Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures

by

Michael I. Gordon

Bachelor of Science, Computer Science and Electrical Engineering
Rutgers University, 2000

Master of Engineering, Computer Science and Electrical Engineering
Massachusetts Institute of Technology, 2002

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2010

Author . . . . . .
Department of Electrical Engineering and Computer Science
May 21, 2010

Certified by . . . . . .
Saman Amarasinghe
Professor
Thesis Supervisor

Accepted by . . . . . .
Terry P. Orlando
Chair, Department Committee on Graduate Students

Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures

by

Michael I. Gordon

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2010, in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Abstract

Given the ubiquity of multicore processors, there is an acute need to enable the development of scalable parallel applications without unduly burdening programmers. Currently, programmers are asked not only to explicitly expose parallelism but also concern themselves with issues of granularity, load-balancing, synchronization, and communication. This thesis demonstrates that when algorithmic parallelism is expressed in the form of a stream program, a compiler can effectively and automatically manage the parallelism. Our compiler assumes responsibility for low-level architectural details, transforming implicit algorithmic parallelism into a mapping that achieves scalable parallel performance for a given multicore target.

Stream programming is characterized by regular processing of sequences of data, and it is a natural expression of algorithms in the areas of audio, video, digital signal processing, networking, and encryption. Streaming computation is represented as a graph of independent computation nodes that communicate explicitly over data channels. Our techniques operate on contiguous regions of the stream graph where the input and output rates of the nodes are statically determinable. Within a static region, the compiler first automatically adjusts the granularity and then exploits data, task, and pipeline parallelism in a holistic fashion. We introduce techniques that data-parallelize nodes that operate on overlapping sliding windows of their input, translating serializing state into minimal and parametrized inter-core communication. Finally, for nodes that cannot be data-parallelized due to state, we are the first to automatically apply software-pipelining techniques at a coarse granularity to exploit pipeline parallelism between stateful nodes.

Our framework is evaluated in the context of the StreamIt programming language. StreamIt is a high-level stream programming language that has been shown to improve programmer productivity in implementing streaming algorithms. We employ the StreamIt Core benchmark suite of 12 real-world applications to demonstrate the effectiveness of our techniques for varying multicore architectures. For a 16-core distributed memory multicore, we achieve a 14.9x mean speedup. For benchmarks that include sliding-window computation, our sliding-window data-parallelization techniques are required to enable scalable performance for a 16-core SMP multicore (14x mean speedup) and a 64-core distributed shared memory multicore (52x mean speedup).

Thesis Supervisor: Saman Amarasinghe
Title: Professor

Acknowledgments

I consider myself wonderfully fortunate to have spent the *Oughts* at the intellectual sanctuary that is MIT. In many ways it was a true sanctuary that offered me the freedom to pursue my passions in multiple fields, in multiple endeavors, and across multiple continents. During the experience, I learned from brilliant individuals, and made lifelong friends. I hope that MIT is a bit better for me having attended, and that I contributed to the advancement of science and technology in some small way.

First and foremost, I wish to thank my advisor Saman Amarasinghe. I truly appreciate the freedom that Saman extended to me combined with his constant accessibility. Saman was always available when I needed help. He has an uncanny ability to rapidly understand a problem, put that problem in perspective, and help one to appreciate the important issues to consider when solving it. Saman believed in the importance of my work in development, and had much patience for (and faith in) my PhD work. I also thank my thesis committee members: Martin Rinard and Anant Agarwal.

The best part of my MIT experience has been working with talented and passionate individuals. William Thies is my partner on the StreamIt project, and I am indebted to him for his collaboration and friendship. Bill is a wildly imaginative and brilliant individual without a hint of conceit. He possesses seemingly limitless patience and is fiercely honest, passionate, and compassionate. Sorry for gushing, but working with Bill was a great experience, and I miss our shared office.

StreamIt is a large systems project, incorporating over 27 people (up to 12 at a given time). I led the parallelization efforts of the project including the work covered in this thesis. William Thies also contributed to the work covered in Chapters 4, 6, and 7 of this dissertation. William led the development of the StreamIt language, incorporating the first notions of structured streams as well as language support for hierarchical data reordering [TKA02, TKG+02, AGK+05]. Many other individuals made this research possible (see [Thi09] for the complete accounting). I thank all the members of the StreamIt group including Rodric Rabbah, Ceryen Tan, Michal Karczmarek, Allyn Dimock, Hank Hoffman, David Maze, Jasper Lin, Andrew Lamb, Sitij Agrawal, and Janis Sermulins, Matthew Drake, Jiawen Chen, David Zhang, Phil Sung, and Qiuyuan Li. I also thank member of the Raw team and the Tilera team: Anant Agarwal, Michael Taylor, Ian Bratt, Dave Wentzlaff, Jonathan Eastep, and Jason Miller. I thank all the members of the Computer Architecture Group at MIT who gave me feedback on my research: Ronny Krashinsky, Marek Olszewski, Samuel Larsen, Michael Zhang, Mark Stephenson, Steve Gerding, and Matt Aasted. I am grateful for all of the help of Mary McDavit. She constantly went above and beyond her duties to help me, and I always enjoyed our conversations on wide-ranging topics.

I come from a small but close-knit family unit. My mother sacrificed much for me in my early years, and I was very lucky to be surrounded by multiple role models growing up. I am thankful that my family provided me with a loving and supportive environment. Thank you Mom, Dad (Steve), Bubby, Grandmom, Uncle Alan and Uncle Mark. I also thank the most recent addition to the family, Annette. In the last 5 years she has loved and supported me through graduate school and through all of my pursuits, no matter how far they have taken me from her. To the members of my family, I love you.

In this final paragraph I thank the friends who added to the richness of life while at MIT. One problem with making friends at MIT is that they are all ambitious individuals who end up in disparate parts of the world. Technology has made it easier to stay it touch, but I will miss my

Band of Brothers: Michael Zhang, Sam Larsen, Ronny Krashinsky, Bill Thies, Marek Olszewski, Ian Bratt, Mark Stephenson, and Steve Gerding. I also thank my non-MIT friends for keeping me somewhat “normal”: Josh Kaplan, David Brown, Lisa Giocomo, Helen O’Malley, John Matogo, and Scott Schwartz.

Dedicated to the memory of my grandfather Ike (*Zayde*) Pitchon.
A mensch who continues to inspire me...

Contents

List of Figures

Chapter 1

Introduction

Given the ubiquity of multicore processor designs, the computer science community has an acute need for accepted high-performance parallel programming languages. Libraries grafted onto imperative languages remain the most utilized solution for developing parallel software for multicore architectures. Due to this, programmers are asked not only to explicitly expose parallelism but also concern themselves with issues of granularity, load-balancing, synchronization, and communication. We cannot accept a lack of programmer uptake of new languages as a cause for this stagnation. Java, C++, C#, Python, and MATLAB are displacing C and FORTRAN because the former bring associated increases in programmability over the latter. Programmers are quick to add to their *répertoire* a system that eases their burden. Although many high-level parallel programming systems exist (e.g., High Performance FORTAN, Occam, Erlang, Chapel, ZPL, Fortress, and X10), no programming system has achieved the combined levels of programmability, portability, and scalability that programmers came to expect from von Neumann languages during the superscalar era. Even embarrassingly parallel domains lack accepted programming systems that increase programmer productivity and automatically manage parallelism. One contributing factor to this deficiency is the general failure of parallelizing compilers to deliver on the performance potential of high-level parallel constructs. Although the languages offer elegant and suitable parallel models, in many cases the parallel execution model is too general, requiring heroic analysis from the compiler for full coverage and demanding too much from architectural designs. Due to this sensitivity, the performance of automatic parallelization often falls short of expectations for implementations perhaps excepting programs included in the designers' benchmark suite [VNL97, YFH97].

Traditionally, language design is driven by the single goal of increasing programmer productivity in implementing the algorithm at hand. Unfortunately, it seems to be the case that little thought is given to the consequences of design choices for the goal of compiler-enabled high-performance. We maintain that the design of a language can satisfy the goals of programmability *and* high-performance. The first condition is the existence of effective compiler analyses and transformations that manage parallelism, enabling robust and portable high-performance for a given domain. Next, considerations of these compiler algorithms must inform the language design process; the programming system is designed holistically such that analyzability is maximized without sacrificing programmability. If a parallel programming system is designed appropriately, it can provide abstractions to model the application class while encouraging the programmer to write analyzable code, allowing the compiler to make all of the parallelization decisions automatically and effectively. In this dissertation we develop compiler technologies that, for an important domain, achieve

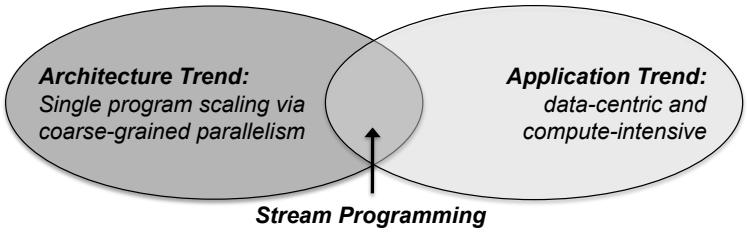


Figure 1-1: Stream programming is motivated by two prominent trends: the end of single-threaded performance increases, and the increase in data-centric computations such as media encoding and editing.

robust, end-to-end high-performance across distinct parallel architectures by automatically managing the issues of granularity, load-balancing, synchronization, and communication. Our compiler assumes responsibility for low-level architectural details, transforming algorithmic parallelism into a mapping that achieves scalable parallel performance for a given multicore target. These compiler technologies, coupled with the language idioms that enable them, can enlighten the designers of the next generation of high-level parallel programming systems.

Motivated by the recent and wide adoptions of higher-than-C-level languages, it is time to evaluate trends in computing to see if we can introduce programming abstractions that provide increases in programmability concomitant with the opportunity for vast performance improvements. The computing landscape has greatly changed in recent years. After experiencing sustained single-program scaling for much of the 1980s, 1990s, and early 2000s, two sweeping and radical trends have emerged in commodity computer architecture and applications:

1. **Microprocessors are scaling in terms of the number of cores.** Multicore processors have become the industry standard as single-threaded performance has plateaued. Whereas in the superscalar era, excess transistors and clock-scaling translated into (software) transparent scaling, multicore designs mean that excess transistors will translate into software visible core counts. As Moore's Law continues, each successive generation of multicores will potentially double the number of cores. Thus, in order for the scaling of multicores to be practical, the doubling of cores has to be accompanied by a near doubling of application performance, the burden of which has now shifted to software. This trend has pushed the performance burden to the compiler, as future application-level performance gains depend on effective parallelization across cores. Traditional programming models such as C, C++ and FORTRAN are ill-suited for automatically mapping a single program to multicore systems because they assume a single instruction stream and a monolithic memory. A high-performance programming model needs to expose all of the parallelism in the application, supporting explicit communication between potentially-distributed memories.

2. **Embedded and data-centric applications are becoming more prevalent.** Desktop workloads have changed considerably over the last decade. Personal computers have become communication and multimedia centers. Users are editing, encoding, and decoding audio and video, communicating, analyzing large amounts of real-time data (e.g., stock market information), and rendering complex scenes for game software. Since 2006, YouTube has been streaming over

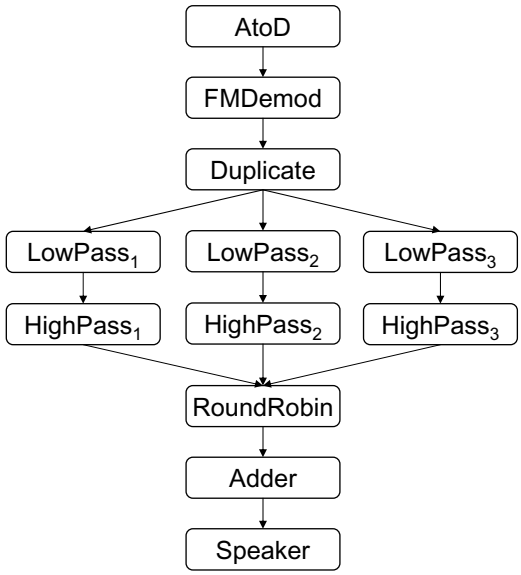


Figure 1-2: Example stream graph for a software radio with equalizer.

250 terabytes of video daily [Wat06]. A common feature of all of these examples is that they operate on streams of data from some external source. This application class elicits different architecture demands versus traditional general purpose or scientific codes, as data is short-lived, being processed for a limited time. Many potential killer applications of tomorrow will come from the domains of multimedia editing, computer vision, and real-time audio enhancement [CCD+08].

At the intersection of these trends is a broad and important application domain that we term *stream programs* (see Figure 1-1). A stream program is any program that is based around a regular stream of dataflow, as in audio, video, and signal processing applications (see Figure 1-2). Examples include radar tracking, software radios, communication protocols, speech coders, audio beamforming, video processing, cryptographic kernels, and network processing. These programs are rich in parallelism and their dependencies and computation can be accurately analyzed. Finally, stream programs have properties that make possible aggressive whole-program transformations to enable effective parallel mappings to multicore architectures.

In this dissertation, we develop and evaluate compiler technologies that enable *scalable* multicore performance from high-level stream programming languages. *Scalable* denotes that our techniques incrementally increase performance as processing cores are added; aiming to achieve a speedup over single core performance that is multiplicative of the number of cores. This dissertation develops effective and aggressive transformations that can be used to inform the design of future parallel programming systems. These transformations facilitate parallel mappings that yield order-of-magnitude performance improvements for multicore architectures. The techniques presented are demonstrated to be portable across multicore architecture with varying communication mechanisms. The impact of this dissertation is that for an important domain our techniques (i) substantiate the scaling mechanism of multicore architectures of using increasing transistor budgets to add cores, and (ii) unburden the programmer from having to match the parallelism and communication of an implementation to the target architecture.

1.1 Streaming Application Domain

Based on the examples cited previously, we have observed that stream programs share a number of characteristics. Taken together, they define our conception of the streaming application domain:

1. **Large streams of data.** Perhaps the most fundamental aspect of a stream program is that it operates on a large (or virtually infinite) sequence of data items, hereafter referred to as a *data stream*. Data streams generally enter the program from some external source, and each data item is processed for a limited time before being discarded. This is in contrast to scientific codes, which manipulate a fixed input set with a large degree of data reuse.

2. **Independent stream filters.** Conceptually, a streaming computation represents a sequence of transformations on the data streams in the program. We will refer to the basic unit of this transformation as a *filter*: an operation that – on each execution step – reads one or more items from an input stream, performs some computation, and writes one or more items to an output stream. Filters are generally independent and self-contained, without references to global variables or other filters. A stream program is the composition of filters into a *stream graph*, in which the outputs of some filters are connected to the inputs of others.

3. **A stable computation pattern.** The structure of the stream graph is generally constant during the steady-state operation of the program. That is, a certain set of filters are repeatedly applied in a regular, predictable order to produce an output stream that is a given function of the input stream.

4. **Sliding window computations.** Each value in a data stream is often inspected by consecutive execution steps of the same filter, a pattern referred to as a *sliding window*. Examples of sliding windows include FIR and IIR filters; moving averages and differences; error correcting codes; biosequence analysis; natural language processing; image processing (sharpen, blur, etc.); motion estimation; and network packet inspection.

5. **High performance expectations.** Often there are real-time constraints that must be satisfied by stream programs. For instance a minimum quality or frame rate has to be achieved. Thus, efficiency is of primary concern. Efficiency for the streaming domain is mainly in terms of throughput, i.e., application outputs per unit of time. Latency is also a concern for certain applications.

While our discussion thus far has emphasized the client context for streaming applications, the stream abstraction is equally important in server-based computing. Examples include XML processing [BCG+03], digital filmmaking, cell phone base stations, and hyperspectral imaging.

1.2 The StreamIt Project

The concept of a "stream of data" has a long and rich history in computer science. Streaming models of computation are often untied to an actual programming language [Kah74, LM87b, Hoa78], however many languages with the notion of a stream do exist [Inm88, AVW93, Arm07, MSA+85, AW77, GBBG86, CPHP87, HCRP91, BG92]; see [Ste97] for a review of the role of streams

in programming languages. The models of computation can generally be considered as graphs, where nodes represent units of computation and edges represent FIFO communication channels. The models differ in the regularity and determinism of the communication patterns, as well as the amount of buffering allowed on the channels. In addition to models of computation, streaming prototyping environments have been employed to simulate and validate the design of complex systems [BHLM91, EJL+03, LHG+89]. Many graph-level optimizations such as scheduling [BML95, BML96, BSL96, ZTB00, SGB06] and buffer management [ALP97, MB01, GGD02, MB04, GBS05] have been considered in the context of prototyping environments. While prototyping environments provide rich, high-level analysis and optimization of stream graphs, most models do not automatically generate efficient and deployable code. In practice most environments require the application to be re-written in a low-level language such as C. A robust and efficient end-to-end programming language environment incorporating streaming optimizations remains out of reach for most developers [Thi09].

Other key shortcomings of many streaming systems is that their models of computation are either (i) too general to be effectively analyzed by a compiler or (ii) contain communication and synchronization primitives that are not a good match to multicore architectures. Given the recent trends in both computer architectures and applications, we have the opportunity to create a language that exposes the regularity of streaming applications for programmer productivity and to enable aggressive optimizations. Offering a fully-automated and efficient end-to-end solution for multicore parallelism will attract programmers, and is a step towards maintaining the single-application performance growth trends that existed in the superscalar era. This is the impact we pursue in the StreamIt project.

StreamIt is a programming language and compiler for high-productivity and high-performance stream programming. StreamIt seeks to occupy the intersection of the application and architecture trends highlighted above. The principle goals of the StreamIt research project are:

1. To expose and exploit the inherent parallelism in stream programs on multicore architectures.
2. To automate domain-specific optimizations known to streaming application specialists.
3. To improve programmer productivity in the streaming domain.

This dissertation addresses the first goal, and we contend that the goal of high-performance is not in conflict with the goal of increasing programmability. Compared to previous effects, the key leverage of the StreamIt project is a compiler-conscious language design that maximizes analyzability without sacrificing programmability. StreamIt achieves this combination by focusing on a very common case in streaming applications – an unchanging stream-graph with statically determinable communication rates. This common case forms the boundary of optimization in the compiler, akin to a procedure in an imperative language and compilation system.

StreamIt is a large systems project, incorporating over 27 people (up to 12 at a given time). I led the parallelization efforts of the project including the work covered in this thesis. The analysis, transformation, scheduling, mapping and code generation infrastructure that I developed has been employed or extended to support many other research projects within StreamIt [STRA05, Won04, Tan09, Ser05, Won04, Zha07, CGT+05].

The StreamIt compiler (targeting shared-memory multicores, clusters of workstations, and the MIT Raw machine) is publicly available [stra] and has logged over 850 unique, registered downloads from 300 institutions. Researchers at other universities have used StreamIt as a basis for their

own work [NY04, Duc04, SLRBE05, JSuA05, And07, So07, CRA10, CRA09, HCW+10, ST09, UGT09a, NY03, UGT09b, HWBR09, HCK+09, HKM+08, KM08, CLC+09].

1.3 Multicore Architectures

Commodity multicore architecture designs began as an outlet for the excess transistors that are a consequence of Moore's Law. In the mid-2000s, general-purpose single-core (superscalar and VLIW) designs reached a wall in terms of power, design complexity, pipelining, and returns of instruction level parallelism (ILP). The first commodity, non-embedded multicore architecture appeared in 2001 with IBM's POWER4. In 2005, Intel and AMD ceased development of single-core desktop microprocessor solutions. For the foreseeable future, growing transistor budgets will translate into successively increasing the number of cores per die for multicore designs. The current generation of Intel microprocessors (Nehalam) includes a server processor with eight cores [int]. The Tilera Corporation is currently offering a 100-core design where each core is powerful enough to run the linux operating system [til].

Multicore designs come in various flavors. In *homogeneous* multicores, each processing core is identical; in *heterogeneous* multicores processing cores can vary. This dissertation is focused on homogeneous designs (or only targeting a set of identical cores in a heterogeneous design), as the techniques assume each core has identical processing power. Multicore designs can also differ in the mechanism that implements communication between cores and the view of memory that is offered to the threads of a single program. Commodity multicores from AMD and Intel implement shared memory via cache coherence; they are termed symmetric multiprocessors (SMP). Newer cache-coherence protocols optimize for streaming memory accesses between cores [Mac]. Distributed shared memory designs, such as Tilera's TILE64, allow software to control where blocks of shared memory reside.

In distributed memory designs each core has its own private address space and inter-core communication is initiated explicitly via software. Streaming is naturally expressed on distributed memory machines, as filters each have their own private address space. In these machines, various mechanisms implement the communication. Different communication mechanisms support communication-and-computation concurrency at varying levels. DMA architectures (e.g., IBM's Cell Processor [Hof05]) include separate logic at each core to implement data movement. The compute pipeline at each core initiates instructions that send from or receive to blocks of memory. These processors can achieve high-levels of computation-and-communication concurrency. Processors such as MIT's Raw [TLM+04] and Tilera's TILE64 offer small-to-large granularity transfers routed over a statically- or dynamically-routed network. Leveraging these mechanisms, the processor is responsible for injecting data to and pulling data from the network, thus communication-and-computation concurrency is limited as cores must rendezvous once hardware buffering is exhausted.

In this dissertation we present compiler techniques that enable scalable parallel performance for architectures that offer varying communication mechanisms: SMP architectures ; distributed shared memory architectures; and fine-grained, near-neighbor, statically-routed communication architectures. The following sections detail the three architectural targets employed to evaluate the techniques of this dissertation. [KM08] demonstrates that similar techniques are portable to the DMA architectures in the context of the IBM Cell processor.

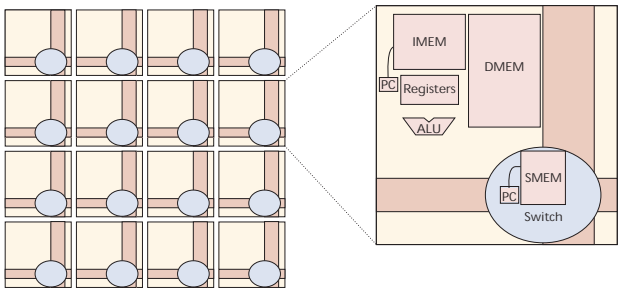


Figure 1-3: Block diagram of the Raw architecture.

