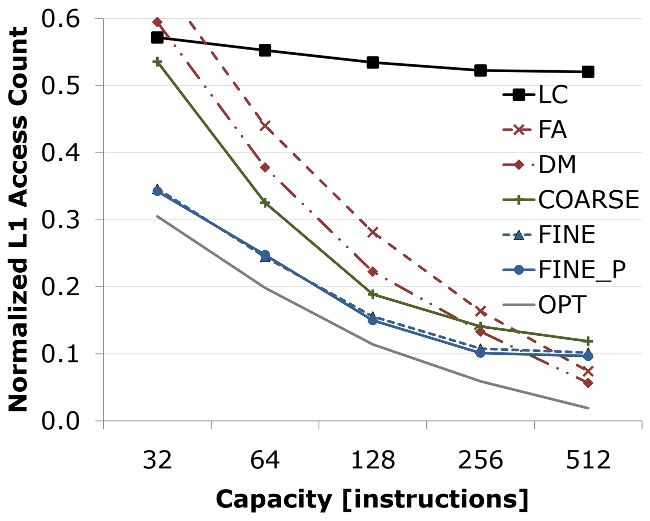
Table 3.4: Energy per operation in pJ. "Refill" is the per cache line size energy for caches and the per word energy for SPMs.

	Hit [pJ]	Miss [pJ]	Refill [pJ]
32-instruction FA	0.28	0.09	3.76
64-instruction FA	0.50	0.17	6.04
128-instruction FA	0.92	0.33	10.60
256-instruction FA	1.74	0.62	19.70
512-instruction FA	3.37	1.18	37.93
32-instruction DM	0.23	0.23	3.73
64-instruction DM	0.39	0.39	5.99
128-instruction DM	0.72	0.72	10.50
256-instruction DM	1.35	1.35	19.55
512-instruction DM	2.64	2.64	37.64
32-instruction SPM	0.11	—	0.33
64-instruction SPM	0.18	—	0.61
128-instruction SPM	0.33	—	1.16
256-instruction SPM	0.63	—	2.26
512-instruction SPM	1.22	—	4.47
16KB L1	20.35	2.68	37.01

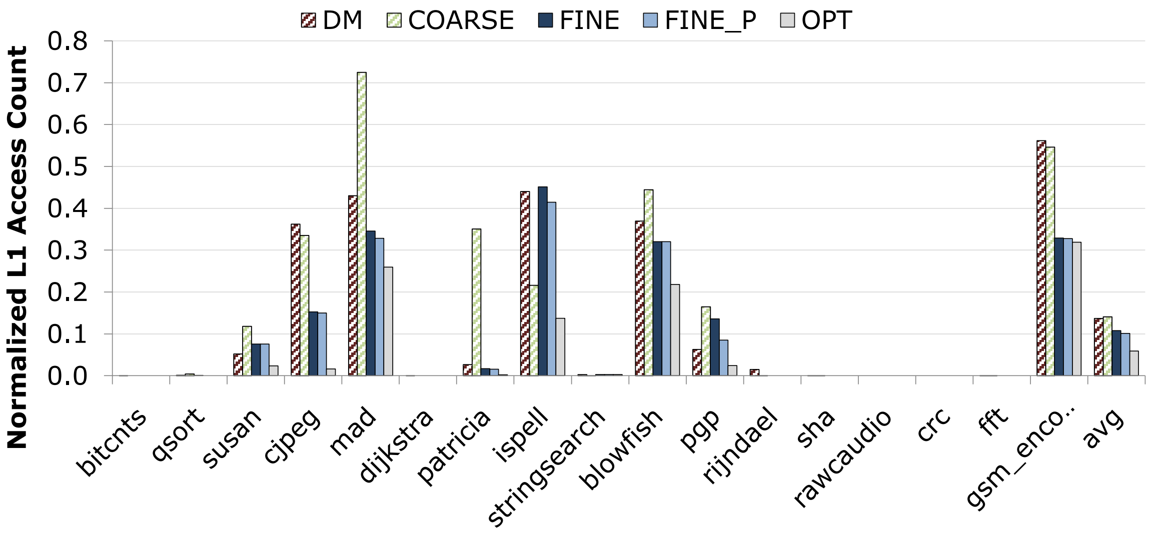
SRAMs to store tags and instructions; the tag array and data array are accessed in parallel, and the tag check is performed after both arrays are accessed. FAs use CAMs to store the tags and SRAMs to store the instructions. FAs and the L1 cache are designed so that the SRAMs is only read when there is a hit in the tag CAM; consequently, a miss consumes less energy, as only the tag array is accessed. When transferring instructions from the L1 cache, the L1 tag is checked once and the instructions are transferred over multiple cycles.

3.5.2 L1 Cache Access

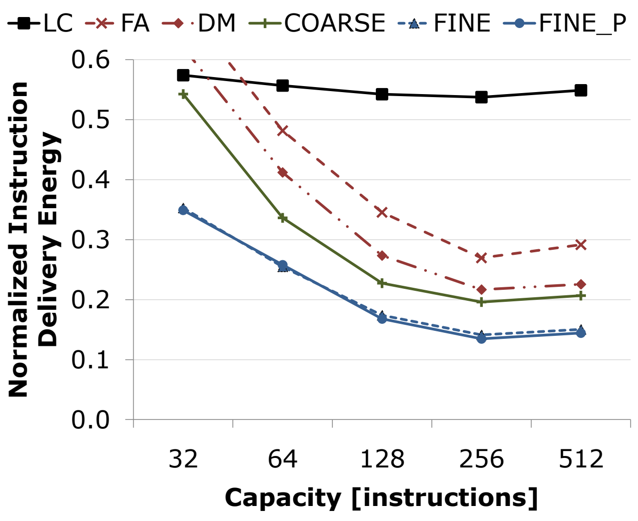
Figure 3.9(a) compares the number of L1 cache accesses for each configuration. The number of L1 cache accesses is normalized to that of the baseline (no L0 store) and accounts for the additional copy instructions in SPM configurations. At smaller capacities, SPMs result in fewer L1 accesses than filter caches since there are many loops whose size is slightly larger than the capacities of SPMs, for which the advantage



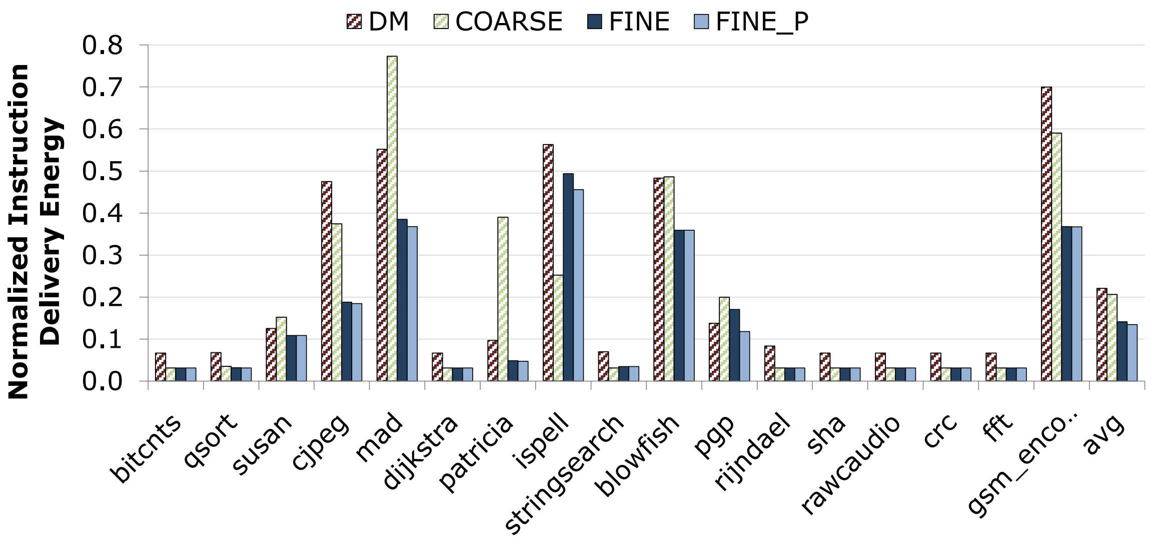
(a) The number of L1 accesses normalized to the baseline.



(b) The number of L1 accesses for 256-instruction configurations normalized to the baseline where no L1 access is filtered.



(c) Instruction delivery energy normalized to the baseline.



(d) Normalized instruction delivery energy for 256-instruction configurations.

Figure 3.9: L1 access and energy consumption results. The averages are obtained by computing arithmetic means over per-instruction-value of each benchmark, then normalizing each mean to the baseline processor configuration.

of SPMs is maximized as illustrated in Figure 3.4. When the capacity is as large as 512, FAs and DMs have fewer L1 accesses because the proportion of instruction reuse not analyzable at compile-time increases as the working set size increases. Despite of its higher associativity, FAs result in more L1 accesses than DMs because the LRU replacement policy works poorly for (almost) straight-line loops that are slightly larger than the cache capacity as described in Section 3.3. LC's L1 access decreases by only 9% as we increase its capacity from 32 to 512 since there are not many straight-line loops larger than 32, which is consistent with Gordon-Ross et al. [53]. If we use fully-associative filter caches instead of direct-mapped filter caches, the number of L1 accesses increases when the capacity is from 32 to 256 and decreases by only 1% when the capacity is 512.

Figure 3.9(b) shows the number of L1 cache accesses for each benchmark for the 256-instruction configurations. Note that we execute whole programs, not just loops. To avoid clutter, we omit LC and FA, which do not show advantages over DM. Our fine-grain instruction placement algorithm (FINE_P and FINE) outperforms DMs on most applications. COARSE suffers from more L1 accesses than DMs on several applications, especially for `susan`, `mad`, `patricia`, and `pgp`. These applications have performance-critical loops bigger than SPMs, where frequently executed portion of the loops can only be captured by a fine-grain placement algorithm or by DMs (e.g., the outer-most loop in Figure 3.7). Although omitted here, we observed that LC works well only if the performance is dominated by straight-line loops (e.g., `sha` and `crc`), but this is too inflexible for other embedded benchmarks. For example, LC exhibits no L1 access reduction for `rawcaudio` because its performance is dominated by a loop with branches.

3.5.3 Energy Consumption

Figure 3.9(c) presents instruction delivery energy for each configuration. Figure 3.9(d) shows the energy consumed in each benchmark for the 256-instruction configurations, where the maximum energy reduction is achieved. FINE_P achieves an 87% reduction, while FA, DM and COARSE achieve 73%, 78% and 80% reduction, respectively.

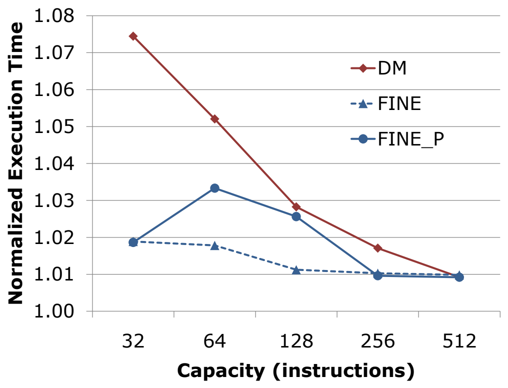
Although COARSE achieves more energy reduction than DMs, their difference, 2%, is small considering the fact that we assume an ideal SPM placement algorithm that achieves zero miss rate (including compulsory misses) in COARSE configuration. Our algorithm without profiling (FINE) does not consume more than 5% additional energy compared to FINE_P, which demonstrates that our fine-grain instruction placement algorithm achieves most of its benefit without profiling. Therefore, our algorithm can be used without profiling by default, allowing programmers to avoid complications from profiling and selecting a representative input data set.

To provide context, an 87% reduction in the energy consumed by the instruction storage hierarchy would result in a 23% reduction in the total dynamic energy consumed in processors such as the StrongARM [115], in which 27% of the total dynamic energy is consumed by the instruction cache.

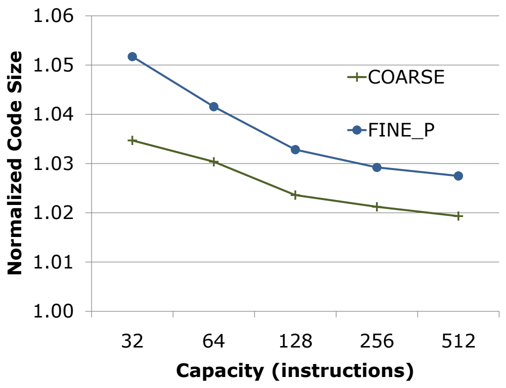
While energy consumption is meaningful as the final metric, L1 access count is the most important variable that the compiler can directly optimize. The relative energy efficiency of an SPM placement algorithm compared to that of FCs varies as the memory hierarchy or circuit design changes. For example, if an SPM placement algorithm achieves smaller energy consumption than FCs despite more L1 accesses, the same SPM placement algorithm may result in worse energy efficiency when we have an L1 cache with a larger capacity or higher associativity, where reducing L1 access is more important that reducing the unit L0 access energy. Conversely, if an SPM placement algorithm achieves smaller energy consumption *with* fewer L1 accesses, its relative energy efficiency compared to that of FCs is less dependent on a specific memory hierarchy or circuit design. Therefore, energy reduction of SPMs must not be reported without the number of L1 accesses in order to measure the energy efficiency of an SPM placement algorithm in a less architecture and circuit dependent manner.

3.5.4 Execution Time and Code Size

To quantify FC's and SPM's impact on execution time, we assume a penalty of 1 cycle for each FC miss as in Hines et al. [70] and Kin et al. [84], and a load-use penalty of 1 cycle for SPMs. Gordon-Ross et al. [53] assume a penalty of 4 cycles for each



(a)



(b)

Figure 3.10: (a) Execution time and (b) code size normalized to the baseline.

FC miss, but the 1 cycle penalty can be achieved by using the critical word first technique [64]. For a copy whose target SPM location is stored in a register (e.g., a copy of function return target), we assume a penalty of 2 cycles. The processor allows one outstanding copy and stalls when a second one is attempted before the first completes. To focus on the aspect of instruction delivery, we disregard L1 data cache miss and branch miss prediction penalty. Within this setup, the 256-instruction FINE_P incurs an average of 1.0% execution time overhead, while 256-instruction DM incurs 1.7% overhead[3]. We optimize the cache line size of DMs for the best energy-delay product [49]. By increasing the cache line size, we capture more spatial locality and miss fewer instructions, resulting in a lower execution time overhead. However, at the same time, this leads to the transfer of more unnecessary instructions from the L1 cache. We find that 8-instruction cache lines balance this trade-off and achieve the best energy-delay product (under the assumption that the L1 instruction cache consumes 27% of the total energy as in the StrongARM processor [115]). For example, by increasing the cache line size from 2 to 8, the execution time overhead of the 256-instruction DM filter cache decreases from 5.8% to 1.7%, while the reduction of energy consumed by the instruction hierarchy changes minimally (from 78.4% to 78.3%).

[3] This is an upper bound of FINE_P's execution time overhead since its baseline is an ideal case without L1 cache and branch miss prediction penalty; e.g., if we assume an L1 cache miss penalty of 32 cycles, a 128-instruction bimodal branch predictor, and a branch miss penalty of 2 cycles, the 256-instruction FINE_P's execution time overhead is reduced to 0.7%.

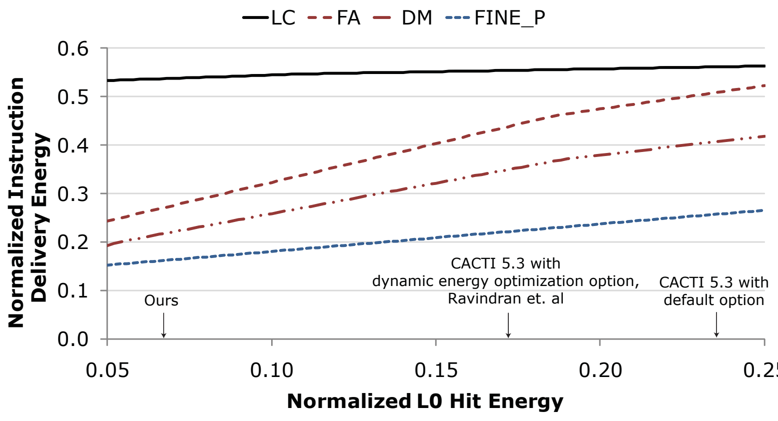


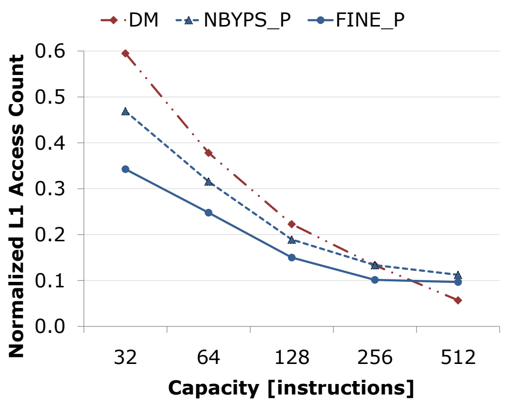
Figure 3.11: Sensitivity of energy saving on the L0 to L1 access energy ratio. Normalized L0 hit energy denotes the hit energy of the 256-instruction DM normalized to that of L1.

Copy instructions increase the code size by, on average, 2.9% with 256-instruction FINE_P and 2.1% with the same capacity COARSE. This demonstrates that the fine granularity of our algorithm does not cause proliferation of copy instructions. Note that, although FINE_P increases the code size, it achieves smaller execution time overhead than DM. This is because FINE_P pre-fetches instructions hiding the L1 access latency.

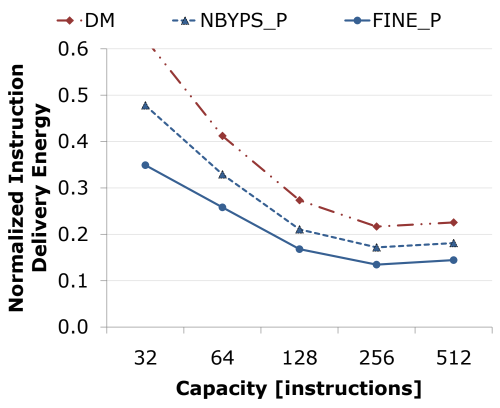
3.6 Discussion

3.6.1 Sensitivity on L1 vs L0 Read Energy Ratio

Figure 3.11 illustrates the sensitivity of energy saving results to the memory energy models and the memory hierarchy configurations by showing how the energy consumption changes as the ratio of the 256-instruction DM hit energy to that of L1 varies. CACTI [163] uses MASTAR (Model for Assessment of CMOS Technologies and Roadmaps) [77] developed by ITRS (International Technology Roadmap for Semiconductors) [76] to estimate device characteristics, while we use models provided by a commercial 45 nm process. Therefore, we cannot directly compare the accuracy of their energy estimations, but we observe that CACTI's estimation of the normalized



(a) The number of L1 accesses normalized to the baseline.



(b) Instruction delivery energy normalized to the baseline.

Figure 3.12: Comparison of FINE_P, NBYPS_P (FINE_P applied to SPMs with no bypassing support), and DM with respect to L1 access and energy consumption.

L0 hit energy is larger than ours [4]. The normalized L0 hit energy can also vary with the capacity or associativity of L1 cache. Since SPMs with our placement algorithm achieve lower L0 access energy *and* lower miss rates than filter caches, they yield consistently better energy efficiency across a wide range of normalized L0 hit energies. In contrast, with an SPM placement algorithm with higher miss rates than filter caches, SPMs can consume more energy than filter caches when the normalized L0 energy is low, even though SPMs with the algorithm were estimated to consume less energy with a high normalized L0 energy (e.g., when CACTI is used or the L1 cache is small).

3.6.2 Effectiveness of Bypassing

SPMs with bypassing support as shown in Figure 3.1(b) have an advantage on reducing L1 accesses by avoiding pollution from instructions with low locality. In addition, bypassing support makes instruction placement algorithm simpler since we do not need to worry about efficiently swap in and out instructions with small temporal locality. A more complicated fine-grain dynamic instruction placement algorithm for

[4] We suspect this is because the cache architecture assumed by CACTI mainly targets caches that are bigger than or equal to typical L1 cache sizes, overestimating energy consumption in small L0 stores.

SPMs without bypassing support is described in our technical report [122]. The key idea behind the algorithm for SPMs without bypassing support is allocating frequently executed instructions to exclusive locations so that they do not conflict with less frequently executed instructions. We call this procedure *pinning*, which is motivated by Lemma 3.4.1.

On the other hand, SPMs without bypassing have several advantages: 1) For bypassing, the processor has to check if an instruction address belongs to the range mapped to the L0 store, which may increase the critical path of the pipeline. 2) With bypassing, the L1 I-cache and the data path between the L1 and L0 should support both word-granularity access and cache-line granularity access. 3) SPMs without bypassing enable a short program counter whose activity factor is very high [12].

Figure 3.12 shows that adding bypassing support reduces normalized number of L1 accesses from 14% to 10% and normalized instruction delivery energy from 17% to 13% at the 256-instruction configuration. Whether adding bypassing support is beneficial depends on how much energy saving we can obtain from the advantages of SPMs without bypassing described above. If the saving is greater than 4% of the L1 cache energy consumption in the baseline configuration, removing bypassing support would make sense. However, the saving in turn depends on specific processor configurations, and whether adding bypassing support in general will save energy remains as future work.

3.6.3 Interaction with Other Architectural Features

Virtual Memory and Context Switching The virtual memory system needs to bypass the translation of addresses mapped to the SPM. We can add a comparator that precedes the TLB and that identifies addresses mapped to the SPM. Alternatively, if the comparator affects the critical path of the processor design, we can add branch instructions that are specifically used for SPM target locations. With the new branch instructions, the processor can change a mode for instruction fetch without the comparator. After a branch instruction targeting an SPM location, the processor changes its instruction fetching mode to "from the SPM". After other branch instructions, the processor changes the mode to "from the L1 cache".

When the operating system switches the context of threads, the SPM-content associated with the old thread need to be saved and the content associated with the new thread need to be restored. Since the typical capacity of L0 instruction SPMs is as small as 256 instructions, the cost of context switching is minimal with a reasonable frequency of context switching. Compared to cache-based designs for L0 instruction stores such as filter caches, the cost of context switching should be similar: cache-based designs do not need to explicitly save and restore the content of L0 instruction stores, but the SPM-content associated with a thread will be evicted anyway when the thread is switched back due to the small capacity of L0 instruction stores. When the capacity of an SPM is as large as that of typical L2 caches, maintaining the context of multiple threads in the SPM can be a challenge, but maintaining the context of multiple threads is not as important for small L0 SPMs.

Debugging When instruction SPMs are used, we need to modify hardware supports for debugging accordingly. For example, when an instruction with a break point is copied into the SPM, we need to add its location in the SPM to the break point list. When the instruction is overwritten, we need to remove its location in the SPM from the break point list. When the break point condition associated with an SPM location is met, continuing the execution within the main memory address space is easier for programmers than executing within the SPM address space. To this end, we need to maintain a mapping from SPM locations of instructions with break points to their main memory addresses. This mapping is also required to generate a core dump which specifies the point of termination with a memory address instead of an SPM location. Evaluating the overhead required for maintaining this mapping remains as future work.

3.7 Chapter Summary

This chapter presents a dynamic instruction placement algorithm for L0 SPMs that shows a notable instruction delivery energy savings (38%) over FCs. This is achieved by 1) fine-grain instruction placement where the length of transfer blocks can be adjusted in increments of one instruction and 2) careful consideration of the tagless and compiler-managed properties of SPMs. Since our fine-grain algorithm achieves 31% instruction delivery energy reduction over even an ideal coarse-grain algorithm, SPMs now have a better chance to become the preferred choice over FCs by providing energy saving that justifies the cost of compiler and instruction set modifications. In addition, processor designers will be able to make well-informed decisions on L0 instruction stores based on our rigorous comparison against the best FC configurations in 17 representative applications and detailed energy model.

Although our algorithm adjusts its block transfer lengths with single instruction granularity and achieves noticeable energy savings over other algorithms, our algorithm is quite simple [5]. In fact, it is quite surprising that no previous work has tried our approach, namely focusing on the simple but most important optimization opportunity: map the most frequently executed instructions of each loop to the SPM. This is because, we believe, the previous algorithms were unnecessarily complicated by blindly formulating the problem as an ILP form or relying on irrelevant metrics such as temporal relation.

As future direction, it would be interesting to see the benefits of hybrid approach, where the copies of a block of instructions from the L1 cache to the SPM are explicitly inserted by the compiler but a tag associated with the block is checked at run-time to avoid unnecessary transfers. Since tags are associated with the blocks, if the typical block size is sufficiently larger than the cache line size, we can keep the tag storage small so that tag lookup overhead is minimal. It would be also interesting to see the synergistic effect of this hybrid approach with block-aware instruction sets such as Zmily and Kozyrakis [165].

In Section 1.1, we have shown that the instruction delivery energy constitutes the largest fraction of energy consumption in conventional embedded processors, which can be significantly reduced by the method presented in this chapter. Nevertheless,

[5]When I presented a shorter version of this chapter at a conference, one of the most common questions was how such a simple algorithm can achieve more energy savings than more complicated previous work.

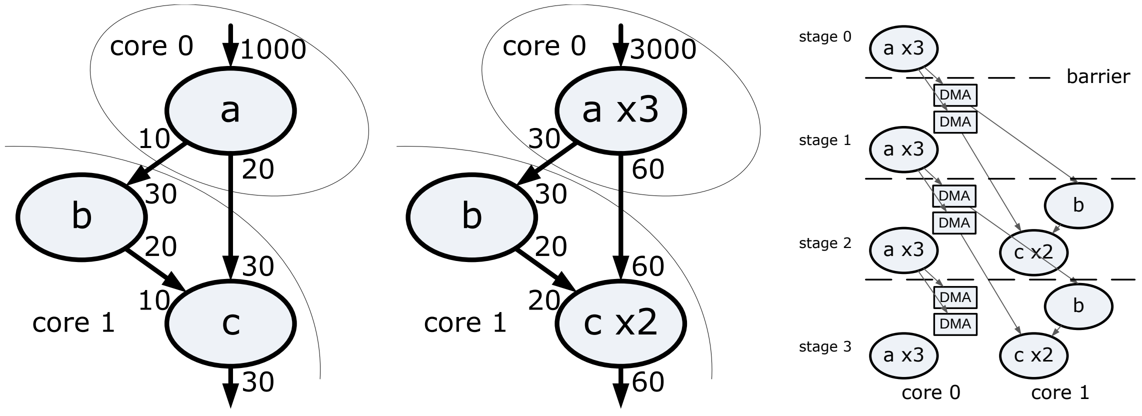
the data delivery energy also constitutes a large fraction. SPMs can be used for reducing the data delivery energy as well using algorithms such as Udayakumaran et al. [147], but SPM placement algorithms for data tend to be significantly more complicated than their counterparts for instructions and have had a limited success for achieving a better energy efficiency than comparably sized caches. In the subsequent two chapters, we focus on how to minimize the additional memory requirement for parallelizing static (Chapter 4) and dynamic (Chapter 5) streaming applications that are commonly found in embedded domain so that the data delivery energy consumption can be reduced.

Chapter 4

Buffers in Static Stream Applications

This chapter presents methods for parallelizing static stream applications with minimal memory space overhead. In static stream applications, the production and consumption rates of actors are close to compile-time constants. We describe an algorithm that computes minimal inter-actor queue capacities that avoid deadlocks and maximize throughput. We present a scheduling algorithm of static stream applications for multi-core architectures called *team scheduling*, which is based on the queue capacity computation algorithm. Compared to previous multi-core stream-scheduling algorithms, team scheduling achieves 1) similar synchronization overhead, 2) smaller buffer spaces for inter-actor queues, and 3) deadlock-free feedback paths. We compare team scheduling to one of the latest stream-scheduling algorithms, SGMS, by evaluating 14 applications on a 16-core Elm processor. Team scheduling successfully targets applications that cannot be validly scheduled by SGMS due to excessive queue capacity requirement (e.g., W-CDMA) or deadlocks in feedback paths (e.g., GSM). Team scheduling consistently satisfies queue capacity constraints imposed by small local memory space of each core in embedded processors, while SGMS fails to do so. For applications that can be validly scheduled by SGMS, team scheduling shows an average of 27% higher throughput within the same local memory space constraints.