1.3.1 Distributed Memory: The Raw Architecture

The Raw Microprocessor [T+02, WTS+97] addresses the wire delay problem [HMH01] by providing direct instruction set architecture (ISA) analogs to three underlying physical resources of the processor: gates, wires and pins. Because ISA primitives exist for these resources, a compiler such as StreamIt has direct control over both the computation and the communication of values between the functional units of the microprocessor, as well as across the pins of the processor.

The architecture exposes the gate resources as a scalable 2-D array of identical, programmable tiles, that are connected to their immediate neighbors by four on-chip networks. Each network is 32-bit, full-duplex, flow-controlled and point-to-point. On the edges of the array, these networks are connected via logical channels [GO98] to the pins. Thus, values routed through the networks off of the side of the array appear on the pins, and values placed on the pins by external devices (for example, wide-word A/Ds, DRAMS, video streams and PCI-X buses) will appear on the networks.

Each of the tiles contains a compute processor, some memory and two types of routers–one static, one dynamic–that control the flow of data over the networks as well as into the compute processor (see Figure 1-3). The compute processor interfaces to the network through a bypassed, register-mapped interface [T+02] that allows instructions to use the networks and the register files interchangeably. In other words, a single instruction can read up to two values from the networks, compute on them, and send the result out onto the networks, with no penalty. Reads and writes in this fashion are blocking and flow-controlled, which allows for the computation to remain unperturbed by unpredictable timing variations such as cache misses and interrupts.

Each tile's static router has a virtualized instruction memory to control the crossbars of the two static networks. Collectively, the static routers can reconfigure the communication pattern across these networks every cycle. The instruction set of the static router is encoded as a 64-bit VLIW word that includes basic instructions (conditional branch with/without decrement, move, and nop) that operate on values from the network or from the local 4-element register file. Each instruction also has 13 fields that specify the connections between each output of the two crossbars and the network input FIFOs, which store values that have arrived from neighboring tiles or the local compute processor. The input and output possibilities for each crossbar are: north, east, south, west, compute processor, to the other crossbar, and into the static router. The FIFOs are typically four or eight elements large.

To route a word from one tile to another, the compiler inserts a route instruction on every intermediate static router [LBF+98]. Because the routers are pipelined and compile-time scheduled, they can deliver a value from the ALU of one tile to the ALU of a neighboring tile in 3 cycles, or more generally, 2+N cycles for an inter-tile distance of N hops.

The results of this paper were generated using btl, a cycle-accurate simulator that models arrays of Raw tiles identical to those in the .15 micron 16-tile Raw prototype ASIC chip. With a target clock rate of 450 MHz, the tile employs as compute processor an 8-stage, single issue, in-order MIPS-style pipeline that has a 32 KB data cache, 32 KB of instruction memory, and 64 KB of static router memory. All functional units except the floating point and integer dividers are fully pipelined. The mispredict penalty of the static branch predictor is three cycles, as is the load latency. The compute processor's pipelined single-precision FPU operations have a latency of 4 cycles, and the integer multiplier has a latency of 2 cycles.

1.3.2 Distributed Shared Memory: Tilera TILE64

The Tilera Corporation's TILE64 Processor is a 64 core system on a chip [WGH+07]. Each core is an identical three-wide VLIW capable of running SMP Linux. In addition to standard RISC instructions, all cores support a SIMD instruction set designed to accelerate video, image and digital signal processing applications. Each core has 64KB L2 cache, and L2 caches can be shared among cores to provide an effective 4MB of shared L3 cache. Cores are connected through five low-latency, two-dimensional mesh interconnects. Two of these networks carry user data, while the other three handle memory and I/O. The combination of mechanisms allows the TILE64 Processor to support both the shared memory and message passing programming paradigms. The code generated by the StreamIt compiler for the TILE64 processor follows the remote store programming (RSP) model [HWA10] in which each process has a private address space, but each process can award remote processes write access to their local memory. When a producer process has write access to a consumer process's memory, the producer communicates directly with the consumer via store instructions whose destination is an address in the consumer's shared memory. Communication is initiated by the producer, and is fine-grained. The consumer reads directly from it's local memory (L2) when accessing input.

1.3.3 SMP: Xeon 16 Core

Our symmetric multiprocessor target is a 16-core architecture that is comprised of four Intel Xeon E7350 multicore processors. Each processor is a 64-bit, quad-core with two dual-core dies. Each die contains a 4 MB L2 cache shared across the two cores. The front-side bus is clocked at 1066 MHz. We utilize the cache coherency mechanism of the architecture for communication between cores.

1.4 Scheduling of Coarse-Grained Dataflow for Parallel Architectures

Computer architectures with multiple central processing units have been with us since the 1960s (one early example is the 4 processor Burroughs D825 introduced in 1962). Shortly after these machines were introduced, researchers began work on execution models for describing and analyzing concurrent systems [Pet62, KM66]. Many of these models represent programs as graphs of computations nodes with edges denoting communication between nodes (e.g., Kahn Process Networks [Kah74], Synchronous Dataflow [LM87b], Actors [HBS73, Gre75, Cli81, Agh85], Petri nets [Pet62, Mur89], Computation graphs [KM66], and Communicating Sequential Processes [Hoa78]). In order to realize the potential of parallel execution of a program, nodes of

the graph must be allocated to processors such that parallelism is achieved between concurrently running nodes, and communication does not overwhelm the benefits of parallel execution. The objective of *scheduling* (also called mapping), is to minimize the completion time of a parallel program by effectively allocating nodes to processors. Chapters 5, 6, and 7 of this dissertation describe scheduling techniques for allocating filters of a stream graph to cores of a multicore architecture. The problem that we solve falls within the realm of classical multiprocessor scheduling. This general scheduling problem has a large body of research dating back 50 years (see [KA99b] for a review). However, the properties of our application domain combined with the properties of multicore architectures allow us to devise novel, aggressive, and effective solutions to our problem.

Broadly, solutions to the scheduling problem fall under two categories: *static* and *dynamic*. In static scheduling, many characteristics of the parallel program are known before execution, and the program is represented as a graph where node weights represent task processing times and edge weights represent data dependencies as well as communication costs between nodes. Before execution, a compiler calculates an assignment of processors to nodes, and an ordering of node execution. In dynamic scheduling, scheduling decisions are made at execution time by a runtime system. Dynamic scheduling includes not only minimizing the programs completion time, but also minimizing the overhead of dynamic scheduling calculations. The techniques presented in this dissertation are all static scheduling techniques.

Static scheduling is NP-complete for most forms except for a few simplified cases. Our problem includes nodes with non-uniform weights and a bounded number of processors. This version of the problem has been shown to be NP-complete [GJ79]. Furthermore, for our problem, we assume an arbitrary graph structure, arbitrary communication and computation costs. Past solutions to this version of the static scheduling problem have taken a heuristic approach that is based on list scheduling [PLW96, CSM93, CR92, ERAL95, MG94]. Generally, these solutions order the nodes according to some heuristic, and then greedily assign nodes to processors. Nodes are duplicated and co-located with consumers to minimize communication costs. Nodes that are duplicated compute on the *same* data, so this is not a form of data parallelism. Parallelism is limited to task parallelism between nodes that do not have a dependence relationship.

Our execution model differs from the models of previous solutions. The nodes of our graphs execute indefinitely conceptually (and in practice nodes execute for many iterations). Our goal is no longer to minimize completion time of the program, but to maximize throughput (outputs per cycle) of the execution. This allows our scheduler to leverage data parallelism and pipeline parallelism in addition to task parallelism. Data parallelism is exploited by duplicating nodes in a SPMD fashion. Pipeline parallelism is exploited by mapping a producer node and a consumer node to distinct cores, and adding buffering to allow the producer to get ahead of the consumer, thus allowing the producer to execute in parallel with the consumer. This ability to introduce parallelism (at the expense of latency) adds considerable flexibility and complexity over the classical scheduling problem. We are no longer just assigning nodes to processors, we also have to calculate the appropriate type and amount of parallelism to exploit.

We mitigate the complexity of our scheduling problems (as compared to classical static scheduling) by leveraging the inherent structure of stream programs. As we will demonstrate in Chapter 4, stream programs are described by graphs with simple producer and consumer relationships. A node rarely has multiple paths to entry or exit with varying computation or communication cost. Task parallel nodes typically execute the same computation (though with different parameters) and have the same ancestors and successors in the graph. For example, in Figure 1-2, notice the

structure of the graph; task parallel nodes (e.g., the LowPass filters) have similar ancestors and successors. This structure allows us to calculate task and data parallelism on task parallel slices of the graph (horizontal slices of the graph), creating slices with balanced computation within a slice, and time-multiplexing execution between slices (see Chapter 6).

The accuracy of static computation cost estimation (or lack thereof) also informs the design of our techniques. Unlike classical scheduling, we try to minimize our reliance on static computation estimation by favoring data parallelism over task and pipeline parallelism. When we exploit data parallelism, the resulting duplicated nodes execute the same code in parallel on different data. In the streaming domain, computation is rarely data dependent, thus the duplicates are load-balanced. We exploit task parallelism between different nodes only to reduce the span of exploited data parallelism, and it tends to be the case in our domain that task parallel nodes perform the same computation. Pipeline parallelism is calculated via a traditional list scheduling (bin-packing) approach, though it is only leveraged when we cannot apply data parallelism to a node.

Finally, multicore architectures enable our exploitation of task, data, and pipeline parallelism. Communication between cores of a multicore is very fast as compared to multiprocessors. Thus, the threshold for computation versus communication where parallelism is profitable is lower for multicores as compared to multiprocessors of the past. Even with this lower threshold we cannot ignore communication costs when targeting multicores. Our data parallelization techniques must reduce communication in various ways to achieve profitability.

1.4.1 The Space-Multiplexing and Time-Multiplexing Continuum

Chapters 5, 6, and 7 present our techniques for solving the scheduling problem introduced above. Generally, each chapter presents techniques that:

1. Transform a stream graph via exploiting coarse-grained task, data, and/or pipeline parallelism

2. Map the filters of the transformed stream graph to cores

3. Schedule interleaving of filters on each core

4. Generate computation and communication code to realize the mapping

Chapters are titled by the covered technique's position on the *time-multiplexing* and *space-multiplexing* continuum. For a pure *time-multiplexing* strategy, each filter is duplicated to all the cores and the stages of the application are "swapped" in and out of the processor. A time-multiplexing mapping does not take advantage of any pipeline or task parallelism. For a pure *space-multiplexing* strategy, all the filters of the application execute concurrently on the processor. Positioning the techniques on the space/time-multiplexing continuum is not without contradictions, but it is helps one to remember the type of parallelism that is *primarily* exploited by the technique.

Chapter 5 covers a space-multiplexing approach to solving the scheduling problem: *hardware pipelining and task parallelism*. Filters of the original stream graph are combined into larger filters in a *partitioning* step. Each of the filters of the partitioned graph is assigned to its own core. The mapping exploits pipeline and task parallelism. This mapping sits at the space-multiplexing end of the space/time multiplexing continuum. Hardware pipelining stands alone and does not interact

nor complement other techniques. The technique achieved modest parallelization speedup results and inspired better solutions.

Chapter 6 covers a set of techniques that sit very near the time-multiplexing end of the space/time-multiplexing continuum: *coarse-grained data and task parallelism*. The technique primarily leverages data parallelism in a time-multiplexed fashion. However, task parallelism is included in the mapping when it can reduce the inter-core communication requirement of the mapping. Task parallelism is harnessed via space-multiplexing. We describe the techniques as "time-multiplexing" because, again, most of the parallelism that exists in the transformed graph is data parallelism, and task parallelism is included only to reduce the communication requirements, not as means of load-balancing. Coarse-grained data and task parallelism is very effective, but it is not robust for all applications. The strategy is complemented by the techniques of Chapter 7.

Chapter 7 presents a strategy that complements data and task parallelism to parallelize filters that cannot be data-parallelized: *coarse-grained software pipelining*. Coarse-grained software pipelining, on its own, exploits pipeline and task parallelism in a space-multiplexed fashion. However, we combine the technique with coarse-grained data and task parallelism to incorporate all three forms of parallelism in the mapping. The combination of techniques sits somewhere in the middle of the time/space continuum, and Chapter 7 presents the results of the combined techniques.

Chapter 8 cover a multiple-stage transformation that creates data parallelism: *optimized fission of peeking filters*. The transformation is guided by the coarse-grained data and task parallelism mapping strategy of Chapter 6. This transformation does not solve the scheduling problem directly, but is an optimized means of creating data parallelism from a filter that computes on overlapping sliding-windows of its input. The results in Chapter 8 first use the coarse-grained data and task parallelism to transform the stream graph via optimized fission of peeking filters, then the transformed graph is then scheduled with the coarse-grained software pipelining strategy.

1.5 Contributions

The specific contributions of this dissertation are as follows:

1. The first fully automatic compilation system for managing coarse-grained task, data, and pipeline parallelism in stream programs.

2. We enable an abstract architecture-independent parallel description of a stream algorithm to be matched to the granularity of the multicore target through a transformation that leverages filter fusion.

3. We solve the problem of load-balancing, while limiting the synchronization of a mapping, by applying data parallelism at appropriate amounts such that task parallel units span all cores.

4. To parallelize filters that include state and thus cannot be data-parallelized, we pioneer software-pipelining scheduling techniques to exploit pipeline parallelism between these filters.

5. For filters that compute on overlapping sliding widows of their input, we enable efficient data-parallelization by converting the state of the sliding window into minimal inter-core communication.

6. Across the 12 benchmarks in our suite, for a 16-core distributed memory multicore, we achieve a 14.9x mean speedup (standard deviation 1.8). For benchmarks that include sliding-window computation, our sliding-window data-parallelization techniques are required to enable scalable performance for a 16-core SMP multicore (14x mean speedup) and a 64-core distributed shared memory multicore (52x mean speedup).

1.6 Roadmap and Significant Results

The overall organization of this dissertation is as follows:

1. **An experience report regarding the characteristics of the StreamIt language from the perspective of exploiting coarse-grained parallelism in the compiler (Chapter 2).** The StreamIt programming model includes novel constructs that simultaneously improve programmability and analyzability of stream programs. We qualitatively evaluate the key language idioms that enable effective parallel mappings to multicore architectures, and relate trends in applications that informed our techniques.

2. **An analytical model of an important class of stream programs targeting multicore architectures (Chapter 3).** The analytical model considers data parallel streaming programs, comparing the scalability prospects of two mapping strategies: time-multiplexed data parallelism and space-multiplexed data parallel. Analytical throughput equations are developed for each, extracting and comparing the key differences in the models in terms of load-balancing effectiveness, communication, and memory requirement. Evidence is offered that shows pipeline parallelism will become increasingly important as multicore continue to scale by adding cores. We develop asymptotic communication bounds for the two strategies given a low-latency, near-neighbor mesh network, concluding that the strategies are asymptotically equal.

3. **A detailed description and analysis of the StreamIt Core benchmark suite (Chapter 4).** Since we first introduced the suite in 2006 [GTA06], the StreamIt Core benchmark suite has gained acceptance as a standard experimentation subject for research in streaming programs [KM08, UGT09b, HWBR09, HCK+09, NY03, UGT09a, ST09, HCW+10, CRA09, CRA10]. We provide a detailed analysis of the suite of 12 programs from the perspectives of composition, computation, communication, load-balancing, and parallelism.

4. **A review of the hardware-synchronized space-multiplexing approach to mapping stream programs including updated performance results and detailed analysis (Chapter 5).** In previous work, we introduced a fully-automatic compilation path for hardware-synchronized space-multiplexing [GTK+02]. This chapter provides updated results for these techniques. Furthermore, the chapter includes thorough performance analysis for multiple benchmarks. The chapter analyzes the positive and negative characteristics of this approach, and when it is most applicable. Finally, we include a retrospective analysis of the hardware support of the Raw microprocessor utilized for synchronization.

5. **The first fully-automated infrastructure for exploiting data and task parallelism in streaming programs that is informed by the unique characteristics of the streaming domain**

and multicore architectures (Chapter 6). Mountains of previous work has covered data-parallelization of coarse-grained dataflow graphs. The work in this chapter differs for previous work in that it automatically matches the granularity of the stream graph to the target without obscuring data-parallelism. The programmer is not required to think about parallelism nor provide any additional information to enable parallelization. When data-parallelism is exploited, the task parallelism of the graph is retained so that the span of the introduced data parallelism can be minimized. This serves to reduce communication and synchronization. These techniques provide a 12.2x mean speedup on a 16-core machine for the StreamIt Core benchmark suite.[1]

6. **The first fully-automated framework for exploiting coarse-grained software pipelining in stream programs (Chapter 7).** Data parallel techniques cannot explicitly parallelize filters with iteration-carried state. Leveraging the outer scheduling loop of the stream program, we apply software pipelining techniques at a coarse granularity to schedule stateful filters in parallel. Our techniques partition the graph to minimize the critical path workload of a core, while reducing inter-core communication. For the stateful benchmarks of the StreamIt benchmark suite targeting Raw, adding coarse-grained software pipelining to data and task parallel techniques improves throughput by 3.2x. The combined techniques intelligently exploit task, data, and pipeline parallelism. Across the entire StreamIt Core benchmarks suite, the mean 16-core throughput speedup over single core for the Raw processor is 14.9x. The standard deviation is 1.8.

7. **A novel framework for reducing the inter-core communication requirement for data-parallelizing filters with sliding window operations (Chapter 8).** Filters that operate on a sliding window of their input must remember input items between iterations. Data-parallelization of these filters requires sharing of input across data-parallel units. We introduce new techniques that reduce the sharing requirement by altering the steady-state schedule and precisely routing communication. For the four benchmarks of the suite that include sliding window computation, the new techniques achieve a 5.5x speedup over previous techniques for the Xeon system, and 1.9x over previous techniques for the 64-core Tile64. Across the entire StreamIt Core benchmark suite, the combination of our mapping and parallelization techniques achieves 14.0x mean speedup for the Xeon system and 52x mean speedup for the TILE64.

Related work and future work are given on a per-chapter basis. We conclude in Chapter 9.

[1] None of the architectures that are targeted in this dissertation include computation and communication concurrency. Thus, our touchstone cannot be a precisely linear speedup.

Chapter 2

Execution Model and the StreamIt Programming Language

The chapter covers the execution model of streaming computation that is employed in this dissertation. The StreamIt programming language is an example of a language that can be mostly represented by the execution model. An overview of the StreamIt programming language is given, with emphasis on parallelization considerations. For more details on the StreamIt language, please consult the StreamIt language specification [strb] or the StreamIt cookbook [Strc].

2.1 Execution Model

The semantics of the model of computation employed in this dissertation are built upon synchronous dataflow (SDF) [LM87a]. Computation is described by composing independent processing nodes into networks. The processing nodes, termed *actors* or *filters*, communicate via uni-directional FIFO data channels. Each filter defines an atomic execution step that is performed repeatedly. The term *synchronous* denotes that a filter will not fire unless all of its input are available. Synchronous dataflow requires that the number of items produced and consumed by a filter during each atomic execution step can be determined statically. This allows the compiler to perform static scheduling of and transformations on the stream graph.

Our model expands upon synchronous dataflow by including additional features that either ease the programmer's burden of expressing the application or ease the compiler's burden in efficiently mapping the program:

1. **Peeking.** Our model explicitly represents filters that do not consume all items read from their input channel. Two separate rate declarations describe a filter's input behavior for each firing: the number of items read and the number of items consumed. The first quantity is termed the *peek* rate and the second quantity is termed the *pop* rate (the peek rate is always greater than or equal to the pop rate). A *peeking* filter is a filter that has a peek rate greater than its pop rate. Peeking is important because it exposes sliding window computations to the compiler. Without peeking, a programmer would have to manually create state to implement the sliding window.

2. **Communication during initialization.** A filter in our model can define an optional initialization step that inputs and/or outputs a statically determinable number of items. This step is

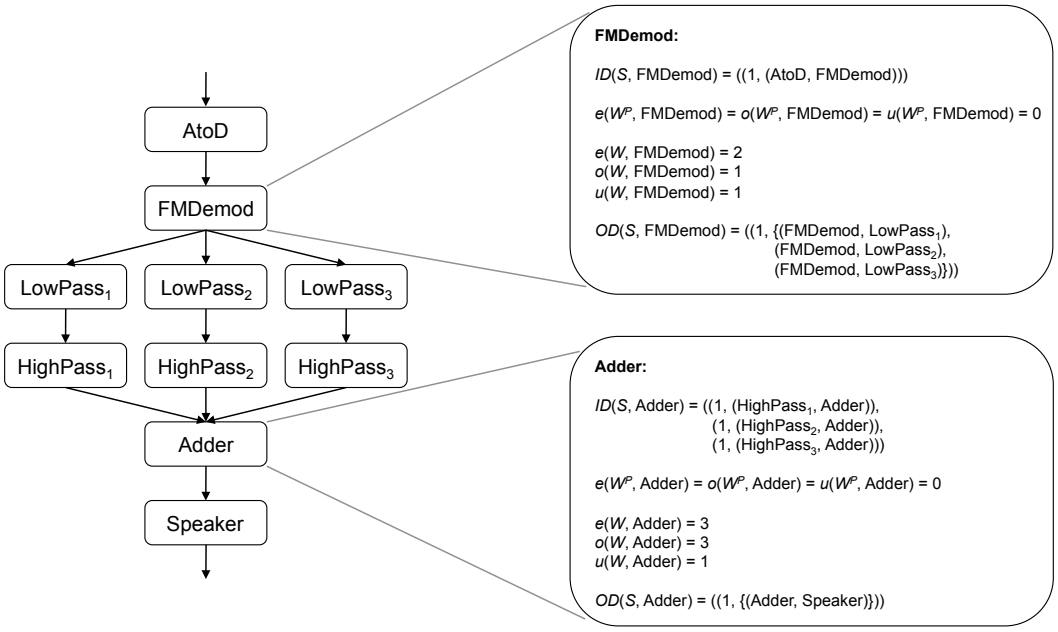


Figure 2-1: On the left is a stream graph that represents an FM radio with a 3-band equalizer. On the right, the details of two filters are given using the notation introduced in this chapter.

defined in a *prework* function. Separate rate declarations are specified for the prework function as opposed to the steady-state behavior.