A shorter version of this chapter is presented in [124].



(a) A part of stream graph

(b) The minimum steady state

(c) Software pipeline

Figure 4.1: An example of Stream Graph Modulo Scheduling (SGMS). We assume that actor *a* is assigned to core 0 and actors *b* and *c* are assigned to core 1 in the partitioning phase that precedes scheduling. Numbers at each edge denote the number of stream tokens that are consumed or produced per actor firing. For example, actor *b* consumes 30 tokens and produces 20 tokens per firing. DMA denotes direct memory access.

Team scheduling is also estimated to save up to 33% (when 16KB local memories are used) of the energy consumed in memory and interconnection compared to SGMS by avoiding costly non-local memory accesses.

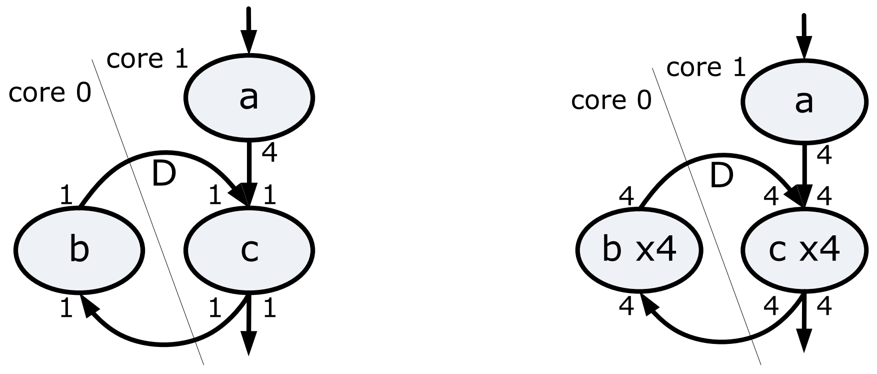
4.1 Overview

Static parts of stream programs, in which the number of tokens consumed and produced per actor firing are compile-time constants, follow the model of computation called *synchronous data flow* (SDF). SDF provides a theoretical background by which we can reduce synchronization overhead and buffer capacities. Lee and Messerschmitt [98] present an algorithm that constructs single-core static schedules with bounded buffers and no synchronization overhead. Bhattacharyya et al. [23] present an algorithm that significantly reduces the buffer requirement of single-core static schedules. For multi-core architectures, [24, 97, 98, 128] present scheduling algorithms based on *homogeneous* SDF *graph* (HSDFG), a graph in which every actor consumes and produces only one token from each of its inputs and outputs [97]. However, constructing an HSDFG from an equivalent SDF graph can take an exponential amount of time [128], and their algorithms do not fully exploit pipeline parallelism [138]. These issues are resolved by Stream Graph Modulo Scheduling (SGMS) implemented in a StreamIt compiler by Kudlur et al. [90].

SGMS applies software pipelining [93,132] to the entire stream graph and synchronizes steady states of the pipeline with barriers. Consider a part of a stream graph shown in Figure 4.1(a). The partitioning phase that precedes scheduling has assigned actor a to core 0 and actors b and c to core 1. Numbers at each edge denote the number of stream tokens that are consumed or produced per actor firing. SGMS first finds the *minimum steady state* [81] in which the number of produced tokens and consumed tokens are balanced at each edge with the minimum number of actor firings. For example, a must be fired three times per b's firing to produce 30 tokens required by b as shown in Figure 4.1(b). An efficient algorithm to find such a minimum steady state is described in Lee and Messerschmitt [98]. After finding the steady state, SGMS constructs a software pipeline as shown in Figure 4.1(c). By starting execution of a producer actor and its consumer actor[1] at different *stages* [132] (a starts at stage 0 while its consumer, b, starts at stage 2), SGMS eliminates intra-stage dependencies so that processor cores do not need to synchronize with each other within a steady state. An actor periodically writes tokens to its output queue, whose data is DMA-transferred at the next stage. Barriers between each stage guarantee that, whenever an actor fires, the input tokens required by the actor are already in place.

SGMS has the advantage of low synchronization overhead (one barrier per steady state), but has the following three drawbacks. First, SGMS requires information that may not be available at compile time. For example, the number of tokens to be produced can vary at run-time for certain streams. We call these *variable-rate streams* (e.g., the output of the Huffman encoder in JPEG). Second, SGMS has little control

[1] More specifically, its consumer actor at a different core since SGMS starts producer and consumer actors at the same stage if they are assigned to the same core.



(a) A stream graph with feedback

(b) Minimum steady state with a deadlock

Figure 4.2: A deadlock in a feedback path caused by SGMS. (a) An example stream graph. "D" at edge (b, c) denotes a single initial token that makes the stream graph deadlock-free. (b) The minimum steady state used by SGMS in which c never fires.

over queue capacities; the minimum queue capacity for each stream[2] is imposed by the minimum steady state. For example, in the steady state shown in Figure 4.1(b), we require queue capacity that accommodates at least 3000 tokens at the incoming stream of actor a. We cannot reduce this queue capacity because the minimum number of a firings between barriers is set to 3 by the steady state. If each core has a 2K-word local memory and the unit token size of a's incoming stream is 1 word, a remote memory must be accessed to further buffer the tokens. This leads to higher energy consumption and less predictable execution time, which makes guaranteeing load balance and real-time constraints at compile-time a challenge. Our evaluation and Lin et al. [107] show that inter-actor queue capacities can grow exponentially in the minimum steady state of real-life applications such as W-CDMA. Third, SGMS does not handle feedback loops satisfactorily. In Kudlur et al. [90], the authors mention that a feedback loop is naïvely handled by fusing the entire loop into a single actor, which results in complete serialization of the loop. If we do not fuse feedback loops into single actors to avoid serialization, SGMS is prone to deadlock. Consider a feedback path $c \rightarrow b \rightarrow c$ shown in Figure 4.2(a). This feedback path is deadlock-free

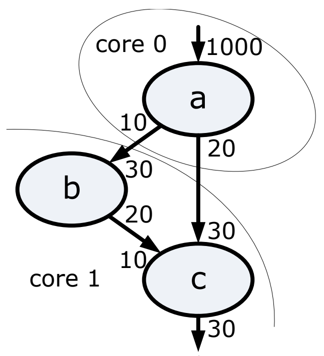
[2] More specifically inter-core stream: the queue capacity of intra-core streams depends on how to schedule actors assigned to a single-core, which is described in [23, 81].

due to an initial token at edge (b, c) denoted as D. The value of the initial token is specified by the programmer and adds a unit delay at (b, c), thus the use of the symbol 'D' commonly found in signal processing [119]. However, in the steady state shown in Figure 4.2(b), actor c cannot be fired because it never receives enough input tokens. The compiler cannot create additional initial tokens because doing so changes the semantic of the application. Our evaluation (Section 4.4) shows that a similar deadlock occurs in a real-life application, GSM.

This chapter presents an alternative algorithm called *team scheduling* that addresses the drawbacks of SGMS, while maintaining a similar synchronization overhead. Team scheduling starts with a simple initial schedule as shown in Figure 4.3(a). Actor firings are pair-wise synchronized through queue empty and full checks. This initial schedule involves high synchronization overhead (i.e., frequent queue empty and full checks). Nevertheless, this is a correct schedule for a wide range of applications including the ones that cannot be validly scheduled by SGMS. Moreover, the synchronization overhead can be minimized with aggregation and amortization of actors as follows. We assume that the partitioning phase precedes scheduling similar to SGMS, where actor-to-core mapping is predetermined when reaching the scheduling phase.

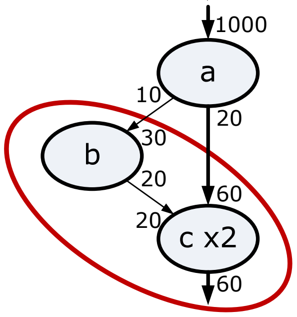
First, we selectively aggregate actors that are assigned to the same core, and form a *team* in which actors are statically scheduled. By statically scheduling actors in a team, we eliminate intra-team synchronizations. For example, in Figure 4.3(b), we form a team by aggregating actors b and c, and eliminate synchronization between them (b_{out0}.isFull() and c_{in0}.isEmpty() checks are removed). We can also eliminate inter-team synchronization such as the one between a and b (explained in Section 4.3.1). In order to construct a static schedule of team $\{b, c\}$, we find its steady state — fire b once and c twice. This is in contrast to SGMS, which must use a steady state of the *entire stream graph*; in team scheduling, the unit of steady state construction is a team whose formation is under the compiler's control. We continue team formation as long as it does not violate constraints such as maximum buffer space per core.

Second, we selectively amortize communication overhead of teams by increasing the number of actor firings per synchronization. Each amortized actor accumulates



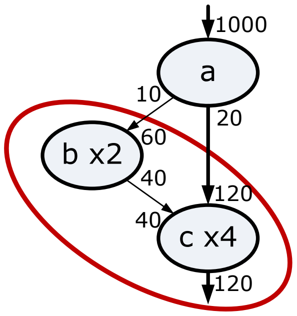
```
while (...) {
  if (!$b_{in0}$.isEmpty() and !$b_{out0}$.isFull())
    doB();
  if (!$c_{in0}$.isEmpty() and !$c_{in1}$.isEmpty() and !$c_{out0}$.isFull())
    doC();
}
```

(a) Initial schedule



```
while (...) {
  if (!$c_{in1}$.isEmpty() and !$c_{out0}$.isFull()) {
    doB(); doC(); doC();
  }
}
```

(b) Team formation



```
while (...) {
  if (!$c_{in1}$.isEmpty() and !$c_{out0}$.isFull()) {
    doB(); doB(); doC(); doC(); doC(); doC();
  }
}
```

(c) Amortization

Figure 4.3: An example team scheduling and its generated code for core 1. (a) An initial schedule (b) Form team $\{b, c\}$ and construct its static schedule, which eliminates synchronization between b and c. Section 4.3.1 describes why synchronization between a and b can also be eliminated. (c) Amortize the team $\{b, c\}$ by a factor of 2.

its output tokens in its local buffer and transfers the accumulated tokens in bulk to the consumer's local memory. For example, in Figure 4.3(c), we amortize team $\{b, c\}$ so that its actors fire twice as often as they do in the minimum steady state of the team. Actor c accumulates 120 tokens in its local buffer and transfers them at once. Amortizing communication overhead is an important optimization scheme, especially for actors with a high computation-to-computation ratio: the locality of memory access is improved, and the fixed cost associated with each data transfer initiation is amortized. In Section 4.4, we show up to 2.1× speed-up from amortization. The same optimization can be done in SGMS but with limited flexibility: minimum amortization factors are predetermined by the minimum steady state; and if we want to amortize an actor by 2, we must amortize all the other actors by 2 as well. Note that the flexibility in choosing amortization factor is crucial for finding the right trade-off between synchronization overhead and queue capacities. For example, in Figure 4.3(c), team scheduling is able to selectively amortize b and c without excessively increasing the queue capacities. On the other hand, SGMS incurs a large queue capacity increase in order to amortize b and c because it must amortize a as well. As we do in team formation, we continue amortization as long as it does not violate constraints such as maximum buffer space per core.

A challenge posed by actor aggregation and amortization is that a deadlock or serialization can be introduced due to insufficient queue capacities. In SGMS, computing queue capacities to avoid serialization is trivial: the difference between the stage of the producer and consumer actor multiplied by the number of tokens produced per producer actor firing. This is because the number of tokens produced per producer actor firing and the number of tokens consumed per consumer actor firing are identical in a steady state. However, when actors are aggregated and amortized in an arbitrary order, the number of tokens produced per producer actor firing and the number of tokens consumed per consumer actor firing are no longer nicely matched. Therefore, we need an algorithm that computes minimum queue capacities to avoid deadlock or serialization in arbitrary stream graph configurations. Here, minimum capacities are desired to reduce memory footprint, particularly in the context of embedded processors since each core commonly has limited local memory space and remote or off-chip memory access incurs a significant energy consumption.

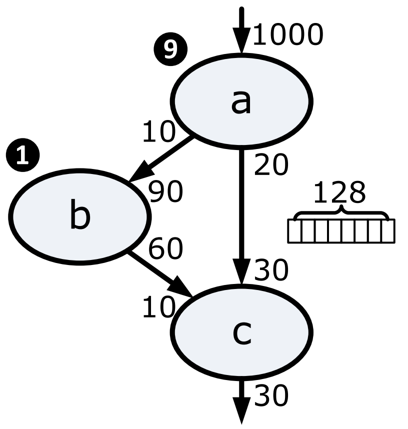
This chapter presents a method for computing minimum queue capacities to avoid deadlock or serialization in static stream applications. As an application of the queue capacity computation method, this chapter presents an algorithm for scheduling stream programs on multi-core architectures called team scheduling that has better control over queue capacities and lower latency than SGMS. We evaluate team scheduling with 14 stream applications on Elm [12]. Our evaluation shows that team scheduling achieves a similar throughput to that of SGMS with lower latency and smaller queue capacities. Our evaluation also shows that team scheduling has better control over buffer space: when we set maximum buffer space per core as a constraint, team scheduling consistently satisfies the constraint while SGMS does not.

The remainder of this chapter is organized as follows: Section 4.2 describes our queue capacity computation algorithm, and Section 4.3 describes the details of team scheduling. Section 4.4 presents simulation results comparing team scheduling with SGMS. Section 4.5 reviews related work and Section 4.6 summarizes this chapter.

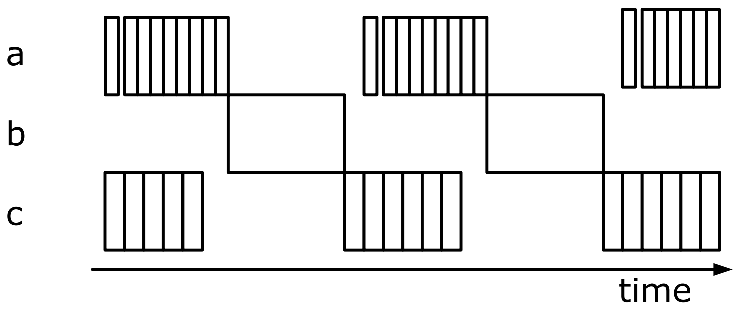
4.2 Queue Capacity Computation Algorithm

When queues have insufficient capacities, deadlock or serialization can occur as shown in Figure 4.4. In Figure 4.4(a), after firing a 6 times, the queue at (a, c) is full and b does not have enough input tokens to be fired, resulting in a deadlock (assume that each actor is assigned to different cores). Note that this is a different kind of deadlock from the ones that occur in the feedback loops shown in Figure 4.2(b). In Figure 4.2(b), deadlock is inherent in the stream graph: we cannot avoid deadlock no matter how large a queue we use for each stream. To avoid the deadlock shown in Figure 4.4(a), we need to increase the capacity of queue at (a, c) to 180. However, this still is not large enough to support serialization-free execution during the latency along path $a \rightarrow b \rightarrow c$ as shown in Figure 4.4(b). To avoid such serialization, the queue capacity must be at least 400. This section presents a method that computes the minimum queue capacity needed to avoid deadlock and serialization.

We need a few definitions from the synchronous data flow (SDF) theory [98]. For a



(a) Deadlock



(b) Serialization when (a, c) queue capacity is 180

Figure 4.4: An example of deadlock and serialization from insufficient queue capacity. ❾ denotes that actor a must be fired at least 9 times to provide enough input tokens for c firing. Assume that the queue at (a, c) can buffer 128 tokens. A deadlock occurs after firing a 6 times. (b) shows a steady state execution when the queue capacity of (a, c) is 180.

stream s, we denote the producer of s as $src(s)$ and the consumer of s as $dst(s)$. We denote the number of tokens produced/consumed per s's producer/consumer firing as $prod(s)$ and $cons(s)$. The *minimum steady state* [81] is where $prod(s)$ and $cons(s)$ are balanced for each s. We can always find the minimum steady state of an SDF application as long as the application is correct (i.e., can be executed with bounded buffers) [98]. The *minimum repetition vector* [98] $\overrightarrow{q_G}$ of stream subgraph G is a vector such that $\overrightarrow{q_G}(a)$ is the number of a firings in the minimum steady state of G ($\overrightarrow{q_G}$ can be denoted as $\overrightarrow{q}$ when G can be unambiguously identified in the context). For example, the minimum repetition vector of the stream graph shown in Figure 4.4(a) is $(9, 1, 6)$ where we index the vector in the order of a, b, and c. We denote the number of tokens produced/consumed at a stream s per minimum steady state as $\overrightarrow{q_G}(s)$.

We first determine the queue capacities of streams along feedback loops. We can bound the queue capacity of a stream along feedback loops s as (see [24])

$$\min_{\text{cycle } C \text{ containing } s} \left(\overrightarrow{q_C}(s) \cdot \sum_{\text{edge } e \in C} \frac{delay(e)}{\overrightarrow{q_C}(e)}\right).$$

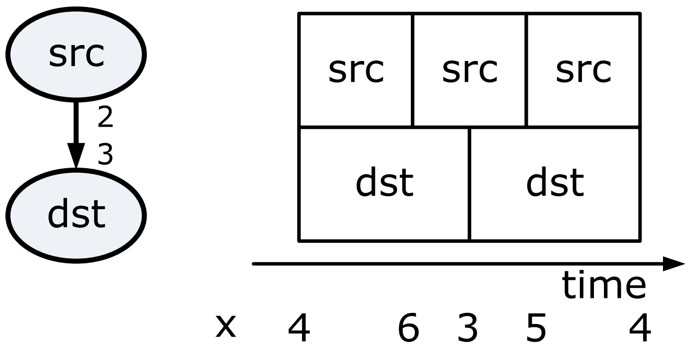


Figure 4.5: A steady state execution of a load balanced producer and consumer pair when $p = 2$, $c = 3$, and $x(0) = 4$.

The intuition behind this equation (Lemma 2 in [24]) is that the number of tokens in any cycle is always conserved up to the amplification factor of each actor (the amplification factor of actor b in Figure 4.4(a) is $\frac{2}{3}$ since 60 tokens are produced per 90 consumed tokens). After bounding queue capacities of feedback streams, we remove an edge from each cycle in the stream graph to construct an *acyclic stream graph*, which is used during the queue capacity computation of other streams .

Second, we compute the queue capacity of each stream by just looking at its producer and consumer pair. The simplest case is a stream s where $prod(s) = cons(s)$. In this case, we need $2 \cdot prod(s)$ queue capacity (a.k.a. double buffering). In general [3] , the queue capacity of each stream s needs to be at least

$$2(prod(s) + cons(s) - gcd(prod(s), cons(s))),$$

when the producer and consumer pair is perfectly load balanced (e.g., each producer firing takes $prod(s)$ time steps and each consumer firing takes $cons(s)$ time steps). The following lemma shows that this capacity prevents serialization between a load balanced producer and consumer pair.

[3] One may wonder whether we generalize too much since either $prod(s)$ or $cons(s)$ divides the other in most cases, which reduces the formula to $2 \cdot max(prod(s), cons(s))$. As will be exemplified by the team scheduling presented in the next section, compiler may want to transform stream graphs and change $prod(s)$ and $cons(s)$, which can result in the case where neither of $prod(s)$ and $cons(s)$ divides the other.

Lemma 4.2.1 *For a load balanced producer and consumer pair that have constant input/output rates and are connected through stream s, $2(p + c - gcd(p, c))$ is the minimum capacity of queue at s to avoid serialization, where p is the number of tokens produced by the producer and c is the number of tokens consumed by the consumer.*

Proof of Lemma 4.2.1. Suppose that s has $x(0)$ tokens at time step 0, when a steady state with respect to the producer-consumer pair begins.

At time step t, the number of tokens at s is

$$x(t) = x(0) + \left\lfloor \frac{t}{p} \right\rfloor \cdot p - \left\lfloor \frac{t}{c} \right\rfloor \cdot c$$

There exists t such that $t \ mod \ p = 0$ (the producer finishes at t) and $x(t) = x(0) + t - \left\lfloor \frac{t}{c} \right\rfloor \cdot c = x(0) + c - gcd(p, c)$ because of the following (where $t = m \cdot gcd(p, c)$ and $c = n \cdot gcd(p, c)$):

$$\begin{aligned} t - \left\lfloor \frac{t}{c} \right\rfloor \cdot c &= m \cdot gcd(p, c) - \left\lfloor \frac{m \cdot \cancel{gcd(p, c)}}{n \cdot \cancel{gcd(p, c)}} \right\rfloor (n \cdot gcd(p, c)) \\ &= (m - \left\lfloor \frac{m}{n} \right\rfloor \cdot n) gcd(p, c) = (m \ mod \ n) \cdot gcd(p, c) \\ &\leq (n - 1) gcd(p, c) = c - gcd(p, c) \end{aligned} \tag{4.1}$$

Note that, there always exists an integer m that holds $m \ mod \ n = n - 1$.

Let l be the queue capacity. Since we need p space to fire the consumer immediately to avoid stalls, $l \geq x(t) + p = x(0) + c - gcd(p, c) + p$.

Similarly, there exists t such that $t \ mod \ c = 0$ (the consumer finishes at t) and $x(t) = x(0) - p + gcd(p, c)$. Since we need c remaining tokens to fire the consumer immediately to avoid stalls, $x(0) - p + gcd(p, c) \geq c$. Therefore, the minimum l that avoids stalls is $2(p + c - gcd(p, c))$, which is achieved by $x(0) = p + c - gcd(p, c)$. Figure 4.5 shows an example where $p = 2$ and $c = 3$. □

For a producer and consumer pair that is not load balanced, $2(p + c - gcd(p, c))$ gives at most twice approximation factor: at the beginning of a steady state, the producer/consumer needs at least p/c space to write/read no matter how widely load imbalanced they are, and, therefore, $2(p+c-gcd(p,c)) < 2(p+c) \leq 2 \cdot$ (lower bound on the buffer space between p and c). The following lemma shows a tighter bound for a producer and consumer pair that is not load balanced.

Theorem 4.2.1 *For a producer and consumer pair where each producer firing takes $0 < p' \leq p$ time steps and each consumer firing takes c time steps, the queue capacity $2p + 2c - (k+2) \cdot gcd(p,c)$ avoids serialization, where $k = \left\lfloor \dfrac{p - p'}{gcd(p,c)} \right\rfloor$. Similarly, when each producer firing takes p time steps and each consumer firing takes $0 < c' < c$ time steps, the queue capacity $2p + 2c - (k+2) \cdot gcd(p,c)$ avoids serialization, where $k = \left\lfloor \dfrac{c - c'}{gcd(p,c)} \right\rfloor$.*

Proof of Theorem 4.2.1. Let us prove the first case since the second one is symmetric. Assume that the producer fires every p time steps, then, similar to Lemma 4.2.1, $l \geq x(0) + p + c - gcd(p,c)$ avoids serialization with respect to the producer.

When the consumer finishes, $t \ mod \ c = 0$, and, similar to Equation 4.1 (let $t = m \cdot gcd(p,c)$ and $p = n \cdot gcd(p,c)$),

$$t \ mod \ p = t - \left\lfloor \frac{t}{p} \right\rfloor \cdot p = (m \ mod \ n) \cdot gcd(p,c) = p - (n - (m \ mod \ n)) \cdot gcd(p,c).$$

For $n-(m \ mod \ n) \leq k$, $t \ mod \ p = p-(n-(m \ mod \ n)) \cdot gcd(p,c) \geq p-k \cdot gcd(p,c) \geq p'$. In other words, the producer is idle and has produced $\left\lfloor \frac{t}{p} \right\rfloor \cdot p + p$ tokens from $t = 0$, not $\left\lfloor \frac{t}{p} \right\rfloor \cdot p$ tokens. Therefore,

$$\begin{aligned}
x(t) &= x(0) + \left\lfloor \frac{t}{p} \right\rfloor \cdot p + p - \left\lfloor \frac{t}{c} \right\rfloor \cdot c \\
&= x(0) + \left\lfloor \frac{t}{p} \right\rfloor \cdot p + p - t \ (\because t \ mod \ c = 0) \\
&= x(0) + p - (t \ mod \ p) \geq c \\
&\Rightarrow x(0) \geq c - p + (t \ mod \ p).
\end{aligned}$$

For $n - (m \ mod \ n) \geq k+1$, $t \ mod \ p = p - (n - (m \ mod \ n)) \cdot gcd(p,c) \leq$

$p - (k + 1) \cdot gcd(p, c) < p'$ (i.e., the producer is not idle). Therefore,

$$\begin{aligned} x(t) &= x(0) + \left\lfloor \frac{t}{p} \right\rfloor \cdot p - t \\ &\geq x(0) - p + (k + 1) \cdot gcd(p, c) \geq c \\ &\Rightarrow x(0) \geq p + c - (k + 1) \cdot gcd(p, c). \end{aligned}$$

Since $k + 1 = \left\lfloor \frac{p - p'}{gcd(p, c)} \right\rfloor + 1 = \left\lfloor \frac{n \cdot gcd(p, c) - p'}{gcd(p, c)} \right\rfloor + 1 \leq n$ ($\because p' > 0$), $p - (k + 1) \cdot gcd(p, c) \geq 0 > -p + (t \ mod \ p)$. Therefore, among all m, m such that $n - (m \ mod \ n) = k + 1$ gives the tightest lower bound on $x(0)$.

Hence, the minimum l that avoids stalls is $2p + 2c - (k + 2) \cdot gcd(p, c)$, which is achieved by $x(0) = p + c - (k + 1) \cdot gcd(p, c)$. □

Notice that when $p' = p$, Theorem 4.2.1 gives $2(p+c-gcd(p, c))$, which is identical to the case with a load balanced pair. When $p = c$, $k = 0$ and we get $p + c$, which is identical to the double buffering case.

After sizing queues locally only based on the information associated with their producers and consumers, we consider global information. The intuition behind the following sequence of procedures is introducing buffers so that delays of any two distinct paths between two actors are *balanced* [135]. For example, along the path $a \rightarrow b \rightarrow c$ in Figure 4.6(a), twice of the minimum steady state (firing a 9 times and firing b once) is required to fire c. In other words, the latency along $a \rightarrow b \rightarrow c$ is twice of the minimum steady state period. We balance the latency along the other path $a \rightarrow c$ by adding 360 buffer space at (a, c). The following describes a step-by-step procedure:

First, we find split-join patterns. An actor is a *splitter* if it has multiple successors, while an actor is a *joiner* if it has multiple predecessors. We define the *split-join pattern* of s and j, G_{sj}, as the actors that are reachable from s and reachable to j. For example, in Figure 4.6(a), $G_{ac} = \{a, b, c\}$.