3. **Filters specify the order in which they read data from input channels, and filters specify the output channels to which each output is forwarded.** This aspect of the model of computation matches cyclo-static dataflow [BELP95, PPL95]. Conceptually, our model includes designated nodes that support reordering and duplication. These nodes can have phases and are allowed to copy an item from an input channel to one or more output channels. In our model, these nodes are folded into the definition of a filter. Each filter includes a node that describes the input reordering and a node that describes the output reordering and duplication.

The model we use for this thesis is a general model that is agnostic of input language. Although the model was inspired by the StreamIt programming language, the model can represent aspects of other streaming programming languages such as Brook [BFH+04], StreamC/KernelC [KRD+03], and SPUR [ZLSL05]. Consider a directed graph $G = (V, E)$ corresponding to a streaming application. $F \in V$ is a filter in the application and $(f, g) \in E$ is an edge in the graph that denotes communication from f to g using a FIFO channel. Edges are also termed *channels*. Note that there can exist multiple edges between 2 filters. Figure 2-1 gives an example stream graph for a FM radio with an equalizer. The figure also includes details for two of the filters; the details are explained below.

The filter is the basic unit of execution in our model. Each filter is described by multiple rate declarations, data reorganization patterns, and compute functions. Basically, each filter defines the number of items it consumers and produces each time it is executed. The execution of a stream graph is represented by a *schedule* of a graph G. A schedule gives a multiplicity for each filter

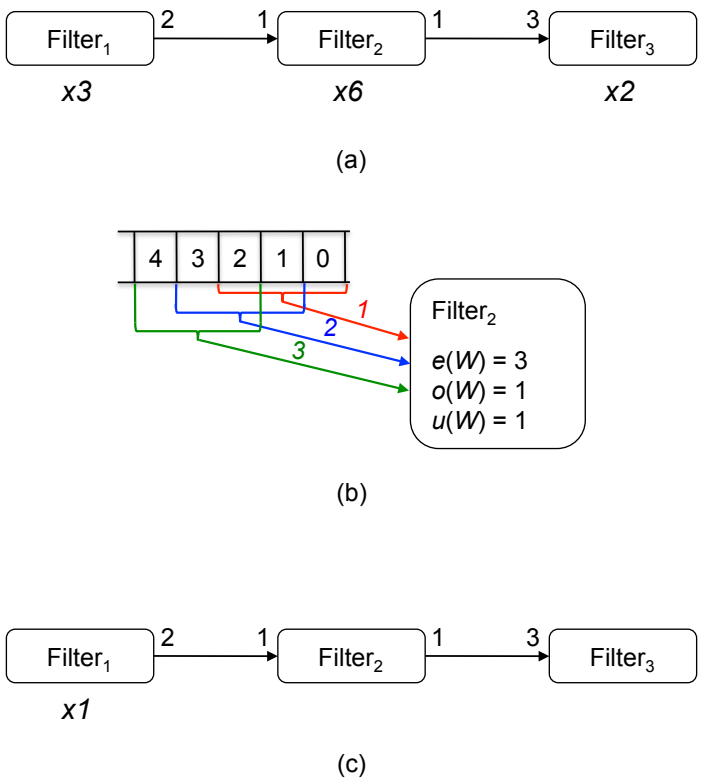


Figure 2-2: A pipeline of three filters. In the figure, the number to the left of a filter is the number of items the filter consumes per firing; the number to the right is the number of items produced per firing. (a) gives a legal steady-state schedule for the three filters. (b) gives more detail for *Filter*$_2$ including its peek rate. The figure shows the window of items read for 3 consecutive firings of the filter. Notice the overlapping in the windows. (c) gives a legal initialization schedule for the graph that enables *Filter*$_2$'s peeking.

$F \in V$ that denotes how many times to fire filter F. When a schedule is executed, each filter fires when it has buffered enough input to satisfy its input requirement (a filter will not fire more times than given by a schedule). In our notation, a schedule is represented by the variable Σ, and it denotes a mapping from filters to non-negative integers. The multiplicity of filter F in schedule Σ is denoted by $M(\Sigma, F)$.

In the streaming domain, input items continuously enter the application, with application output continuously produced as items flow from producer filters to consumer filters. To model this behavior statically, a schedule can be calculated such that all filters fire in the schedule, and the schedule can be repeated indefinitely. Since our model of execution adheres to SDF, a *steady-state* schedule, S, of a graph can be statically calculated such that the quantity of items on each of the filter's input buffer and output buffer remains unchanged by the complete execution of the schedule [LM87b]. Thus this schedule can be repeated indefinitely because buffers to not grow or shrink in size. Steady-state execution of the graph entails repeating the steady-state schedule for as much input as is expected. Due to this, execution of the stream graph is conceptually wrapped in an outer loop that continuously executes the steady-state schedule. Figure 2-2(a) gives an example of a steady-state schedule calculated for a graph that consists of three filters in a pipeline.

Our model does not deal with arbitrary schedules of filter executions. The steady-state schedule, S, is explicitly represented. Furthermore, an *initialization* schedule, I, is explicitly represented that enables the steady-state schedule in the presence of peeking. Thus, in our model $\Sigma \in \{I, S\}$. More details on the initialization schedule are given below.

For each filter, $F \in V$, a *work* function, W_F, is defined. The work function defines the atomic execution step for each filter. When a filter fires, its work function is called once, consuming the items and producing the items statically defined. For each filter, $F \in V$, a *prework* function, W_F^P, can also be defined. Prework describes a computation step that executes once at the first firing of a filter. The prework function is executed only once per execution of the *application*; it describes any special initialization behavior required for a filter. Many filters do not require a prework function that differs from the work function. If a filter does not define a prework function, the work function is called on the first execution of the filter. We denote a function with the variable $\mathcal{F} \in \{W^p, W\}$. We sometimes leave out the subscript that denotes the filter if it is clear which work or prework function is intended.

For each filter $F \in V$ we define the following:

- $o(\mathcal{F}, F)$, the number of items dequeued from F's input buffer per firing of $\mathcal{F}$. This quantity is termed the *pop* rate.
- $e(\mathcal{F}, F)$, 1 + the greatest index that is read (but not necessarily dequeued) from F's input buffer per firing of $\mathcal{F}$. This quantity is termed the *peek* rate.
- $u(\mathcal{F}, F)$, the number of items enqueued to F's output buffer per firing of $\mathcal{F}$. This quantity is termed the *push* rate.

If F does not define a prework function, $e(W_F^P, F) = o(W_F^P, F) = u(W_F^P, F) = 0$. A filter with peek rate greater than pop rate is termed a *peeking* filter. Figure 2-2(b) demonstrates a peeking filter. The figures shows that the window of items read for consecutive firings of the filter overlap. For the work function execution, a peeking filter F requires $e(W_F, F)$ items to be on its input channel(s) to fire. F will consume (dequeue) the first $o(W_F, F)$ items after the firing of the work function.

An initialization schedule is required if peeking is present in a graph to enable the calculation and execution of a steady-state schedule [Kar02]. After the initialization schedule executes, each filter F is guaranteed to have at least $e(W_F, F) - o(W_F, F)$ items in its input buffer. The initialization schedule is required to calculate a steady-state schedule for a graph with an F such that $e(W_F, F) - o(W_F, F) > 0$ (if the prework function peeks then an initialization schedule is also required) [KTA03]. Figure 2-2(c) gives a legal steady-state schedule for the pipeline that enables the peeking in filter *Filter*$_2$. During application execution, the initialization schedule is executed once followed by an infinite repetition of the steady-state schedule. Figure 2-3 shows how the execution of the initialization and steady-state schedules unfold for the pipeline example of Figure 2-2. The number of items remaining on F's input channel(s) after execution of the initialization schedule is represented by $C(F)$. In the case of Figure 2-3, $C(\textit{Filter}_2) = 2$ and $C(\textit{Filter}_1) = C(\textit{Filter}_3) = 0$.

A filter may have multiple incoming edges and/or multiple outgoing edges. Let $\text{OUT}(F)$ represent the set of output edges of F, and let $\text{IN}(F)$ represent the set of input edges of F. Filter inputs are organized into a single internal FIFO buffer for the filter to read according to an *input distribution pattern*, and filter outputs are distributed from a single internal output FIFO buffer according to an *output distribution pattern*. The input distribution pattern is represented as:

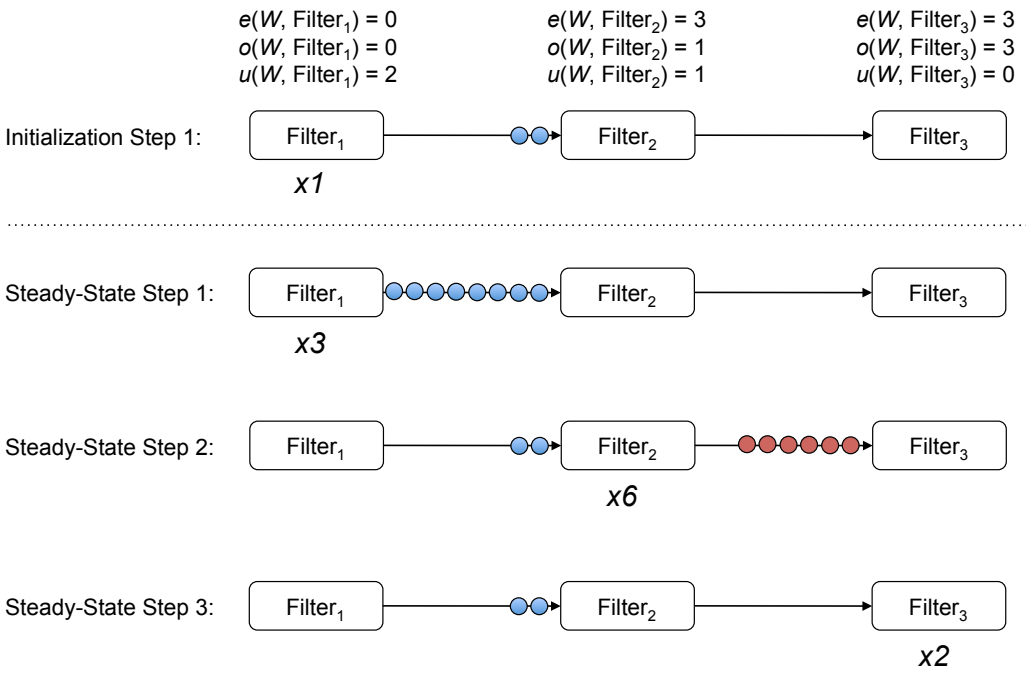


Figure 2-3: Timeline of execution for the pipeline example of Figure 2-2. The rates for the filters are given at the top (no prework functions are defined). The initialization stage ensures that Filter_2 has at least 2 items ($e(W, \mathit{Filter}_2) - o(W, \mathit{Filter}_2)$) buffered in its input edge. The steady-state steps then fire each filter in dataflow order. Each filter fires the number of times prescribed by the steady-state schedule in Figure 2-2(a). Notice that the number of items remaining on each buffer is the same before and after the steady-state schedule completely executes.

$$\mathit{ID}(\Sigma, F) \in (\mathbb{N} \times E)^n = ((w_1, e_1), (w_2, e_2), ..., (w_n, e_n))$$

Where n is the width of the input distribution pattern. The input distribution describes the round robin joining pattern for organizing the input data into the filter's single internal FIFO buffer, where w_i items are received from edge e_i before proceeded to the next edge, e_{i+i}.

The output distribution pattern describes both round-robin splitting and duplication in a single structure:

$$\mathit{OD}(\Sigma, F) \in (\mathbb{N} \times (P(E) - \emptyset))^m = ((w_1, d_1), (w_2, d_2), ..., (w_n, d_n))$$

Each d_i is called the *dupset* of weight i. The dupset d_i specifies that w_i items be duplicated to the edges of the dupset. Each tuple denotes that w_i output items of F are duplicated to the edges of d_i before moving on to the next tuple.

For both the *OD* and the *ID*, the variable Σ specifies the schedule which the distribution pattern describes, either I for initialization, or S for steady-state. The input and output distributions are repeated as needed for the schedule that is being executed. The start of each steady-state iteration resets the input and output distributions to the first tuple. Figure 2-4 highlights an example of a filter with both multiple inputs and multiple outputs, and how the distribution patterns translate into scattering and gathering.

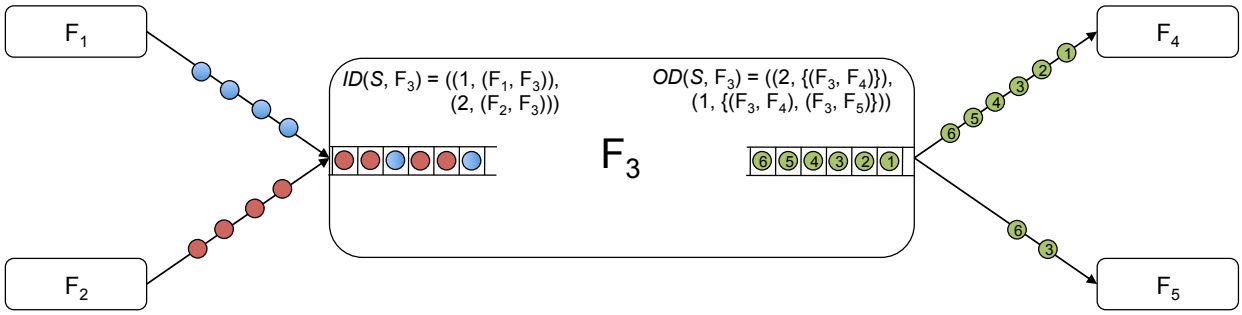


Figure 2-4: An example of a filter with non-trivial input and output distribution patterns. Filter F_3 has both multiple inputs and multiple outputs. The input items from the multiple inputs are joined in the pattern described by its input distribution (ID): receive 1 item from F_1 and 2 items from F_2, and repeat. F_3's output items are distributed to its multiple outputs as described by the output distribution pattern (OD): 2 items to F_4, then 1 item is duplicated to both F_4 and F_5. The output items of F_3 are indexed to show how they are distributed to F_4 and F_5.

Let $\textit{RO}(F_1, F_2, \Sigma)$ be the ratio of output items F_1 splits along the edge (F_1, F_2) to the total number of items that F_1 produces in the schedule Σ. This can be calculated from F_1's output distribution pattern for Σ. For example, in Figure 2-4, $\textit{RO}(F_3, F_4) = 1$ and $\textit{RI}(F_3, F_5) = \frac{1}{3}$. Conversely, let $\textit{RI}(F_1, F_2, \Sigma)$ be the percentage of total input items that F_2 receives from F_1 for Σ. For example, in Figure 2-4, $\textit{RI}(F_2, F_3) = \frac{2}{3}$.

Let $s(\mathcal{F}, F)$ denote the execution time (in cycles, with respect to a given real or conceptual machine) of function $\mathcal{F}$ of filter F. For many of the static scheduling techniques covered in this thesis, we calculate a static estimation of the total amount of work for a filter F in the steady-state, $s(W, F) \cdot M(S, F)$. We often use the term *work estimation* to denote this quantity.

We define one final property of a filter. For filter F, given that there is sufficient input data and output data will be ordered properly, if it is required to execute firing i of F before firing $i + 1$, then F is termed *stateful*. If firing i of F can be executed before (or in parallel with) $i + 1$, then F is termed a *stateless* filters. This section does not define how the computation of the work and prework functions is specified. In the next section we will see that for StreamIt, a work function is imperative code that can include (field) variables live across firings of the filter. In this case, a filter F is stateless if F does not write to a field variable that is read during a subsequent firing. This will create a loop-carried dependence and prevent data-parallelization.

In general, if all of the quantities defined above can be statically determined for graph $F \in G$, then we say G is *static*. Static graphs match closely the original formulation of synchronous dataflow (with the exception of peeking and prework). The scheduling, partitioning, and mapping techniques presented in this thesis are enabled by and operate on static graphs.

Appendix A includes a single-page guide to our notation.

2.2 The StreamIt Programming Language

One example of a high-level programming language that can be mostly expressed via the execution model of the previous section is the StreamIt programming language. We say mostly, because our execution model currently does not represent the StreamIt feature of *teleport mes-*

```
float->float filter FIR(int N) {
  float[N] weights;

  init {
    for (int i=0; i<N; i++) {
      weights[i] = calcWeight(i, N);
    }
  }

  work push 1 pop 1 peek N {
    float sum = 0;
    for (int i=0; i<N; i++) {
      sum += weights[i] * peek(i);
    }
    push(sum);
    pop();
  }
}
```

(a)

```
void init_FIR(float* weights, int N) {
  int i;

  for (i=0; i<N; i++) {
    weights[i] = calc_weight(i, N);
  }
}

void do_FIR(float* weights, int N,
            int* src, int* dest,
            int* srcIndex, int* destIndex,
            int srcBufferSize, int destBufferSize) {

  float sum = 0.0;
  for (int i = 0; i < N; i++) {
    sum += weights[i] *
           src[(*srcIndex + i) % srcBufferSize];
  }
  dest[*destIndex] = sum;
  *srcIndex = (*srcIndex + 1) % srcBufferSize;
  *destIndex = (*destIndex + 1) % destBufferSize;
}
```

(b)

Figure 2-5: Two implementations of an FIR filter: (a) the StreamIt version; and (b) the C version.

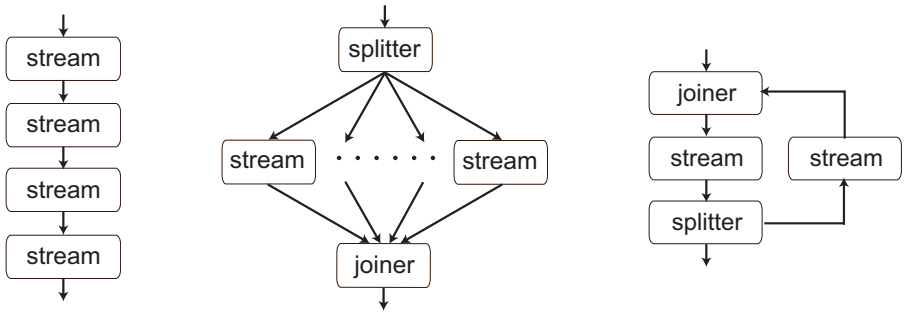
saging [TKS+05]. This section presents the StreamIt programming language, and discusses the mapping from StreamIt to the execution model of the previous sections.

2.2.1 StreamIt Filters

In StreamIt, the basic unit of computation is called a *filter*. The filter represents a programmer-defined computational node with (at most) a single input channel and a single output channel. Each filter has its own private address space and private program counter. Communication between filters is achieved via operations on the channels connecting filters. A StreamIt filter corresponds almost directly a filter in the execution model of the previous section (though the execution model's filters can support input that is received from multiple sources and output that is forwarded to multiple destinations).

Figure 2-5(a) gives an example of a filter. This filter performs a sliding window, multiply and accumulate operation. The computation is parametrized by the window length (taps) N. At compilation time, any and all parameters passed to each filter is resolved, and the *init* function of each filter is called. In the case of the FIR filter, the parameter N is resolved, and the init function initializes the weights of the FIR (the impulse response). The weights are stored in an array private to each FIR filter. Just as with the execution model, StreamIt defines two stages of execution for an application: initialization and steady state. During the steady-state execution of the filter, the work function is called repeatedly (as part of a larger steady-state schedule).

Within the work function, filters communicate via operations on their input and output FIFO channels: `push(val)` enqueues `val` onto the output channel, `pop()` dequeues and returns an item



(a) A pipeline. (b) A splitjoin. (c) A feedbackloop.

Figure 2-6: Hierarchical stream structures supported by StreamIt.

```
float -> float pipeline Main() {
  add Source();  // code for Source not shown
  add FIR();
  add Output();  // code for Output not shown
}
```

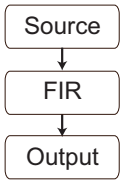


Figure 2-7: Example pipeline with FIR filter.

from the input channel, or `peek(index)` return, do not dequeue, the item at a given index on the input channel. Filters can also declare a separate *prework* function that is executed once for the first filter execution of the application.

Prework and work definitions require the programmer to define push, pop, and peek rates. Just as with the execution model, the push rate defines the number of items that are enqueued onto the output channel for each call to work or prework. The pop rate defines the number of items dequeued from the input channel. The peek rate defines the maximum number of items that may be inspected on the input channel. Note that the peek rate is always greater than or equal to the pop rate. In order for the compiler to statically schedule the graph, all rate expressions must be resolvable to integer constants at compile time.

2.2.2 Building Stream Graphs in StreamIt

One of the novel ideas introduced in the StreamIt language is to enforce a structured programming model when building stream graphs of filters. StreamIt does not allow the programmer to connect filters in an arbitrary manner. The language specifies three hierarchical structures for building larger graphs from smaller structures. Figure 2-6 illustrates the three basic stream structures: a pipeline, a splitjoin, and a feedbackloop. Each structure has a single input channel and a single output channel so that they can be composed hierarchically and interchanged easily. We use the term *stream* to refer to any of the single input, single output structures including a filter.

The pipeline construct introduces a sequential composition of its child streams. The child streams are specified via successive calls to `add` from within the pipeline. Figure 2-7 is an example of a pipeline that is composed of three filters: a Source, a FIR, and an Output. The `add` statement can be mixed with regular imperative code to parametrize the construction of the stream graph.

```
void->void pipeline FMRadio(int N, float lo, float hi) {
  add AtoD();
  add FMDemod();
  add splitjoin {
    split duplicate;
    for (int i=0; i<N; i++) {
      add pipeline {
        add LowPassFilter(lo + i*(hi - lo)/N);
        add HighPassFilter(lo + i*(hi - lo)/N);
      }
    }
    join roundrobin;
  }
  add Adder();
  add Speaker();
}
```

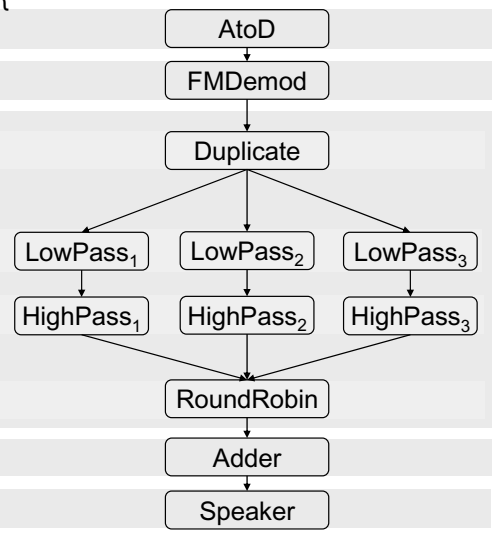


Figure 2-8: Example of a software radio with equalizer. There is a natural correspondence between the structure of the code and the structure of the graph. In the code, stream structures can be lexically nested to provide a concise description of the application.

A splitjoin represents independent parallel streams that diverge from a common *splitter* and merge into a common *joiner*. There are two types of splitters: 1) *duplicate*, which replicates each data item and sends a copy to each parallel stream, and 2) *roundrobin(*w_1*, ..,* w_n*)* which sends the first w_1 items to the first stream, the next w_2 items to the second stream and so on, repeating in a cyclic fashion. *roundrobin* is the only possible type for a joiner; its function is analogous to a roundrobin splitter. The splitter and joiner are specified with the keywords `split` and `join`, respectively (see Figure 2-8). The parallel streams are specified by successive calls to `add`, with the i'th call setting the i'th stream in the splitjoin.

The feedbackloop construct provides a way to create cycles in the stream graph. Each feedbackloop contains: 1) a body stream, which is the block around which a backwards "feedback path" is being created, 2) a loop stream, which can perform some computation along the feedback path, 3) a splitter, which distributes data between the feedback path and the output channel at the bottom of the loop, and 4) a joiner, which merges items between the feedback path and the input channel at the top of the loop. Feedbackloops are not common in StreamIt application; the techniques discussed in Chapter 7 do not support feedbackloops.

In StreamIt programmers build graphs of filters using structured, hierarchical composition. This requirement is analogous to the structured control flow introduced to replace unwieldy GOTO statements in imperative programming languages in the 1960s. See [Thi09] for a discussion of the benefits of structured streams. Figure 2-8 gives graph composition StreamIt code for our FMRadio benchmark with the resulting graph. Notice the strong correspondence between the code (especially the indentation of the code) and the resulting StreamIt graph. This correspondence allows programmers to visualize the structure of the graph while inspecting the code.

Hierarchical, structured graph composition also eases reusability and composability of components. All streams in StreamIt are single input/single output and can be reused easily because

of this uniformity property. Furthermore, streams can be parametrized by parameters that are resolved statically. These parameters can be simple constants or they can affect the graph structure as imperative code (e.g., `if`s, `for`s) can be used to build graphs. Notice that in Figure 2-8 the outer pipeline is parametrized by an integer N. N controls the width of the inner splitjoin (the equalizer in this example).

Hierarchical, structured streams also benefit the compiler during high-level analyses and transformations. Algorithms can reason locally about graphs, knowing that streams are single input / single output. The hierarchy allows for elegant recursive algorithms. These properties of the StreamIt graph eased formulation of phased scheduling [Kar02, KTA03], linear optimizations [LTA03, Lam03, Agr04, ATA05], and mapping to the compressed domain [THA07].

2.2.3 Other StreamIt Features

In StreamIt, the input and/or output rates of filters may be declared to be *dynamic*. Declaring a rate to be dynamic indicates that the filter will produce or consume a statically indeterminable number of data items. A dynamic input or output rate of a filter can be declared as a range (with minimum, maximum, and an average hint), with any or all of the characteristics designated as unknown. Dynamic rates are required for some application including MPEG-2 [MDH+06]. Applications with dynamic rates cannot be represented via SDF. We have developed techniques that break a dynamic rate application into multiple sub-components with statically determinable rates within the sub-components [CGT+05]. The techniques described in this thesis can then be applied to the static sub-components. It is beyond the scope of this thesis to describe the dynamic rate support in detail.

StreamIt also supports one other expansion to the synchronous dataflow model termed *teleport messaging* [TKS+05]. The techniques described in this thesis currently do not support teleport messaging.

2.2.4 Translation of StreamIt to the Execution Model

The execution model presented in Section 2.1 can represent computation described by the StreamIt programming language. The translation is straightforward, and will be summarized here. Previous work describes how to calculate initialization and steady-state schedules for StreamIt programs [Kar02]. Once schedules have been calculated, each StreamIt filter can be transformed directly into a filter in the execution model, with all of the appropriate information translated (push, pop, and peek rates for work and prework; and initialization and steady-state multiplicity). Since each filter in StreamIt is single-input and single-output, the input and output distributions are trivial to calculate. Section 5.3 describes how the compiler can calculate static work estimations for the filters.

Splitter and joiner nodes in StreamIt can be directly translated into filters in the execution model. These filters perform the identity function. For a StreamIt joiner, the weighted round-robin pattern is translated directly into the equivalent weighted round-robin pattern in the filter's input distribution. For a StreamIt duplicate splitter, the output distribution for the translated filter is a single duplicated dupset for all the destinations with weight 1. For a StreamIt weighted round-robin splitter, the output distribution of the translated filter is equivalent to the splitter's pattern. Note that since the execution model is a flat representation and since a single output distribution

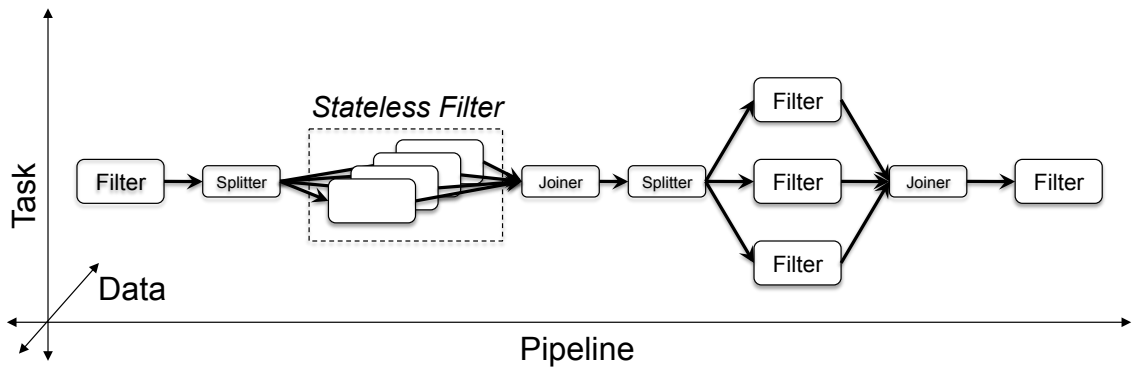


Figure 2-9: The three types of coarse-grained parallelism available in a StreamIt application. Note that normally stream graphs are presented vertically, but due to space considerations, we present this graph horizontally.

can duplicate and round-robin, multiple contiguous splitters (or joiners) can be cascaded into a single output (or input) distribution folded into the upstream (or downstream) filters. Section 8.3 describes an algorithm that removes filters in the execution graph that represented splitters and joiners in the original StreamIt graph.

In the remainder of this thesis we make the distinction between a *StreamIt graph* and a *stream graph*. The StreamIt graph for an application is the structured, hierarchical graph of splitjoins, feedbackloops, pipelines and filters. Many high-level transformations operate on this graph (these transformations can be thought of as source-to-source translations). The stream graph is the flat graph as described in Section 2.1. We use the term *filter* to denote the computational nodes of both graphs. Hopefully this will not cause confusion.

2.3 Parallelism in Stream Graphs

Stream programs offer three types of coarse-grained parallelism: task, data, and pipeline parallelism. Figure 2-9 illustrates the three types of parallelism by locating them to different dimensions of a stream graph. Despite the abundance of parallelism in stream programs, it is nonetheless a challenging problem to obtain an efficient mapping to a multicore architecture. Often the gains from parallel execution can be overshadowed by the costs of communication and synchronization. In addition, not all parallelism has equal benefits, as there is sometimes a critical path that can only be reduced by running certain filters in parallel. Due to these concerns, it is critical to leverage the right combination of task, data, and pipeline parallelism while avoiding the hazards associated with each.

Task parallelism refers to pairs of filters that are on different parallel branches of the original stream graph, as written by the programmer. That is, the output of each filter never reaches the input of the other. In stream programs, task parallelism reflects logical parallelism in the underlying algorithm. In StreamIt, task parallelism is introduced via the splitjoin construct; the branches of splitjoin are task parallel. In Figure 2-9 task parallelism can be found on the y axis between filters of the right-most splitjoin. It is easy to exploit by mapping each task to an independent processor and splitting or joining the data stream at the endpoints. The hazards associated with

task parallelism are the communication and synchronization associated with the splits and joins. Also, as the granularity of task parallelism depends on the application (and the programmer), we will demonstrate that it is not sufficient as the only source of parallelism. We demonstrate that it is important to consider task parallelism as a means to reduce the communication requirement of data-parallelization (see Section 6.3).

Data parallelism refers to any filter that has no dependences between one execution and the next. Such "stateless" filters[1] offer unlimited data parallelism, as different instances of the filter can be spread across any number of computation units. In Figure 2-9 data parallelism is present on the z axis between copies of a single stateless filter. A StreamIt programmer has no control on the amount of data parallelism exploited. It is the compiler's decision to duplicate stateless filters to introduce data parallelism. However, while data parallelism is well-suited to vector machines, on coarse-grained multicore architectures it can introduce excessive communication overhead. Previous data-parallel streaming architectures have focused on designing a special memory hierarchy to support this communication [KRD+03]. However, data parallelism has the hazard of increasing buffering and latency, and the limitation of being unable to parallelize filters with state. We demonstrate that, when present, data parallelism is the most effective form of parallelism to exploit, though it must be exploited at the proper granularity (see Section 6.2).

Pipeline parallelism applies to chains of producers and consumers that are directly connected in the stream graph. In Figure 2-9 pipeline parallelism is present on the x axis between filters. The compiler can exploit pipeline parallelism by mapping clusters of producers and consumers to different cores and using an on-chip network for direct communication between filters. Compared to data parallelism, this approach offers reduced latency, reduced buffering, and good locality. It does not introduce any extraneous communication, and it provides the ability to execute any pair of stateful filters in parallel. However, this form of pipelining introduces extra synchronization, as producers and consumers must stay tightly coupled in their execution. In addition, effective load balancing is critical, as the throughput of the stream graph is equal to the minimum throughput across all of the processors. We demonstrate that pipeline parallelism is required to parallelize applications that include significant amounts of stateful computation. We introduce new scheduling techniques that leverage pipeline parallelism by unrolling the steady-state scheduling loop of stream programs (see Chapter 7).

2.3.1 Data Parallelization of Sliding Window Computation

Figure 2-5(a) presents the StreamIt filter implementation of an FIR filter. The StreamIt version of the FIR filter allows the compiler to data-parallelize the computation without heroic analyses. The StreamIt compiler will realize that the filter does not include any mutable state in the steady-state (the `weights` array is modified only during initialization). Furthermore, the sliding window computation of the FIR algorithm is expressed using StreamIt's peek idiom. As we will see, the compiler can exploit data parallelism and create many parallel copies of the FIR filter; each copy computing on its own input data. The sliding window is shared as necessary between the data parallel copies via intelligently replicating input items between the copies. Furthermore, the amount of sharing between the copies can be parametrized by altering the steady-state schedule (see Chapter 8). Finally, the `weights` array can be replaced by multiple scalar variables by completely

[1] A stateless filter may still have read-only state.

unrolling all loops and propagating constants calculated in the init function to the work function. This process is called scalar replacement [STRA05].

Figure 2-5(b) shows how one could implement an FIR in C. This implementation requires sophisticated compiler analyses in order to parallelize and optimize. The programmer is forced to use a circular buffer to represent the sliding window of the FIR computation. Modulo operations are used in the address calculation for the circular buffer. Modulo operations are typically not handled by array dependence analysis frameworks. So in the presence of modulo operations, the compiler must conservatively assume that each read and write can access any location in the array. C's support for pointers makes parallelization of `do_FIR` difficult because the compiler would have to prove that the written arrays and scalars (`dest`, `srcIndex`, and `destIndex`) do not overlap the read arrays (`weights` and `src`). Finally, the legality of parallelizing calls to `do_FIR` depends on the sizes of the buffers chosen by the programmer.

The root cause of these obstacles for is that the programmer is writing an inherently streaming computation in C. C is an iterative programming language without support for streaming computation. The C programmer is forced to explicitly manage the buffering and scheduling policies of the computation, greatly over-specifying and obscuring the parallelism inherent to the computation. StreamIt's philosophy is that scheduling and buffering policies are best left to the compiler so that the compiler is free to apply program-wide transformations towards the goal of efficient parallel execution.

The goal of including peeking support in language, execution model, and compiler is to prevent stateful computation. Stateful filters cannot be data-parallelized because of the dependences that exist between successive firings of the filter. The peek idiom eliminates one common form of stateful computation, allowing the state of a sliding window to be exposed directly to the compiler. Once the sliding window is exposed to the compiler, data-parallelization of the sliding window filter can be enabled via duplication of input items across the data-parallelized products (see Section 6.3.3 and Chapter 8).

2.4 High-Level StreamIt Graph Transformations: Fusion and Fission

As we will see, it is necessary to adjust the granularity of the stream graph (and its filters) as it is instantiated by the programmer. One liberating aspect of stream programming is that the programmer does not have to worry about the underlying architecture(s) that she is targeting. A streaming application should be written in a way that most easily represents the algorithmic specification. Thus, it is often necessary to combine filters because they are too small, incurring a large penalty for context switching and communication between them. *Fusion* (verb form *fuse*) is the process of combining multiple filters into a single aggregate filter. Please see [Gor02] for a detailed description of StreamIt graph fusion.

Furthermore, stream programming discourages the programming from manually data-parallelizing filters (in fact, our compiler first removes all programmer-introduced data parallelism). Thus, it is necessary for the compiler to introduce data-parallelism. *Fission* (verb form *fiss*) is the process of data-parallelizing a stateless filter by duplicating the filter a certain number of ways, and wrapping the duplicates in a round-robin splitter and joiner to distribute the input data and collect the output data (SPMD). The duplicated filters are referred to as *products*. Please see [Gor02] for a detailed description of StreamIt graph fission.

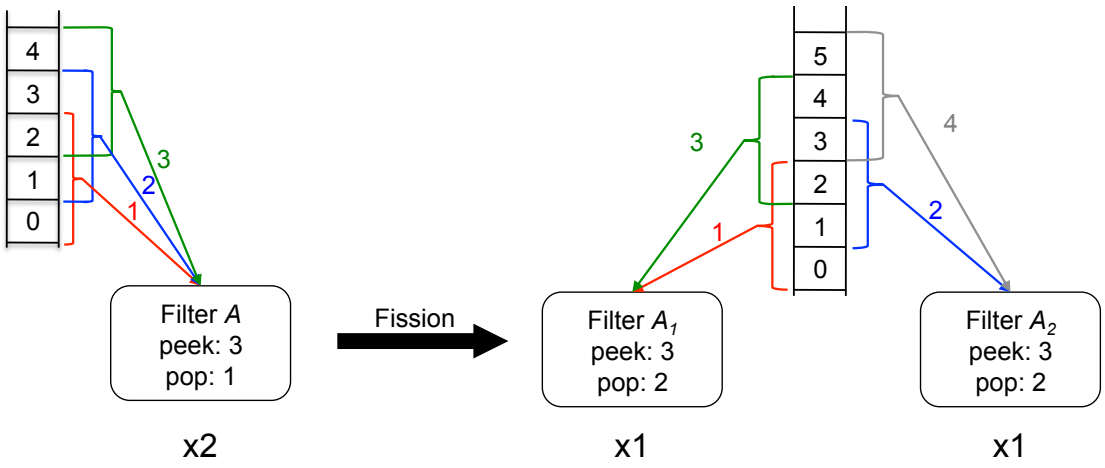


Figure 2-10: An example of the input sharing for fission of a peeking filter. Filter A has a multiplicity of 2 and is fissed to A_1 and A_2 each of which have multiplicity 1. For A because of its peek rate, items are shared across iterations of the filter. The input channel to A is annotated with the portions of the input buffer that each firing requires. After fission is performed, A_1 and A_2 share items from the input channel. The shared input channel is annotated with the items that are required for each firing of the fission products. The products share every input item during the steady-state.

Fission and peeking interact to require the sharing of input items across fission products. Figure 2-10 demonstrates a simple example of the sharing required by the fission of a peeking filter. Successive iterations of the fission products access overlapping portions of the input stream. The exact implementation of the sharing depends on the communication substrate of the target, e.g., for an architecture with fine-grained near-neighbor communication, each input can be duplicated to all products, items that are not needed by a product can be ignored. In Chapter 8, we introduce fission on the stream graph of the execution model. The formulation of fission on the graph of the execution model leverages the more expressive (as compared to StreamIt splitters) output distribution of filters to optimize data parallelization of peeking filters.

In the general case, when fissing a filter that peeks, i.e., a filter f with $C(f) > e(W, f) - o(W, f) > 0$, by P, the producers of f need to duplicate output items to an average of:

$$\max\left(1 + \frac{C(f)}{M(S, f) \cdot o(W, f)/P}, P\right) \tag{2.1}$$

fission products of f. Figure 2-11 gives a more complex example of the required sharing for a fission application. The filter f is duplicated 4 ways, has $C(f) = 9$, $o(W, F) = 4$, and $M(S, f) = 16$. From the above formula, each item is duplicated to an average of 3.25 fission products.

2.5 Sequential StreamIt Performance

Before the discussion begins on the parallelization prospects of StreamIt implementations, let us briefly discuss StreamIt's single-core performance. We need to establish a baseline for single-core performance to substantiate the techniques of this thesis. Figure 2-12 provides results for

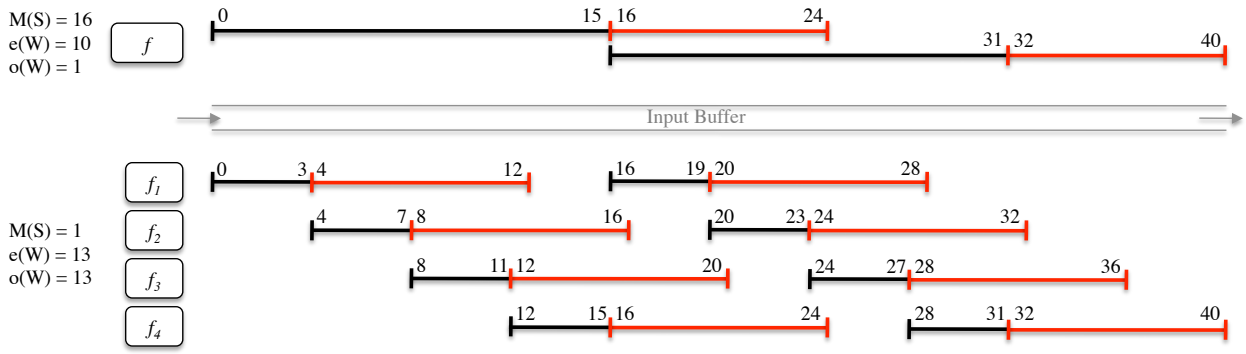


Figure 2-11: An example of the duplication of items required by fission. Filter f is fissed by 4 into f_1 - f_4. Two steady-states of the item indices required for both f and the fission products are shown. Item indices that are inspected by not dequeued are in red. For the fission products, it is required to duplicate 1 out of 4 items to all 4 filters, and the remaining 3 items are duplicated to 3 filters. The rates for the product filters have changed; Chapter 8 covers fission of peeking filters in detail. Notice that there is no longer peeking in the product filters.

5 benchmarks included in the StreamIt Core benchmark suite, comparing StreamIt's single-core throughput to C implementations of the benchmarks. The C versions are compiled with GCC with optimization level 3 enabled. For all of the benchmarks, the StreamIt implementation has higher throughput than the C implementation. This is because the StreamIt compiler includes aggressive optimizations that specialize the generated code of filters for the parameters with which they are instantiated. These optimizations include constant propagation and folding, array scalarization, and loop unrolling. These optimizations are applied across filter boundaries. GCC includes many of the same optimizations, however, they are rarely applied across filter boundaries in the C code, and constants are not propagated across functions. These results establish that the throughput parallelization speedups achieved by our techniques are normalized to an appropriate baseline.

2.6 Thoughts on Parallelization of StreamIt Programs

Over the past nine years, we have gained significant experience in mapping StreamIt applications to parallel architectures. When we began the StreamIt project in 2001, multicore architectures only existed in niche settings or computer architecture research groups. The Computer Architecture Group (CAG) at MIT anticipated the plateauing of single thread performance due to increasing energy budgets, design complexity, wire delay, and diminishing returns for ILP structures. Now, in 2010, multicore architectures have become the norm in personal computing, and multicore architectures are beginning to invade the mobile and embedded domains.