Second, we compute $x_j(a)$ for each $a \in G_{sj}$, the minimum number of a firings to fire j at least once. In Figure 4.6(a), $x_c(a) = 9$. This can be computed by the

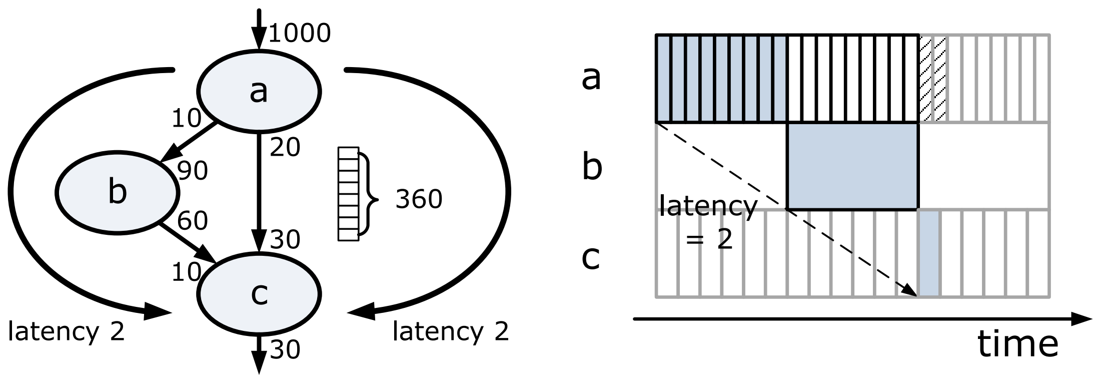


Figure 4.6: An illustration of queue capacity computation algorithm applied to the stream graph shown in Figure 4.4(a).
(a) Intuitively, we balance latencies of paths between a and c. Latencies are normalized to the period of minimum steady state (in this example, the minimum steady state is equivalent to one b firing); the path $a \rightarrow b \rightarrow c$ has latency 2 because we need to fire a 9 times (1 minimum steady state) and b once (1 minimum steady state) to fire c.
(b) A serialization-free steady state execution. We denote actor firings that involve the longest latency between a and c as shaded rectangles. During the longest latency, we need buffer space large enough to sustain serialization-free execution depicted as rectangles with dark boundaries. Our algorithm finds that 360 buffer space is needed at (a, c) to balance the latency between path $a \rightarrow c$ and $a \rightarrow b \rightarrow c$ as shown in (a). On top of 360 buffer space, we need additional 40 for two firings of a depicted as rectangles filled with diagonal lines.

following algorithm: We initialize $x_j(j) = 1$. We traverse G_{sj} in a reverse topological order (recall that we traverse an acyclic stream graph where feedback streams are removed). For each actor a we visit, we compute $x_j(a)$ as follows.

$$\begin{aligned} & x_j(a) \\ &= \max_{\text{successor } b \text{ of } a} (\text{number of } a \text{ firings to fire } b \; x_j(b) \text{ times}) \\ &= \max_{\text{successor } b \text{ of } a} (\left\lceil \frac{x_j(b) \cdot cons(s)}{prod(s)} \right\rceil) \end{aligned}$$

Third, we find the longest latency path from s to j with the latency defined as follows: Let $l_j(a) = \dfrac{x_j(a)}{\overrightarrow{q}(a)}$ be the latency of actor a normalized to the period of repeating the minimum steady state of the application. In Figure 4.6(a), $l_c(a) = \frac{9}{9}$ and $l_c(b) = \frac{1}{1}$. Therefore, the longest latency path is $a \to b \to c$ with normalized latency 2. The actor firings involved in this longest latency path are depicted as shaded rectangles in Figure 4.6: a must be fired 9 times ($x_c(a) = 9$) and b must be fired once ($x_c(b) = 1$), and they together result in the longest latency, twice of the minimum steady state. Buffers along each path from a to c must balance the longest latency to support serialization-free execution. Rectangles with dark boundaries in Figure 4.6 illustrate a serialization-free execution during the longest latency. Denote the number of s firings during the longest latency as $y_{sj} = \lceil \overrightarrow{q}(s) \cdot$ (the longest latency from s to j) $\rceil$. In Figure 4.6(a), $y_{ac} = 9 \cdot 2 = 18$. We use Bellman-Ford algorithm [20] to compute the longest latency, whose time complexity is $O(|V| \cdot |E|)$, where $|V|$ denotes the number of actors and $|E|$ denotes the number of streams.

Fourth, we simulate firing actors in the split-join pattern until s is fired y_{sj} times while streaming as many as tokens as possible to downstream actors. In the simulation, we do not fire j and we set queue capacities of j's incoming streams to infinity. Other queue capacities are set to the ones found by Theorem 4.2.1 or other split-join patterns that are already processed. Let $z_{sj}(i)$ be the number of residual tokens in j's incoming stream i after the simulation, which corresponds to the buffer capacity required at i to balance the latency between s and j so that serialization-free execution during the longest latency from s and j can be supported. During the steady state, the buffer at i requires the following additional space (Theorem 4.2.1), where p denotes the number of tokens produced per i's producer (i.e., *prod*(i)) and c denotes the number of tokens produced per i's consumer (i.e., *cons*(i)):

$$\begin{cases} p + c - (k+1) \cdot gcd(p,c), & \text{where } i\text{'s producer takes } p \text{ time steps and } i\text{'s consumer} \\ & \text{takes } c' < c \text{ time steps per firing, and } k = \left\lfloor \dfrac{c - c'}{gcd(p,c)} \right\rfloor \\ p + c - gcd(p,c), & \text{otherwise} \end{cases}$$

Therefore, we increase the queue capacity of i to $z_{sj}(i) + p + c - gcd(p,c)$ or

$z_{sj}(i) + p + c - (k+1) \cdot gcd(p, c)$ (the latter is used when the pair is not load balanced and the producer is the bottleneck). In Figure 4.6(a), a is fired 18 times during the simulation and leaves 360 tokens at (a, c) $(z_{ac}((a, c)) = 360)$. In Figure 4.6, rectangles filled with diagonal lines depict two a firings that require additional 40 (= 20 + 30 - 10) buffer space. Therefore, we set the queue capacity of (a, c) to 360 + 20 + 30 - 10 = 400.

In SGMS, $prod(i) = cons(i) = 1$ for every stream i since it uses the minimum steady state of the entire application as scheduling units. Therefore, $x_j(a)$, the minimum number of a firings to fire j at least once, is 1 for every joiner j and every actor a inside a split-join pattern G_{sj}. A longest latency path from s to j reduces to a longest path from s to j where each stream in the stream graph has unit length. As a result, our algorithm sets the queue capacity at each stream to (the difference between *stage* numbers of its producer and consumer) + 1, which is identical to the buffer sizing scheme described in Kudlur et al. [90].

4.3 Team Scheduling

This section describes the team scheduling algorithm, which is an application of the queue capacity computation algorithm described in the previous section. Figure 4.7 shows pseudo-code of team scheduling.

4.3.1 Team Formation

We start from an initial schedule in which each actor forms a separate team. For example, in Figure 4.3(a), actors a, b, and c each form teams on their own. In a pair-wise manner, we merge teams in the same core starting with the pair that leads to the highest gain. We compute the gain as synchronization reduction divided by additional buffer requirement resulting from team merge. This is a greedy heuristic chosen to maximize synchronization overhead reduction (i.e., the reduction of queue empty or full checks) per additional buffer space requirement. We maintain a *team graph* that represents the connectivity of teams. The team graph is initially identical

```
construct an initial schedule;
initial queue sizing; // Feedback queues are sized here.

// merge teams
$q$ = a priority queue with pairs of teams that do not introduce a cycle;
**while** (!$q$.isEmpty()) {
 $\langle a, b\rangle$ = $q$.remove();
 **if** (merging $a$ and $b$ does not exceed buffer limit and does not deadlock) {
   $m$ = merge($a$, $b$);
   remove all team pairs containing $a$ or $b$ from $q$;
   **for each** (neighbor $c$ of $a$ or $b$) {
     **if** (merging $m$ and $c$ does not introduce a cycle)
       add $\langle m, c\rangle$ to $q$;
   }
 }
}

// amortize teams
$q$ = construct a priority queue with teams;
**while** (!$q$.isEmpty()) {
 $a$ = $q$.remove();
 **if** (amortizing $a$ does not exceed buffer limit and does not deadlock) {
   amortize $a$;
   add $a$ to $q$;
 }
}
```

Figure 4.7: Pseudo code of team scheduling

to the stream graph, and then we contract the corresponding nodes for each team merge.

We adhere to the following four constraints when merging. First, we do not merge across variable-rate streams. Second, we do not merge teams if doing so exceeds the buffer limit per core. Suppose that actors a, b, and c are all assigned to the same core in Figure 4.3. If each core has a 2K-word local memory and the unit token size of a's incoming stream is 1 word, we avoid merging a with any other actor. The buffer capacities are computed by the method described in the previous section (Section 4.2). Third, we must not introduce a cycle to the team graph since it may result in deadlocks. Suppose again that actors a, b, and c are all assigned to the same core in Figure 4.3. We avoid merging a with c because it forms a cycle $b \rightarrow \{a, c\} \rightarrow b$. Fourth, a merge must not introduce any deadlock in an existing cycle (i.e., in a feedback loop). We can check for such deadlocks by inspecting *precedence expansion graphs* (PEG) [98] of each cycle containing the merged team in the team graph. If every PEG is acyclic, it is guaranteed that the team merge does not introduce any deadlock. This is because a PEG has a cycle if and only if the corresponding stream graph has a deadlock [128]. Suppose that we are about to merge a and c in Figure 4.2(a). If we construct a PEG of the cycle $b \rightarrow c \rightarrow b$ after the merge, we see a cycle in the PEG since merging a and c introduces a deadlock. For the details of PEG construction, refer to Appendix I of [136]. A PEG can grow exponentially when the number of actor firings in the minimum steady state of a team or the number of cycles in the stream graph is exponential. In this case, we use a heuristic described in Pino et al. [128] that conservatively but quickly checks for deadlocks.

After merging a team, we construct a static schedule of the actors within the team. There are several ways of constructing such a single-core schedule [23, 81], but we find that *loose interdependence scheduling framework* (LISF) [22] works well in our evaluation (Section 4.4). For most applications, LISF finds a *single appearance schedule* in which each actor lexically appears only once, resulting in a minimal code size [22] (refer to Section 4.5 for the further details). For example, b $2c$ (fire b once, then fire c twice) is a single appearance schedule of team $\{b, c\}$ shown in Figure 4.3(b). Other single-core scheduling methods such as *push schedule* or *phased schedule* save significant buffer space at the expense of marginal increase in code size when applied to the entire stream graph [81]. However, when we target multi-cores, applications are partitioned into small pieces, and applying either scheduling method to each piece shows little buffer space saving (on average 8% for 16 cores).

By constructing a static schedule of a team, we eliminate intra-team synchronizations such as the one at edge (b, c) in Figure 4.3(b). We can also eliminate certain inter-team synchronizations such as the one at edge (a, b). This is possible because the production-to-consumption ratios of streams between a given team pair (with no variable-rate streams) are constant as proven by the following claim.

Claim 4.3.1 *The production-to-consumption ratios of streams with static rates between a team pair are constant.*

Proof of Claim 4.3.1. For teams T and U in stream graph G, according to Bhattacharyya et al. [23]:

$$\exists \text{ an integer } m \text{ such that } \forall a \in T,\ \overrightarrow{q_G}(a) = m \cdot \overrightarrow{q_T}(a) \tag{4.2}$$

$$\exists \text{ an integer } n \text{ such that } \forall a \in U,\ \overrightarrow{q_G}(a) = n \cdot \overrightarrow{q_U}(a) \tag{4.3}$$

Since the number of tokens produced and consumed at a stream are equal in the minimum steady state of G, $\overrightarrow{q_G}(src(s)) \cdot prod(s) = \overrightarrow{q_G}(dst(s)) \cdot cons(s)$, which is called the *balanced equation* [23]. Substituting Equation (4.2) and (4.3) into the balanced equation shows that, for each stream s from T to U, $m \cdot \overrightarrow{q_T}(src(s)) \cdot prod(s) = n \cdot \overrightarrow{q_U}(src(s)) \cdot cons(s)$. This means that the ratio of the number of tokens produced at s by each T firing $(\overrightarrow{q_T}(src(s)) \cdot prod(s))$ to the number of tokens consumed from s by each U firing $(\overrightarrow{q_U}(dst(s)) \cdot cons(s))$ is a constant, $\dfrac{n}{m}$. $\square$

For example, at $(a, \{b, c\})$ in Figure 4.3(b), production-to-consumption ratios are $\frac{10}{30} = \frac{20}{60}$. To generalize, consider the streams from team T to team U denoted as S_{TU} (in Figure 4.3(b), $S_{TU} = \{(a, b), (a, c)\}$ when $T = \{a\}$ and $U = \{b, c\}$).

Let s_1 be the stream in S_{TU} that is enqueued last in T's static schedule (in Figure 4.3(b), s_1 is (a, c) if T enqueues tokens to (a, b) before (a, c)). Consider the situation when we check conditions to fire U. Assume that we have checked that s_1 is not empty. This implies that all the other streams in S_{TU} are also not empty, which makes checking if any of those streams is empty unnecessary. This can be shown through contradiction as follows. Suppose that the queue at $s_3 \in S_{TU} - \{s_1\}$ is empty. Then due to the constant production-to-consumption ratios of S_{TU}, the queue at s_1 must be empty which contradicts our assumption. Therefore, among the conditions with respect to S_{TU} that we need to check before firing U, we can eliminate everything except the check for whether s_1 is not empty (in Figure 4.3(b), checking whether the queue at (a, b) is not empty is redundant when s_1 is (a, c)).

Let s_2 be the stream in S_{TU} that is dequeued last in U's static schedule (in Figure 4.3(b), s_2 is (a, c)). Suppose that the queue lengths of S_{TU} are proportional to their respective number of tokens produced by each T firing. Consider the situation when we check conditions to fire T. Similar to the argument in the previous paragraph, we can show that, if s_2 is not full, all the other streams in S_{TU} must not be full as well. Therefore, among the conditions with respect to S_{TU} that we need to check before firing T, we can eliminate everything except the check for whether s_2 is not full.

4.3.2 Amortization

After team formation, we amortize communication cost of teams starting from the one that leads to the highest synchronization reduction per additional buffer requirement. As in the team formation procedure, we do not amortize a team if doing so exceeds buffer space limit or introduces deadlock in a feedback path.

We define amortization as follows: For each stream subgraph G that is statically scheduled (the entire stream graph in SGMS or a team in team scheduling), we define the *repetition vector* $\overrightarrow{r_G}$ such that $\overrightarrow{r_G}(a)$ is the number of a firings in the current static schedule of G. We call $\overrightarrow{r_G}(a)$ the *repetition* of actor a. For example, the repetition vector of team $\{b, c\}$ in Figure 4.3(b) is the same as its minimum repetition vector, $(1, 2)$, because the team has not been amortized. In Figure 4.3(c), the repetition vector of team $\{b, c\}$ is $(2, 4)$. In this chapter, *amortization* of stream subgraph G

by a factor of k means multiplying G's repetition vector by k. For example, in Figure 4.3(c), amortization of team $\{b, c\}$ by a factor of 2 has updated its repetition vector from $(1, 2)$ to $(2, 4)$.

Note that in SGMS the repetition vector is identical to the minimum repetition vector of the stream graph before any amortization. If we amortize a schedule by a factor of 2, we multiply the repetition of *every* actor by 2. In team scheduling, each team has its own repetition vector, and each team is amortized separately.

We use the following method of selecting amortization factors: Suppose that we are about to amortize team T in stream graph G. If $\exists$ an integer $k > 1$ such that $\forall a \in T,\ \overrightarrow{q_G}(a) = k \cdot \overrightarrow{r_T}(a)$, we amortize T by the smallest integer greater than 1 that divides k. For example, for team $\{a\}$ in Figure 4.3(b), $\overrightarrow{q_G}(a) = 3$ and $\overrightarrow{r_{\{a\}}}(a) = 1$, thus $k = 3$. Otherwise, we amortize T by a factor of 2. We use this method in order to first amortize T up to the minimum steady state of the entire graph and to additionally amortize T by a factor of 2 thereafter. This is in turn motivated by our conjecture that schedules resemble the minimum steady state are preferable when other factors such as queue capacities are the same.

4.3.3 Time Complexity

The time complexity of team scheduling shown in Figure 4.7 is dominated by the longest path algorithm for buffer requirement computation. Let $|V|$ be the number of actors and $|E|$ be the number of streams. We compute buffer requirement $O(|V|log(b))$ times (lines 8 and 22), when b is the buffer space limit per core and we amortize each team at least by a factor of 2. As described in Section 4.2, we use Bellman-Ford algorithm [20] to compute the longest distance, whose time complexity is $O(|V| \cdot |E|)$. Bellman-Ford is invoked $O(|V|)$ times per each buffer requirement computation; thus the time complexity of team scheduling is $O(|V|^3|E|log(b))$.

The time complexity for cyclicity checks is $O(|V|^3)$: cyclicity check is done $O(|V^2|)$ times (line 12) and each check takes $O(|V|)$ by using reachability matrix [23]. The time complexity of finding split-join patterns is $O((|E|log(|E|) + |V|^2)|V|log(b))$: we apply the concept of *dominance frontier* used for constructing *static single assignment*

Table 4.1: Evaluated Applications.

Benchmark	Description	Input
`bitonic_sort`	A parallel bitonic sorting	sort 8 integers 2,560 times
`channel_vocoder`	A channel voice coder	16 filters. 6K floats
`dct`	A 16×16 IEEE reference DCT	80 16×16 integer blocks
`des`	A pipelined version of the DES encryption algorithm	4K integers
`fft`	A 256-element FFT	12 blocks
`filterbank`	A filter bank for multi-rate signal processing	8 filters. 2.5K floats
`fmradio`	An FM radio with an equalizer	6-band equalizer. 18K integers
`gsm`	A Global Standard for Mobile communication encoder	48 blocks, 160 integers each
`mpeg2`	The block and motion vector decoding parts of an MPEG-2 decoder	A 352×240 frame
`radar`	A radar array front-end	12 channels, 256 samples each
`serpent`	A serpent encryption algorithm implementation	8K integers
`tde`	A time delay equalization for a radar processing application called Ground Moving Target Indicator (GMTI)	6 channels, 36 samples each. 16K complex numbers
`vocoder`	A bit-rate reduction vocoder	15 filters, 4K floats
`w-cdma`	The searcher part of Wideband Code Devision Multiple Access protocol	15K complex numbers

form in the compiler [37], which takes $O(|E|log(|E|) + |V|^2)$ [37, 102].

If the time complexity is unacceptable (e.g., in just-in-time compilation), we can stop during algorithm execution since we maintain a valid schedule throughout the algorithm execution.

4.4 Evaluation

This section describes the experimental setup for evaluating team scheduling and the analysis of results.

4.4.1 Experimental Setup

We use the same set of StreamIt benchmark applications that were used for SGMS evaluation [90] plus GSM encoder and W-CDMA searcher [157] (shown in Table 4.1. We have ported the StreamIt applications into the Elk stream programming language that extends StreamIt (the details of Elk is described in Section 2.2.2). The ported source code is available at our Elk website [2]. During the porting, we occasionally made structural modifications when the original implementation in StreamIt was inefficient. For example, actors in the DES implementation in the StreamIt benchmark communicate data bit by bit, which is too inefficient unless the target machine supports bit-level communication or the compiler is able to aggregate the the bit-level communication in words. We modified the DES implementation so that data are transfered in word-level. W-CDMA searcher is ported from a proprietary benchmark from Qualcomm^{TM}. GSM encoder is ported from MiBench [55] and contains a feedback path. We use the input size that is large enough to execute at least one iteration of the minimum steady state of an SGMS and a team schedule.

We use Elm [12] that supports DMA-like stream memory instructions that transfer a block of data to other cores' local memory in the background, and these stream memory instructions are used to implement queue operations. Elm has an ensemble organization in which four cores share their local memory. We made each core have its own separate local memory, changed the local memory size to 256KB, and used 16 cores to make the evaluation setup similar to that of the SGMS paper [90] which uses Cell processors.

The Elk compiler generates C++ code from Elk code, and an LLVM-based [95] C++ compiler called `elmcc` [122] generates Elm assembly code. The assembly code is executed in a cycle-accurate Elm simulator. We model interconnection as a mesh network with word-wide channels and canonical 4-stage pipeline routers [39]. Therefore, the latency of a message is $4(d + 1)$ cycles if the Manhattan distance to the destination is d. For SGMS, we idealistically assume that every core can access a dedicated memory in 1 Manhattan distance latency (8 cycles), and we implement a sense-reversing barrier [66] using `fetch-and-add` instructions on the dedicated memory. This results in 75 cycles per barrier while each barrier takes 1600 cycles in the

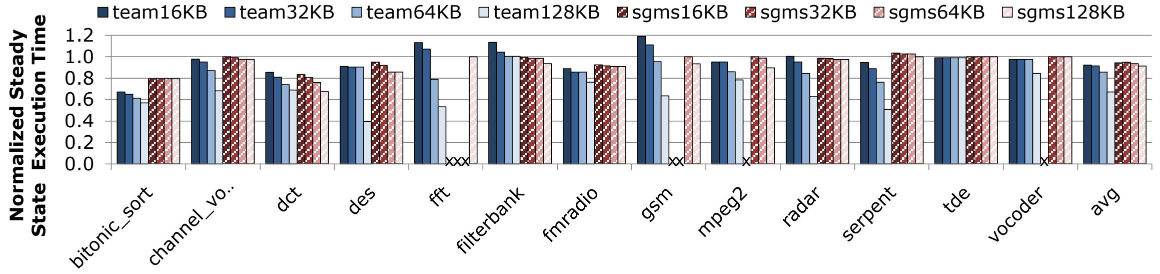


Figure 4.8: Steady state execution time when team scheduling and SGMS amortize actors within different buffer space constraints. Execution times are normalized to those of the original SGMS algorithm without any amortization.

SGMS paper [90].

As in previous work [51, 52, 90], partitioning is done before scheduling. We first fission stateless actors with high computation requirement so that every stateless actor has at most 1/16 of the total computation requirement. Then we assign actors to cores using METIS, a graph partitioning package [82].

In the first experiment, we compare the throughput of team scheduling with that of SGMS as we change the buffer space limit per core from 16KB to 128KB. This experiment measures the efficiency of using limited local memory space, which is critical for multi-core embedded processors. In the second experiment, we intentionally avoid exploiting the amortization flexibility of team scheduling by limiting the maximum repetitions to the ones in the minimum steady state of the entire stream graph, and compare throughput, latency, and buffer usage of the two algorithms. This experiment compares performance of the two scheduling algorithms independent of amortization effects.

4.4.2 Buffer Space Limited Experiment

Figure 4.8 compares throughput of both algorithms as we change the buffer space limit per core from 16KB to 128KB. Steady state execution time is measured as the time between the generation of the first and the last output of the furthest downstream actor, and is inversely proportional to throughput. Steady state execution times are normalized to those of SGMS without any amortization. The average speed-up from

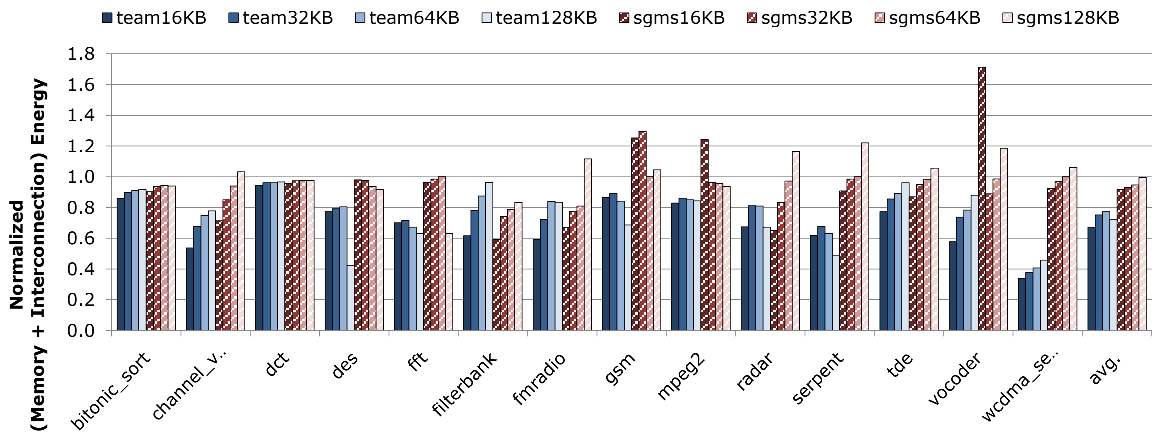


Figure 4.9: Estimated energy consumptions in memory and interconnection normalized to those of the original SGMS algorithm without any amortization and with 64KB local memories.

single-core executions is 11×. The results for WCDMA are not shown here because SGMS requires an excessive buffer space even without amortization, which will be shown in Section 4.4.3. We fuse feedback loops in GSM to single actors for SGMS since SGMS results in a deadlock similar to that shown in Figure 4.2(b) without fusion. For the particular case of GSM, SGMS does not show its disadvantage with respect to complete serialization of feedback loops since the feedback loops in GSM do not have parallelism that can only be exploited by team scheduling.

SGMS does not satisfy the buffer space constraint for `fft`, `gsm`, `mpeg2`, and `vocoder` when the space constraint is as small as 16KB. In contrast, team scheduling satisfies buffer space constraints across all configurations. The averages are computed only on the applications that satisfy the buffer space constraint with both scheduling algorithms (e.g., the averages for `team16KB` and `sgms16KB` are computed excluding `fft`, `gsm`, `mpeg2`, and `vocoder`). When the buffer space limit is as large as 128KB, team scheduling achieves an average of 37% higher throughput than that of SGMS, which is especially apparent in `des`, `fft`, `gsm`, `radar`, and `serpent`. We can see the importance of amortization from its up to 2.1× speed-up (`serpent` at `team128KB`).

As mentioned in Section 4.3.1, there are several ways to schedule actors that belong to a team (in team scheduling) or that are assigned to the same core and the

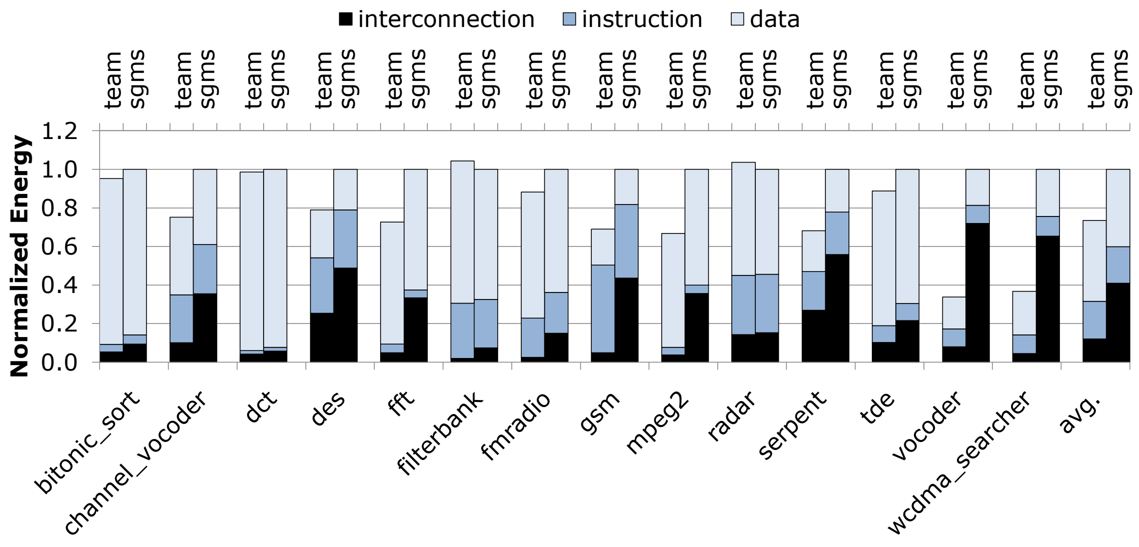
Table 4.2: Energy of operations in pJ that is estimated in a 45nm low-leakage process. SPM and SRAM access energy is per word (32-bit) except for 8MB SRAM whose access energy is per line (16-word). DRAM access energy is per line and is estimated using the results from Udipi et al. [148]. The interconnection energy is per hop per word.

Energy per operation [pJ]	read	write
256-entry SPM	0.63	2.26
16KB SRAM	3.02	2.76
32KB SRAM	3.92	3.22
64KB SRAM	6.31	5.43
128KB SRAM	8.43	6.83
8MB SRAM (16 tiles)	96.95	83.81
4GB off-chip DRAM	25,067.00	25,067.00
1 hop in interconnection	14.29 (2.23 channel + 12.06 router)	

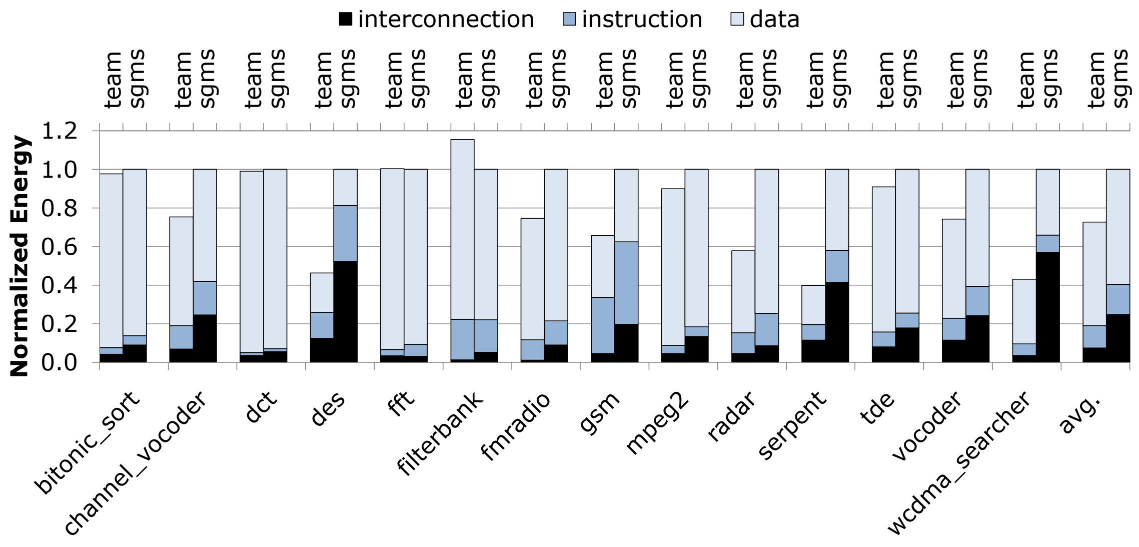
same software pipeline stage (in SGMS). However, push schedule — the most buffer space efficient single-core schedule [81] — saves only an average of 8% of buffer space compared to single appearance schedule. In addition, push schedule does not make SGMS meet buffer space constraints of any application which does not already satisfy the constraint in single appearance schedule. If we apply push schedule to the entire stream graph, we achieve a significant reduction in buffer space requirement compared to single appearance schedule as shown in Karczmarek et al. [81]. However, when we target 16 cores, the application is already partitioned into 16 pieces, and there is little difference between using either scheduling method on each small piece.

Figure 4.9 shows estimated energy consumption in the memory and interconnection subsystem. Figure 4.10 shows estimated energy consumption breakdown when the local memory capacity is 16KB (Figure 4.10(a)) and 128KB (Figure 4.10(b)). Notice that this is a rough estimation to illustrate an energy saving trend achieved by team scheduling. The energy consumption is estimated by the following equations:

$$\text{data memory energy} = \sum_{x \in \{\text{local, L2, DRAM}\} \times \{\text{read, write}\}} (\#\text{ of data accesses in } x) \times (\text{access energy of } x)$$



(a)



(b)

Figure 4.10: Energy consumption breakdown when the capacity of local memory is (a) 16KB and (2) 128KB. Energy consumptions are normalized to those of SGMS.

instruction memory energy =

$$\sum_{x \in \{\textsc{spm}, \text{ local, L2, } \textsc{dram}\} \times \{\text{read, write}\}} (\text{\# of instruction accesses in } x) \times (\text{access energy of } x)$$

interconnection energy =

(sum of # of hops of word-sized phits[4]) $\times$ (energy per hop per word)

Table 4.2 lists the energy of each operation per word. SRAM access energies are estimated by CACTI [163] 5.3 with 4 banks, dynamic read power optimization, and 45nm low-leakage process options selected. We assume that the L2 memory is an 8MB on-chip SRAM that is distributed to each core as 16 tiles. The off-chip DRAM access energy incorporates only the energy consumed in bit-lines, which dominates the total DRAM access energy [148]. We assume a 4GB DRAM, with two 2 GB ranks, each consisting of 256 MB, 4-bank devices as in [148]. The row-buffer hit rates of DRAM is assumed as 35% following the results for 16-core cases presented in [148]. As a rough estimation, DRAM write energy is assumed to be the same as the read energy. The estimated interconnection energy is the sum of energy consumed in channels and energy consumed in routers. The channel energy is estimated based on 1.1V Vdd, 50% activity factor, and 0.23fF/μm wire cap in a 45nm low-leakage process, and the router energy is estimated based on the RTL model presented by Becker and Dally [18]. We assume that the channel length is 1mm and width is one word (32-bit). We assume that each L2 memory access incurs a 4-hop traversal through the interconnection network (4-hop is the average hop count in a 4$\times$4 mesh network with a uniform-random traffic).

Team scheduling saves the energy consumed in memory and interconnection from 23% (64KB) to 33% (16KB), compared to the baseline [5]. The energy consumption difference between team scheduling and SGMS is particularly significant in `fft`, `gsm`,

[4] The physical unit of information that is transferred a channel in one cycle.

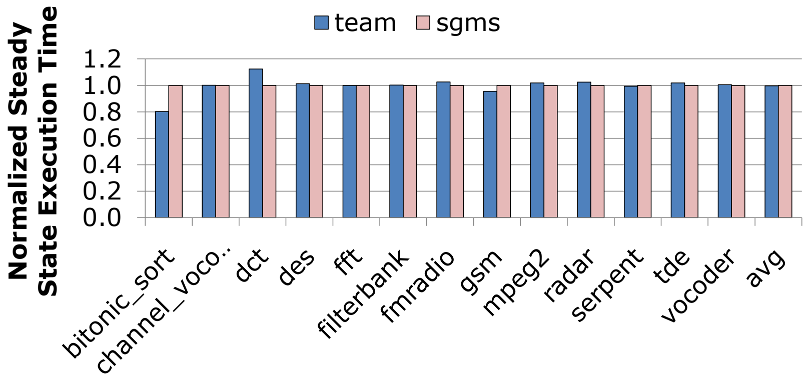
[5] To provide context, these memory and interconnection energy savings would result in from 13% (64KB) to 20% (16KB) reduction in the total dynamic energy consumed in Elm processors if we assume that the ratio of energy consumed in the instruction SPM to that in datapath, clock, and control is 32 to 49 as presented in Balfour et al. [13]. The average power consumption in memory and interconnection is from 53 mW to 74 mW depending on configuration when the clock frequency is 500 mHz.

`mpeg2`, `vocoder` and `wcdma_searcher` since SGMS incurs costly accesses to the L2 memory and its associated interconnection network traversals when it fails to allocate all buffers in local memories. Figure 4.10(a) shows that, in `des` and `serpent`, team scheduling achieves 48% and 52% reduction in energy consumption of the interconnection subsystem compared to SGMS. In addition, Figure 4.10(b) shows that team scheduling reduces the interconnection energy by 54% and 42% in `des` and `serpent` when the capacity of local memories increases from 16KB to 128KB, whereas SGMS achieves <10% interconnection energy reduction. This demonstrates that team scheduling more effectively uses limited local memory space for amortizing communication overhead (e.g., overhead from DMA packet headers and ACK messages).

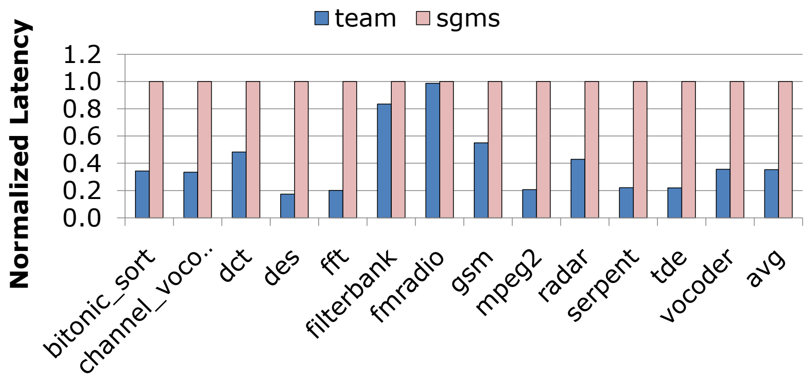
In this experiment, we show that team scheduling has better control over buffer space than SGMS has: given limited local memory space, team scheduling has a better chance of fitting buffers in local memories and achieves a higher throughput by efficiently utilizing the limited buffer space for amortization (i.e., team scheduling achieves balanced trade-off between synchronization overhead and buffer space requirement).

4.4.3 Amortization Factor Limited Experiment

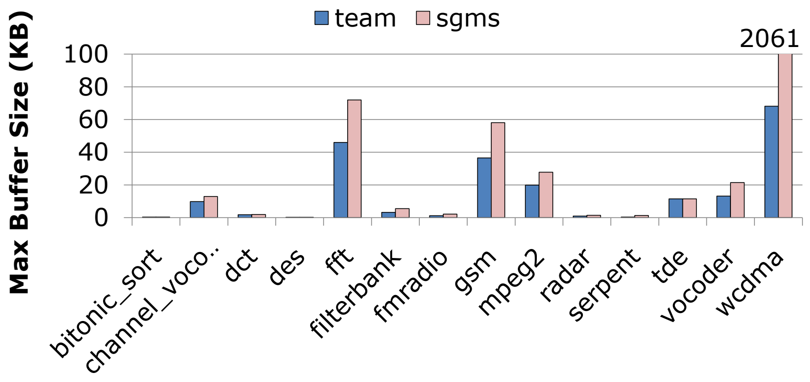
Figure 4.11 shows throughput, latency, and buffer requirement of both algorithms while we limit repetition factors to those in the minimum steady state of the entire stream graph. In this experiment, we set the buffer space limit per core to 64KB for team scheduling, which achieves a similar (0.4% higher) throughput to that of SGMS as shown in Figure 4.11(a) — a larger buffer space limit improves throughput at the expense of longer latency. In Figure 4.11(b), latency is measured as the time until the first output of the furthest downstream actor is generated. Team scheduling shows 65% lower latency and 46% smaller buffer requirement when its throughput is similar to that of SGMS. SGMS has high latency because of poor load balancing in its software pipeline prologue, resulting in idle cycles while the processor waits for barriers. Team scheduling does not suffer from this problem since actors are pair-wise synchronized and can be fired whenever input tokens are ready.



(a) Steady state execution time normalized to SGMS



(b) Latency normalized to SGMS



(c) Buffer requirement

Figure 4.11: Results of amortization factor limited experiment with team scheduling and SGMS for 16 cores.

Since we set the buffer space limit to 64KB for team scheduling, team scheduling uses less than 64KB for every application as shown in Figure 4.11(c). SGMS requires 2MB buffer space for `wcdma`, which is well over the local memory size of each core, 256KB. Hence, we omit `wcdma` in Figure 4.11(a) and (b). In `wcdma`, there is a series of reduction actors that produce fewer tokens than it consumes, thus actors upstream must be executed hundreds of times, consuming hundreds of KB of data in steady state. Team scheduling avoids excessive buffer requirement from the upstream actors by decoupling the scheduling of upstream and downstream actors.

4.4.4 Discussion: Sensitivity to Architectural Parameters

This section qualitatively discusses the sensitivity of the performance of team scheduling to architectural parameters.

The Number of Cores: It is expected that the energy-efficiency and performance gap between team scheduling and SGMS will grow as the number of cores increases. This is because team scheduling uses localized point-to-point synchronizations, while SGMS uses barriers that require global communication. Although the barrier synchronization primitive is typically optimized by techniques such as tree reduction, it is still less scalable than localized point-to-point communications generated by team scheduling.

The Capacity of Local Memories: Team scheduling shows the most clear advantage over SGMS when each core has local memories with limited capacity, the common case for embedded processors. When the capacity of each local memory is large enough to perform actor amortizations beyond the point of diminishing return even with SGMS, a benefit of team scheduling — more efficient minimization of synchronization and communication overhead given the same local memory space — will not play an important role any more.

The Support for DMA: As long as there is energy efficiency or performance benefits of fitting data in smaller memories, team scheduling should also be useful for architectures with no software-managed data transfers such as DMA. However, as will be discussed in Section 5.5.3, there is no benefit of fitting data with producer-consumer communication pattern in smaller memories (e.g., L1 caches instead of L3 caches) in the current x86 architectures. I expect that future processors will provide architecture supports for efficient producer-consumer communications, where team scheduling will show benefits similar to those shown in architectures with software-managed memories such as Elm.

4.5 Related Work

4.5.1 SDF Scheduling for Single-core Architectures

Lee and Messerschmitt [98] lay the foundation of synchronous data flow (SDF) including a necessary and sufficient condition to the existence of a valid static schedule which does not deadlock and requires bounded buffer space. They define the *topology matrix* associated with a stream graph, where the (i, j)th entry in the matrix is the amount of data produced by actor j on edge (i.e., stream) i each time the actor is fired. If actor j consumes data from stream i, the number of is negative. By observing the rank of the topology matrix, we can easily check whether the input program has a valid schedule. Let $|V|$ be the number of actors and Γ denote the topology matrix. Then, $\text{rank}(\Gamma) = |V| - 1$ is a necessary condition for the existence of a periodic single-core schedule that does not deadlock and requires bounded buffer space (Theorem 1 in [98]). Intuitively, the null space of the topology matrix must not be empty to admit periodicity (i.e., there must be a q such that $\Gamma \cdot q = 0$), and the connectivity of the stream graph imposes $\text{rank}(\Gamma) \geq |V| - 1$. Here, the periodicity is required to bound the schedule size finite. We say a stream graph that violates the condition above has a *sample rate inconsistency.* They also present a scheduling algorithm that finds a valid periodic single-core schedule whenever such a schedule exists (Theorem 3 in [98]).

While the scheduling algorithm described in Lee and Messerschmitt [98] guarantees to produce a valid schedule with a bounded buffer space for applications without sample rate inconsistencies, there can be many valid schedules with significantly different memory requirements. Bhattacharyya et al. [22, 23] develop a series of algorithms that improve two major contributers to the memory pressure: code size and data buffer size.

As a code size optimization, Bhattacharyya et al. [22] refine Lee and Messerschmitt's scheduling algorithm [98] so that it finds a single appearance schedule whenever such a schedule exists. As defined in Section 4.3.1, a single appearance schedule is a static schedule in which each actor lexically appears exactly once. For example, $2b\ 4c$ (fire b twice, then fire c four times) is a single appearance schedule. All single appearance schedules result in the same minimal code size if we ignore code size overhead from looping (this assumes that actor code is inlined in the schedule so the actor code dominates the total code size of the schedule). They define a topological property of a strongly connected subgraph of the input program called *loose independence* (Definition 2 in [22]). They prove that an SDF application has a single appearance schedule if and only if every nontrivial connected subgraph is loosely independent (Theorem 1 in [22]). They develop a family of scheduling algorithms called *loose interdependence algorithms* that yields a single appearance graph whenever every nontrivial connected subgraph of the input program is loosely independent. They also show that every nontrivial connected subgraph is loosely independent in the majority of practical applications (Section IV in [22]).

As a data buffer size optimization, Bhattacharyya et al. [23] present an algorithm that improves an existing single appearance schedules so that the buffer requirements can be reduced. There can be many possible single appearance schedules for a given input program, and, depending on how actors are grouped, the buffer requirement can significantly vary. For example, $2(b\ 2c)$ will require smaller buffers than $(2b\ 4c)$ although both are single appearance schedules. They use *pairwise grouping of adjacent nodes* (PGAN) heuristic to find a grouping that yields a small buffer requirements, in which some aspects are similar to our team formation procedure (grouping can be viewed as actor aggregation), but their algorithm assumes an acyclic stream graph and works in the context of single-core scheduling.

[87, 92, 134] present SDF *vectorization* that amortizes the cost associated with actor interactions. For example, Ko et al. [87] develop a vectorization algorithm that reduces context switch overhead and increases memory locality within memory constraints, which is similar to amortization described in this chapter in some aspects but is done in the context of single-core scheduling. Amortization, or vectorization, plays a more important role in multi-core scheduling because it not only improves the locality of memory access but also amortizes fixed costs associated with each DMA initiation.

4.5.2 SDF Scheduling for Multi-core Architectures

Ha and Lee [57] propose a taxonomy of data-flow scheduling for multi-core architectures, which consists of four types of scheduling methods. In *fully dynamic* scheduling, all the scheduling decisions are made at run-time. Any actor whose input tokens are available can be assigned to any idle processor. The load balancing work stealing scheduling [29] is an example of fully dynamic scheduling. In *static allocation* scheduling, actors are assigned to a processor at compile time, and a local run-time scheduler determines the order of actor invocations. In *self-timed* scheduling, the order of actor invocation inside each processor is also determined by the compiler, but the specific timing when an actor is fired is determined at run-time (e.g., the processor waits for data to be available for the next actor, and, then, fires the actor). In *fully static* scheduling, the compiler determines the exact firing time of actors as well. According to this taxonomy, the initial schedule of our team scheduling algorithm falls into the static allocation category, while aggregated parts (i.e., teams) follow the self-timed scheduling.

In [24], Bhattacharyya et al. present a post-pass optimization scheme that eliminates redundant synchronization in an existing multi-core self-timed schedule for SDF applications. Their method eliminates the same set of redundant inter-team synchronizations (e.g., synchronization at (a, b) in Figure 4.3(b)) as team scheduling, but team scheduling does so without running a sophisticated analysis by exploiting a property of team scheduling, namely constant production-to-consumption ratios of streams between a team pair. They also present a method of reducing synchronization overhead by introducing a few feedback paths that make full checks of other queues unnecessary. However, this method requires additional buffer space, and it will be interesting to evaluate whether using the buffer space for eliminating queue full checks as in [24] is more beneficial than using the same buffer space for amortization as in team scheduling.

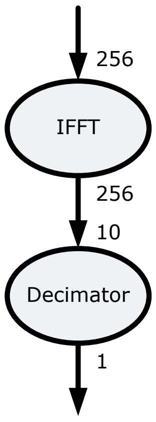
The body of SDF work [23, 24, 97, 98, 128, 138] provides a solid theoretical background for optimizing static parts of stream programs. However, their multi-core scheduling algorithms [24, 97, 98, 128] are based on *homogeneous* SDF *graph* (HSDFG), whose construction from an equivalent SDF graph can take an exponential amount of time [128]. In addition, their algorithms do not overlap the execution of different HSDFG iterations, resulting in a smaller degree of parallelism.

Lin et al. [107] point out exponential buffer space growth in their W-CDMA evaluation and present an algorithm that applies software pipelining in a hierarchical manner. However, in their algorithm, the programmer must define the hierarchy, and the authors failed to keep the scheduling algorithm free from exponential buffer space growth when they designed their later work, SGMS [90]. They formulate the partitioning problem as integer linear programming (ILP) to find an optimal combination of data, task, and pipeline parallelism. However, as Gordon [50] points out in his thesis, it is unclear whether the cost of using an ILP solver can be justified for finding an "optimal" partitioning, considering that computation requirement of each actor is not completely accurate, which renders the ILP formulation an approximation anyway. In addition, their ILP formulation does not attempt to minimize the communication, which contributes a large fraction of energy consumption. Choi et al. [35] presents an extension to SGMS that considers memory constraints during their partitioning phase formulated as ILP. Essentially, the only optimization opportunity in their approach is passing buffers between producers and consumers, while the total buffer space is preserved. Consequently, their approach still does not avoid exponential buffer space growth.

Gordon *et al.* [50, 52] point out deadlocks in split-join patterns. However, they just mention that "Each architecture requires a different deadlock avoidance mechanism and we will not go into a detailed explanation of deadlock here. In general, deadlock occurs when there is a circular dependence on resources. ... If the architecture does not provide sufficient buffering, the scheduler must serialize all potentially deadlocking dependencies". We failed to find any other documentation that describes their detailed deadlock avoidance scheme, but we can infer from [50, 52] that their scheme is confined to a specific case where the packet size of a splitter and that of its corresponding joiner are unmatched (Figure 13 in [52]). Consequently, their deadlock avoidance scheme cannot handle the case shown in Figure 4.4(a). In addition, as opposed to the quoted rather qualitative statement, our queue capacity computation algorithm tells us that precisely how large buffer is needed at each stream to avoid deadlock. Although not described in this chapter to focus on scheduling problems, our pre-processing step to partitioning roughly follows the one described in Gordon et al. [51]: we first selectively fuse adjacent stateless actors to coarsen the granularity, and then we fission stateless actors just enough to fill the cores. After the pre-processing step, the main partitioning step described in [51] uses a greedy packing algorithm that minimizes load imbalance and communication, but we did not find a compelling reason for using their algorithm since a popular graph partitioning package, METIS [82], also minimizes load imbalance and communication. It would be interesting to see how their algorithm performs compared to METIS, but we did not perform the comparison since our main interest was memory-space efficient scheduling algorithms, not partitioning algorithms.

4.5.3 Loop Transformations

Since actors in stream applications work on a sequence of input tokens, we can conceive of implicit loops that enclose the actors. For example, when the stream graph shown in Figure 4.12(a) is written in C, each actor code will be enclosed by loops as shown in Figure 4.12(b). The first `for` loop corresponds to the inverse fast Fourier transform actor and writes its output to the array `B` with size `N`, where `N` is the number of tokens transferred between two actors. The next `for` loop reads the array `B` and writes the array `C` with size `N/10`. Once stream graphs are written in a



(a)

```
for (int i = 0; i < N/256; i++) {
  inverse256PointFFT(&A[256*i], &B[256*i]);
}

for (int i = 0; i < N/10; i++) {
  // 10-to-1 decimation
  C[i] = B[10*i];
}
```

(b)

```
for (int i = 0; i < N/1280; i++) {
  for (int j = 0; j < 5; j++) {
    inverse256PointFFT(&A[256*(5*i + j)], &B[256*(5*i + j)]);
  }

  for (int j = 0; j < 128; j++) {
    // 10-to-1 decimation
    C[128*i + j] = B[10*(128*i + j)];
  }
}
```

(c)

Figure 4.12: (a) A stream graph, (b) its equivalent implementation in C, and (c) after loop fusion.

form with loops and arrays as shown in Figure 4.12(b), numerous loop transformations [44, 45, 94, 104, 105, 135, 164] can be applied. In fact, some of them subsume stream graph transformations described in Gordon et al. [51, 52]. For example, actor fusion can be viewed as loop fusion, and actor fission can be viewed as loop fission. Compared to generic loop transformations, stream programming provides more information explicitly at the expense of targeting a restricted class of applications. This can facilitate more optimization opportunities since the compiler may not be able to infer the explicitly provided information with a reasonable amount of effort.

Figure 4.12(c) shows a C code equivalent to Figure 4.12(b) but after a loop fusion where each outermost loop iteration corresponds to the minimum steady state of the stream graph [90]. To balance the number of tokens transferred between two loops, `IFFT` is repeated 5 ($= \frac{lcd(256, 10)}{256}$) times, and `Decimator` is repeated 128 ($= \frac{lcd(256, 10)}{10}$) times per the outermost loop iteration. These repetition counts correspond to *minimum repetition* [98] of each actor described in Section 4.2. SGMS can be viewed as an application of software pipelining to the loop such as the outermost loop in Figure 4.12(c).

Sarkar and Gao [135] present array contraction transformation whose purpose is maximizing the amount of producer-consumer pipeline parallelism with minimum buffer storage. This is the same objective as the queue capacity computation algorithm described in Section 4.2; a difference is that our algorithm applies to stream graphs whereas array contraction applies to generic imperative code with loops as the one shown in Figure 4.12(b). They introduce a notion of *loop communication graph*, which captures the communication pattern of a series of loops with producer-consumer relations. If we construct loop communication graph from Figure 4.12(b), the resulting graph is effectively same as the stream graph shown in Figure 4.12(a). After publishing a shorter version of this chapter [124], it was realized that the idea of balancing latencies of paths between actors used in our queue capacity algorithm is a rediscovery of Sarkar and Gao's algorithm on *extra buffer* allocation. However, they only consider producer-consumer pairs with *consistent* communication, where corresponding array writes and reads have the same affine expression except for constant terms. Consequently, their algorithm does not contract the array `B` in Figure 4.12(b).

4.6 Chapter Summary

A compiler algorithm that computes the minimum queue capacities of static stream applications that avoid deadlocks and serialization is presented. Since the queue capacity computation algorithm works for arbitrary configuration of static stream applications, not only does it minimizes the queue capacity requirement for a given configuration, but also it allows to flexibly transform the stream application to trade-off synchronization/communication overhead with the queue capacity requirement. Team scheduling is presented as an example of applying our queue capacity computation algorithm. In team scheduling, we apply actor aggregation or amortization in a flexible order to find a sweet spot of the trade-offs between synchronization/communication overhead and the queue capacity requirement. Applying actor aggregation or amortization in an arbitrary order is prone to deadlocks or serialization without our generic queue capacity computation algorithm.

Team scheduling realizes key performance features such as deadlock-free feedback loops, low latency, and flexible queue capacity control since it is not constrained by the minimum steady state of the entire application. Due to its flexibility in queue capacity control, team scheduling efficiently utilizes limited local memory space of each core. This is clearly shown by the fact that team scheduling consistently satisfies the queue capacity constraint whereas SGMS fails to do so when the space limit per core is small (no larger than 16KB). In the case where the space limit is as large as 128KB, team scheduling achieves on average 37% higher throughput than SGMS. It is also estimated that team scheduling saves the energy consumed in memory and interconnection up to 46% (when 16KB local memories are used) compared to SGMS by avoiding costly non-local memory accesses. These results demonstrate team scheduling as a critical optimization scheme in stream compilers for a large class of applications targeting embedded multi-core processors, which commonly have limited local memory space.

Although many applications such as digital signal processing algorithms follow the synchronous data flow (SDF) model, it is true that there are many other stream applications with dynamic behavior. The methods described in this chapter are however still useful for the dynamic applications as tools for optimizing their static subparts. For example, in an MP3 decoder, the execution time and output rate of the Huffman stage vary over time, but the other parts of the decoder follow the SDF model. For this type of applications, we can find stream subgraphs that follow the SDF model and apply team scheduling to each of them to minimize the memory footprint while maintaining the throughput. The next chapter presents a queue design called QED (Queue Enhanced with Dynamic Sizing) that can be used for the rest part of applications and variable-rate streams that connect static stream subgraphs.

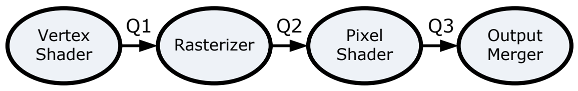
Chapter 5

Buffers in Dynamic Stream Applications

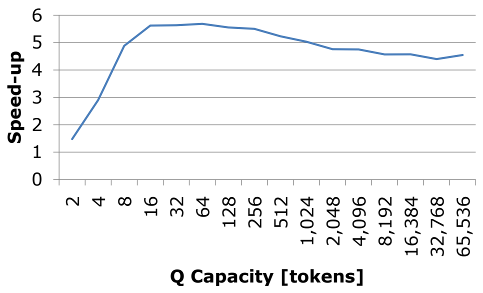
This chapter presents a non-blocking queue design called QED (Queue Enhanced with Dynamic sizing) that minimizes memory footprint of inter-actor queues in *dynamic* stream applications (or dynamic pipeline parallel applications in general). This is complementary to the methods described in the previous chapter that compute the minimum queue capacities that maximize the throughput in *static* stream applications at *compile time*. Many applications have dynamic behavior, and hence require a *run-time* mechanism that adapts the queue capacities for the run-time variation.

In QED, capacities are dynamically adjusted to maximize the throughput while minimizing memory footprint. Despite its dynamic nature, queues are based on circular arrays avoiding the overhead associated with dynamic memory allocation of linked-list based queues. The capacity adjustment is based on the approximated time variance of the number of tokens present in the queue. A large array is allocated initially, but only a small portion of it is actively used at any given time.

We compare the performance of QED with four alternative methods, three of which use statically-sized, array-based queues and the other which uses the Michael and Scott's queue. The first alternative method, which approximates optimal statically-sized queues, achieves the highest throughput among them, and we analyze the performance degradation in other methods. We show that QED achieves a throughput



(a)



(b)

Figure 5.1: (a) A Direct3D pipeline and (b) its parallelization speed-up as we change the inter-actor queue capacities while keeping all three queue capacities the same. The experimental setup is detailed in Section 5.4.

similar to that of the first alternative method, while eliminating long search times required for the alternative method.