Predictions show that Moore's law will continue for the foreseeable future through new materials and lithography advances. In the multicore era, Moore's Law potentially translates to a doubling of the number of cores on a chip every 18 months. Current multicore and operating systems directly expose the number of cores to the executing program. The burden for continued performance gains with successive generations of multicore architectures now falls squarely on the

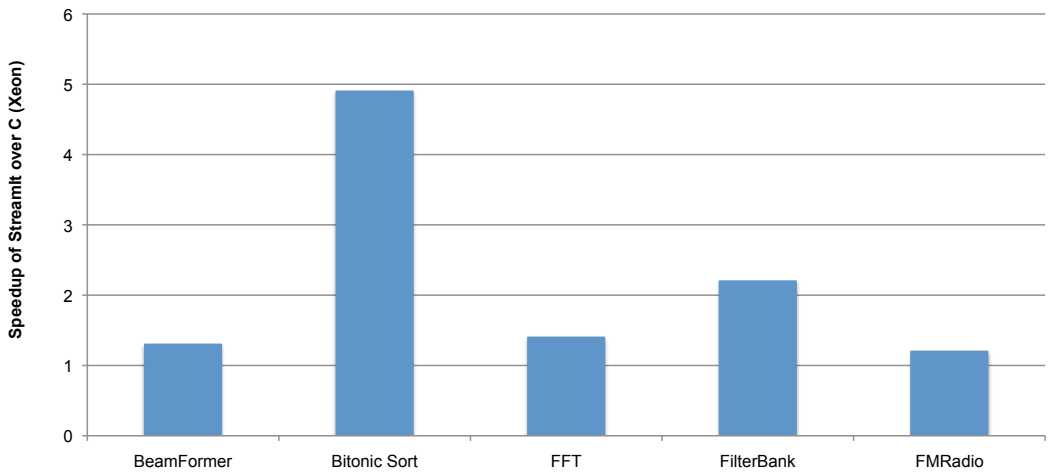


Figure 2-12: The performance of single-core StreamIt code compared to C on our Xeon evaluation system. The throughput (outputs per second) of the StreamIt code is normalized to the throughput of the C code. The chart includes the benchmarks for which we have C implementations.

programming system. The programming system must effectively leverage the additional cores to increase program performance.

The importance of a language that offers flexibility and scalability for multicore architectures is clear. We cannot ask programmers to rewrite software for each new multicore generation. Software must be written in a system that offers scalability (the ability to use additional cores without a software rewrite) and portability (the ability to be efficiently mapped to a number of multicore with different architectural structures and properties). Furthermore, the programming system must provide for software design properties that ease the burden of the programmer such as malleability (easy to make algorithmic or parameter changes to a program) and modularity (easy to reuse components of software). For more details regarding the benefits of StreamIt for the programmer, please see [Thi09].

It has been our experience that the following characteristics of StreamIt make it an effective, flexible, and scalable programming system for multicore architectures:

1. **Important class of algorithms that naturally fit into streaming domain.** Desktop workloads have changed considerably over the last decade. Personal computers have become communication and multimedia centers. Users are editing, encoding, and decoding audio and video, communicating, analyzing large amounts of real-time data (e.g., stock market information), and rendering complex scenes for game software. A common feature of all of these examples is that they operate on streams of data from some external source. The data is short-lived, being processed for a limited time unlike traditional scientific codes. Many potential killer applications of tomorrow will come from the domains of multimedia editing, computer vision, and real-time audio enhancement [CCD+08].

   StreamIt's philosophy is that the programmer does not have to worry about parallelization and domain specific optimization issues. She is free to focus on implementation correctness, malleability, and modularity of the code. StreamIt naturally captures algorithms that are organized as operations on streams of data. Furthermore, these codes have regular and repeating communication that can be captured via a StreamIt graph.

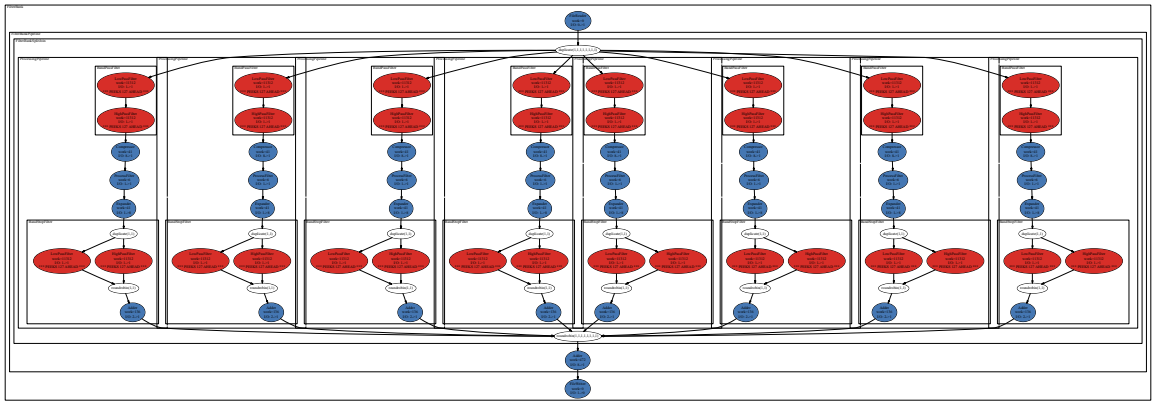


Figure 2-13: StreamIt graph for the FilterBank benchmark. Filters are colored according to the static work estimation. Red filters perform much more work than the blue filters.

2. **Parallelism in StreamIt is either explicit or easy to identify.** Task parallelism and pipeline parallelism are both explicit in StreamIt. Task parallelism is found in the splitjoin construct while pipeline parallelism is found in the pipeline construct. Data-parallelism is found by analyzing the work function of a filter and determining if the work function writes to a location that is read by a subsequent invocation of work. With the absence of pointers, this analysis is simple. The compiler does not have to perform impossible analyses to find the coarse-grained parallelism that exists in a program.

   Although parallelism is easy to identify in StreamIt, the compiler must be intelligent about leveraging parallelism in the mapping of the application; the most appropriate types of parallelism at the correct granularity must be exploited. This aspect is the main focus of this thesis. As an example let us consider the FilterBank benchmark given in Figure 2-13. The application has clear pipeline and task parallelism; additionally all filters of the application are stateless and can therefore be data-parallelized. There is abundant parallelism in the FilterBank benchmark, but it is illustrative to consider different techniques for this exploiting parallelism when compiling to a 16 core machine: exploiting data parallelism without adjusting the granularity achieves a speedup of 5.3x over a single core; exploiting task and hardware pipeline parallelism achieves a 7.5x speedup over a single core; exploiting only task parallelism and adjusting the granularity achieves an 11x speedup over a single core; and exploiting task and data parallelism and adjusting the granularity achieves a 14x speedup over a single core. We will explain these techniques and results in more detail in subsequent chapters, but the variability in these results show that it is important to be intelligent about the types of parallelism exploited and at what granularity.

3. **The filter is an appropriate atomic unit for parallel computation.** For compilers targeting von Neumann machines, the basic block is an atomic unit of computation that allows for localized reasoning for algorithms and transformations. A basic block is a simple structure with a single entrance and a single exit that can be analyzed and its computation summarized. We feel that StreamIt's filter construct offers the same benefits to compilers targeting multicore architectures. A filter is a both private and independent, with a private address space and its own program counter, thus they can be reasoned with locally. Filters are single input and single

output, so transformations do not have to account for complex communication topologies. The effects of a filter on dataflow can be summarized via its rates when known.

The concept of a basic block is hidden from the programmer; the compiler breaks down the code into basic blocks. However, in StreamIt, users create filters to express the computation on data streams. Since the user creates filters as dictated by their expression of the algorithm, the compiler must frequently alter the size of filters for performance reasons, combining (fusing) filters to reduce communication and context switching overhead, and duplicating (fissing) filters for parallelization. The design of the filter construct makes these transformations simple and efficient [Gor02].

Finally, the code of the work function for a filter is very close to C (though by design it lacks pointers). This imperative code can be optimized using the accumulated techniques of last 60 years. The structure of StreamIt programs mirrors the structure of multicore architectures: commodity multicore architectures are collections of von Neumann cores and StreamIt codes are collections of filters. Also, because of the lack of pointers, dataflow analyses are accurate. This allows the compiler to infer when a filter computes a linear function of its inputs, subject to aggressive optimization [LTA03]. Also, arrays of constants can be converted into multiple scalar variable constants by unrolling loops and performing constant propagation [STRA05].

4. **Static rates are found in many applications.** Many full applications or components of applications can be described by filters with static rates. In StreamIt, these static rates can depend on compile-time parameters to allow for code malleability. Static rates allow the StreamIt compiler to employ precise the scheduling, mapping, and communication techniques described in this thesis. These techniques do not bear the overhead of a runtime system.

   Static work estimates can be calculated for filters to allow for load-balancing for pipeline and task parallelism (although we will show that this is a difficult problem). Static rates allow for exact communication patterns to be calculated, and communication code to be generated for efficient, low-level mechanisms (e.g., the static network on Raw and remote writes for Tilera). Optimized fission and fusion transformations can be implemented for static rate filters.

5. **Peeking allows the compiler to parallelize sliding window computations.** Peeking is a useful construct that allows the programmer to represent a variety of sliding window computations. The most common pattern is a FIR filter where a filter peeks at N items, pushes a weighted sum of the items to the output, and pops one from the input. Another common pattern is when a filter peeks at exactly one item beyond its pop window [Thi09].

   Without including the peeking idiom in StreamIt, the patterns described above would have to be implemented in a stateful manner where the programmer explicitly introduces state to remember the dequeued item across work function firings. This state would inhibit data-parallelization (fission), as a dependence would exist between successive filter invocations. With explicit peeking, the compiler is allowed to data-parallelize peeking filters by intelligently replicating any shared input items across the fission constituents. The replication is performed via the scatter node that is the input to all the constituents.

6. **Implicit outer loop enclosing entire stream graph.** It is a property of the streaming domain that applications operate on huge amounts of data. Conceptually, StreamIt programs run infinitely, and in practice, applications execute a large number of iterations of their stream graph.

The scheduling and mapping techniques described in this thesis take advantage of this property when exploiting both hardware and software pipeline parallelism. Data-parallelization also takes advantage of the outer loop as the fission transformation often increases the multiplicities of filters in the steady-state schedule (at the expense of latency).

7. **Prework functions are useful for expressing startup conditions, and for eliminating associated state.** The prework function allows a filter to have different behavior on its first invocation. The most common use of prework is for implementing a delay; on the first execution, the filter pushes N placeholder items, while on subsequent executions it acts like an Identity filter. Without prework, the delayed items would need to be buffered internally to the filter, introducing state into the computation.

8. **Regular and repeating communication allows compiler to optimize communication and synchronization.** Communication in StreamIt is regular and repeating. Changes to the graph topology of a streaming application are meant to occur infrequently. In combination with static rates and precise mapping techniques, the compiler can calculate the ordering and routing of all data items communicated between filters. Furthermore, synchronization in StreamIt is implicit through the channel that connect filters. Filters can fire only when all of their input data is available. Thus, synchronization is local between producers and consumers, and does not require expensive global barriers.

   These properties allow the compiler to target efficient, low-level, and fine-grained communication structures of some multicore architectures. In the case of the Raw microprocessor, the compiler generates instructions to control the switch processors, achieving precise communication in parallel with computation, synchronization via the network's flow control, parallelized scattering and gathering, and duplication for free.

   StreamIt's communication mechanisms do not limit it to one architecture. Efficient backends for the Cell processor [Hof05, ZLRA08], commodity SMPs [Tan09], and Tilera [WGH+07] demonstrate that it is a portable language for diverse multicore architectures.

2.7 Related Work

The concept of computing on a stream of data has a long history in programming languages; see [Ste97] for a review. In general, streaming models are considered as graphs, where nodes represent units of computation and edges represent communication patterns. The models differ in the regularity and determinism of the communication pattern, as well as the amount of buffering allowed on the channels. Some of the most prevalent models are Kahn Process Networks [Kah74], Synchronous Dataflow (SDF) [LM87b], Computation graphs [KM66], Actors [HBS73, Gre75, Cli81, Agh85], Petri nets [Pet62, Mur89], and Communicating Sequential Processes [Hoa78]. We cover the most related models in more detail:

1. **Kahn Process Networks** is a simple model in which output is non-blocking, but input will block until data is ready (it is not possible to test for the presence or absence of data on input channels). Assuming that each node performs a deterministic computation, these properties imply that the entire network is deterministic. It is undecidable to statically determine the

amount of buffering needed on the channels, or to check whether the computation might deadlock [Thi09].

2. **Communicating Sequential Processes (CSP)** is a model where nodes communicate via rendezvous communication. There is no buffering on the communication channels. CSP is non-deterministic as a node may make a non-deterministic choice when choosing to read input from its input channels. Deadlock freedom is undecidable.

3. **Computation graphs** are a generalization of SDF that supports finite systems. Nodes stipulate that in order to execute a threshold number of items must be available on an input channel, this threshold can exceed the number of items consumed by the node.

Compared to other models, SDF offers a unique combination of statically determinable rates, FIFO buffering, infinite execution, and synchronicity. In SDF, unlike many models, the amount of buffering can be determined statically, and the communication pattern is static. Many variations of synchronous dataflow have been defined, including cyclo-static dataflow [BELP95, PPL95] and multidimensional synchronous dataflow [ML02].

The StreamIt language is one example of a stream programming language. Here we compare StreamIt to its contemporaries. The Brook stream language [BFH+04] focuses on data parallelism as stateful computation nodes are not supported. Brook, unlike StreamIt, has special support for reductions (in StreamIt, a programmer has to manually implement reductions). Sliding windows are supported via stencils, which indicate how data elements should be replicated across multiple processing instances. An independent comparison of the two languages concludes that StreamIt was designed by compiler writers (it is "clean but more constrained") while Brook was driven by application developers and architects, and is "rough but more expressive" [ML03].

Several stream languages were developed by the computer graphics community. Cg exploits pipeline parallelism and data parallelism, though the programmer must write algorithms to exactly match the two pipeline stages of a graphics processor [MGAK03]. The sH language is similar to Cg, though it supports arbitrary length pipelines [MQP02, MTP+04]. sH nodes are required to be stateless. Like StreamIt, sH specializes stream kernels to their constant arguments, and fuses pipelined kernels in order to increase their granularity. Unlike StreamIt, sH performs these optimizations dynamically in a Just-In-Time (JIT) compiler, offering increased flexibility. However, StreamIt offers increased expressiveness.

StreamC/KernelC preceded Brook and operates at a lower level of abstraction; kernels written in KernelC are stitched together in StreamC and mapped to the data-parallel Imagine processor [KRD+03]. SPUR adopts a similar decomposition between "microcode" stream kernels and skeleton programs to expose data parallelism [ZLSL05].

StreamIt is not the first language to expose sliding window computation. Printz's "signal flow graphs" [Pri91] and ECOS graphs [HMWZ92] allow actors to specify how many items are read but not consumed. The Signal language allows access to the window of values that a variable assumed in the past [GBBG86]; and the SA-C language contains a two-dimensional windowing operation [DBH+01].

In summary, whereas all languages covered above expose data parallelism, StreamIt places more emphasis on exposing task and unconstrained pipeline parallelism. By building upon the synchronous dataflow model of execution, StreamIt focuses on well-structured and long-running

programs that can be aggressively optimized statically. Spidle [CHR+03] is also a recent stream language that was influenced by StreamIt.

Proebsting and Watterson [PW96] present a filter fusion algorithm that interleaves the control flow graphs of adjacent nodes. However, they assume that nodes communicate via synchronous `get` and `put` operations; StreamIt's asynchronous peek operations and implicit buffer management fall outside the scope of their model.

2.8 Chapter Summary

This chapter describes the execution model employed for this paper. The execution model is built upon the synchronous dataflow model and adds the additional features of cyclic data distribution, peeking, and communication during initialization. In our model, computation nodes termed filters communicate via FIFO channels, with the number of items produced and consumed by each firing of the filter statically determinable. With peeking support, our model explicitly represents filters that do not consume all the items read from their input buffer. Our execution model explicitly represents the steady-state of an application that can be repeated indefinitely as is does not grow nor shrink buffers after execution. Each filter defines rate declarations for its firing in the steady-state. Filters can be multiple input and/or multiple output where data distribution is described by cyclic schedules of sources (for inputs) and destinations (for outputs).

The StreamIt language is an example of a high-level programming language that can be expressed by our model of execution. In StreamIt the computation step of a filter is described by imperative code, and the programmer declares rates for each filter. StreamIt enforces a structured and hierarchical approach to constructing graphs. The programmer has three containers available to construct graphs: pipelines, splitjoins, and feedback loops. The translation from StreamIt code to the execution model is trivial.

There exists three types of coarse-grained parallelism in our execution model: task, pipeline, and data parallelism. With peeking support, our execution model exposes one common source of stateful computation to the compiler, allowing the compiler to parallelize these computations. The author's experience with parallelizing StreamIt codes is distilled. The main points being that stream programming is a natural representation for many important algorithms, and the execution model naturally exposes parallelism. Static rate declarations are necessary for the static scheduling techniques of this dissertation. Regular and repeating communication allows compiler to optimize communication and synchronization. Finally, the calculation of the steady-state schedule and the fact that it is repeated many times, allows a compiler to schedule computation across multiple steady-states in order to effectively schedule parallelism.

Chapter 3

A Model of Performance

In this chapter we develop an analytical throughput model for stream programs targeting multicore architectures. We employ the model to compare the scalability prospects of two scheduling strategies: time-multiplexed data parallelism (TMDP) and space-multiplexed data parallelism (SMDP) The model is restricted to stream programs that can be represented as a pipeline of stateless filters. We develop the model's throughput calculation for the scheduling strategies by considering how they compare in terms of communication cost, load-balancing effectiveness, and memory requirement. Closed-form solutions for communication cost are calculated for architectures offering fine-grained, near-neighbor, grid communication.

3.1 Introduction

In Chapter 2 we introduce the various forms of parallelism inherent to stream programs: data, pipeline and task. In this chapter we begin to examine the tradeoffs of the various forms of parallelism and different mapping strategies for harnessing parallelism. Chapter 4 demonstrates that most of our streaming application benchmarks are comprised solely of data parallel (stateless) filters. In this chapter we develop a model of parallelism for stream programs that can be represented as a pipeline of stateless filters, examining two mapping strategies. Furthermore, we compare the exact communication costs of this class of programs when mapped to a grid architecture with near-neighbor communication. In subsequent chapters we present empirical performance results for points within the two general mapping strategies modeled here.

There are many possible strategies for mapping a stream graph to a multicore architecture. We model two strategies that are positioned at opposite ends of the *pipeline parallelism* spectrum. The traditional mapping strategy is *time-multiplexing* where each filter is duplicated to all the cores and the stages of the application are "swapped" in and out of the processor [GTA06, KRD+03]. A time-multiplexing mapping does not take advantage of any pipeline parallelism. An alternate method is *space-multiplexing*. In this method, all the filters of the application execute concurrently on the processor. Each filter must be mapped to at least one core, but a filter can be data-parallelized (fissed) so that it occupies more than one core. In this manner, filters with differing amounts of relative work can be load-balanced by assigning each to a different number of cores. There exists data-parallelism between the duplicated filters of the original application and pipeline parallelism between the different data-parallel groups.

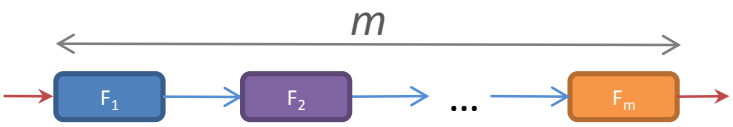


Figure 3-1: The initial version of the stream graph.

In this chapter, we compare the two mapping strategies using our model of streaming computation. We qualitatively and quantitatively compare the scalability prospects of the methods in terms of communication requirement, load-balancing, and caching behavior. We show that neither strategy is perfect for all programs and architectures. However, pipeline parallelism will become increasingly important if trends in multicore architectures continue towards increasing the number and coupling of cores and increasing the simplicity and efficiency of each core. Time-multiplexing, the traditional approach, offers perfect static load-balancing but may overtax the memory system. This approach increases buffer requirement and duplicates code across cores. Alternately, in a space-multiplexed mapping, load-balancing becomes more difficult as the different filters need to be balanced, but reduced memory footprints can be achieved. Exposing pipeline parallelism reduces memory requirements because data parallel units do not have to span all the cores of the processor. Interestingly, we show that the communication requirements of space-multiplexing and time-multiplexing are asymptotically equivalent when considering architectures with low-latency, near-neighbor networks.

3.2 Preliminaries

Much of our analysis is applicable to any multicore architecture with n cores. We assume that n remains fixed during execution. In Section 3.5, we develop a closed form solution for the communication cost of the two scheduling strategies. For the communication cost analysis we model multicore architectures with low-latency, near neighbor communication such as the IBM/Toshiba/Sony Cell [Hof05], the Tilera Tile64 [WGH+07], and MIT's Raw [TLM+04].

We will restrict ourselves to graphs which begin as a pipeline of m stateless filters, $F_1 \ldots F_m$, where F_{i+i} directly consumes the items produced by F_i, $1 \leq i < m - 1$ (see Figure 3-1). This is an important class of applications. Many stream languages, by construction, support only data-parallel filters [BFH+04, KRD+03, ZLSL05, MGAK03]. Pipelines of stateless filters may seem overly restrictive, but many streaming applications can be represented by this model. In fact, the model captures 7 of the 12 benchmarks in the StreamIt Core benchmark suite introduced in Chapter 4. This is because in the domain of SDF two or more neighboring filters can be *fused* into a single filter [GTK+02]. This fused filter executes a schedule of its constitute filters. Repeatedly applying the fusion transformation can reduce a complex stream graph to the pipeline of filters that we consider.

For the analysis in this chapter, we are concerned with the aggregate steady-state characteristics of a filter F. If we remember the notation from Section 2.1, we define peek, pop, and push rates of a filter F per firing of the work function as $e(W, F)$, $o(W, F)$, and $u(W, F)$ respectively, where W is the work function of F. For this chapter, we define the following shorthand to denote rates of a filter over the entire steady-state:

- $o(F) \equiv o(W, F) \cdot M(S, F)$
- $e(F) \equiv (e(W, F) - o(W, F)) + o(F)$
- $u(F) \equiv u(W, F) \cdot M(S, F)$

where $M(S, F)$ defines the multiplicity of F for the steady-state schedule S. We also use the shorthand $s(F)$ to denote the total execution time of all the work function firings of F in the steady state (i.e., $s(F) \equiv s(W, F) \cdot M(S, F)$, where $s(W, F)$ defines the work estimation of a firing of F's work function W).