5.1 Overview

When production and consumption rates of actors are static, the compiler can determine minimum queue capacities that avoid serialization using the queue capacity computation algorithm presented in Chapter 4. However, many applications have parts that are not static, and hence requires larger queue capacities to accommodate the run-time variation. For example, Figure 5.1(b) shows parallelization speed-ups of a Direct3D pipeline as we change the capacities of inter-actor queues. When the capacity is too small (e.g., less than 32 tokens), the run-time variation of actors cannot be hidden. Conversely, when the capacity is too large, the memory system stalls intermittently due to events such as TLB misses, resulting in up to a 23% throughput decrease from the maximum at capacity 64. In addition to improving the throughput, using appropriate capacities instead of “large enough” ones yields better locality, thereby saving energy by reducing costly access to non-local memories.

To the best of our knowledge, computing appropriate inter-actor queue capacities in dynamic stream applications however has been understudied. Arbitrary “large enough” numbers are often chosen for inter-actor queue capacities. To address this, we present a non-blocking[1] queue design called *QED (Queue Enhanced with Dynamic sizing)* that adjusts its capacity at run-time to maximize the throughput with minimal memory footprint. Even though the capacity is dynamically adjusted in QED, queues are based on circular arrays avoiding linked-list based queues' overhead associated with dynamic memory allocation.

We compare the performance of QED with four alternative methods for finding inter-actor queue capacities that maximize the throughput of dynamic stream applications. In the first method, we use statically-sized circular-array based queues (in short, static queues) and search for their optimal capacities by measuring the throughput as we change the capacities. In order to avoid exponential search steps of exhaustive search, a heuristic search is performed. In short, the first method approximates an optimum of statically-sized queues. In the second method, we construct an optimal dynamic queue capacity adjustment using traces collected from executions with unbounded queues. Then, we set capacities of statically-sized queues to 90% percentile of capacities used during the constructed optimal capacity adjustment. In the third method, we estimate optimal capacities of static queues by applying an analytical model based on queueing theory that was developed by Navarro et al. [117]. In the fourth method, we use the Michael and Scott's queue (MS queue) [112], considered to be one of the most efficient and scalable linked-list based non-blocking queue designs in the literature [63, 66, 88, 91, 146].

We show that QED achieves a throughput similar ($< 1\%$ difference) to that of static queues whose capacities are determined by the first alternative method (i.e., heuristic search). However, the search time of the first alternative method can be hours for executing applications multiple times while changing queue capacities, and we need to redo the search whenever we modify the application or encounter a new architecture. In addition, it can be hard to choose a representative input data set for the search so that we can find capacities optimal across a wide range of input data. The second alternative method (i.e., using 90% percentile of capacities used for optimal dynamic adjustments) requires considerably fewer application runs, but shares a problem with the heuristic search — choosing a representative input data

[1] Refer to Section 5.6 for a definition and benefits of non-blocking algorithms.

set for profiling. The third method (i.e., analytical estimation method) requires no search time, but it does not accurately model the application behavior, resulting in an average of 16% slowdown. In the last method, the MS queue is 18 % slower and incurs 3.9× TLB misses than optimally-sized static queues due to its un-throttled execution of upstream actors and its overhead associated with dynamic memory allocations.

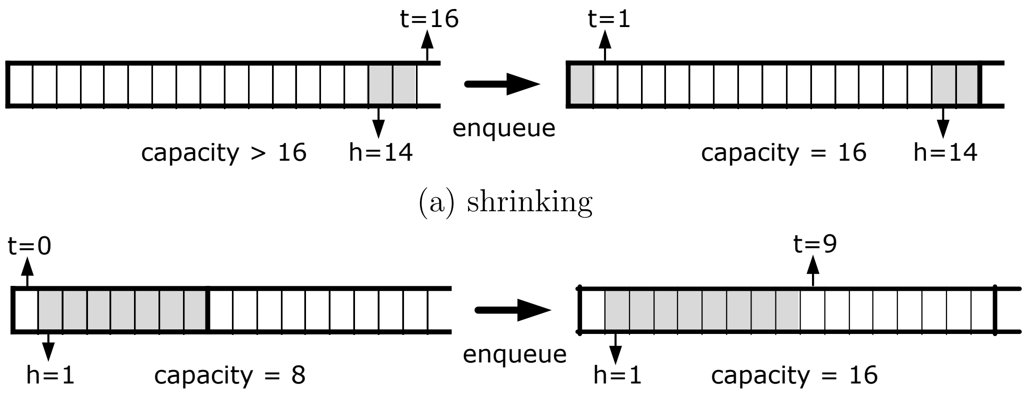
The remainder of this chapter is organized as follows: Section 5.2 presents QED focusing on its capacity adjustment procedure and non-blocking property. Section 5.3 describes four alternative methods for finding inter-actor queue capacities, namely search, dynamic optima profiling, analytical estimation, and MS queue. Section 5.4 describes the applications and evaluation setup we use throughout this chapter. Section 5.5 compares the performance of QED with the alternative methods. Section 5.6 reviews related work, and Section 5.7 summarizes this chapter.

5.2 QED (Queue Enhanced with Dynamic sizing)

This section introduces a non-blocking queue design that is array-based yet dynamically sizable, called QED. Section 5.2.1 presents the main idea behind QED and how the capacity is adjusted so that the throughput can be maximized with minimal memory footprint. Section 5.2.2 describes QED's reserve-commit interface for in-place computation, and Section 5.2.3 details its implementation focusing on its non-blocking property.

5.2.1 Capacity Adjustment

Initially, a queue allocates a large array (the current implementation uses an array with 2^{16} elements by default). We however actively use only a part of the array and define the size of the actively used part the *current capacity* of the queue. We define *occupancy* as the number of tokens that have been enqueued but yet to be dequeued, which can be measured by the difference between the tail and head index. The capacity opportunistically shrinks when the time variance of the occupancy is small, and expands later only if the variance increases up to a certain level.



(a) shrinking

(b) expanding

Figure 5.2: Examples of capacity adjustment.

```
// Adjust capacity. t: tail index, c: capacity
int adjustC(int t, int c, int occ) {
  if (t == c && maxOcc - minOcc > c/2 && occ > 1)
    return 2c; // expand
  if (isPowerOf2(t) && maxOcc - minOcc < t/2 && occ < t)
    return t; // shrink
  return c;
}
```

Figure 5.3: Capacity adjustment procedure.

Figure 5.2 illustrates examples of shrinking and expanding the capacity. In the left side of Figure 5.2(a), we are about to enqueue a token to the tail index, 16. Suppose that the time variance of the occupancy has been small relative to the current capacity. Then, instead of writing the token to index 16, we shrink the capacity to the tail index value, wrap-around the tail index to 0, and write the token to index 0, as shown in the right side of Figure 5.2(b). In the left side of Figure 5.2(b), we are about to enqueue a token to index 0 after wrapping around the tail index. Suppose that the time variance has been large relative to the current capacity, 8. Then, we double the capacity and write the token to index 8.

The capacity is adjusted based on the variance of occupancy, which reflects the

```
template<typename T> class queue {
  bool reserveEnqueue(int *index);
  void commitEnqueue(int index);

  bool reserveDequeue(int *index);
  void commitDequeue(int index);

  T array[MAX_C]; // circular array
};
...
int dequeueIndex, enqueueIndex;
while (!inQ.reserveDequeue(&dequeueIndex));
InputToken *in = &inQ.array[dequeueIndex];
foo(in->field1) // operations on in
...
while (!outQ.reserveEnqueue(&enqueueIndex));
OutputToken *out = &outQ.array[enqueueIndex];
out->field2 = boo(in->field3); // access on in and out
...
inQ.commitDequeue(dequeueIndex);
...
out->field4 = bar(); // access on out
outQ.commitEnqueue(enqueueIndex);
```

Figure 5.4: Reserve-commit interface and its usage example.

run-time variation of application behavior. The variance of occupancy can lead to stalls from *transient* empty or full states, which can be controlled by adjusting the capacity. When the variance of occupancy is large relative to the current capacity, we are likely to have many transient stalls; thus the capacity should expand. Conversely, when the variance of occupancy is relatively small, we are unlikely to have transient stalls; thus the capacity can shrink to reduce memory footprint. Note that QED does not attempt to avoid *recurring* empty or full states resulting from the difference between the long term time averages of enqueue and dequeue rate. The load imbalance resulting from the difference in long term averages should be addressed by load balancing scheduling instead of the capacity adjustment.

We need to answer the following two design questions with respect to the capacity adjustment: 1) How often do we check whether we need to shrink or expand the capacity? 2) How can we measure the variance of the occupancy with minimal overhead?

Regarding the first question, we check the condition for shrinkage when the tail index is a power of two and check for expansion when the tail index wraps around to 0. This design choice has the following advantages: First, the capacity is always a power of two, which allows us to efficiently wrap around indices using bit-wise `and` operations and compress the capacity value by storing its binary logarithm (detailed in Section 5.2.3). Second, the overhead of checking the conditions is minimized by reducing its frequency (log_2C instead of C per tail index wrap-arounds, where C is the capacity).

For the second question, we measure the difference between the minimum and maximum occupancy of the queue during the last *epoch*, where epochs are defined as the times between consecutive wrap-arounds of the tail index. The difference between the minimum and maximum occupancy during the last epoch approximates the variance of the occupancy, and we adjust the capacity based on this approximated variance as shown in Figure 5.3. In the `adjustC` method shown in Figure 5.3, variables `occ`, `minOcc`, and `maxOcc` denote the current occupancy, the minimum and maximum occupancy during the last epoch, respectively. When (`maxOcc - minOcc`) is large (more than half of the current capacity in our implementation), we double the capacity (the second `if` condition of line 3 in Figure 5.3). When (`maxOcc - minOcc`) is small (less than half of the current tail index in our implementation), we shrink the capacity to the tail index value (the second `if` condition of line 5 in Figure 5.3). The reasons for the last `if` conditions of lines 3 and 5 are described in Section 5.2.3.

5.2.2 Reserve-commit Interface

QED provides in-place access and avoids dynamic allocations by supporting the two phase *reserve-commit* interface shown in lines 1-9 of Figure 5.4, which is adopted from the GRAMPS graphics pipeline programming model [141]. Through the pointer argument `index`, `reserveEnqueue/Dequeue` returns an index that points to a part of the circular array associated with the queue. The return value of `reserveEnqueue/Dequeue` is `false` when the queue is full/empty. The reserve-commit interface guarantees the caller exclusive access to the part of the array pointed by the `index` until it receives a corresponding commit call. Lines 11-23 in Figure 5.4 show an example of using this reserve-commit interface. Note that operations on input and output tokens are directly applied to parts of the circular arrays (lines 13, 14, 17, 18, and 22).

On the other hand, the conventional one-phase interface typically incurs overhead from either copies or dynamic memory allocations of tokens. In the applications we evaluate, the size of each token transferred between actors can be quite large (e.g., the token size of the most upstream queue in the MP3 decoder shown in Figure 5.9(f) is 3KB). Copying the value of each large token to and from queues incurs significant overhead. To avoid this overhead, pointers to the tokens, instead of values, are often enqueued and dequeued, but this typically requires dynamic allocation of tokens. Theoretically, one can manually implement a memory pool to reduce the overhead from dynamic memory allocations. However, the circular array in QED already lends itself to a memory pool specialized for the producer-consumer access pattern: memory chunks released by dequeued tokens are reused for tokens enqueued later in a FIFO manner. The reserve-commit interface exposes this memory pool inherent in circular-array based queues to the caller and saves efforts to manually implement a memory pool. Wrapping the reserve-commit interface as the conventional one-phase interface is straightforward and desirable when tokens have primitive types with small copy cost.

5.2.3 Non-blocking Implementation

Figure 5.4 shows pseudo C++ code of our QED implementation. In lines 1-4, two `structs`, `PackedIndex` and `PackedIndexAndC`, pack their fields into a single 64-bit word (they are actually `unions` in order to be atomically accessed through their 64-bit

```
struct PackedIndex { int logical, physical; };
struct PackedIndexAndC {
 int logical, physical:24, log2C:8;
}; // log2C: log2(capacity)

template<typename T> class Qed {
 int presence[MAX_C]; // initialized as zeros
 PackedIndex head; PackedIndexAndC tail;
 int minOcc = INT_MAX, maxOcc = 0;
   // last epoch's min/max occupancy

 bool reserveEnqueue(PackedIndex *ret) {
   PackedIndexAndC old, new; int occ;
   do {
     old = tail; // atomic copy of tail to a local variable
     int c = $2^{old.log2C}$;
     occ = old.logical - head.logical;
     if (occ $\geq$ c || presence[old.physical%c])
       return false; // queue is full
     new.log2C = $\log_2$(adjustC(old.physical, c, occ));
     ret->logical = old.logical;
     new.logical = old.logical + 1;
     ret->physical = old.physical%$2^{new.log2C}$;
     new.physical = ret->physical + 1;
   } while (!cas(&tail, old, new));
   if (ret->physical == 0) minOcc = maxOcc = occ;
   minOcc = min(minOcc, occ);
   maxOcc = max(maxOcc, occ);
   return true;
 }

 void commitEnqueue(PackedIndex reservedIndex) {
   presence[reservedIndex.physical] = 1;
 }
35  bool reserveDequeue(PackedIndex *ret) {
36    PackedIndex old, new;
37    do {
38      PackedIndexAndC t = tail; // atomic copy of tail
39      *ret = old = head; // atomic copy of head
40      ret->physical %= $2^{t.log2C}$;
41      if (ret->logical $\geq$ t.logical || !presence[ret->physical])
42        return false; queue is empty
43      new.logical = ret->logical + 1;
44      new.physical = (ret->physical + 1)%$2^{t.log2C}$;
45    } while (!cas(&head, old, new));
46    return true;
47  }
48
49  void commitDequeue(PackedIndex reservedIndex) {
50    presence[reservedIndex.physical] = 0;
51  }
52 } // end of class Qed
```

Figure 5.4: QED implementation.

fields, but this is omitted to avoid clutter). In these `struct`s, the logical index is a 32-bit integer that points to a location in a logical unbounded array, whereas the physical index is used to index the physical circular array. As opposed to statically-sized circular-array queues, one cannot be inferred from the other since the capacity can continuously change. Note that the 2's complement arithmetic maintains the correct semantic of logical indices in the presence of integer overflows: e.g., even after the tail index wraps around to a negative value, (`tailIndex.logical` - `headIndex.logical`) still results in a positive number. In addition to the logical and physical indices, `PackedIndexAndC` uses its 8 bits to store log_2 of the capacity. The physical index of `PackedIndexAndC` is represented by 24 bits, which limits the maximum capacity to 2^{24} = 16M, which we believe to be a practically sufficient number.

We have a presence flag associated with each array element (line 7) to check the queue's full or empty state. Simply inspecting the difference between the logical tail and head index is not sufficient for full or empty checks since `commitEnqueue/Dequeues` can be performed out of order. However, inspecting presence flags alone is also insufficient when there is a lagging producer/consumer that takes a long time to `commitEnqueue/Dequeue` after a `reserveEnqueue/Dequeue`. Suppose that thread t_i enters `reserveEnqueue` when the physical tail index is 1. When there is a lagging thread t_j that has reserved physical index 1 but has not committed, without the first `if` condition in line 18, the physical index 1 will be double-reserved since `presence[1]` is still 0. Therefore, we need to inspect both the presence flag and the difference between the logical tail and head index as shown in lines 18 and 41.

The states atomically updated in `reserveEnqueue` are the logical/physical tail indices and the capacity. We atomically copy these states to local variables (line 15), check if the queue is not full (line 18), modify the local variables (lines 20-24), and attempt to atomically update the states using a `cas` (compare-and-swap) operation (line 25). The ABA problem [66] is unlikely to happen since a part of the value we compare-and-swap, the logical tail index, is a 32-bit sequence number. We do not atomically update the minimum and maximum occupancy since race conditions resulting from the non-atomic updates are benign ones that marginally affect only the accuracy of the occupancy variation. The `reserveDequeue` method operates similarly except that it updates only the logical and physical head indices.

In `reserveDequeue`, the tail indices and capacity can refer to out-dated values when there is an enqueuer that concurrently updates their values. This however does not lead to race conditions as long as the tail indices and the capacity are atomically stored from enqueuers and atomically loaded from dequeuers. Suppose that an enqueuer thread t_i shrinks the capacity to 16, while a dequeuer thread executes `reserveDequeue` with the physical head index 16. In this case, a race condition of setting `ret->physical` to 16 using the old capacity in line 40 (the correct value is 0) and returning `true` from `reserveDequeue` cannot happen because of the following reason: Due to line 6 of `adjustC` in Figure 5.3, right before t_i has shrunk the capacity, the physical tail index must have been 16, and, therefore, the logical tail and head indices must have been equal ("`tail.logical - head.logical` == capacity" cannot hold due to the third `if` condition in line 5 of `adjustC`). Since `PackedIndexAndC` is atomically stored and loaded, an old capacity implies an old tail index, rendering the first condition of line 41 true.

Suppose that an enqueuer thread t_i expands the capacity from 16 to 32, while a dequeuer thread executes `reserveDequeue` with the physical head index 15. In this case, a race condition of setting `new.physical` to 0 using the old capacity in line 44 (the correct value is 16) and returning `true` cannot happen because of the following reason: Due to the first `if` condition in line 3 of `adjustC`, the physical tail index must have been 16 right before t_i has expanded the capacity, rendering the third `if` condition in line 3 of `adjustC` false.

Notice that we need modular operations in both lines 40 and 44 to avoid race conditions that come from capacity adjustments between consecutive `reserveDequeue`s. Suppose that we are about to execute line 40 with the physical head index 16, and the capacity has shrunk to 16 after the last successful `reserveDequeue`. In order to set `ret->physical` to the correct value 0 instead of 16, we need the modular operation in line 40. Suppose that we are about to execute line 44 with the physical head index 15, and the capacity has expanded from 16 to 32 thereafter. In order to set `new.physical` to the correct value 0 instead of 16, we need the modular operation in line 44.

We have implemented a few performance optimizations, which are omitted in Figure 5.4 to focus on presenting the principal ideas of QED. First, measuring the occupancy as the difference between the logical tail and head index can be inaccurate since there can be tokens that have been reserved but yet to be committed. To consider this, we maintain a counter called `reservedEnqueueCounter` that is atomically incremented per `reserveEnqueue` and decremented per `commitEnqueue`, and another similar counter called `reservedEnqueueCounter` for dequeues. The value of `minOcc` in line 9 of Figure 5.4 is computed as the minimum value of (`tail.logical` - `head.logical` - `reservedEnqueueCounter`) during the last epoch,. Similarly, `maxOcc` is computed as the maximum value of (`tail.logical` - `head.logical` + `reservedDequeueCounter`) during the last epoch. Note that the computation of both variables are biased toward easier expansions since we prioritize avoiding stalls due to buffers too small over avoiding TLB or cache misses from larger capacities. Second, we optimize QED for the case of single producer and/or single consumer. For example, `cas` is not necessary

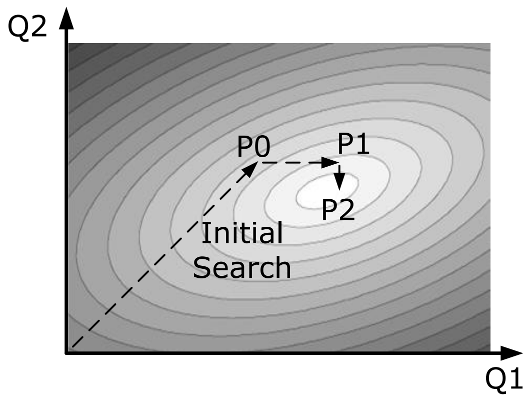


Figure 5.5: Illustration of heuristic search procedure.

in `reserveDequeue` when only a single dequeuer exists. Refer to `SpQed`, `ScQed`, and `SpscQed` in our Google code website [5] for the details of optimizations for the single producer and/or single consumer cases.

5.3 Alternative Methods

This section describes methods for finding inter-actor queue capacities that are alternative to QED. The methods described in this section use a statically-sized queue implementation (in short, static queue). The static queue implementation is similar to that of Tsigas and Zhang [146] with respect to its non-blocking and circular-array based property. However, similar to QED, our static queue implementation provides the *reserve-commit* interface described in Section 5.2.2. For the further details, refer to `qed_static.h` in our Google code website [5].

5.3.1 Static Optimum Approximation

Figure 5.5 illustrates our heuristic search procedure when there are two inter-actor queues. First, we start an initial search that finds the capacity that maximizes the profit (defined below), while keeping the capacities of each queue the same. We search along the line from the origin to P0 while keeping Q1 and Q2 equal, and find that P0 maximizes profit. If we confine our definition of profit to speed-up only, capacities

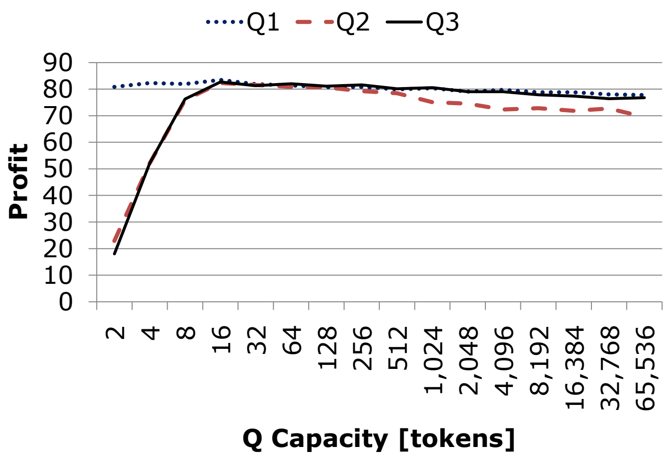


Figure 5.6: The profit of `d3d10` as we change the capacity of Q1, Q2, and Q3.

may unnecessarily be increased even after near-maximum speed-up has been achieved (e.g., in Figure 5.1(b), when the capacity is greater than or equal to 16).

Therefore, we define profit as $log_k(speedup) - log_2(\bar{q})$, where the speed-up is *regularized* [31] by the queue capacity. Here, $\bar{q}$ denotes the average queue capacity, and k is the regularization factor (i.e., how large a speed-up is considered to be beneficial by increasing the overall queue capacities by a factor of 2). For example, $k = 1.02$ means that we are willing to increase queue capacities by a factor of two if the speed-up from doing so exceeds 2%. Figure 5.1(b) shows speed-up during an initial search. Although the speed-up is maximized when the capacity is 64, our profit function is maximized at 16 when $k = 1.02$. Although we find that profit varies with approximately convex forms in the applications we evaluate, there can be local minima resulting from measurement noise (e.g., when capacity is 32K in Figure 5.1(b)). To avoid local minima during the search from the origin to P0 in Figure 5.5, we continue the search until the profit drops by more than 5% from the maximum value seen so far. This initial search quickly finds a configuration where no queue is a particular bottleneck, which is used as the starting point of the main search described in the following:

Starting from the point found in the initial search (e.g., P0 in Figure 5.5), the main search iterates over each inter-actor queue. For each iteration, we find the queue capacity that maximizes the profit by varying only the capacity of the current queue.

In both directions (smaller and larger capacity), we continue the search until the profit drops by more than 5%. For example, in Figure 5.5, the first iteration changes Q1 from P0 and finds that P1 maximizes the profit. Then, the second iteration changes Q2 from P1 and finds that P2 maximizes the profit. Figure 5.6 shows the profit of `d3d10` as we change the capacity of Q1, Q2, and Q3. Since the profit is maximized when the capacities are 16 in an initial search for `d3d10`, we start the main search with Q1 = 16, Q2 = 16, and Q3 = 16. For Q1, we measure the profit when its capacity is 8, 4, and 2 along the negative Q1 direction and find that 16 still achieves the maximum profit. Then, we measure the profit as we increase the capacity from 32 to 8K, where the profit drops by more than 5%. Similarly, we select the capacity of Q2 and Q3 to be 16.

One can view this procedure as a search through n dimensional space where each dimension corresponds to an inter-actor queue capacity. Seen in this light, our heuristic search resembles a coordinate descent search [7], which finds an optimum when the target function is convex and differentiable [108]. In the coordinate descent method, iterating over each dimension is repeated multiple times. For example, after searching along lines P0-P1 and P1-P2, the coordinate descent method repeats more iterations, checking whether the profit further improves by varying Q1 and Q2 again. However, since our main objective is minimizing the number of search steps, we stop our search after iterating over each dimension once. Note that each search step involves multiple application runs to reduce measurement noise. Gradient-descent is a more popular method to optimize convex functions and has better convergence properties, but it requires measuring application throughputs multiple times to accurately compute a gradient in our discrete search space.

Table 5.1 shows the results of our heuristic search procedure. We use $k = 1.02$ (i.e., we are willing to tolerate twice the queue capacities if it leads to more than 2% of parallelization speed-up increase), and the search range for the queue capacities is powers of two between 2 and 1M in the number of tokens. To be more conservative on increasing queue capacities, one can increase the value of k. We can see that our heuristic search achieves parallelization speed-up that is close to the maximum found by an exhaustive search in `d3d10`, `djpeg`, `hmmcalibrate`, and `mad`. For `dedup`,

Table 5.1: Results of exhaustive search (ES), heuristic search (HS), dynamic optimal approximation (DO), and analytical estimation (AE).The search range is powers of two between 2 and 1M (in number of tokens). The results of exhaustive search for `dedup`, `ferret`, and `packet_tracer` are not available because hundreds of days are needed. The results of analytical estimation for `packet_tracer` are not presented since it is not clear how to apply the model based on queueing theory to applications with feed-back loops. For each step of exhaustive search and heuristic search, we run the application 20 times and compute the average to consider measurement noise. In event-driven simulation and analytical estimation, the time required for profiling is included, and we run the application 20 times to compute averages during the profiling.

App.	Method	Queue Capacities [tokens]						Search	Search	Speed
		Q1	Q2	Q3	Q4	Q5	Q6	steps	time	up
d3d10	ES	8	64	64				8K	15 hours	5.75
	HS	16	16	16				35	4 min.	5.62
	DO	64	64	128				1	9 sec.	5.25
	AE	4	32	16				1	7 sec.	5.31
dedup	ES	N/A	N/A	N/A	N/A			160K	>500 days	N/A
	HS	256	2	2,408	2,048			63	5 hours	5.41
	DO	8	4	2,048	4,096			1	5 min.	5.40
	AE	4	4	8	32			1	5 min.	4.41
djpeg	ES	64	4	128	16			160K	10 hours	1.60
	HS	64	4	64	8			37	3 sec.	1.60
	DO	4	4	128	4			1	2 sec.	1.60
	AE	4	4	8	4			1	<1 sec.	1.51
ferret	ES	N/A	N/A	N/A	N/A	N/A		3.2M	>50K days	N/A
	HS	2	2	2	128	128		98	40 hours	7.12
	DO	4	4	4	128	128		1	24 min.	7.12
	AE	4	4	8	32	16		1	24 min.	4.10
hmm calibrate	ES	32						20	10 hours	7.65
	HS	32						11	7 hours	7.65
	DO	64						1	17 min.	7.65
	AE	32						1	17 min.	7.65
mad	ES	32	4	8				8K	8 days	2.96
	HS	64	4	8				39	1 hours	2.96
	DO	16	4	4				1	1 min.	2.96
	AE	8	4	4				1	1 min.	2.92
packet tracer	ES	N/A	N/A	N/A	N/A	N/A	N/A	64M	>100K days	N/A
	HS	64	64	1,024	32	16	2,048	117	6 hours	7.39
	DO	4	1,024	32	32	32	16	1	4 min.	7.30
	AE	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

`ferret`, and `packet_tracer`, an exhaustive search was not possible because it requires hundreds of days. Our heuristic search finishes within two hundred steps in every application.

5.3.2 Dynamic, Local Optimum Approximation

Although the heuristic search described in the previous section significantly reduces the number of application runs necessary for approximating static optimal inter-actor queue capacities, a considerable amount of time can still be required for each run. In addition, the search may need to be repeated when the application is modified or a different multi-core architecture is encountered. This section presents how queues can be sized with even much fewer application runs by inspecting locally-optimal dynamic capacity adjustments. Here, "*locally*-optimal" indicates that the constructed capacity adjustment of a queue is optimal with respect to the producer-consumer actor pair of the queue. Constructing globally optimal dynamic capacity adjustments remains as future work.

Figure 5.7 illustrates the construction of locally-optimal dynamic capacity adjustments. First, we run the application with unbounded queues so that enqueues never block, and gather enqueue and dequeue event traces. Figure 5.7(a) is an example trace of enqueue and dequeue events, where e_i denotes ith enqueue time and d_i denotes ith dequeue time. Second, we delay enqueue events to reduce queue occupancies as much as possible subject to the following two constraints, where e'_i denotes the ith delayed enqueue time:

$$\begin{aligned} e'_i &\leq d_i && ,\forall i \\ e'_i - e'_{i-1} &\geq e_i - e_{i-1} && ,\forall i \end{aligned}$$

The first constraint is due to causality between ith enqueue and ith dequeue event. The second constraint is due to the time required for computing ith token in the producer. Lastly, we compute occupancies at delayed enqueue events. The optimal adjustment with respect to the enqueuer and dequeuer actors is setting queue capacities to the computed occupancies at each delayed enqueue event.

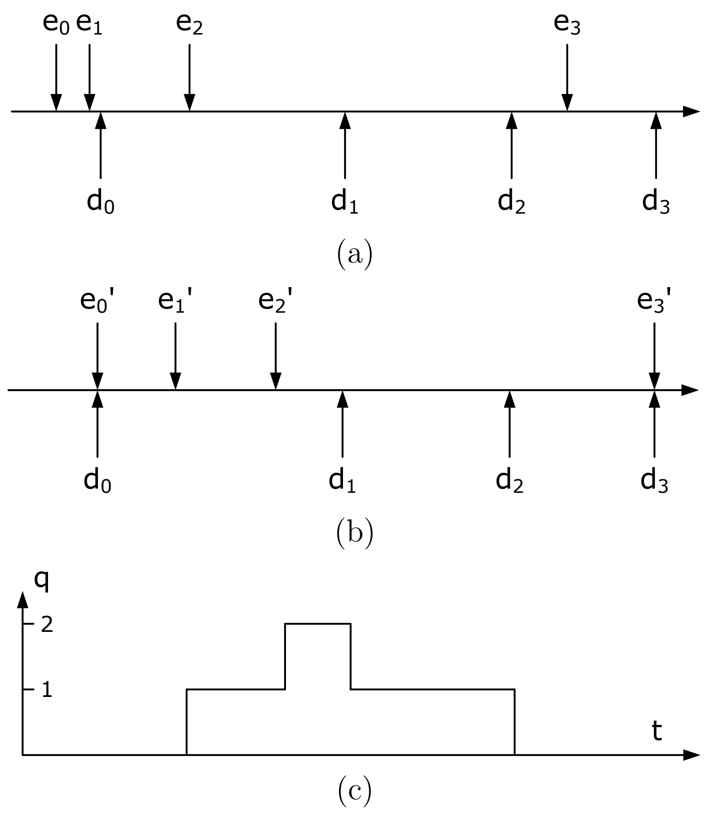


Figure 5.7: Illustration of computing the optimal queue capacity adjustment. (a) A trace from an execution with unbounded queues, where e_i denotes ith enqueue and d_i denotes jth dequeue. (b) Delay enqueues as much as possible. (c) Compute the occupancy at each delayed enqueue event.

Figure 5.8 shows an example locally-optimal dynamic capacity adjustment of Q2 in `d3d10` shown in Figure 5.9(a) with `courtyard` as the input. Note that, up to 0.03 second, optimal queue capacities are similar to the occupancies during an execution with unbounded queues, since enqueue events cannot be delayed there (i.e., the producer is the bottleneck). After 0.03 second, optimal queue capacities are significantly smaller than the occupancies of unbounded queues by delaying enqueue events (i.e., the consumer is the bottleneck after 0.03 second).

Ideally, we can achieve the dynamic optimal by following the constructed sequence of adjustments. However, each run leads to different dynamic optima, and following the optimal sequence of adjustments during run-time can incur significant overheads

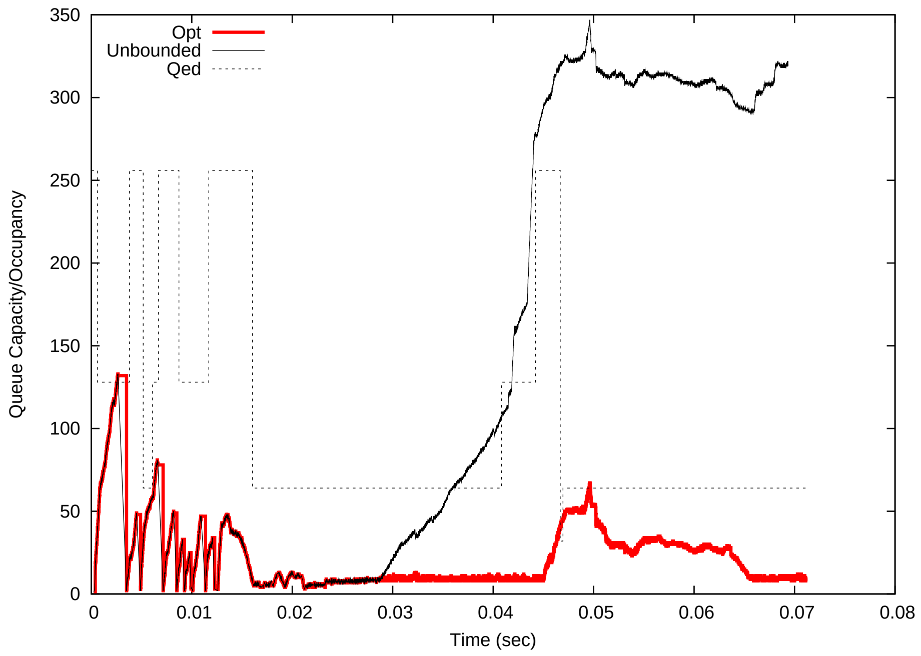


Figure 5.8: An example dynamic optimal capacity adjustment of Q2 in `d3d10` with `courtyard` scene. The thick red line denotes the optimal capacity adjustment, the thin solid black line denotes the occupancy during an execution with unbounded queues, and the thin dotted line denotes a sequence of capacity adjustments by QED.

for storing a long sequence of adjustments. Instead, we use a heuristic that sets the capacities to 90% percentile of capacities used during optimal adjustments constructed from 20 profiling runs, and we call this method dynamic, local optimum approximation. For example, if 20 profile runs result in optimal capacity adjustments similar to that is shown in Figure 5.8, we set the capacity of queue to 64.

In Table 5.1, the results of dynamic optimal approximation (DO) are shown for each application. The dynamic optimal approximation method achieves a throughput within 1% of the heuristic search on average. Considering that the time required for the dynamic optimal approximation is significantly faster than that of the heuristic search, the dynamic optimal approximation is an attractive method for computing queue capacities. However, the dynamic optimal approximation shares a problem with the heuristic search — choosing a representative input data set for approximation so that the found capacities are optimal across a wide range of input data. We expect that run-time queue capacity adjustment is used in conjunction with load balancing scheduling, as will be discussed in Section 5.5.3. In this setting, the pipeline configuration keeps changing in run-time, further complicating the application of methods based on profiling.

The method described in this section does not construct globally-optimal adjustments because delayed enqueue events affect optimal dequeue timings of upstream queues. Note that our method does not adjust the dequeue timings of upstream queues and constructs adjustments in isolation per queue. If actors form an acyclic connectivity, it is easy to extend the locally-optimal method described in this section to construct globally-optimal adjustments: we visit queues in a reverse topological order and construct adjustments of them by using the method described in this section while using dequeue timings as the ones dictated by the delayed enqueue timings of downstream actors. However, it is presently not clear how to further extend this method to actors with a cyclic connectivity, which remains as future work.

5.3.3 Analytical Estimation

In the previous section, we show that our dynamic optimum approximation method achieves a throughput similar to that achieved by the heuristic search method, while significantly reducing the search time. Alternatively, we can reduce the search time by using an analytical model of pipeline-parallel applications, which is examined in this section.

Navarro et al. [117] develop an analytical model of pipeline-parallel applications based on queueing theory. Based on the analytical model, they also present a method to compute the minimum queue capacities that do not degrade the throughput. They model each pipeline stage as an $M/M/c/K/K$ queue where the first M denotes that inter-arrival time follows a time-independent exponential distribution, the second M denotes that the service time also follows a time-independent exponential distribution,

c denotes the number of servers (i.e., threads), the first K denotes the queue capacity, and the second K denotes the client population. Since there are only K clients, the arrival rate of clients decreases as more clients are present in the queue, whereas the rate does not change in an infinite population model. For the details of this analytical model, refer to [117]. In an $M/M/c/K/K$ queue, the probability that n clients are present (receiving service or waiting) in the queue, P_n, can be derived as follows (Section 6.5 in [16]):

$$P_n = \begin{cases} \binom{K}{n}\left(\dfrac{\lambda}{\mu}\right)^n P_0 & , n < c \\ \dfrac{K!}{(K-n)!c!c^{n-c}}\left(\dfrac{\lambda}{\mu}\right)^n P_0 & , n \geq c \end{cases}$$

Here, λ denotes the arrival rate, and μ denotes the service rate of one server (i.e., thread). P_0 is a normalization factor that makes the sum of P_n values 1. Using these P_n values, we can compute the effective arrival rate of the i the stage, λ_{ei}, as follows [16, 117]:

$$\lambda_{ei} = \sum_{n=0}^{K}(K-n)\frac{P_n(\lambda = \frac{1}{T_{arr}}, \mu = \mu_i, c = c_i)}{T_{arr}}$$

Here, the slowest average service time, T_{arr}, is the maximum among $\dfrac{1}{c_i\mu_i}$, where c_i denotes the number of threads for the ith pipeline stage and μ_i denotes the average service rate of the ith pipeline stage. When there is no limitation on queue capacity, this slowest average service time determines the throughput. When queue capacities are too small, throughput can be reduced from that is determined by the slowest average service time, which we want to avoid. Therefore, for each stage, the following condition should hold:

$$\lambda_{ei} \geq \frac{1}{T_{arr}} \Rightarrow \sum_{n=0}^{K}(K-n)P_n(\lambda = \frac{1}{T_{arr}}, \mu = \mu_i, c = c_i) \geq 1$$

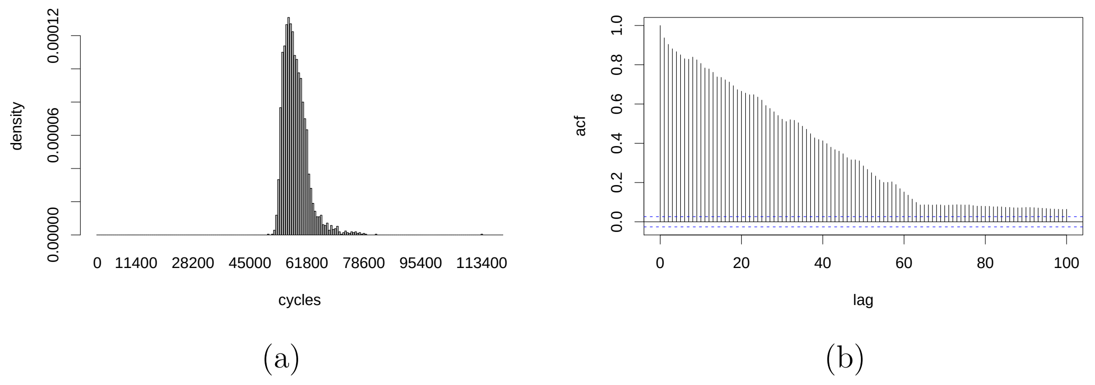


Figure 5.9: Statistics of applications (measured by PAPI [32]). (a) A histogram of the number of cycles per `out` stage iteration in `ferret`. (b) An auto-correlation function of the number of cycles per `PixelShader` stage iteration in `d3d10` with input `fairy`. These dynamic/figures are generated by R [6], a statistics package. In (b), dotted horizontal lines indicate the point of statistical significance — values between these lines are not statistically significant.

Using this condition[2], we can find the minimum queue capacity, K, that avoids throughput reduction. We set the queue capacity to the smallest power of 2 that is no smaller than K[3].

In Table 5.1, analytical estimation (AE) rows present the result of applying the analytical method developed by Navarro et al. [117]. We can see that their method tends to underestimate the queue capacities, particularly in `dedup` and `ferret`, resulting in a maximum of 74% and an average of 16% slowdown. This can be explained by the following two facts: First, execution times of some pipeline stages do not follow an exponential distribution. For example, Figure 5.9(a) shows a histogram of the number of cycles per `out` stage iteration in `ferret`, which deviates from that of an exponential distribution. Second, even those stages whose execution times follow an exponential distribution not time-independent. For example, Figure 5.9(b) shows the auto-correlation of the number of cycles per `PixelShader` stage iteration in `ferret`.

[2] This condition can be interpreted as follows: The average number of empty entries in the queue is greater than or equal to 1.

[3] When we use queues with the reserve-commit interface described in Section 5.2.2, each producer reserves one buffer space during its iteration. Therefore, we add the number of producers to K.

We can see high auto-correlation values at non-zero smaller lags, which implies that the execution times are bursty and not time-independent.

5.4 Experimental Setup

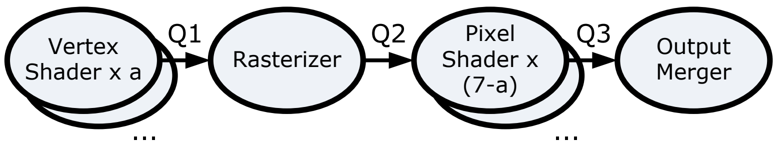
This section describes the evaluation setup that we use throughout this chapter.

Figure 5.9 shows the pipeline configuration of each application we evaluate. As opposed to the applications that have been evaluated for stream programming systems [51, 124], we select applications that have actors with variable input/output rates. Dotted lines denote single-producer, single-consumer queues. Queues associated with solid lines are shared by multiple producers and/or multiple consumers. For example, Q3 in `dedup` (Figure 5.9(b)) is shared by five `Compress` threads. The details of applications we evaluate are as follows:

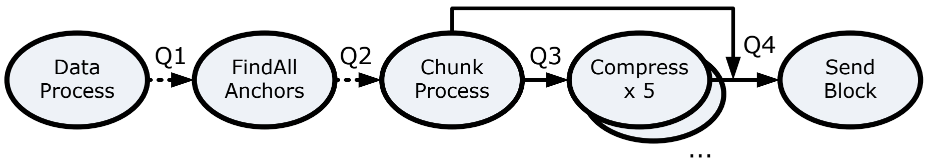
d3d10 (Figure 5.9(a)): A Direct3D pipeline that is ported from GRAMPS [141]. `Vertex Shader` and `Pixel Shader` share the same thread pool. When both `Vertex Shader` and `Pixel Shader` task can be executed, we give `Pixel Shader` priority to reduce memory pressure as in Sugerman et al. [141]. Since a software rasterizer completely dominates the performance of graphics pipelines [141], our `Rasterizer` actor emulates a hardware rasterizer by emitting pre-computed rasterized outputs that correspond to incoming triangles. We use `courtyard`, `fairy`, and `teapot` used in [141], which represent a fragment-heavy scene, a vertex-heavy scene, and a mixed scene, respectively. A fragment-heavy scene imposes heavier load on `Pixel Shader` while a vertex-heavy scene imposes heavier load on `Vertex Shader`.

dedup (Figure 5.9(b)): Data stream compression using "de-duplication" in PARSEC benchmark suite [25]. We use the largest input set, `native`.

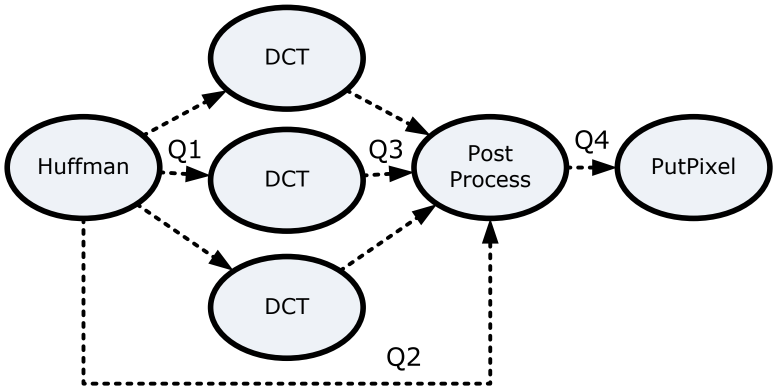
djpeg (Figure 5.9(c)): A JPEG decoder. We parallelize the 6b version implemented by Independent JPEG Group [75] included in MediaBench [4, 96]. Instead of sharing the input and output queues of `DCT` threads, we split the inputs and join the outputs of `DCT` threads in round robin. This does not introduce load imbalance between `DCT` threads since their execution time per input token is close to constant. Run-time variation mainly comes from the `Huffman` actor whose execution time and



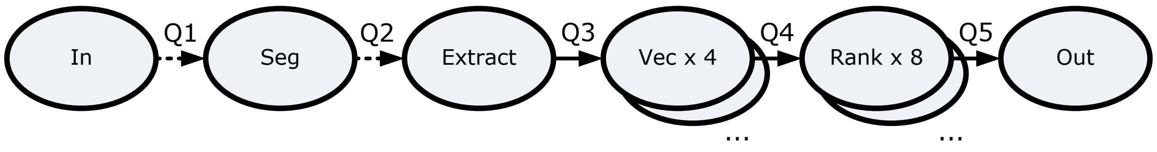
(a) `d3d10`: Direct3D pipeline



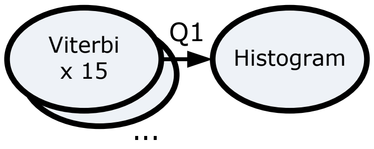
(b) `dedup`: data compression



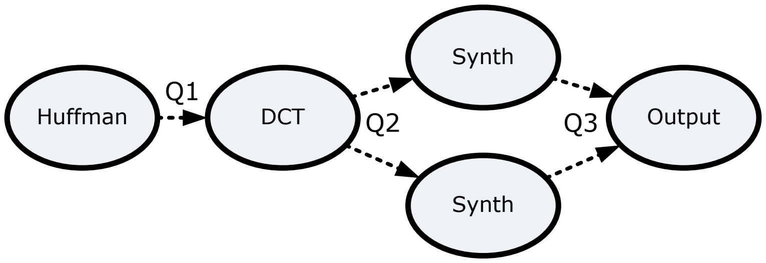
(c) `djpeg`: JPEG decoder



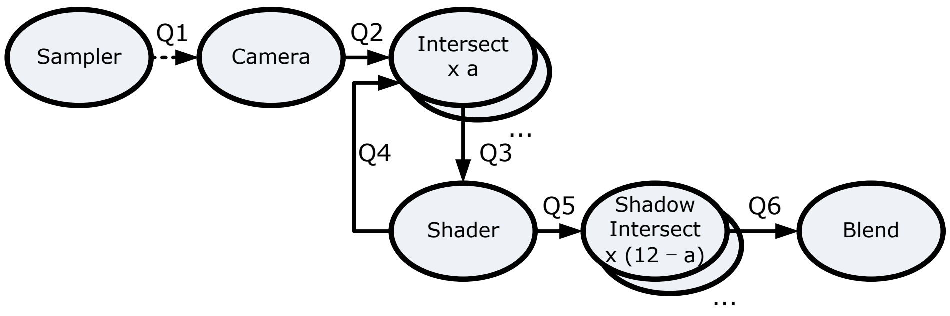
(d) `ferret`: similarity search



(e) `hmmcalibrate` : hidden Markov model calibration



(f) `mad`: MP3 decoder



(g) `packet_tracer` : a ray tracer implementation

Figure 5.9: Pipeline configuration of applications. `Compress`×`5` in `dedup` denotes that 5 threads are assigned to the `Compress` stage. In `d3d10`, 7 threads are shared by two shader stages. Similarly, in `packet_tracer`, 12 threads are shared by two intersect stages.

amount of output data depend on the entropy of the stream. As the input data set, we use `input_base_4CIF_96bps.jpg` and `testimg.jpg` from MediaBench, and `input_small.jpg` and `input_large.jpg` from MiBench [55].

ferret (Figure 5.9(d)): Image similarity search in PARSEC benchmark suite. We use the largest input set, `native`.

hmmcalibrate (Figure 5.9(e)): Calibrating hidden Markov model for biosequence analysis from SPEC2006.

mad (Figure 5.9(f)): We parallelize the 0.14.2b version of MAD MP3 decoder [149] included in MiBench [55]. As in `djpeg`, we split the inputs and join the outputs of `Synth` threads in a round robin manner. Run-time variation mainly comes from `Huffman` as in `djpeg`. We use the 10 most popular free downloadable MP3 files from Amazon.com.

packet_tracer (Figure 5.9(g)): A ray tracer implementation with a feedback loop that is ported from GRAMPS [141]. `Intersect` and `Shadow Intersect` share the same thread pool. Similar to shaders in `d3d10`, we give `Shadow Intersect` priority. When neither input queue of `Intersect` is empty, the input queue along the feedback loop (Q4) gets priority. We use the same input set used for `d3d10`.

We use two 2.26 GHz E5520 quad-core Intel Nehalem processors with hyper-threading, 32KB 8-way set associative private L1 D-caches, 256KB 8-way set associative private L2 caches, and 8MB 16-way set associative shared L3 caches. We use Ubuntu server with Linux kernel version 2.6.32, and exclusive access to the machine is provided. Applications are compiled by gcc 4.4.3 with -O3 option, and each thread is assigned appropriate affinity value. All parallelization is done by pthread API. Performance numbers are computed as the average of 20 runs. TLB misses and L1/L2 cache misses are measured by PAPI 4.1.2 [32], a performance counter library. L3 cache misses are measured by `perf`, a performance analysis tool in Linux [4]. The hardware pre-fetcher is turned on when the execution time is measured and turned off when TLB and cache misses are measured.

It would be ideal to evaluate with the Elm architecture as well to be consistent with the evaluation in Chapter 4. In static applications evaluated in Chapter 4, executing a few iterations of the steady state often suffices to accurately capture the application behavior. In contrast, we typically need to execute applications from beginning to end with large enough inputs to capture the dynamic behavior of applications we evaluate in this chapter. Since the Elm simulator takes about 5K cycles to simulate 1 cycle, it would have taken about 4 days to simulate `ferret` as `native.dat` the input. Instead of resorting into smaller input data, which may degrade the accuracy of capturing the dynamic behavior of applications, we decided to use a real machine (an Intel Nehalem machine in our case). The design of QED should be applicable to architectures with more software controls such as Elm, although more optimizations such as the ones exploiting scatter-gather DMA operations are possible. How much benefit we can get from this additional optimization opportunity remains as future work.

Before starting any experiment with respect to queue capacities, we find the number of threads for each stage that achieves the maximum speed-up, which is shown in Figure 5.9. An important optimization for pipeline-parallel applications is *packetization*: send tokens as a packet to amortize communication and synchronization

[4] The last level cache misses are counted by the uncore in the Intel Nehalem architecture, and PAPI presently does not handle performance counters associated with the uncore.

overhead [141]. We use the degree of packetization that achieves maximum speed-up. For example, we use packets with 128 tokens in `d3d10`.

5.5 Evaluation

5.5.1 Execution Time

Table 5.2 shows execution times and the number of TLB misses normalized to those of static queues whose capacities are determined by the heuristic search method. QED achieves execution times and TLB misses similar to those of approximated static optima found by the heuristic search method.

The input set for measuring the execution time is the same as the one used for searching the capacities of static queues (i.e., training and validation sets are the same), which is favorable for the search method. When we use the static queues for an input outside the training set, their capacities may not be optimal. For example, when we increase the rendering resolution of `d3d10` to 2048×2048, QED is 6% faster than the static queues whose capacities are trained for 1024×1024 resolution. The variance of the `Rasterizer` stage's output rate increases at higher resolution, to which QED adapts by expanding the capacity of Q3 by 30% on average.

The dynamic optimum approximation method achieves an execution time within 1% of the heuristic search on average, while reducing the search time from several hours to half an hour (the details of search times are presented in Table 5.1). The analytical estimation based on queueing theory [117] results in a maximum of 74% and an average of 16% slowdown since it does not accurately model the behavior of applications (elaborated in Section 5.3.3).

The MS queue is on average 18% slower and incurs 3.9× TLB misses than the heuristic search method. This is due to the MS queue's un-throttled execution of upstream stages and its overhead associated with dynamic memory allocation. In order to separate these two effects, we also evaluate MS queues whose capacities are bounded by those determined by the heuristic search (`Bounded MS` in Table 5.2). `Bounded MS` reduces the number of TLB misses by 18% and improves the execution

Table 5.2: (a) Execution time and (b) the number of TLB misses normalized to those of static queues with capacities determined by the heuristic search method described in Section 5.3.1. Normalized execution time and TLB misses are presented with 95% margin of error assuming each application run is independent and identically distributed (i.i.d.).

- DO: static queues whose capacities are determined by dynamic, local optimum approximation presented in Section 5.3.2.
- Analytical: analytical estimation based on queueing theory presented in Section 5.3.3.
- MS: MS queues.
- Bounded MS: MS queues whose capacities are bounded by those determined by the heuristic search method.
- Large enough: "large enough" (2K tokens) static queues.

(a) Normalized execution time

	QED	DO	Analytical	MS	Bounded MS	Large enough
`d3d10`	0.99 ±0.04	1.07 ±0.05	1.06 ±0.10	1.26 ±0.06	1.20 ±0.08	1.18 ±0.08
`dedup`	1.00 ±0.04	1.00 ±0.02	1.23 ±0.02	0.98 ±0.01	0.98 ±0.02	1.00 ±0.03
`djpeg`	1.00 ±0.02	1.00 ±0.02	1.06 ±0.02	2.45 ±0.09	2.45 ±0.07	1.15 ±0.02
`ferret`	1.01 ±0.00	1.00 ±0.00	1.74 ±0.01	1.01 ±0.00	1.00 ±0.00	1.00 ±0.00
`hmmcalibrate`	0.99 ±0.00	1.00 ±0.00	1.00 ±0.00	1.00 ±0.00	1.00 ±0.00	1.00 ±0.00
`mad`	0.96 ±0.00	1.00 ±0.01	1.01 ±0.01	1.04 ±0.01	1.03 ±0.01	1.05 ±0.03
`packet_tracer`	1.02 ±0.00	1.01 ±0.00	N/A	1.02 ±0.00	1.02 ±0.00	1.00 ±0.00
geomean	1.00	1.01	1.16	1.18	1.17	1.05

(b) Normalized TLB misses

	QED	DO	Analytical	MS	Bounded MS	Large enough
`d3d10`	1.30 ±0.10	1.03 ±0.10	1.02 ±0.05	4.46 ±0.15	2.82 ±0.09	2.02 ±0.10
`dedup`	0.98 ±0.11	1.10 ±0.15	0.94 ±0.10	0.99 ±0.15	1.02 ±0.10	0.97 ±0.06
`djpeg`	1.45 ±0.19	1.00 ±0.17	0.93 ±0.23	53.82 ±0.24	44.36 ±0.21	2.87 ±0.20
`ferret`	0.95 ±0.08	1.01 ±0.05	0.94 ±0.07	0.78 ±0.12	0.77 ±0.13	1.00 ±0.04
`hmmcalibrate`	1.00 ±0.03	1.01 ±0.04	1.00 ±0.04	1.25 ±0.06	1.22 ±0.04	1.00 ±0.03
`mad`	0.89 ±0.10	0.86 ±0.05	0.86 ±0.02	14.72 ±0.36	12.92 ±0.18	17.39 ±0.20
`packet_tracer`	0.99 ±0.12	1.03 ±0.20	N/A	3.76 ±0.09	2.09 ±0.09	1.13 ±0.05
geomean	1.07	1.00	0.95	3.86	3.17	1.96

time by 1% by throttling execution of upstream stages. For example, in `d3d10` with unbounded MS queues, the `Rasterizer` stage advances unnecessarily faster than the bottleneck stage `PixelShader` and accumulates many tokens between `PixelShader` and `Rasterizer`. This un-throttled execution is avoided by bounded capacities in `Bounded MS` and static queues, and by capacity adjustment based on the variance of occupancy in QED. To make sure that memory allocation is not the parallelization bottleneck, we also measured the MS queue's performance with the Hoard allocator [21], but we observed a negligible ($< 1\%$) execution time difference.