In the streaming domain, performance is measured in terms of throughput in the steady-state. Throughput is defined as the number of application outputs per cycle of the global clock. For architectures with communication/computation concurrency (i.e., DMA) [Hof05, WGH+07]. The throughput equation for a graph G is:

$$\mathit{THRU}(G, \mathit{COMP}_G, \mathit{OUTPUTS}_G, \mathit{COMM}_G) = \frac{\mathit{OUTPUTS}_G}{\max(\mathit{COMP}_G, \mathit{COMM}_G)} \tag{3.1}$$

$\mathit{OUTPUTS}_G$ is the number of outputs per steady-state. COMP_G is the computation cost for the steady-state. COMM_G is the communication cost for the steady-state. The numerator calculates the number of outputs. The denominator estimates the total number of cycles required for the steady-state. Since we have communication and computation concurrency, it is a maximum of the communication cost and computation cost.

It is easy to extend the model to architectures without computation and communication concurrency. The number of cycles required for the steady-state is the sum of computation and communication:

$$\mathit{THRU}(G, \mathit{COMP}_G, \mathit{OUTPUTS}_G, \mathit{COMM}_G) = \frac{\mathit{OUTPUTS}_G}{\mathit{COMP}_G + \mathit{COMM}_G} \tag{3.2}$$

3.3 Filter Fission

Each filter of our pipeline is stateless and can be parallelized by duplicating it across cores. Each duplicated filter will receive its own input item(s) and the outputs of the duplicated filters will be joined in a round-robin fashion. *Fission*, introduced in Chapter 2, describes the process of filter duplication, input splitting, and output joining. Fission of filter F_i by P_i results in P_i filters called *fission products*, $F_{(i,1)}, F_{(i,2)}, \ldots, F_{(i,P_i)}$. This section presents examples of filter fission in more depth focusing on the *width* of the communication required by the fission. Width of communication will be a factor that affects the scalability of a streaming application.

Let us consider two filters in our pipeline application, F_i and F_{i+1} (see Figure 3-2). Note that $u(F_i) = o(F_{i+1})$ because if did not, executing the steady-state schedule would increase or decrease the number of items remaining on the channel between F_i and F_{i+1} and this is a contradiction. The simple fission case is illustrated in Case 1 of Figure 3-2. In this case $e(F_{i+1}) = o(F_{i+1})$ and we fiss F_i by P_i and F_{i+1} by P_{i+1} where $P_i = P_{i+1} = P$, parallelizing each to P cores. In this case, each duplicated $F_{(i,p)}$ communicates to only one filter, $F_{(i+1,p)}$, $1 \leq p \leq P$. We say that the width of the communication required for this fission is 1.

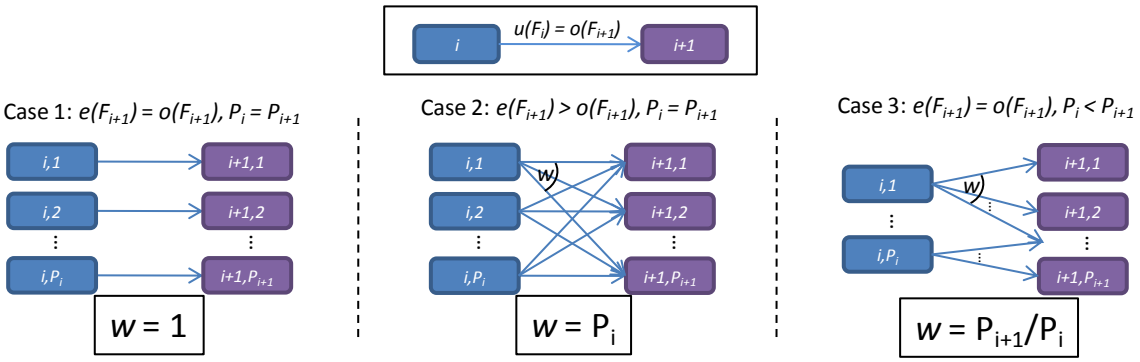


Figure 3-2: Three cases of fission on two filters F_i and F_{i+1}.

It is possible for the width of communication of each $F_{(i,p)}$ to equal P_{i+i}, the fission amount of F_{i+1}. Consider the case in which F_{i+1} *peeks*, meaning $e(F_{i+1}) > o(F_{i+1})$. When fissing a peeking filter F_{i+1}, the fission product $F_{(i+1,j)}$ must read items from $F_{(i,j)}$ but must also read input items that were produced by other $F_{(i,q)}$ (where $1 \leq q \leq P$ and $q \neq j$) because of F_{i+i}'s peek rate. As illustrated in Case 2 of Figure 3-2, when F_{i+1} is duplicated, each product also peeks; $F_{(i+1,p)}$ must read from multiple products of F_i. An implementation could simply duplicate the outputs from each $F_{(i,p)}$ to all $F_{(i+1,p)}$. Each $F_{(i+1,p)}$ would then disregard the inputs that it did not need [Gor02]. In this case the width of communication would equal P and this is considered this a global communication.

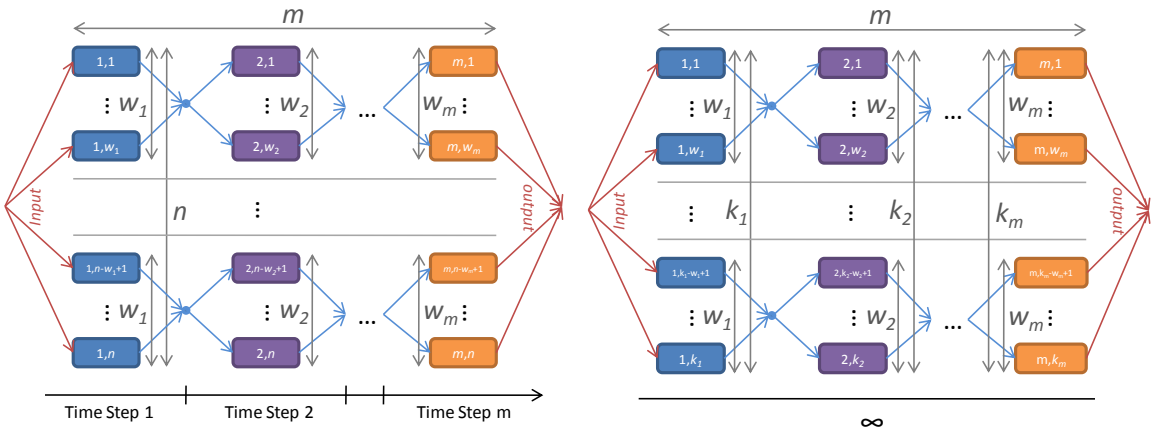
Finally, it is possible for the width of communication for each $F_{(i,p)}$ to range between 1 and P_{i+1}. As an example, consider the case in which $P_i < P_{i+1}$, meaning F_i is fissed by less than F_{i+1} (see Case 3 of Figure 3-2). This is useful when attempting to load-balance a pipeline of filters and $s(F_i) < s(F_{i+1})$. In this case, each $F_{(i,p)}$ outputs to exactly $\frac{P_{i+1}}{P_i}$ of the $F_{(i+1,p)}$'s. We still require that the items enqueued onto the channels by all the $F_{(i,p)}$'s equal the number of items dequeued from the channels by all the $F_{(i+1,p)}$'s (to maintain the steady-state property). In this case the width of communication is $\frac{P_{i+1}}{P_i}$.

The three cases given above demonstrate that it is possible for fission to require differing types of communication given varying application, mapping, and implementation scenarios. The discussion in this section did not rigorously define the width of communication for all cases of fission (see Chapter 8 for complete coverage of fission of peeking filters). The point of this section is to prove by example that w can vary from 1 to P_{i+1} for a filter at stage i. It is important to include the width of communication, w, for the communication component of our model. We will see later that the width of the communication is an important factor affecting scalability.

3.4 Scheduling Strategies

This section describes the two scheduling strategies we are comparing, time-multiplexed and space-multiplexed data parallelism. In each strategy, data parallelism is leveraged through fission of filters. At the high level, the strategies differ in that time-multiplexing does not leverage any pipeline parallelism while space-multiplexing does leverage pipeline parallel.

Our model considers applications that are composed of m stateless filters arranged in a pipeline. The application has been *fused* down to this simple structure by repeatedly applying the fusion



(a) Time-Multiplexed Data Parallelism.

(b) Space-Multiplexed Data Parallelism.

Figure 3-3: Two scheduling strategies: (a) Time-multiplexed data parallelism; and (b) Space-multiplexed data parallelism.

transformation on neighboring filters. We call this pipeline (Figure 3-1) the *intermediate form* of the application. This intermediate configuration is the input to the scheduler.

3.4.1 Time-Multiplexed Data Parallelism

Time-multiplexed data parallelism (TMDP) is the traditional strategy for exploiting coarse-grained data-parallelism in a variety of domains [Cha01, KRD+03]. Figure 3.4.2(a) displays the result of applying a TMDP scheduler to the intermediate form of the application. Each filter F_i, $1 \leq i \leq m$, is fissed n ways, where n is the number of cores of the target. Each fission results in its own width of communication based on the characteristics of the original filter and the fission of its downstream consumer.

During execution, on time step 1, the n fission products of F_1 occupy the entire chip. The application input has been distributed to these product filters. After the products of F_1 execute, their output is distributed as needed so that the fission products of F_2 can execute. On step 2, the products of F_2 occupy the chip and so on, until m steps are completed . This is considered the new steady-state. The steady-state repeats itself if there is more input to process. There are $\sum_{i=1}^{n} F_{(m,i)}$ outputs per steady-state.

3.4.2 Space-Multiplexed Data Parallelism

Space-multiplexed data parallelism (SMDP) leverages the pipeline parallelism exposed in the intermediate form the of application. Each filter F_i, $1 \leq i \leq m$, is fissed k_i ways, such that $\sum_{i=1}^{m} k_i = n$. This strategy is shown in Figure 3.4.2(b). The goal is to use data-parallelism (fission) to load balance the pipeline stages of the intermediate form, i.e., we would like $s(F_{(1,1)}) = s(F_{(2,1)}) = \ldots = s(F_{(m,1)})$. It is sufficient to use the first product because all the fission products of a single filter are load balanced, i.e., $\forall i \in 1 \ldots m[s(F_{(i,1)}) = s(F_{(i,2)}) = \ldots = s(F_{(i,j)}), 1 \leq j \leq k_i]$. In practice, this is usually the case since filters in the streaming domain contain little data-dependent control flow and the fission products are duplicates of the same filter.

Each filter of the space-multiplexed graph, $F_{(i,j)}$, $1 \leq i \leq m$ and $1 \leq j \leq k_i$, is mapped to its own core and executes on that core for the duration of the execution. The schedule takes advantage of the pipeline parallelism of the intermediate form because stages directly communicate and a producer can work on a new input while a consumer is working on the producer's output. A steady-state is an execution of all the filters of the space-multiplexed graph such that each fires once. $\sum_{i=1}^{k_m} F_{(m,i)}$ outputs are produced per steady-state.

In this model we only consider data-parallel filters, but a space-multiplexed scheduling strategy can parallelize stateful filters as well, mapping them to a pipeline and exploiting pipeline parallelism. A time-multiplexed approach cannot parallelize stateful filters because it relies on data-parallelism as its sole form of parallelism.

3.5 Communication Cost

The comparison of the two mapping strategies begins by quantifying their communication cost, *COMM*. This quantity will be a factor in the overall performance comparison of the two techniques. Determining the communication requirement of a parallel mapping is crucial to evaluating its effectiveness. A mapping that seems otherwise perfect (i.e., it is load balanced and has good caching behavior) could still overwhelm the communication substrate, rendering the mapping ineffective. In this section, we develop a closed-form asymptotic bound for communication cost that assumes a specific type of communication substrate. We use this bound to analyze the scalability prospects of the mapping strategies.

We model a near-neighbor, fine-grained communication implementation for grid topologies. This design reappeared recently with MIT's Raw microprocessor [TLM+04] and sees continued development and acceptance with Tilera's lineup of multicore processors [til]. This design offers extreme scalability as there are no structures that grow with the number of cores, and no wire is longer than a single core. Routing is regular, and data is streamed between cores. Our conclusions can be qualitatively extended to other communication implementations such as shared memory. However, calculating closed form solutions for complex hardware sharing protocols is extremely difficult because of unknown topologies, complex cache hierarchies, non-uniform communication costs, cache-line issues, and prefetching strategies.

We quantify the *maximum link bandwidth* (*MLB*) of TMDP and SMDP under some further restrictions on the intermediate form of the application. The maximum link bandwidth is defined as the maximum number of items that traverse a communication link of our target architecture during one steady-state execution of the scheduled and mapped graph. Maximum link bandwidth is important because we model a parallel communication substrate; all links can perform a transfer in parallel. Maximum link bandwidth is directly related to the number of cycles required to perform the communication.

3.5.1 Communication Substrate

We consider both a 1D array of n cores and a 2D array of $\sqrt{n} * \sqrt{n} = n$ cores. We model a wormhole routed, near-neighbor interconnect. Cores are connected by bi-directional, full-duplex communication links that can transfer one item per cycle in each direction. The interconnect flow-controls the cores, meaning a core will block when attempting to receive from an empty link and

a core will block when attempting to send to a link with an item occupying it. A core in the 1D array has a link to its east and west neighbor while a core in the 2D array has a link to its north, south, east, and west neighbors. Links on the periphery of the processor are connected to off-chip I/O and off-chip memory.

Items are wormhole, dimension-order routed in first the x dimension and then the y dimension. If an item injected into the network has multiple destinations, the duplication is handled by the network. More specifically, this means that the item need only be injected into the network once and a copy of the item is duplicated only when needed along its path in first the x dimension and then the y dimension. Input is read from single source off-chip and streams onto the chip using one of the links on the periphery of the chip. Output written is to single source using a single link on the periphery.

3.5.2 Application Constraints

For this analysis the intermediate form of the application is constrained such that it is a *load-balanced* pipeline of stateless filters such that, $s(F_1) = s(F_2) = \ldots = s(F_m)$. Additionally, assume that each filter consumes one item and produces one item ($o(F_i) = u(F_i) = 1$ for $1 \leq i \leq m$). Forcing this situation makes it possible to directly compare the two strategies because an exact form of the each can be computed. $e(F_i)$ is left unspecified.

In this analysis, the communication width, w, measures the number of filters that each filter communicates to in the scheduled graph (see Figure 3.4.2). w is constant across the scheduled graph such that $w_1 = w_2 = \ldots = w_m = w$. Each filter F_i is data-parallelized (fissed) into a *fission group*, i.e., filters $F_{(i,1)} \ldots F_{(i,k_i)}$, note that $k_i = n$ for TMDP. Each fission group is segmented into $\frac{k_i}{w}$ *clusters*. A cluster can communicate with only one input cluster and one output cluster. This segments the scheduled graph into *cluster pipelines*, pipelines of communicating clusters, where communication does not enter or leave the cluster pipeline. Note that there is no pipeline parallelism between cluster pipelines in TMDP. Clustering the scheduled graph as such simplifies our analysis while still modeling the width of communication. It will allow us to determine if width is an important factor in determining performance and scalability.

For TMDP, $1 \leq w \leq n$, m is unbounded, and there are n outputs per steady-state.

For SMDP, since each filter in the intermediate pipeline performs the same amount of work, each filter is fissed by the same amount, k (i.e., $k_1 = k_2 = \ldots = k_m = k$). Finally, $1 \leq w \leq k$ and $m * k = n$. This last equality specifies that the number of filters in the scheduled graph is equal to the number of cores on the target architecture.

The input channel streams onto the processor to one of the boundary cores where it is distributed by the network in around robin fashion to the fission products of F_1. The output is collected by the network from each of the fission products of F_m and streams off of the processor from one of the boundary cores. The exact location of these boundary cores is noted if important. Between each fission group, items must be distributed to the proper filters. This is called a *reorganization stage*. Within each reorganization stage, each filter forwards its output item to each filter of its destination cluster. Remember that the duplication is achieved in the network; only one item is injected into the network (see Section 3.5.1).

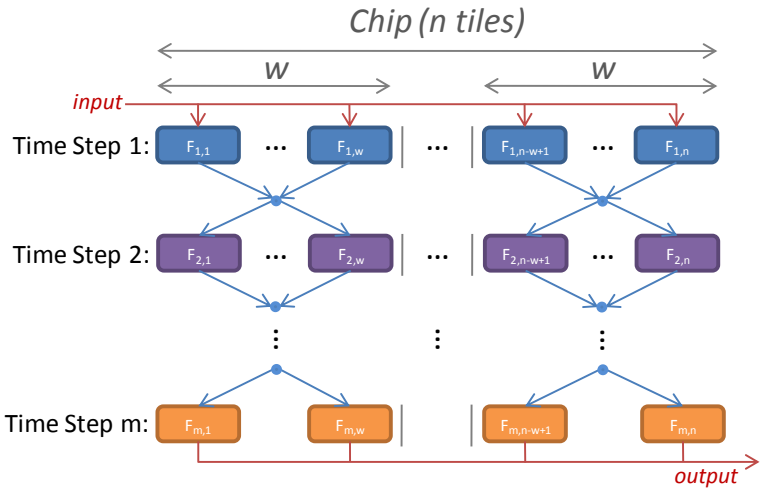


Figure 3-4: The mapping and schedule for TMDP on a linear array of n cores.

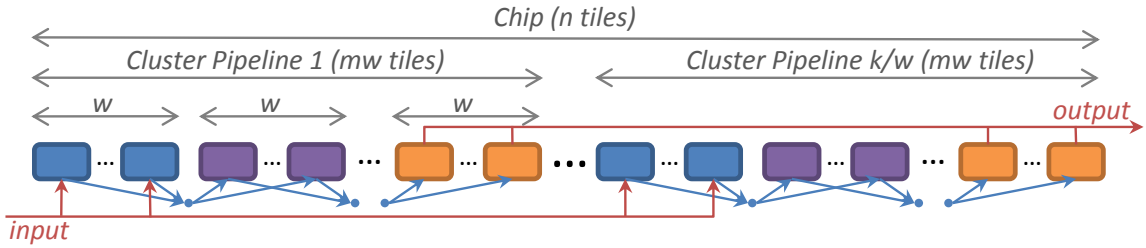


Figure 3-5: The mapping for SMDP on a linear array of n cores.

Theorem 3.5.1 *The maximum link bandwidth for TMDP on a linear array of n cores is $O(n + mw)$.*

Proof. The mapping is given in Figure 3-4. For each link, L_l (l links from the left), $n - l$ input items and l output items traverse the link, summing them generates n I/O items. For one of the reorganization stages, for each cluster, at most w items traverse because all communication stays with a cluster. There are $m - 1$ redistribution stages. Summing I/O and the redistribution for each link produces $n + w(m - 1) = O(n + mw)$. □

Theorem 3.5.2 *The maximum link bandwidth for SMDP on a linear array of n cores is $O(w + k)$.*

Proof. A sensible mapping is given in Figure 3-5. In this mapping items are flowing from left to right in a pipeline fashion, and the cluster pipelines are assigned contiguously. For the link between the two rightmost cores in a cluster, w input items from the upstream cluster are communicated into the rightmost core and $w - 1$ output items from this cluster pass through this link on their way to the downstream cluster. For the application input and output, at most k items flow through each link. So, $2w + k - 1$ items flow across this link, $= O(w + k)$. □

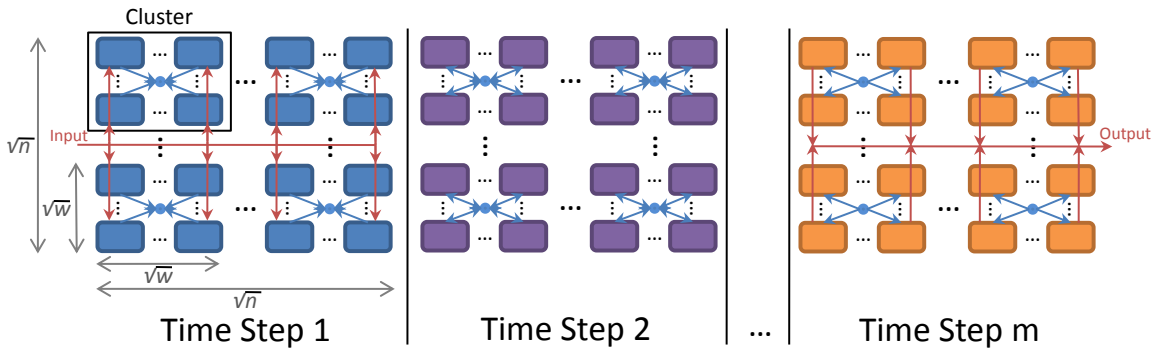


Figure 3-6: The mapping and schedule for TMDP on a 2D array of n cores.

3.5.3 n Processor 2-D Array ($\sqrt{n} \times \sqrt{n}$)

Theorem 3.5.3 *The maximum link bandwidth for TMDP on a 2D array of n cores is $O(n + mw)$.*

Proof. The mapping and layout is given in Figure 3-6. In the mapping, each cluster occupies a contiguous block of cores of length $\sqrt{w}$ and width $\sqrt{w}$. The downstream cluster is mapped to an identical set of cores so all communication during the redistribution stages remain within the block of cores assigned to the cluster.

For the input and output, as an upper bound $2n$ items will traverse any link (in actuality, no on-chip link achieves this because the I/O is distributed across the processor).

Since all duplication is performed in the network and items are first routed in the x (horizontal) direction, the link with the maximum number of hops traversing it will be a vertical link. More specifically, given a core t, when receiving input from the upstream cluster during a redistribution stage, t will receive input from only t's own row on the horizontal links, but on the vertical links t will receive input from all the cores north and south of t.

Consider one of the southernmost cores in a cluster block. For its north link, it will receive $w - \sqrt{w}$ input items of the previous cluster from the cores north. This core will send north its own output plus the outputs of the other cores of its row, $\sqrt{w}$ items. There are $m - 1$ redistribution stages. Summing the input and accounting for all the redistribution stages produces $2n + (m - 1)w = O(n + mw)$. □

Theorem 3.5.4 *The maximum link bandwidth for SMDP on a 2D array of n cores is $O(w + k)$.*

Proof. Let us first consider the simple case in which $m = \sqrt{n}$ and $k = \sqrt{n}$ as shown in Figure 3-7. For application input and output, an upper bound on the number of items that traverses a link is $2k$.

Now consider the link bandwidth of the redistribution between stages of a cluster pipeline. Since all duplication is performed in the network and items are first routed in the x dimension, the outputs of a stage in the cluster pipeline travel horizontally (left to right the figure), then each output is routed to each core in the vertical directions (north and south). So each vertical link will have w input items traversing it, one item for each of the outputs of the upstream cluster. Adding application I/O and the redistribution stage we end up with: $w + 2k = O(w + k)$. Note that our bound does not change if we rotate the mapping given Figure 3-7 by $90°$.

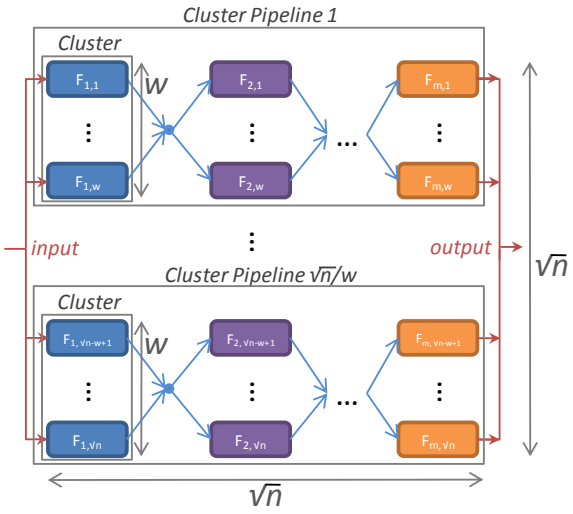


Figure 3-7: A simple mapping for SMDP on a 2D array of n cores if $m = \sqrt{n}$ and $k = \sqrt{n}$.

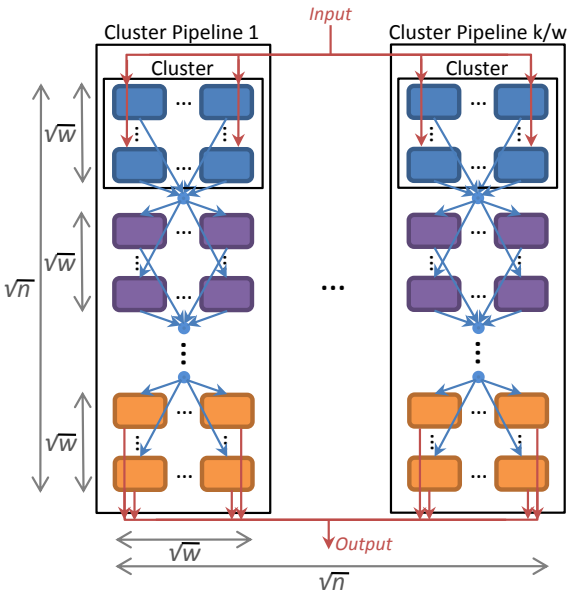


Figure 3-8: An alternate mapping for SMDP on a 2D array of n cores each cluster occupies a $\sqrt{w} \times \sqrt{w}$ block and communicating clusters are neighboring.

Now consider the alternate mapping given in Figure 3-8 where we have a $\sqrt{w} \times \sqrt{w}$ block of cores for each cluster and communicating clusters are neighboring, forming a pipeline. As before, for application input and output, in the worst case, $2k$ items will traverse any link.

For the duplication and routing scheme, the maximum link bandwidth will occur when we have 3 stages communicating in a vertical direction. Assume a *downward* vertical direction of items. The northern link of a bottom-most core in the middle stage will have w input items (from the upstream cluster) traversing it and $w - \sqrt{w}$ of its own stage's output items traversing it (the output items from its own row in the block do not traverse the link). Summing the I/O with the redistribution yields $(2k + 2w - \sqrt{w}) = O(w + k)$ □

3.5.4 Dimensionality of the Communication Substrate

It is interesting to note that the dimensionality of the communication substrate does not affect the maximum bandwidth of each mapping. The maximum link bandwidth of the 1D array is equal to the 2D array for each strategy. This is because item duplication is handled by the network and in the 2D array, the network adopts a dimension-ordered routing scheme (x then y). In the 2D array, when an item is duplicated to each core, the item first spreads out across one row of horizontal links but then traverses all of the vertical links. This unfairness causes the vertical links to be over-utilized. In the 1D case, when duplicating an item, the item must travel each link.

Our 2D routing scheme models the dimension-ordering routing schemes of some distributed-memory multicore processors [WGH+07, TLM+04]. A more fair (but complex) routing scheme would allow the 2D array to have a lower maximum link bandwidth for each strategy.

3.5.5 Normalizing to Inverse Throughput

In the streaming domain, performance is measured in terms of throughput, i.e., items per cycle. So far, the analysis in this section has not accounted for the fact that TMDP and SMDP output different number of items per steady-state. When we normalize each asymptotic bound by the accounting for the number of outputs, the two strategies are revealed to be equivalent in term of maximum link bandwidth.

For TMDP, the *MLB* bound is $O(n + mw)$. TMDP outputs n items per steady-state. We normalize the maximum link bandwidth to a single output (the inverse of throughput) by dividing by n yielding $O(\frac{mw}{n})$. Note that $w \leq n$ but that there is no bound on m so this quantity might be greater than 1.

For SMDP, the *MLB* bound is $O(k + w)$. SMDP outputs k items per steady-state. Normalizing this bound we get $O(\frac{w}{k}) = O(\frac{mw}{n})$, since $n = km$ (see Section 3.4.2 for why $n = km$). Note that $O(\frac{mw}{n}) \leq 1$ because $n = km$ and $w \leq k$. This does not mean that SMDP has a lower communication requirement than TMDP; the bound is less than or equal to one because we constrain the resulting scheduled graph that SMDP produces to have $n = km$. There is no such constraint for TMDP.

Intuitively, we are comparing the mw term of the TMDP non-normalized bound to the w term of the SMDP non-normalized bound. These terms represent the bandwidth required to perform the data reorganization stage(s). For TMDP, there are $m - 1$ stages that are serialized, each requiring bandwidth of w. In SMDP, the stages occur in parallel (in a pipeline), so m is not a factor. However, for SMDP, the number of outputs, k, is constrained by n and m. This constraint on how much SMDP can data-parallelize accounts for the asymptotic equality of the communication requirement of the strategies.

3.6 Memory Requirement and Cost

Although the normalized asymptotic communication requirement is equal, TMDP and SMDP differ in terms of the memory requirement each mapping imposes. The memory hierarchy attributes vary with each multicore design. No matter the specifics, performance suffers if the total memory required by some core for a parallel mapping exceeds the core's local memory. Access to an address not resident in local memory will force communication with the address's non-local home. This is problematic because a miss in the core's local memory could require access to a shared resource, serializing otherwise independent and parallel filters.

Quantitative cache modeling for parallel architectures is difficult and, thus, rarely attempted (a notable exception is [CGKS05], modeling shared memory). Our analysis will be mainly qualitative but it will elucidate the differences between the mapping strategies and the benefits of pipeline parallelism. Total memory requirement for a filter has four components: instructions, procedure stack, read-only state, and input buffer. Let $r(F_i)$ denote the total memory requirement for a filter F_i.

After a TMDP scheduling and mapping has been applied to the intermediate graph, each core is assigned what is essentially a copy of the entire intermediate graph (See Figures 3-4 and 3-6). The total memory requirement of each core for TMDP is then $\sum_{i=1}^{m} r(F_i)$. Depending on the application, this requirement could be quite large. As streaming programs move from small kernels to full applications, this requirement could be a barrier to performance scaling.

Assume that each filter of the intermediate graph, F_i, had an $r(F_i)$ less than a core's local memory. A TMDP schedule and mapping could have per-core memory requirement that is greater than the a core's local memory. Assuming a core's local memory caches main memory using a simple LRU policy, during a steady-state execution, a startup cost would be paid for each new time step (when a new fission product filter begins to execute). The startup cost accounts for the fact that the code and data of the filter are not resident in local-memory when it begins to execute. They were evicted during the execution of another filter. This startup cost can be amortized by increasing the scheduling multiplicity of each filter but that would also the increase buffering requirement of each filter [STRA05].

In an SMDP scheduling and mapping, each core is assigned a *single* filter that is a fission product of one of the filters of the intermediate graph. In the case where each filter of the intermediate graph, F_i, has $r(F_i)$ less than a core's local memory, a SMDP steady-state execution would proceed without any references to non-locally-resident addresses. In the steady-state, SMDP would not incur a startup cost associated with acquiring non-local addresses. The exploitation of pipeline parallelism in an SMDP schedule and mapping is the cause for the lower memory requirement of the strategy.

SMDP also has the benefit that the memory requirement may *decrease* as the number of cores increase. If n grows, then intermediate form of the application could have more filters (remember that for SMDP $m \leq n$). More filters in the intermediate form would imply each filter has less code because less fusion would occur. For TMDP the memory requirement does not depend on the number of cores.

In our performance model, memory capacity costs are represented by a term *MEM* that is added to the computation cost term, *COMP*, for TMDP only (see Section 3.8). This is because when we are comparing the two models, any memory capacity costs incurred by SMDP will also be incurred by TMDP. If a filter's memory requirement is greater than the size of a core's local memory the filter will incur the same memory-capacity cost in both strategies when it executes. Note that this is different from startup cost and is not explicitly modeled. The exact form of *MEM* is dependent on the target architecture.

3.7 Load Balancing

A component of the computation time is how well the computation can be load-balanced. The scheduler cannot expect the intermediate form of the application to consist of a load-balanced pipeline (we assumed this in Section 3.5 only to calculate a closed-form for maximum link bandwidth). Remember that the intermediate form is a result of repeated applications of the fusion transformation on the original graph of the application. It is difficult to fuse a complex application into a pipeline while balancing the load of the amalgamated filters [GTK+02]. Fusion transformations are restricted to merging contiguous sections of the stream graph. This is a problem with leveraging pipeline parallel in coarse-grained dataflow, but it is overcome by a SMDP strategy.

In TMDP, the load-balancing is optimal for a static strategy. Each filter in the intermediate form is fissed to all n cores. During each time step, each core is executing the same code on different input. Furthermore, in the streaming domain, most filters do not have data-dependent control flow. There is little variation in the computation requirement of cores [GTA06].

For SMDP two issues affect the load-balancing, (i) accuracy comparing the static work estimations, the $s()$'s, and (ii) symmetry between the application and the architecture. These are caveats with any scheduling strategy that leverages pipeline parallelism. SMDP calculates the amount to fiss a filter, F_i using the equation:

$$P_i = n * \frac{s(F_i)}{\sum_{j=1}^{m} s(F_j)} \tag{3.3}$$

Any inaccuracy in the estimation of the $s()$'s will cause inefficiencies in the steady-state schedule as pipeline stages will be unbalanced. Filters with less work will have to block while waiting for filter with more work to complete. Estimation could be achieved by profiling filters on the target architecture. However, any method for estimation could have inaccuracies given the complexity of the application and the architecture. Our model employs the term *MIS-EST* to represent the increase in computation cost due to the error of calculating and comparing the static work estimations of filters of the intermediate form. This term is included only for SMDP as a component of its computation cost. *MIS-EST* is dependent upon the specifics of the architecture, application, and the estimation method.

In most cases, when calculating P_i from Equation 3.3 we will be forced to round to an integral number. Rounding is caused by an asymmetry between the architecture and the application. There might be too few cores such that fission does not have the flexibility to balance the load between filters. This drawback of SMDP is termed *load inflexibility*.

As example, consider only the computation costs of a pipeline of 2 filters F_1 and F_2 such that, $s(F_1) = 10$ and $s(F_2) = 3$. All input and output rates are 1. If $n = 8$, then $P_1 = 6.2$ and $P_2 = 1.9$. We will round P_1 to 6 and P_2 to 2. We must schedule $F_{(2,1)}$ and $F_{(2,2)}$ to each execute 3 times per steady-state because there are 3x as many fission products of F_1 as F_2. Since SMDP executes as a pipeline, the steady-state computation time is defined by the filter with the most work. Each $F_{(1,i)}$, $1 \leq i \leq 6$, requires 10 units of work. There are 6 outputs so the throughput is 0.6.

However, in the case of TMDP, we will achieve a perfect load balancing. F_1 and F_2 are each fissed to 8 cores. Since the rates are matched, the multiplicities of each filter can remain 1. The steady-state requires 13 units and produces 8 outputs, for a throughput of 0.615.

Our analysis does not include an explicit term for the load imbalance inflexibility of SMDP. Instead, this inflexibility manifests itself in the calculation of the work estimation of the fission products of SMDP, the $s(F_{(i,j)})'s$. The work estimation of fission products is used in the estimation of the computation cost of both SMDP and TMDP in Section 3.8. The example above is a sample of the full analysis.

3.8 Computation Cost

$COMP_{TMDP}$ is equal to the sum of the work estimates of all the time steps. Each time step's work is defined by one of its fission products because each product is executing the same code. To this sum we add the memory capacity cost described in Section 3.6:

$$COMP_{TMDP}(G) = \sum_{i=1}^{m} s(F_{(i,1)}) + MEM \tag{3.4}$$

$\mathit{COMP}_{\mathit{SMDP}}$ is calculated as the pipeline stage with the maximum amount of work. The work of a pipeline stage can be defined by the work of its first fission product for reasons cited previously. Added to the filter with the maximum work is the penalty for relying on static work estimation, *MIS-EST* (see Section 3.7):

$$\mathit{COMP}_{\mathit{SMDP}}(G) = \max_{i=1...m} s(F_{(i,1)}) + \mathit{MIS\text{-}EST} \tag{3.5}$$

We can refine the throughput equation for architectures with computation and communication concurrency (Equation 3.1) to account for the similarities of the strategies:

$$\mathit{THRU}(G, \mathit{COMP}_G, \mathit{MLB}_G) = \frac{P_m \cdot u(F_{(m,1)})}{\max(\mathit{COMP}_G, \mathit{MLB}_G)} \tag{3.6}$$

The numerator calculates the number of outputs. Remember that $u(F_{(m,1)}) = u(F_{(m,2)}) = \ldots = u(F_{(m,P_m)})$. The denominator estimates the total number of cycles required for the steady-state.

3.9 Discussion

One of the surprising conclusions of this chapter is that SMDP and TMDP schedules have the same asymptotic communication cost for near-neighbor communication networks. Initially, we expected SMDP to have a reduced of communication requirement compared to TMDP. Of course, SMDP has other important advantages over TMDP (and vise versa) but this equivalency simplifies any comparison of the two strategies. Thus, when comparing SMDP and TMDP strategies for a given application targeting a specific architecture, we compare the memory capacity cost of TMDP (*MEM*) to the load-balancing inflexibility and load estimation error (*MIS-EST*) of SMDP. The load-balancing inflexibility affects the calculation of the work of the fission products in Equation 3.5. Our model cannot reveal a concrete conclusion without developing more detailed values or equations for the terms *MIS-EST* and *MEM*. However, the model does allow us to predict the performance of the resulting schedules at a high-level considering the characteristics of an application. The importance of our model in its current form is that it predicts the scalability of the scheduling policies across applications with respect to the maturity of multicore architectures.

As Moore's Law continues, one possible target for the increasing transistor density is a doubling of cores with every multicore generation. Can we expect the two scheduling policies to scale indefinitely? Note that at first glance it looks like we can always create parallel work by fissing filters. To determine scalability we need to analyze Equation 3.1. Any factor that could reduce the growth of throughput as the number of cores increases could potential limits scalability. The numerator, the number of outputs per steady-state, will only increase with the number of cores as any scheduler would have more cores for parallelizing the application.

Maximum link bandwidth, *MLB*, appears in the denominator so we must analyze how it could increase with the number of cores. In Section 3.5, we calculate a closed form solution for the normalized maximum link bandwidth for a restricted version of the intermediate form of the application. Given a single application, w, the width of communication, was the only factor that could grow with n. If the width of communication is kept constant, meaning it does not grow with n, our model demonstrates the scalability of the strategies will not be limited by communication costs. If w grows with n and the computation cost remains constant, once the communication cost

begins to outweigh the computation cost, scaling benefits will begin to diminish. The width of communication is dependent upon the rates of the application and the fission implementation of the compiler. It behooves compiler writers to optimize this transformation. Naïve implementations will limit scalability.

The computation cost of the steady-state, *COMP*, also presents in the denominator and it may increase in certain situations. If the trend is towards simpler, more efficient cores over successive generations of multicore architectures, the computation cost for a steady-state may increase. This increase in computation cost may be larger for a TMDP schedule because of its higher memory requirement. Simpler cores may have smaller local memory capacities[1] which means more applications might have a memory requirement that exceeds the capacity.

The shortcoming of SMDP, its load-balancing inflexibility, is eased as the number of cores increases. A SMDP scheduler would still need to accurately estimate the load of the filters, but it would have more flexibility in load-balancing the pipeline stages. The memory requirement of SMDP is also reduced as cores scale. TMDP shortcoming is not eased as the number of cores grow. However, it is also an option to use the increasing transistor budget to design more powerful cores. As the local memory capacity of a core increases, fewer applications would incur the memory capacity cost of TMDP (*MEM*). As mentioned above, though, this option goes against recent multicore design progressions.

The key difference between the strategies is that SMDP leverages pipeline parallelism to reduce the per core memory requirements as compared to TMDP. For SMDP, the code and data of the application is split among the cores. There is some duplication of code and data when data-parallelizing, but this is required for load-balancing. Current generations of commodity multicore architectures often preclude the use of SMDP because they have few cores (2-8 currently). However, as multicores scale in the number of cores the benefits of SMDP could allow it to displace TMDP as the traditional scheduling approach.

The remaining chapters of this thesis present empirical results for different points within the two mapping strategies discussed here. We will refer back to this analysis when analyzing results of our implementations.

3.10 Related Work

Since multicore parallel architectures are securely entrenched as the commodity processor architecture, it is surprising to find a lack of recent literature that attempts to model current workloads targeting these parallel systems. Parallel modeling papers of the past, of which there are many, deal with different hardware attributes and application classes. For a survey, see [MMT95]. Many models employ queuing theory to predict the behavior of dynamic parallel systems [ML90, CG02], but our work differs in that we model a static system. This allows our model to develop closed-form solutions for communication and directly compare different scheduling policies on the same application class.

Elegant analytical cache models exist, though mainly to motivate the design of hardware caching structures [AHH89, CFKA90]. Our work does not require the same amount of detail in

[1] At the very least, recent history is demonstrating that core local memory is stable. Intel's successive multicore lines of Core 2 (Penryn, Wolfdale and Yorkfield), Core 2 Quad (Allendale and Conroe), and i5/i7 (Lynnfield and Clarkdale) have equal total L1 cache capacity per core.

cache modeling as currently we are interested in asymptotic behavior. Also, if our model is to be incorporated into a compiler, profiling would suffice to calculate any memory properties. Clement and Quinn develop an analytic model of parallel systems exploring many of the same issues as our work [CQ93]. However, their model remains very high-level and focuses on the relationship of parallel to non-parallel portions of the application. [SRG94] makes the case for the importance of modeling communication in parallel algorithms but fails to give an analytical model, concluding with examples of complex algorithms. Crovella et al. describe the interaction of communication and computation by modeling a parallel quicksort algorithm to determine the appropriate amount of parallelism for the algorithm/architecture paring [CL94].

In addition to our previous work [GTK+02, GTA06], many others have explored scheduling and mapping techniques for computation graphs, specifically SDF, targeting parallel architectures [PL95, PBL95, MSK87, KA99b, EM87]. Our work differs in that we develop a rich analytical model for a constrained application space. We can then compare our techniques in terms of communication, memory, and load balancing. Furthermore, our model allows us to forecast the scalability of techniques. We are not alone in promoting pipeline parallelism. Of course, pipeline parallel is pervasive at the hardware level. Many software systems have been developed that leverage pipeline parallelism [TCA07, DFA05, ORSA05]. Our focus, though, is to promote pipeline parallelism as a means of extreme scalability through memory requirement reduction.

3.11 Chapter Summary

In this chapter we develop a model for a class of stream programs targeting a general multicore architecture. The model provides a function that models the throughput of a data-parallel streaming application. We employ the model to compare the scalability of two scheduling strategies, time-multiplexed data parallelism (TMDP) and space-multiplexed data parallelism (SMDP). We develop the inputs to the model's throughput function for the scheduling strategies by considering how they compare in terms of communication cost, load-balancing effectiveness, and memory requirement.

TMDP parallelizes each filter of the original application to all cores; the executing stage of the application varies over time. SMDP leverages pipeline parallelism to create a load-balanced pipeline of application stages. The full application executes concurrently in a pipeline with stages occupying different numbers of cores depending on their load. Our model demonstrates that TMDP offers more effective load balancing, while SMDP has a lower memory requirement. A surprising conclusion is that the strategies have equivalent asymptotically communication cost when considering near-neighbor communication networks. The asymptotic communication cost dependents on the properties of data-parallelization of the filters composing the application.

The properties of SMDP are more congruent with recent trends in multicores architectures that favor increasing the number of simpler, more efficient cores. As we increase the number of cores, the load-balancing effectiveness of SMDP increases and the memory requirement of SMDP decreases. Conversely, the memory requirements of TMDP remain unaltered as we increase the number of cores. These findings demonstrate that pipeline parallelism should become more popular as multicore architectures mature.