The "Large Enough" configuration represents a current common-practice of using arbitrary large queues. We set the capacity of every queue to 2K tokens. We could have used other arbitrary "Large Enough" capacities (e.g., `dedup` uses 1M tokens in its original implementation), but we select the maximum capacity among those chosen by the heuristic search (2,048). In `d3d10`, `djpeg`, and `mad`, execution times increase by 18%, 15%, and 5% from those achieved by the heuristic search. These applications have queue tokens larger than 1KB; therefore, large queue capacities result in a high TLB miss rate. On the other hand, in the applications with smaller queue tokens such as `dedup`, `ferret`, `hmmcalibrate`, and `ferret`, "Large Enough" shows small slowdowns. This small slowdown mainly stems from the inefficiency of the cache architecture implemented in present-day processors in handling producer-consumer communication, which will be discussed in Section 5.5.3.

5.5.2 Energy Consumption

Figure 5.10 shows estimated energy consumption in the memory hierarchy normalized to that of static queues whose capacities are determined by the heuristic search method. Notice that this is a rough estimation to illustrate an energy saving trend achieved by QED, which is based on the energy per operation listed in Table 5.3. The estimated power consumption in the memory hierarchy is from 0.12 W (`packet_tracer`) to 1.60 W (`djpeg`). QED achieves an average of 15% energy savings over MS and "Large Enough" configurations. While `d3d10`, `djpeg`, and `mad` with QED show >10%

Table 5.3: Cache access energies that are estimated in a 45nm low-leakage process using CACTI [163] and DRAM access energy estimated using the results from Udipi et al. [148]. Access energy of L1 is per word, and that of others is per cache line (16-word). The DRAM access energy incorporates only the energy consumed in bit-lines, which dominates the total DRAM access energy [148]. We assume a 4GB DRAM, with two 2 GB ranks, each consisting of 256 MB, 4-bank devices as in [148]. The row-buffer hit rates of DRAM is assumed as 35% following the results for 16-core cases presented in [148]. As a rough estimation, DRAM write energy is assumed to be the same as the read energy.

Energy [nJ]	read	write
32KB 8-way L1	0.008	0.006
256KB 8-way L2	0.134	0.118
8MB 16-way L3	0.515	0.401
DRAM	25.067	25,067

differences from either MS or "Large Enough" configuration, differences in other applications are small. In `hmmer` and `packet_tracer`, a large fraction of energy is consumed by L1-3 caches. Smaller queue capacities do not translate into fewer accesses to L1-3 caches in the cache coherency architecture used in the Nehalem processors, which explains small energy savings of QED in `hmmer` and `packet_tracer`. In `dedup` and `ferret`, DRAM accounts for a large fraction of energy consumption, but the most of them come from cold misses. Since token sizes of `dedup` and `ferret` are smaller than those of `d3d10`, `djpeg`, and `mad`, increasing queue capacities to 2K tokens does not incur many capacity misses in the L3 cache (which is also reflected in the number of TLB misses shown in Table 5.2(b)).

5.5.3 Discussion

Long-term Variation

As described in Section 5.2.1, the main objective of QED is accommodating a short-term variation of application behavior. This motivates the usage of short-term time variance for adjusting the capacity in QED. Consequently, a long-term variation must be handled by load balancing scheduling algorithms such as work stealing [29].

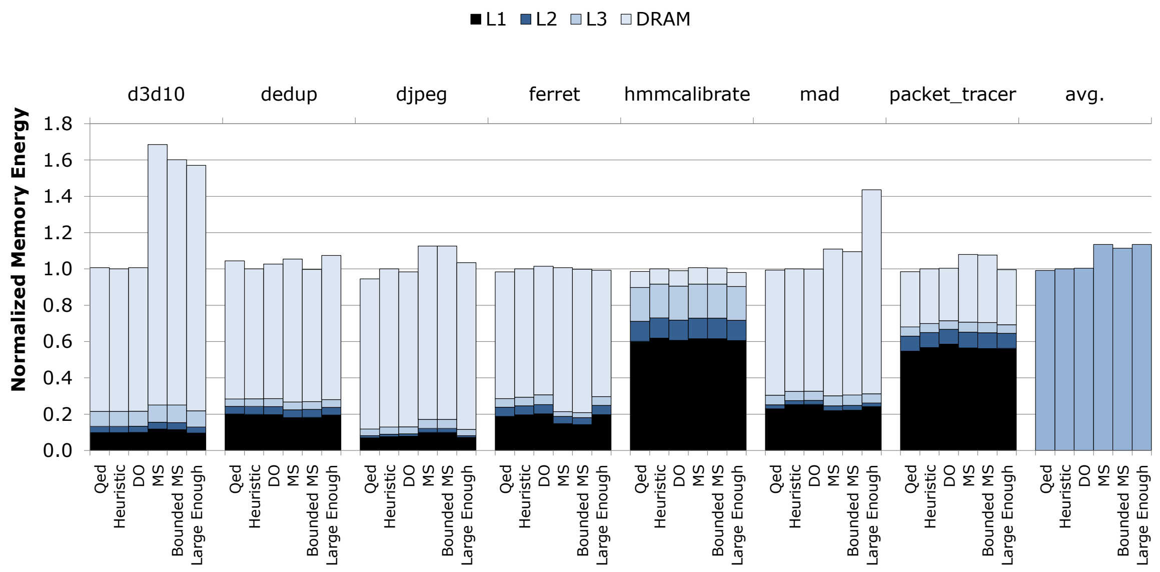


Figure 5.10: Estimated energy consumption in the memory hierarchy normalized to that of static queues with the heuristic search method.

For example, in `dedup`, the `ChunkProcess` stage encounters mostly duplicated chunks during the first one second, and, then, starts encountering chunks that have not seen before. Duplicated chunks are directly sent to the `SendBlock` stage bypassing the `Compress` stage, while newly seen chunks are processed by the `Compress` stage. This results in queue capacity adjustments shown in Figure 5.11, where the `Compress` stage becomes the bottleneck from 1 to 3 seconds while processing a burst of newly seen chunks. QED adjusts its capacity from 256 to 512 at the beginning of the burst but does not further expand the capacity, even though expansions up to 1500 will result in a faster execution as indicated by the optimal adjustment. This is because QED adjusts its capacity based on short-term time variance of occupancy. The long term load imbalance from 1 to 3 seconds should be handled by a load balancing scheduler that assigns more cores to the `Compress` stage.

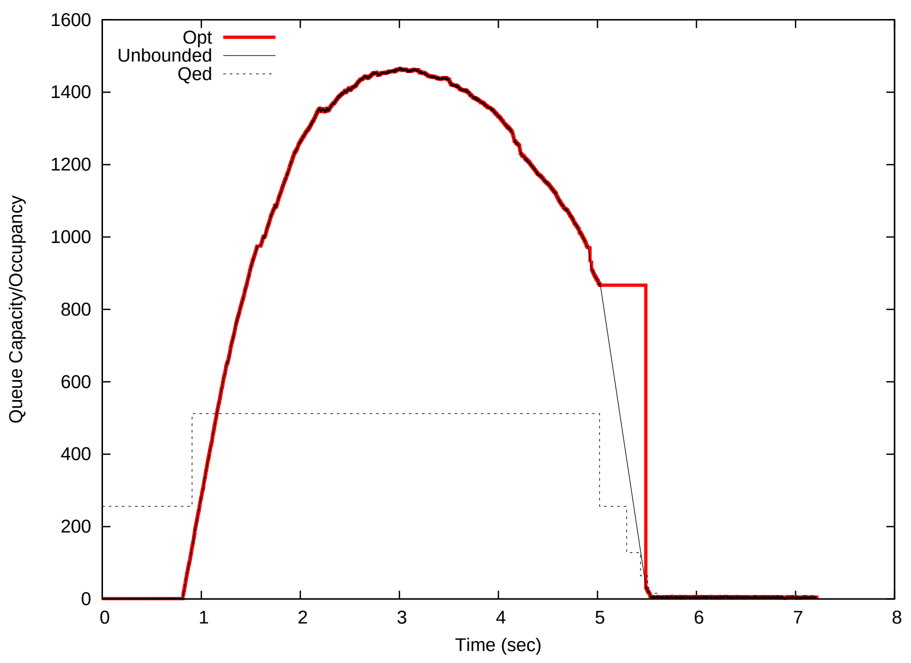


Figure 5.11: An example capacity adjustment of Q3 in `dedup` shown in Figure 5.9(b) that illustrates the need for a long-term load balancing mechanism that complements QED's handling of short-term variation. The thick red line denotes the optimal capacity adjustment, the thin solid black line denotes the occupancy during an execution with an unbounded queue, and the thin dotted line denotes a sequence of capacity adjustment by QED.

Inefficiencies of the Current Cache Architecture Regarding Producer-consumer Communications

Modest execution time differences (5%) between QED and static queues with arbitrary large capacities are shown in Section 5.5.1. This modest difference mainly stems from the inefficiency of the cache architecture implemented in present-day processors in handling producer-consumer communication. Future processors that resolve this inefficiency will further improve speed-ups and energy savings achieved by QED.

Figure 5.12 shows the throughput of a synthetic benchmark as we change the queue

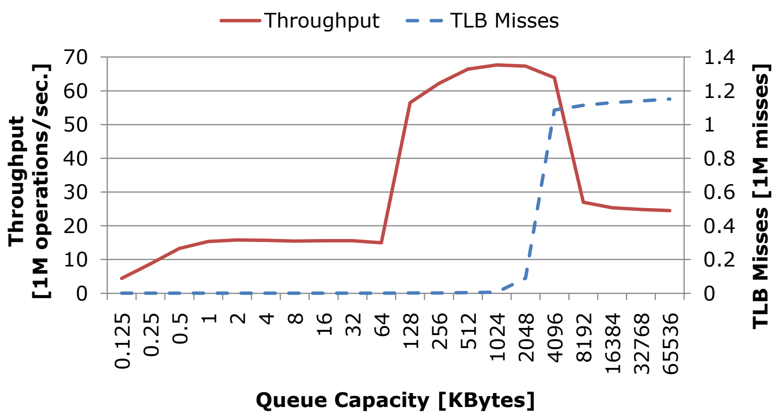


Figure 5.12: Throughput and TLB misses of a synthetic benchmark as we change the queue capacity.

capacity. The synthetic benchmark has two threads, a producer and a consumer. The producer enqueues packets with cache line size (16 integer numbers) 32M times, and the consumer computes the sum of the numbers present in the received packets. We assign affinity values to the threads so that they are mapped to the cores that share an L3 cache. We observe that the throughput steeply increases when the queue does not fit in the L2 private cache at 128KB [5] (the throughput decreases due to TLB misses above 2MB, and it further decreases when the queue does not fit in the L3 cache at 8MB).

One may consider this as abnormal (performance improves when the working set does not fit in caches), but this is a natural behavior of the cache architecture of Nehalem processors where L2 caches are private and invalidate-based cache coherency protocol is implemented. Ha et al. [56] observe a similar behavior in their work on concurrent dynamic analysis framework that extensively uses concurrent queues. They explain that, when a queue fits in the L2 private cache, the consumer must read a cache line with `modified` coherency state that resides in the remote L2 cache owned by the producer, resulting in long latencies (measured to be 75 cycles in Molka et al. [114]). When the queue does not fit in the L2 private cache and has sufficiently

[5] This does not occur precisely at L2 cache capacity (256KB) since instructions and data other than queue tokens are also present.

large occupancy, queue tokens are evicted to the shared L3 cache by the time consumers read them, resulting in shorter latencies (measured to be 38 cycles in [114]). As a result, stream applications need to either thrash private caches (when the queue does not fit in the private cache) or suffer long latencies from remote caches (when the queue fits in the private cache). Other inefficiencies of the current cache architectures with respect to producer-consumer communication such as superfluous refills [103] are discussed in Section 5.6 where we review related work.

To address this inefficiency, processors should support a mechanism of associating locality hints with cache lines. For example, Wang et al. [158] propose associating *evict-me* hints with cache lines that have no temporal locality. Note that, although the current Intel processors do support non-temporal memory operations, they bypass the entire cache hierarchy and go directly to the main memory rather than using on-chip L3 caches. With architecture support such as *evict-me*, we should be able to consistently achieve maximum throughput (more than 60M operations per second) at queue capacities smaller than 128KB when we perform measurements similar to those shown in Figure 5.12. Alternatively, we can use an update-on-write coherency protocol instead of an invalidate-on-write one for cache lines that are tagged with a certain locality hint. Such architecture support renders smaller queue capacities more preferable since consumers can read data from their private caches when queues fit in the cache.

A Guideline to Select a Queue Design

This chapter focuses on queue designs for buffering data between actors in dynamic stream applications. In other settings, general-purpose queue designs such as the MS queue may be a better choice.

When we target static stream applications, statically-sized queues are simpler to implement and have lower overhead than QED. As presented in Chapter 4, appropriate inter-actor queue capacities of static stream applications can be computed at the compile time, and there is no need for dynamic adjustment of the capacities. Although we expect that the implementation and verification cost of QED can be amortized over many applications, it may not be desirable to implement QED from

Table 5.4: A guideline to select a queue design. Each "+" denotes that the design is preferable when the corresponding condition is met.

		QED	Statically-sized	MS
Inter-actor queue?	Yes	+		
	No			+
Static stream application?	Yes		+	
	No	+		+
Have a QED implementation?	Yes	+		
	No		+	+
Application behavior varies considerably depending on inputs?	Yes	+		+
	No		+	
The number of queues?	Many	+		+
	Few		+	
Token size?	Large	+		
	Small		+	+

scratch when a QED implementation is not already available for the target language or architecture. Depending on the language or architecture, synchronization primitives such as compare-and-swap may not be available, which demands partial reimplementation of QED and verification of whether the implementation is free from race conditions. In this case, using statically-sized queues and determining their capacities by the heuristic search method presented in Section 5.3.1 can be a better option provided that the application behavior does not significantly vary over input data and the number of queues is small enough to perform the search in a reasonable time.

When tokens are small and when we target an architecture where fitting queues in L1 or L2 caches has little impact on the performance as in the current x86 processors (discussed in the previous section), using statically-sized queues with "large enough" capacities is a simple and efficient solution. Table 5.4 summarizes the guideline to select a queue design.

5.6 Related Work

An algorithm is non-blocking if the suspension of a thread does not stop the progress of the other threads. Non-blocking algorithms are less prone to deadlock/livelock and robust with respect to abrupt thread terminations compared to lock-based algorithms. They are also less prone to priority inversion, where a low-priority thread holding a lock is preempted for a high-priority thread waiting for the lock, wasting cycles.

There are multiple levels of progress conditions for non-blocking data structures (Chapter 3.7 in Herlihy and Shavit [66]). The strongest progress condition is *wait-free*. A method is wait-free if it guarantees that every call finishes its execution in a finite number of steps. This guarantees that every thread makes progress, hence starvation-free. The second strongest progress condition is *lock-free*. A method is lock-free if it guarantees that some threads make progress (system-wide progress). The weakest progress condition is called *obstruction-free*. A method is obstruction-free, if it executes in isolation, it finishes in a finite number of steps. QED satisfies the obstruction-free property. Theoretically, an obstruction-free data structure can make no system-wide progress but this only happens with a pathological scheduling of threads. Obstruction-free retains the most important benefits of non-blocking algorithms: it is less prone to deadlock and priority inversion. Herlihy et al. [65] argue that obstruction-freedom admits substantially simpler implementations, and they believe that, in practice, it provides the benefits of wait-free and lock-free implementations.

There is a large body of work [63,71,91,112,113] on designing efficient and scalable non-blocking queues. QED is designed specifically for buffering data between actors in stream applications, and, therefore, it should be distinguished from general-purpose queue designs such as the MS queue.

Giacomoni et al. [46] present a cache-optimized concurrent queue implementation. Their queue is lock-free and avoids cache-line thrashings that occur when the queue is almost empty or full by maintaining a certain distance between the tail and head index (called *temporal slipping*). However, theirs is a single-producer and single-consumer queue, while multi-producer and multi-consumer ones are necessary for load balancing when multiple threads are assigned to a pipeline stage with dynamic execution time per iteration[6]. In our evaluation, since tokens are typically bigger than cache lines due to *packetization* [141], the importance of temporal slipping has decreased.

Navarro et al. [117] apply queueing theory to performance modeling of pipeline-parallel applications. We show that application behavior does not accurately follow the assumption of their model and that their method under-estimates the queue capacities. In order to analyze queueing models with less restrictive assumptions, discrete-event simulations are typically used [16, 30], but they are not likely to yield more accurate or faster modeling than our dynamic optimal profiling method.

Molka et al. [114] measure the latency and throughput of Nehalem memory system for different sharing patterns. When a cache line is modified by a remote core that shares an L3 cache, the latency in reading the cache line is 75 cycles when the working set fits in L2, while the latency *decreases* to 38 cycles when the working set does not fit in L2 but fits in L3. In the same setting, the read throughput is 13GB/s when the working set fits in L2, while it *increases* to 19.9GB/s when the working set does not fit in L2 but fits in L3. This latency and throughput difference prevents realizing the full benefit of optimal queue capacities: while we want to find the smallest queue capacities that accommodate the run-time variation of applications, the architecture provides shorter latencies and higher throughputs to larger queue capacities.

Leverich et al. [103] compare two competing models of chip multiprocessor memory systems: hardware-managed coherent caches and software-managed streaming memory. They conclude that a cache-based memory architecture can achieve similar performance even in applications that have motivated the emergence of streaming architectures. However, they also explain that their conclusion holds only if the cache architecture resolves its inefficiencies with respect to producer-consumer sharing patterns. For example, since caches often use write-allocate policies, when a producer incurs a store miss due to modification by another producer, the entire cache line is refilled from a remote or larger cache. This unnecessarily increases the latency and wastes bandwidth because the refilled cache line will be used as write-only by the producer, thus called superfluous refills.

[6] We observe an average of 7% slowdown from using single-producer and single-consumer static queues with round robin distribution.

5.7 Chapter Summary

This chapter presents QED, a non-blocking queue design for minimizing memory footprint of inter-actor queues in dynamic stream applications. This is complementary to the compile-time queue capacity analysis for static applications presented in Chapter 4. In QED, the capacity of a queue is dynamically adjusted depending on the run-time behavior of applications. The capacity expands only when the high run-time variation of application behavior demands so. Otherwise, it is kept to a minimum, throttling the execution of non-bottleneck actors. Although queues are dynamically sizeable, they are based on circular arrays, thereby avoiding overhead associated with dynamic memory allocations.

We compare the performance of QED with four alternative methods: heuristic search, dynamic optimum approximation, analytical estimation, and the MS queue. The analytical estimation method does not accurately model the application behavior. The MS queue incurs un-throttled execution of upstream stages and overhead associated with dynamic memory allocation. The heuristic search method approximates optima of statically-sized queues, achieving the highest throughput among the alternative methods. However, the heuristic search requires long search times (up to 40 hours) and representative training set selection. We show that QED achieves a throughput similar to that of the heuristic search method, while requiring neither search time nor training set selection.

Chapter 6

Conclusion

6.1 Summary and Contributions

This dissertation presented a collection of software mechanisms that improve the energy efficiency of embedded computing, focusing on its most energy-hungry part — instruction and data delivery. For instruction delivery energy, a compiler algorithm that manages instruction scratch-pad memories (SPMs) is presented (Chapter 3). For data delivery energy in stream applications commonly found in the embedded domain, I presented a compile-time queue capacity computation analysis (Chapter 4) and a queue data structure that adjusts its capacity at run-time (Chapter 5). These mechanisms were estimated to save 87% of instruction delivery energy, 33% of data delivery energy in static stream applications, and 15% of data delivery energy in dynamic stream applications. This contributes to a large fraction of energy savings of the Elm architecture compared to conventional RISC embedded processors.

Our instruction SPM management algorithm (Chapter 3) is differentiated from previous SPM management algorithms in that SPMs with our algorithm achieves not only smaller L0 access energy and faster execution times but also fewer L1 accesses than filter caches. Key ideas behind our algorithm are 1) fine-grain instruction placement and 2) careful consideration of the tagless and compiler-managed properties of SPMs.

The compile-time queue capacity computation analysis for static stream applications (Chapter 4) determines minimum queue capacities that sustain throughput of stream graphs with arbitrary connection. This analysis enables flexible transformations of stream graphs, which was previously avoided due to potential deadlocks or throughput degradation. This dissertation exemplifies this advantage by team scheduling, in which actors are aggregated and amortized to minimize communication and synchronization.

The dynamically-sized array-based queue design called QED (Chapter 5) throttles the execution of non-bottleneck actors and expands its capacity only when the high run-time variation of application behavior demands so. The capacity adjustment is based on the approximated time variance of queue occupancy. Although this adjustment is a heuristic, it is shown that QED achieves a throughput similar to that of the approximated static and dynamic optimum presented in Chapter 5. QED has an advantage over the methods computing approximated static and dynamic optimum since QED requires neither long search times nor careful training set selection.

A central theme of the Elm architecture design was improving energy efficiency by exposing controls over deep and distributed storage hierarchies to the compiler [12]. From the compiler's perspective, critical design decisions were 1) the selection of resources to be controlled by the compiler, 2) architecture supports to make the software control efficient, and 3) compiler algorithm design strategies.

A resource should be controlled by the compiler only if the compiler can analyze the usage of the resource accurately enough to realize energy efficiency improvements. This may seem obvious but careful evaluation and insight on the system is required to do so. For example, note that our SPM management algorithm targets small L0 instruction SPMs, where the majority of locality captured comes from loops, for which the compiler has a proven ability to analyze. In contrast, according to our evaluation, energy efficiency benefit of using SPMs compared to caches is not as promising at other levels of the memory hierarchy with larger stores, where the locality from irregular control flows plays an important role. It is also important to complement compiler analyses with run-time mechanisms to cover portions that are not accurately analyzable in compile time. For example, a dynamically-sized queue data structure

(Chapter 5) was necessary to optimize the data delivery of dynamic parts of stream applications to complement the static queue capacity computation analysis (Chapter 4).

In order to achieve effective compiler control over resources, it is crucial to give the compiler *flexibility* and *uniformity* so that the compiler has ample space to optimize and is free from dealing with corner cases. To this end, Elm provides several architectural features. For example, instruction SPMs in Elm provide wrap-around access to make instruction SPM placement algorithm simpler and suffer less fragmentation. Instruction SPMs in Elm also provide non-blocking instruction transfers so that the SPM management algorithm can primarily focus on optimizing energy consumption without excessive consideration on performance overhead. DMA operations with strided/scatter/gather accesses allow the compiler to flexibly choose memory layout of inter-actor queues, which is particularly important for reducing the overhead of actors that split or join streams.

Regarding algorithm design strategy, we prefer searching for sub-optimal answers over large solution spaces to searching for optimal answers over restricted solution spaces. For example, several previous SPM management algorithms [41, 139, 156] formulate SPM placement problem as integer linear programming (ILP). Even though their ILP formulations find optimal solutions, due to the time complexity of ILP solvers, their solution space is restricted to transferring instructions in blocks with coarse granularity. Instead, our algorithm expands the solution space to one with finer-grain blocks and uses a heuristic (our algorithm is optimal only with respect to non-nested loops as shown in Claim 3.4.1). As another example, whereas SGMS [90] restricts synchronization and communication boundaries as the minimum steady state of the entire application, team scheduling relaxes the restriction to apply synchronization and communication minimizing transformations in a flexible order starting from those that yield the maximum gain (Chapter 4).

The following section lists future directions that can improve the work presented.

6.2 Future Directions

Application to Other Domains

Although embedded computing and high performance computing (HPC) are the low-end and high-end extreme of the computing spectrum, respectively, they share numerous characteristics. Both computing domains are severely constrained by energy consumption: embedded devices are constrained by their battery life and the biggest challenge for next generation HPC computing is identified as energy efficiency [89]. In addition, applications in both embedded and HPC tend to have regular control flow and data access patterns compared to their desktop computing counterparts. Therefore, they are amenable to interesting compile-time and run-time locality optimizations that improve energy efficiency. I believe that the techniques presented in this dissertation, particularly the instruction SPM management algorithm, are applicable to the HPC. The question is how many changes will be needed in the algorithms due to different cost structures in HPC (e.g., HPC applications typically work on larger data sets and use floating point operations that are more energy-hungry than fixed-point operations in embedded applications).

Hybrid Instruction Scratch-pad Memory

As a hybrid design, instruction SPMs can have a few tags associated with instruction blocks that are transferred by the compiler. Before transferring a block to the instruction SPM, we can look up the tags to avoid unnecessary transfers. This compensates a disadvantage of compiler-based approaches with respect to the lack of run-time information — the compiler must re-transfer instructions to the SPM if there is even a slight chance of them having been overwritten. Since tags are associated with blocks, if the typical block size is sufficiently large, we can keep the tag storage small so that tag lookup overhead is minimal.

Interaction with Dynamic Scheduling

Dynamic scheduling algorithms need to be incorporated to complement the static scheduling implemented in `elmhc`. Without dynamic scheduling such as work stealing [29], dynamic stream applications parallelized in pipelining can suffer from severe load imbalance. Perhaps, a load balancing scheduling specialized for streaming applications can offer better locality and response time. The GRAMPS project [141] investigates scheduling algorithms tailored to pipeline-parallel applications, expanding its scope from graphics pipelines. It would be also interesting to investigate the interaction between load balancing dynamic scheduling and the run-time inter-actor queue capacity adjustment mechanism presented in Chapter 5. As discussed in Section 5.5.3, the capacity adjustment mechanism is designed for accommodating short-term load imbalance, while dynamic scheduling needs to handle long-term load imbalance. It would be ideal if imbalance handling is optimally divided so that the queue capacity adjustment deals with load imbalance if and only if the duration of imbalance is not long enough to justify overheads associated with load balancing scheduling.

Multi-modal Stream Applications

Dynamic stream applications typically have static parts in them, and these parts can be statically scheduled to minimize communication and synchronization overhead while a dynamic scheduling algorithm is used globally. It would be interesting to look at how statically scheduled parts and dynamically scheduled parts can constructively coexist, maximizing the communication and synchronization overhead reduction in statically scheduled parts while minimizing the overall load imbalance.

Interaction with Data-level Parallelism

Only trivially data-parallel actors without any state are presently parallelized (i.e., fissioned) in `elmhc`. Affine partitioning algorithms [44, 45, 105] can be applied to exploit more data-level parallelism from actors. As a complement to affine partitioning algorithms, explicit data parallelization can be performed as in the Sequoia programming language [43]. A question is how to attain the best combination of data-level parallelism with pipeline parallelism, and how to efficiently orchestrate vertical data movements optimized for data-level parallelism (e.g., blocking [94]) and horizontal data movements optimized for pipeline parallelism (e.g., DMAs between producers and consumers). Gordon et al. [50, 51] investigate the first question and present a series of reasonable heuristics to exploit data-level and pipeline parallelism together. However, they do not consider two key factors for finding a good combination of both parallelisms: 1) the locality between data-parallel partitions of adjacent stages and 2) the amount of state associated with each stage. These two factors determine how much non-local communication will be incurred when data-level parallelism is exploited. An extreme case is when 1) the entire pipeline can be fused into a single stage, 2) we can find an affine partitioning with no communication for the fused stage, and 3) the fused stage is associated with no state. In this case, data-level parallelism results in no communication and excellent load balance. However, there can be cases where partitions for data-level parallelism result in non-local communication between stages. State associated with stages also leads to cache misses when the sum of the state size of all stages exceeds the cache capacity and data-level parallelism is solely exploited. A partitioning algorithm that is aware of the locality trade-offs from these two factors should be able to achieve better parallelism speed-up and energy efficiency.

Multi-dimensional Streams

For applications such as image processing, whose data are inherently multi-dimensional, it is often awkward to describe data reuse pattern in one dimension. The Elk programming language is designed so that its syntax can be easily extended to multi-dimensional streams. However, how to efficiently orchestrate the data movement of multi-dimensional streams is still an open question. Generalized Multidimensional Synchronous Data Flow [116] and Windowed Synchronous Data Flow [83] frameworks support multi-dimensional streams but require significant time complexity. Black-Schaffer [26, 28] suggests support for two-dimensional streams as a reasonable compromise that captures most of the applications with multi-dimensional streams while keeping the time complexity manageable. It would be interesting to quantify the benefits obtained from supporting two-dimensional streams with respect to performance and productivity.

Bibliography

[1] ANTLR Praser Generator Webpage. http://www.antlr.org.

[2] ELM Webpage. Concurrent VLSI Architecture Group, Stanford University. http://cva.stanford.edu/projects/elm/software.htm.

[3] Janino Embeddable Java Compiler Webpage. http://janino.codehaus.org.

[4] MediaBench II Benchmark. `http://euler.slu.edu/~fritts/mediabench/mb2/index.html`.

[5] QED: Queue Enhanced with Dynamic sizing, Google Code website. `https://code.google.com/p/qed`.

[6] The R Project for Statistical Computing. `http://www.r-project.org`.

[7] T. Abatzoglou and B. O'Donnell. Minimization by Coordinate Descent. *Journal of Optimization Theory and Applications*, 36(2):163–174, 1982.

[8] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2007.

[9] Federico Angiolini, Luca Benini, and Alberto Caprara. Polynomial-Time Algorithm for On-chip Scratchpad Memory Partitioning. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 318–326, 2003.

[10] Federico Angiolini, Francesco Menichelli, Alberto Ferrero, Luca Benini, and Mauro Olivieri. A Post-Compiler Approach to Scratchpad Mapping of Code. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 259–267, 2004.

[11] M. Armand and J.-M. Tarascon. Building better batteries. *Nature*, 414(6861):359–367, 2001.

[12] James Balfour. *Efficient Embedded Computing*. PhD thesis, Stanford University, 2010.

[13] James Balfour, William J. Dally, David Black-Schaffer, Vishal Parikh, and Jongsoo Park. An Energy-Efficient Processor Architecture for Embedded Systems. *Computer Architecture Letters*, 7(1):29–32, 2008.

[14] James Balfour, R. Curtis Harting, and William J. Dally. Operand Registers and Explicit Operand Forwarding. *Computer Architecture Letters*, 8(2):60–63, 2009.

[15] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems. In *International Conference on Hardware Software Codesign*, pages 73–78, 2002.

[16] Jerry Banks, John Carson, Barry L. Nelson, and David Nicol. *Discrete-Event System Simulation*. Prentice Hall, 2004.

[17] Luiz André Barroso. The Price of Performance. *ACM Queue*, 3(7):48–53, 2005.

[18] Daniel U. Becker and William J. Dally. Allocator Implementations for Network-on-Chip Routers. In *Conference on Supercomputing (SC)*, pages 1–12, 2009.

[19] L. A. Belady. A Study of Replacement Algorithms for Virtual-storage Computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[20] Richard Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[21] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *International Conference on Architecture Support for Programming Language and Operating Systems (ASPLOS)*, pages 117–128, 2000.

[22] Shuvra S. Bhattacharyya, Joseph T. Buck, Soonhoi Ha, and Edward A. Lee. Generating Compact Code from Dataflow Specifications of Multirate Signal Processing Algorithms. *IEEE Transactions on Circuits and Systems*, 42(3):138–150, 1995.

[23] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. APGAN and RPMC: Complementary Heuristics for Translating DSP Block Diagrams into Efficient Software Implementations. *Design Automation for Embedded Systems*, 2(1):33–60, 1997.

[24] Shuvra S. Bhattacharyya, Sundararajan Sriram, and Edward A. Lee. Optimizing Synchronization in Multiprocessor DSP Systems. *IEEE Transactions on Signal Processing*, 45(6):1605–1618, 1997.

[25] Christian Biena, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, 2008.

[26] David Black-Schaffer. *Block Parallel Programming for Real-Time Applications on Multi-core Processors.* PhD thesis, Stanford University, 2008.

[27] David Black-Schaffer, James Balfour, William J. Dally, Vishal Parikh, and Jongsoo Park. Hierarchical Instruction Register Organization. *Computer Architecture Letters*, 7(2):41–44, 2008.

[28] David Black-Schaffer and William J. Dally. Block-Parallel Programming for Real-Time Embedded Applications. In *International Conference on Parallel Programming (ICPP)*, pages 297–306, 2010.

[29] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Strealing. *Journal of ACM (JACM)*, 46(5):720–748, 1999.

[30] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor Shridharbhai Trivedi. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications.* Wiley-Interscience, 2006.

[31] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization.* Cambridge University Press, 2004.

[32] S. Browne, J. Dongarra, N. Garner, G. Ho, and Mucci P. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[33] G. J. Chaitin. Register Allocation & Spilling via Graph Coloring. In *Compilter Construction*, pages 98–105, 1982.

[34] Anantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen. Low-Power CMOS Digital Design, 1992.

[35] Yoonseo Choi, Yuan Lin, Nathan Chong, Scott Mahlke, and Trevor Mudge. Stream Compilation for Real-time Embedded Multicore Systems. In *International Symposium on Code Generation and Optimization (CGO)*, pages 210–220, 2009.

[36] John Cocke and Ken Kennedy. An Algorithm for Reduction of Operator Strength. *Communications of the ACM*, 20(11):850–856, 1977.

[37] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Language and Systems (TOPLAS)*, 13(4):451–490, 1991.

[38] William J. Dally, James Balfour, David Black-Schaffer, James Chen, R. Curtis Harting, Vishal Parikh, Jongsoo Park, and David Sheffield. Efficient Embedded Computing. *IEEE Computer*, 41(7):27–32, 2008.

[39] William J. Dally and Brian Towles. *Principles and Practices of Interconnection Networks.* Morgan Kaufmann, 2004.

[40] Jan Edler and Mark D. Hill. Dinero IV Trace-Driven Uniprocessor Cache Simulator.

[41] Bernhard Egger, Chihun Kim, Choonki Jang, Yoonsung Nam, Jaejin Lee, and Sang Lyul Min. A Dynamic Code Placement Technique for Scratchpad Memory Using Postpass Optimization. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 223–233, 2006.

[42] Bernhard Egger, Jaejin Lee, and Heonshik Shin. Dynamic Scratchpad Memory Management for Code in Portable System with an MMU. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(2):1–38, 2008.

[43] Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Conference on Supercomputing (SC)*, 2006.

[44] Paul Feautrier. Some Efficient Solutions to the Affine Scheduling Problem: I. One-dimensional Time. *International Journal of Parallel Programming*, 21(5):313–348, 1992.

[45] Paul Feautrier. Some Efficient Solutions to the Affine Scheduling Problem: II. Multidimensional Time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[46] John Giacomoni, Tipp Moseley, and Manish Vachharajani. FastForward for Efficient Pipeline Parallelism. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 43–52, 2007.

[47] Nikolas Gloy and Michael D. Smith. Procedure Placement using Temporal-ordering Information. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):977–1027, 1999.

[48] Ricardo Gonzalez. Xtensa: A Configurable and Extensible Processor. *IEEE Micro*, 20(2):60–70, 2000.

[49] Ricardo Gonzalez and Mark Horowitz. Energy Dissipation in General Purpose Microprocesors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1284, 1996.

[50] Michael I. Gordon. *Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures.* PhD thesis, Massachusetts Instituite of Technology, 2010.

[51] Michael I. Gordon, William Thies, and Saman P. Amarasinghe. Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs. In *International Conference on Architecture Support for Programming Language and Operating Systems (ASPLOS)*, pages 151–162, 2006.

[52] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffman, David Maze, and Saman P. Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *International Conference on Architecture Support for Programming Language and Operating Systems (ASPLOS)*, pages 291–303, 2002.

[53] Ann Gordon-Ross, Susan Cotterell, and Frank Vahid. Tiny Instruction Caches for Low Power Embedded Systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 2(4):449–481, 2003.

[54] Jayanth Gummaraju and Mendel Rosenblum. Stream Programming on General-Purpose Processors. In *International Symposium on Microarchitecture (MICRO)*, pages 343–354, 2005.

[55] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *IEEE 4th Annual Workshop on Workload Characterization*, pages 83–94, 2001.

[56] Jungwoo Ha, Matthew Arnold, Stephen M. Blackburn, and Kathryn S. McKinley. A Concurrent Dynamic Analysis Framework for Multicore Hardware. In *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 155–174, 2009.

[57] Soonhoi Ha and Edward A. Lee. Compile-Time Scheduling and Assignment of Data-Flow Program Graphs with Data-Dependent Iterations. *IEEE Transactions on Computers*, 40(11):1225–1238, 1991.

[58] S. Habnic and J. Gaisler. Status of the LEON2/3 processor development. In *Data Systems in Aerospace (DASIA)*, 2007.

[59] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding Sources of Inefficiencies in General-Purpose Chips. In *International Symposium on Computer Architecture (ISCA)*, pages 37–47, 2010.

[60] Amir H. Hashemi, David R. Kaeli, and Brad Calder. Efficient Procedure Mapping using Cache Line Coloring. In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 171–182, 1997.

[61] Paul Havlak. Nesting of Reducible and Irreducible Loops. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(4):557–567, 1997.

[62] M. S. Hecht and Jeffrey D. Ullman. Characterizations of Reducible Flow Graphs. *Journal of the ACM (JACM)*, 21(3):367–375, 1974.

[63] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat Combining and the Synchronization-Parallelism Tradeoff. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 355–364, 2010.

[64] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 2003.

[65] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *International Conference on Distributed Computing Systems*, pages 522–529, 2003.

[66] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[67] Stephen Hines, Joshua Green, Gary Tyson, and David Whalley. Improving Program Efficiency by Packing Instructions into Registers. In *International Symposium on Computer Architecture (ISCA)*, pages 260–271, 2005.

[68] Stephen Hines, Gary Tyson, and David Whalley. Reducing Instruction Fetch Cost by Packing Instructions into Register Windows. In *International Symposium on Microarchitecture (MICRO)*, pages 19–29, 2005.

[69] Stephen Hines, Gary Tyson, and David Whalley. Addressing Instruction Fetch Bottlenecks by Using an Instruction Register File. In *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 165–174, 2007.

[70] Stephen Hines, David Whalley, and Gary Tyson. Guaranteeing Hits to Improve the Efficiency of a Small Instruction Cache. In *International Symposium on Microarchitecture (MICRO)*, pages 433–444, 2007.

[71] Moshe Hoffman, Ori Shalev, and Nir Shavit. The Baskets Queue. *Principles of Distributed Systems*, pages 401–414, 2007.

[72] H. Peter Hofstee. Power Efficient Processor Architecture and the CELL Processor. In *International Symposium on High-Performance Computer Architectures (HPCA)*, pages 258–262, 2005.

[73] Steven Hsu, Vishak Venkatraman, Sanu Mathew, Himanshu Kaul, Mark Anders, Saurabh Dighe, Wayne Burleson, and Ram Krishnamurthy. A 2GHz 13.6mW 12x9b Multiplier for Energy Efficient FFT Accelerators. In *European Solid-State Circuit Conference*, pages 199–202, 2005.

[74] Wen-mei W. Hwu and Pohua P. Chang. Achieving High Instruction Cache Performance with an Optimizing Compiler. In *International Symposium on Computer Architecture (ISCA)*, pages 242–251, 1989.

[75] Independent JPEG Group. `http://www.ijg.org`.

[76] International Technology Roadmap for Semiconductors. `http://www.itrs.net`.

[77] International Technology Roadmap for Semiconductors. Model for Assessment of CMOS Technologies and Roadmaps (MASTAR). `http://www.itrs.net/models.html`.

[78] Andhi Janapsatya, Aleksandar Ignjatović, and Sri Parameswaran. A Novel Instruction Scratchpad Memory Optimization Method based on Concomitance Metric. In *Asia and South Pacific Design Automation Conference*, pages 612–617, 2006.

[79] Murali Jayapala, Francisco Barat, Tom Vander Aa, Francky Catthoor, Henk Corporaal, and Geert Deconinck. Clustered Loop Buffer Organization for Low Energy VLIW Embedded Processors. *IEEE Transactions on Computers*, 54(6):672–683, 2005.

[80] M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnana, I. Kadayif, and A. Parikh. Dynamic Management of Scratch-Pad Memory Space. In *Design Automation Conference (DAC)*, pages 690–695, 2001.

[81] Michal Karcmarek, William Thies, and Saman Amarasinghe. Phased Scheduling of Stream Programs. In *Conference on Language, Compiler, and Tool Support for Embedded Systems (LCTES)*, pages 103–112, 2003.

[82] George Karypis and Vipin Kumar. METIS: Unstructured Graph Partitioning and Sparse Matrix Ordering System. Technical report, Department of Computer Science, University of Minnesota, 1995.

[83] J. Keinert, C. Haubelt, and J. Teich. Modeling and Analysis of Windowed Synchronous Algorithms. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 14–19, 2006.

[84] Johnson Kin, Munish Gupta, and William H. Mangione-Smith. The Filter Cache: An Energy Efficient Memory Structure. In *International Symposium on Microarchitecture (MICRO)*, pages 184–193, 1997.

[85] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. 220(4598):671–680, 1983.

[86] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy Code Motion. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 224–234, 1992.

[87] Ming-Yung Ko, Chung-Ching Shen, and Shuvra S. Bhattacharyya. Memory-constrained Block Processing for DSP Software Optimization. *Journal of Signal Processing Systems*, 50(2):163–177, 2008.

[88] Alex Kogan and Erez Petrank. Wait-Free Queues with Multiple Enqueuers and Dequeuers. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2011.

[89] Peter Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, and Katherine Yelick. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. 2008.

[90] Manjunath Kudlur and Scott Mahlke. Orchestrating the Execution of Stream Programs on Multicore Platforms. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 114–124, 2008.

[91] Edya Ladan-Mozes and Nir Shavit. An Optimistic Approach to Lock-Free FIFO Queues. *Distributed Computing*, 20(5):323–341, 2007.

[92] Kumar N. Lalgudi, Marios C. Papaefthymiou, and Miodrag Potkonjak. Optimizing Computations for Effective Block-Processing. *ACM Transactions on Design Automation of Electronic Systems*, 5(3):604–630, 2000.

[93] Monica Lam. Software Pipelining: An Effective Scheduling Technique on VLIW Machines. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 318–328, 1988.

[94] Monica Lam, Edward Rothberg, and Michael Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, pages 63–74, 1991.

[95] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 75–86, 2004.

[96] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems. In *International Symposium on Microarchitectures (MICRO)*, pages 330–335, 1997.

[97] Edward A. Lee. *A Coupled Hardware and Software Architecture for Programmable Digital Signal Processors*. PhD thesis, University of California, Berkeley, 1986.

[98] Edward A. Lee and David G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, 36(1):24–35, 1987.

[99] Lea Hwang Lee, Bill Moyer, and John Arends. Instruction Fetch Energy Reduction Using Loop Caches for Embedded Applications with Small Tight Loops. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 267–269, 1999.

[100] Lea Hwang Lee, Bill Moyer, and John Arends. Low-Cost Embedded Program Loop Caching - Revisited. *Technical Report CSE-TR-411-99, University of Michigan*, 1999.

[101] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. In *International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, pages 46–57, 1998.

[102] Thomas Lengauer and Robert E. Tarjan. A Fast Algorithm for Finding Dominators in Flowgraph. *ACM Transactions on Programming Language and Systems (TOPLAS)*, 1(1):121–141, 1979.

[103] Jacob Leverich, Hideho Arakida, Alex Solomatnikov, Amin Firoozshahian, Mark Horowitz, and Christos Kozyrakis. Comparative Evaluation of Memory Models for Chip Multiprocessors. *ACM Transactions on Architectures and Code Optimization (TACO)*, 5(3):1–30, 2008.

[104] Amy W. Lim, Gerald I. Cheong, and Monica S. Lam. An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication. In *International Conference on Supercomputing*, pages 228–237, 1999.

[105] Amy W. Lim and Monica S. Lam. Maximizing Parallelism and Minimizing Synchronization with Affine Transforms. In *Symposium on Principles of Programming Languages (POPL)*, pages 201–214, 1997.

[106] Kevin Lim, Parthasarathy Ranganathan, Juchuan Chang, Chandrakant Patel, Trevor Mudge, and Steven Reinhardt. Understanding and Designing New Server Architectures for Emerging Warehouse-Computing Environments. In *International Symposium on Computer Architecture (ISCA)*, pages 315–326, 2008.

[107] Yuan Lin, Manjunath Kudlur, Scott Mahlke, and Trevor Mudge. Hierarchical Coarse-grained Stream Compilation for Software Defined Radio. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 115–124, 2007.

[108] Zhi-Quan Luo and Paul Tseng. On the Convergence of the Coordinate Descent Method for Convex Differentiable Minimization. volume 72, pages 7–35, 1992.

[109] Yves Mathys and André Châtelain. Verification Strategy for Integration 3G Baseband SoC. In *Design Automation Conference (DAC)*, pages 7–10, 2003.

[110] Peter Mattson. *A Programming System for the Imagine Media Processor*. PhD thesis, Stanford University, 2002.

[111] S. McFarling. Program optimization for instruction caches. In *International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, pages 183–191, 1989.

[112] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 267–275, 1996.

[113] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using Elimination to Implement Scalable and Lock-Free FIFO Queues. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 253–262, 2005.

[114] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S. Muller. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 261–270, 2009.

[115] James Montanaro, Richard T. Witek, Krishna Anne, Andrew J. Black, Elizabeth M. Cooper, Daniel W. Dobberpuhl, Paul M. Donahue, Jim Eno, Gregory W. Hoeppner, David Kruckemyer, Thomas H. Lee, Peter C. M. Lin, Liam Madden, Daniel Murray, Mark H. Pearce, Sribalan Santhanam, Kathryn J. Snyder, Ray Stephany, and Stephen C. Thierauf. A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor. *IEEE Journal of Solid-State Circuits*, 31(11):1703–1714, 1996.

[116] Praveen K. Murthy and Edward A. Lee. Multidimensional Synchronous Dataflow. *IEEE Transactions on Signal Processing*, 50(8):2064–2079.

[117] Angeles Navarro, Rafael Asenjo, Siham Tabik, and Calin Cascaval. Analytical Modeling of Pipeline Parallelism. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 281–290, 2009.

[118] Nghi Nguyen, Angel Dominguez, and Rajeev Barua. Memory Allocation for Embedded Systems with a Compile-Time-Unknown Scratch-Pad Size. *ACM Transactions on Embedded Computing Systems (TECS)*, 8(3):1–32, 2009.

[119] Alan V. Oppenheim, Alan S. Willsky, and S. Hamid Nawab. *Signals & Systems*. Prentice Hall, 1997.

[120] Amit Pabalkar, Aviral Shrivastava, Arun Kannan, and Jongeun Lee. SDRM: Simultaneous Determination of Regions and Function-to-Region Mapping for Scratchpad Memories. *High Performance Computing*, pages 569–582, 2008.

[121] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications. In *European Design and Test Conference*, pages 7–11, 1997.

[122] Jongsoo Park, James Balfour, and William J. Dally. Maximizing the Filter Rate of L0 Compiler-Managed Instruction Stores by Pinning. Technical Report 126, Concurrent VLSI Architecture Group, Stanford University, 2009.

[123] Jongsoo Park, James Balfour, and William J. Dally. Fine-grain Dynamic Instruction Placement for L0 Scratch-Pad Memory. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 137–146, 2010.

[124] Jongsoo Park and William J. Dally. Buffer-space Efficient and Deadlock-free Scheduling of Stream Applications on Mulit-core Architectures. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–10, 2010.

[125] Jongsoo Park and William J. Dally. Guaranteeing Forward Progress of Unified Register Allocation and Instruction Scheduling. Technical Report 127, Concurrent VLSI Architecture Group, Stanford University, 2011.

[126] Jongsoo Park, Sung-Boem Park, James Balfour, David Black-Schaffer, Christos Kozyrakis, and William J. Dally. Register Pointer Architecture for Efficient Embedded Computing. In *Conference on Design, Automation and Test in Europe (DATE)*, pages 600–605, 2007.

[127] Karl Pettis and Robert C. Hansen. Profile Guided Code Positioning. In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 16–27, 1990.

[128] Jose Luis Pino, Shuvra S. Bhattacharyya, and Edward A. Lee. A Hierarchical Multiprocessor Scheduling Systems for DSP Applications. Technical Report UCB/ERL M95/36, University of California, Berkeley, 1995.

[129] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T. N. Vijaykumar. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 90–95, 2000.

[130] Isabelle Puaut and Christophe Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Conference on Design, Automation and Test in Europe (DATE)*, pages 1484–1489, 2007.

[131] P. Puschner and Ch. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, pages 159–176, 1989.

[132] Bob Ramakrishna Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *International Symposium on Microarchitecture (MICRO)*, pages 63–74, 1994.

[133] Rajiv A. Ravindran, Pracheeti D. Nagarkar, Ganesh S. Dasika, Eric D. Marsman, Robert M. Senger, Scott A. Mahlke, and Richard B. Brown. Compiler Managed Dynamic Instruction Placement in a Low-Power Code Cache. In *International Symposium on Code Generation and Optimization (CGO)*, pages 179–190, 2005.

[134] Sebastian Ritz, Matthias Pankert, Vojin Živojnović, and Heinrich Meyr. Optimum Vectorization of Scalable Synchronous Dataflow Graphs. In *International Conference on Application-Specific Array Processors (ASAP)*, pages 285–296, 1993.

[135] Vivek Sarkar and Guang R. Gao. Optimization of Array Accesses by Collective Loop Transformations. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 194–205, 1991.

[136] Gilbert Christopher Sih. Multiprocessor Scheduling to Account for Interprocessor Communication. *Memorandum No. UCB/ERL M91/29, University of California, Berkeley*, 1991.

[137] Olli Silven and Kari Jyrkkä. Observations on Power-efficiency Trends in Mobile Communication Devices. *EURASIP Journal of Embedded Systems*, 2007(1):17–17, 2007.

[138] Sundararajan Sriram and Shuvra S. Bhattacharyya. *Embedded Mutiprocesors: Scheduling and Synchronization.* CRC, 2009.

[139] Stefan Steinke, Nils Grunwald, Lars Wehmeyer, Rajeshwari Banakar, M. Balakrishnan, and Peter Marwedel. Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory. In *International Symposium on Systems Synthesis*, pages 213–218, 2002.

[140] Stefan Steinke, Lars Wehmeyer, Bo-Sik Lee, and Peter Marwedel. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *Conference on Design, Automation and Test in Europe (DATE)*, pages 409–415, 2002.

[141] Jeremy Sugerman, Kayvon Fatahalian, Solomon Boulos, Kurt Akeley, and Pat Hanrahan. GRAMPS: A Programming Model for Graphics Pipeline. *ACM Transactions on Graphics (TOG)*, 28(1):1–11, 2009.

[142] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. WCET Centric Data Allocation to Scratchpad Memory. In *International Real-Time Systems Symposium (RTSS)*, pages 223–232, 2005.

[143] Gustavo E. Téllez, Amir Farrahi, and Majid Sarrafzadeh. Activity-Driven Clock Design for Low Power Circuits. In *International Conference on Computer-Aided Design (ICCAD)*, pages 62–65, 1995.

[144] William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs. In *International Symposium on Microarchitectures (MICRO)*, pages 356–369, 2007.

[145] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A Language for Streaming Applications. In *Compilter Construction*, pages 49–84, 2002.

[146] Philippas Tsigas and Yi Zhang. A Simple, Fast and Scalable Non-Blocking Concurrent FIFO Queue for Shared Multiprocessor Systems. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 134–143, 2001.

[147] Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. Dynamic Allocation for Scratch-Pad Memory Using Compile-Time Decisions. *ACM Transactions on Embedded Computing Systems (TECS)*, 5(2):472–511, 2006.

[148] Aniruddha N. Udipi, Naveen Muralimanohar, Niladrish Chatterjee, Rajeev Balasubramonian, Al Davis, and Norman P. Jouppi. Rethinking DRAM Design and Organization for Energy-Constrained Multi-Cores. In *International Symposium on Computer Architecture (ISCA)*, pages 175–186, 2010.

[149] Underbit Technologies, Inc. MAD: MPEG Audio Decoder. `http://www.underbit.com/products/mad`.

[150] US Environmental Protection Agency. Report to Congress on Server and Data Center Efficiency. `http://www.energystar.gov/ia/partners/prod_development/downloads/EPA_Datacenter_Report_Congress_Final1.pdf`.

[151] Jan-Willem van de Waerdt, Stamatis Vassiliadis, Sanjeev Das, Sebastian Mirolo, Chris Yen, Bill Zhong, Carlos Bastos, Jean-Paul van Itegem, Dinesh Amirtharaj, Kulbhushan Kalra, Pedro Rodriguez, and Hans van Antwerpen. The TM3270 Media-Processor. In *International Symposium on Microarchitecture (MICRO)*, pages 331–342, 2005.

[152] Richard van Nee and Ramjee Prasad. *OFDM for Wireless Multimedia Communications*. Artech Hoise, Inc., Norwood, MA, USA, 2000.

[153] Manish Verma and Pter Marwedel. Overlay Techniques for Scratchpad Memories in Low Power Embedded Processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(8):802–815, 2006.

[154] Manish Verma, Klaus Petzold, Lars Wehmeyer, Heiko Falk, and Peter Marwedel. Scratchpad Sharing Strategies for Multiprocess Embedded Systems: A First Approach. In *Workshop on Embedded Systems for Real-Time Multimedia*, pages 115–120, 2005.

[155] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Cache-Aware Scratchpad Allocation Algorithm. In *Conference on Design, Automation and Test in Europe (DATE)*, 2004.

[156] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Dynamic Overlay of Scratchpad Memory for Energy Minimization. In *International Conference on Hardware/software Codesign and System Synthesis*, pages 104–109, 2004.

[157] Yi-Pin Eric Wang and Tony Ottosson. Cell Search in W-CDMA. *IEEE Journal on Selected Areas in Communications*, 18(8):1470–1482, 2000.

[158] Zhenlin Wang, Kathryn S. McKinley, Arnold L. Rosenberg, and Charles C. Weems. Using the Compiler to Improve Cache Replacement Decisions. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 199–208, 2002.

[159] Mark N. Wegman and F. Kenneth Zadeck. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Language and Systems (TOPLAS)*, 13(2):181–210, 1991.

[160] Lars Wehmeyer, Urs Helmig, and Peter Marwedel. Compiler-optimized Usage of Partitioned Memories. In *Workshop on Memory Performance Issues*, pages 114–120, 2004.

[161] Lars Wehmeyer and Peter Marwedel. Influence of Memory Hierarchies on Predictability for Time Constrained Embedded Software. In *Conference on Design, Automation and Test in Europe (DATE)*, pages 600–605, 2005.

[162] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for Reduced CPU Energy. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 13–23, 1994.

[163] Steven J.E. Wilton and Norman P. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, 1996.

[164] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 30–44, 1991.

[165] Ahmad Zmily and Christos Kozyrakis. Block-Aware Instruction Set Architecture. *ACM Transactions on Architectures and Code Optimization (TACO)*, 3(3):327–357, 2006.