Chapter 4

The StreamIt Core Benchmark Suite

In this chapter we describe the StreamIt Core Benchmark suite. The suite is used to evaluate the techniques covered in this dissertation. The benchmark suite in comprised of 9 real-world applications and 3 kernels, ranging in size from 60 lines to 4233 lines. Detailed descriptions for each benchmark is given. For each benchmark, we provide annotated stream graphs that give the work distribution for the filters of the graph. A detailed analysis of the benchmark suite is presented that includes work distribution, percent of stateless computation, task parallelism, counts of peeking filters, and computation to communication ratio.

4.1 The StreamIt Core Benchmark Suite

The techniques covered in later chapters in this thesis are evaluated in the context of the StreamIt language on a set of benchmarks developed by researchers both inside and outside of the StreamIt group. The collection is named the "StreamIt Core Benchmark suite" to denote that it is a subset of the 67-program "StreamIt benchmark suite" described in [Thi09]. The StreamIt Core benchmark suite was first introduced in 2006 [GTA06], and has since gained acceptance as a standard experimentation subject for research in streaming programs [KM08, UGT09b, HWBR09, HCK+09, NY03, UGT09a, ST09, HCW+10, CRA09, CRA10].

An overview of the StreamIt Core benchmark suite is given in Table 4-1. The suite contains 12 programs including 3 kernels (FFT, DCT, and TDE) and 9 realistic applications. Benchmarks range in size from 60 lines (DCT) to 4233 lines (MPEG2Decoder), with a total of 6853 non-comment, non-blank lines in the suite. The Information Sciences Institute developed TDE, researchers at UC Berkeley [NY04] developed FFT and bitonic sort, and Vocoder was implemented with support from Seneff [Sen80]. Other benchmarks were adapted from a reference implementation in C, Java, or MATLAB.

4.2 The Benchmarks

In this section we provide a description and the StreamIt graph for each benchmark in the suite. The stream graph reflects the structure of the original input program, prior to any transformations by the compiler. In practice, the compiler canonicalizes each graph by removing redundant synchronization points, flattening nested pipelines, and collapsing data-parallel splitjoins. This canonicalization is disabled to illustrate the programmer's original intent.

Benchmark	Description	Author	Libraries Used	Lines of Code[1]
Static apps (12):				
BitonicSort	Bitonic sort (fine-grained, iterative)	Mani Narayanan	--	121
ChannelVocoder	Channel voice coder	Andrew Lamb	--	135
DCT	NxN IDCT (IEEE-compliant integral transform, reference version)	Matthew Drake	--	60
DES	DES encryption	Rodric Rabbah	--	567
FFT	512-point FFT (coarse-grained)	Michal Karczmarek	--	116
Filterbank	Filter bank for multi-rate signal processing	Andrew Lamb	--	134
FMRadio	FM radio with equalizer	*multiple*	--	121
Serpent	Serpent encryption	Rodric Rabbah	--	550
TDE	Time-delay equalization (convolution in frequency domain)	Jinwoo Suh	FFT	102
MPEG2Decoder	MPEG-2 Block and Motion Vector Decoding	Matthew Drake	--	4233
Vocoder	Phase vocoder, offers independent control over pitch and speed (Seneff, 1980)	Chris Leger	--	513
Radar	Radar array front end (fine-grained filters)	*multiple*	--	201

[1] Only non-comment, non-blank lines of code are counted. Line counts do not include libraries used.

Table 4-1: Overview of StreamIt Core benchmark suite.

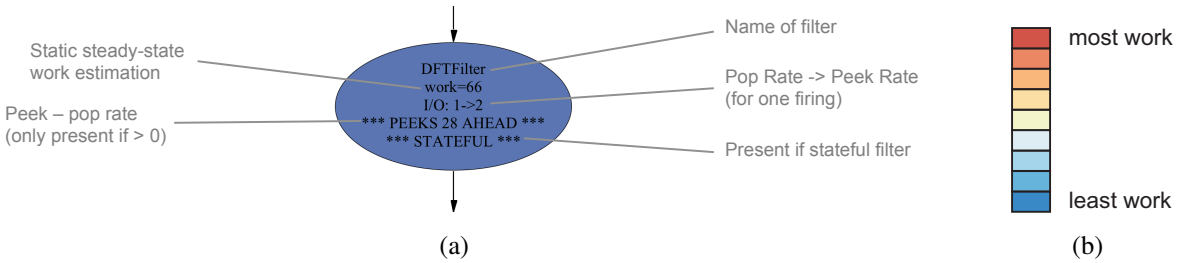


Figure 4-2: Filter annotations for the stream graphs presented in this chapter: (a) filter text describes rates, peeking, work estimation, and presence of state, (b) filter color indicates their approximate amount of work relative to other filters in the same program.

In the stream graphs, each filter is annotated with the following information (see Figure 4-2):

- The filter name.
- The number of items pushed and popped per execution of the filter.
- The estimated work (number of cycles) per execution of the filter.
- Peeking filters are annotated with the number of items peeked (but not popped) per execution.
- Stateful filters are annotated as such.

Filters are also colored to indicate their approximate amount of work relative to other filters in the same program. The heaviest and lightest filters in a program are assigned fixed colors, and intermediate filters are colored on a linear scale between the two (see Figure 4-2(b)). Work estimates are gathered statically and may differ by 2x or more from actual runtime values. Work estimation is implemented as a pass in the StreamIt compiler that iterates over the statements and expression of the work function of each filter. Since most streaming applications do not include data-dependent control flow, and due to the static I/O rates in StreamIt, most loops within `work` can be unrolled, allowing a close approximation of the actual cycle count for the `work` function. Mathematical expressions and library function calls are assigned cycle latencies based on empirical experiments of the Raw microprocessor. The work estimation pass requires a steady-state schedule

so that the work of filters can be accurately compared. Work estimates are not available for the graphics benchmarks as a whole because of the presence of dynamic rates. In Chapter 5.9 we will see how we can estimate the work of filters within static rate sub-components of an application with dynamic rates.

4.2.1 BitonicSort

The BitonicSort benchmark (see Figure 4-3) implements a high-performance Batcher's bitonic sort network for sorting power-of-2 sized key sets. The sort of n keys is accomplished using $\log(n) \cdot (\log(n)+1)/2$ comparator stages, where for each stage there are $n/2$ comparators. For the entire network, there are $\Theta(n \cdot \log(n)^2)$ comparators [bit, Knu98]. The benchmark is instantiated with $n = 8$.

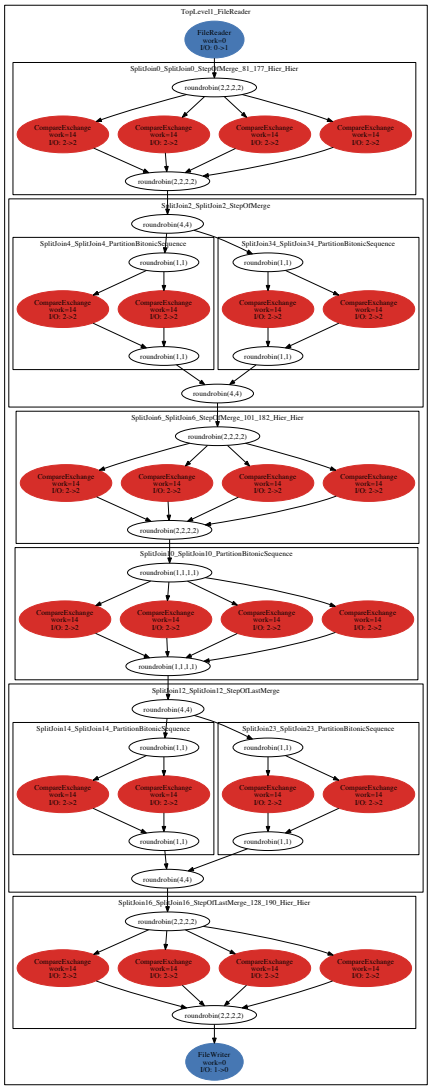


Figure 4-3: StreamIt graph for BitonicSort.

4.2.2 ChannelVocoder

The ChannelVocoder benchmark (see Figure 4-4) is the analyzer portion of a *source-filter model* speech coder that uses short-time Fourier analysis to code the *filter* portion of the speech model. The ChannelVocoder first includes a lowpass filter to filter out high frequency noise. The resulting signal is duplicated to a pitch detector and a bank of bandpass filters that split the incoming speech signal into different frequency bands (16). The pitch detector determines if a frame is voiced by first center clipping the signal, then pitch is determined using the autocorrelation function. In the benchmark the pitch window is 100 samples of the signal. The envelope of each band is determined using a magnitude and lowpass filtering method. Each band is then decimated (in this case by dropping 49 out of every 50 elements of the signal) to achieve a reduction in bit rate [Gre].

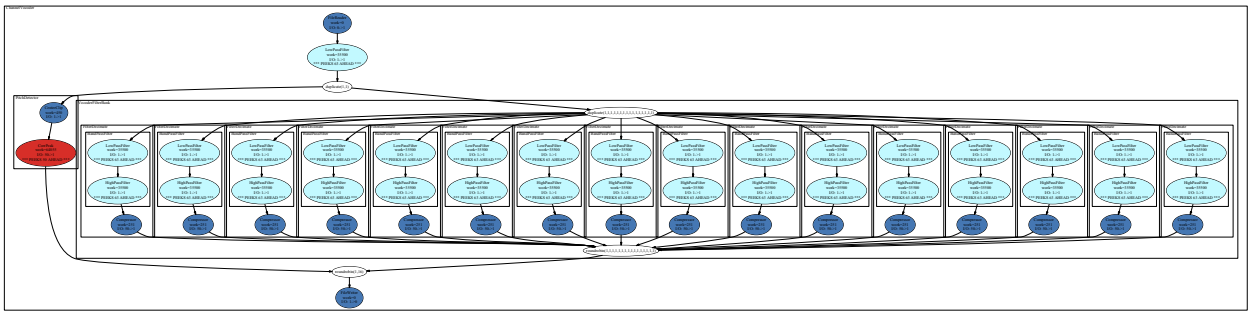


Figure 4-4: StreamIt graph for ChannelVocoder.

4.2.3 DES

The DES benchmark (see Figure 4-5) implements the Data Encryption Standard [Cop94] block cipher. DES is based on a symmetric-key algorithm that uses a 56-bit key. Until recently, DES was a widely-used encryption algorithm, selected by the US government as a standard and secure scheme. DES as adopted by the US government requires 16 stages of processing (rounds) but the StreamIt benchmark performs 4 rounds. The algorithm operates on 64 bits at time, a block. The StreamIt version use a 32-bit integer to represent each bit because the multicore backend does not support a bit type.

Before the rounds, the block is first permuted on the bit level. This is performed inside a single filter by pushing the value at a peek index stored in a lookup table that encodes the permutation. Next the block is divided into 2 halves, and processed via a criss-crossing scheme that is evident in the StreamIt graph structure. Since some data has to pass through untouched to the next round, an identity filter and a decimation filter (`nextL`) is present. Inside each round is a Feistel function which performs 4 stages of processing: expansion, key mixing, substitution, and permutation. The Feistal function is performed over 5 filters in the benchmark, and each of the filters makes heavy use of lookup tables and the peek construct. All key schedules are calculated once at initialization time. Finally, after the last round, another permutation is performed (the inverse of the pre-round permutation).

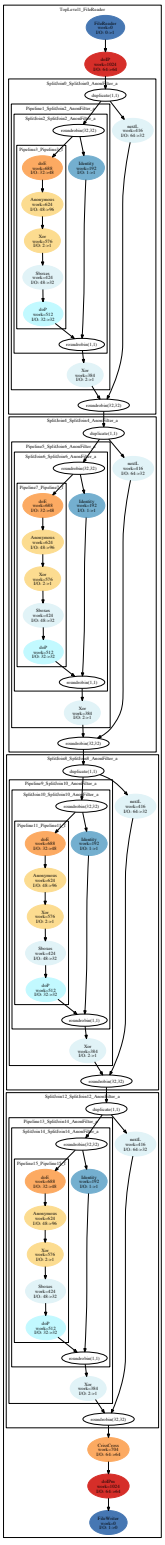


Figure 4-5: StreamIt graph for DES.

4.2.4 DCT

The DCT benchmark (see Figure 4-6) implements a 2-dimensional inverse DCT. DCT transforms a 16x16 signal from the frequency domain to the signal domain using an inverse Discrete Cosine Transform in accordance with the IEEE specification for a 2-dimensional 8x8 iDCT. This iDCT specification is used in both JPEG and MPEG-2 coding (on 8x8 blocks). Data is streamed to the benchmark by rows (x dimension). The DCT first performs the iDCT on the Y dimension by splitting the columns using a round-robin splitter with weight 1. So each iDCT in the first splitjoin operates on its own column. The data is reorganized via a round-robin joiner with weight 1. The data is next split by row using a round-robin splitter with weight 16 so that each iDCT operates on its own row. Finally the stream is reassembled by a round-robin joiner with weight 16.

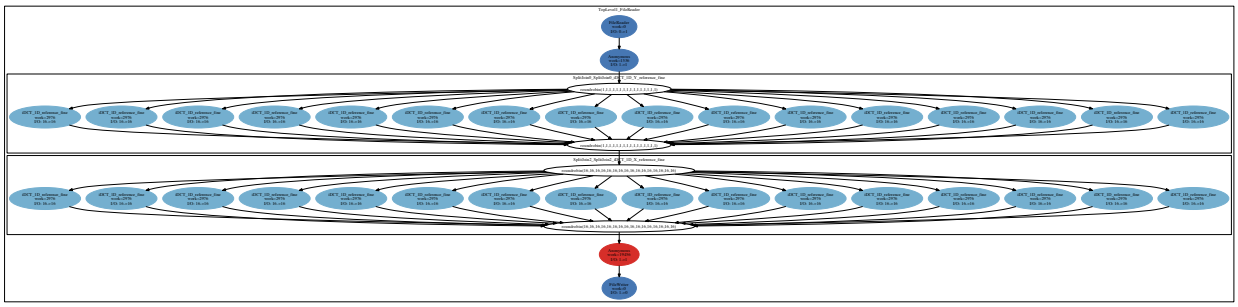


Figure 4-6: StreamIt graph for DCT.

4.2.5 FFT

The FFT benchmark (see Figure 4-7) implements a radix-2 decimation-in-time fast Fourier transform using a version of the Cooley-Tukey FFT algorithm for problem size of $N = 2^M$ [TTRT00]. In the FFT benchmark, $N = 512$. The bit-reversal reordering of the input is achieved via a pipeline of $M - 2$ filters that consecutively reorder smaller portions of the input $(2^{(m)}, \ldots, 2^2)$). The N-size DFT is then divided into $M - 1$ smaller DFTs by repeatedly doubling the size of the DFTs (the Danielson-Lanczos lemma). In the DFTs, Euler's formula is employed to interpret the interpolating trigonometric polynomial as a sum of sine and cosine functions. The output is left in bit-reversed order as many applications can work on data in this order.

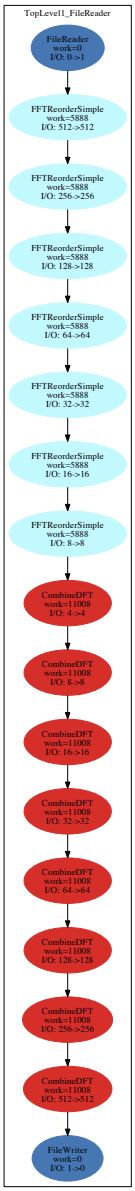


Figure 4-7: StreamIt graph for FFT.

4.2.6 Filterbank

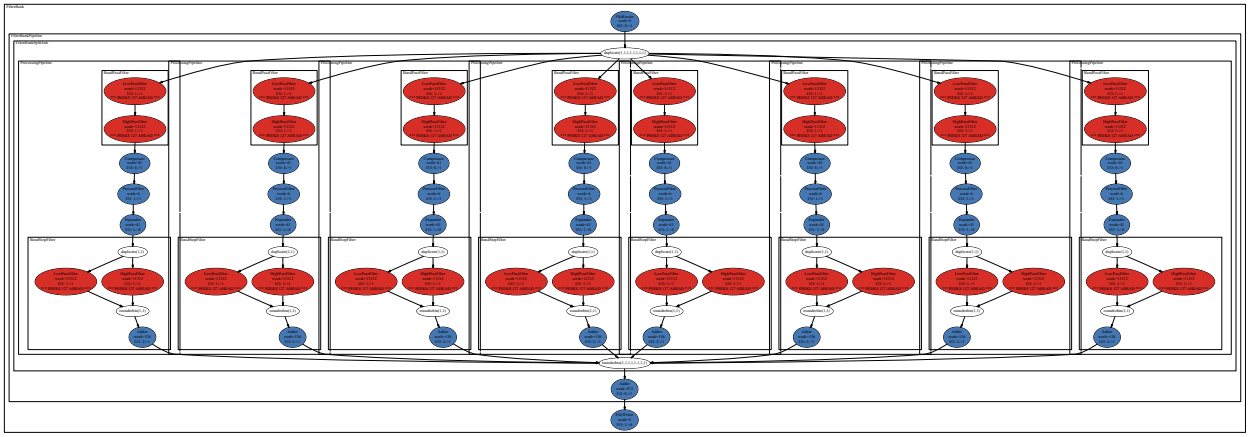


Figure 4-8: StreamIt graph for Filterbank.

The Filterbank benchmark (see Figure 4-8) implements a multi-rate signal decomposition processing block common in communications and image processing. The incoming signal is duplicated to 8 processing branches, each operating on their own frequency sub-band. Each branch includes a bandpass filter, a decimation filter, a process filter, an upsampler, and finally a bandstop filter. The work of the process filter is specific to the application of the filterbank, and in this case the process filter performs the identity function. The bandpass filter is implemented as a lowpass filter cascaded with a highpass filter. The bandstop filter is as a lowpass filter and highpass filter with their outputs combined. All lowpass and highpass filters have 128 taps.

4.2.7 FMRadio

The FMRadio benchmark (see Figure 4-9) implements an FM radio with multi-band equalizer. The input passes through a demodulator to produce an audio signal, and then an equalizer. The equalizer is parametrized by the number of bands, in this case 6. The equalizer is implemented as a splitjoin with a number of band-pass filters; each band-pass filter is implemented as the difference of a pair of low-pass filters (each with 128 taps). Finally the signal is combined via an adder filter. While it is possible to derive more efficient representations of an Equalizer, in this work we focus on the natural expression of the algorithm. Automatic elimination of redundant operations can be done using linear analysis [LTA03].

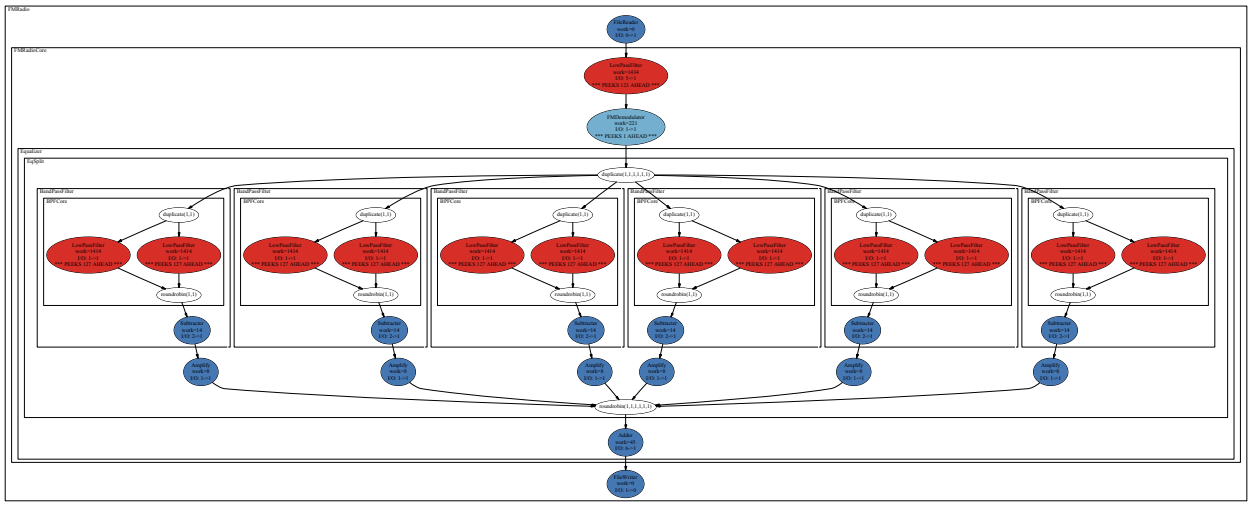


Figure 4-9: StreamIt graph for FMRadio.

4.2.8 Serpent

The Serpent benchmark (see Figure 4-10) implements the Serpent block cipher [ABK]. The Serpent algorithm was submitted as a replacement for the DES standard in the late 1990s (it was not selected as the replacement). The algorithm is similar in structure to DES (see Section 4.2.3). The submitted version of Serpent has 32 rounds with a variable-length key; the StreamIt version employs 8 rounds with a 256-bit key. The algorithm includes an initial permutation before the rounds, and a final permutation after the rounds.

Each round consists of a mixing operation, pass through S-boxes, and a linear transformation (omitted for the last round). The mixing operation XORs the round's input with the round's key. Each round has its own key that is computed from the overall 256-bit key at initialization time. The output of the mix is then permuted in the round's Sbox which operates on 4-bits at a time. Each Sbox performs some bitwise operations and shifts to implement the permutation. The final action for a round is a linear transformation (`rawL`) that is implemented via a series of peek, push, and exponent operations.

Figure 4-10: StreamIt graph for Serpent.

4.2.9 TDE

The TDE benchmark (see Figure 4-11) implements the Time Delay Equalization phase from the Ground Moving Target Indicator (GMTI) application [Reu04]. TDE essentially performs frequency domain convolution. TDE operates on data coming from 6 sensors with 36 input samples per sensor per quantum (N), there are 15 pulse repetitions per sample (M). The benchmark first performs a $N \times M$ transpose, followed by a 64-way FFT, decimation, a 64-way inverse FFT, decimation, and transpose. The FFT implementation is described in more detail in Section 4.2.5.

Figure 4-11: StreamIt graph for TDE.

4.2.10 MPEG2Decoder

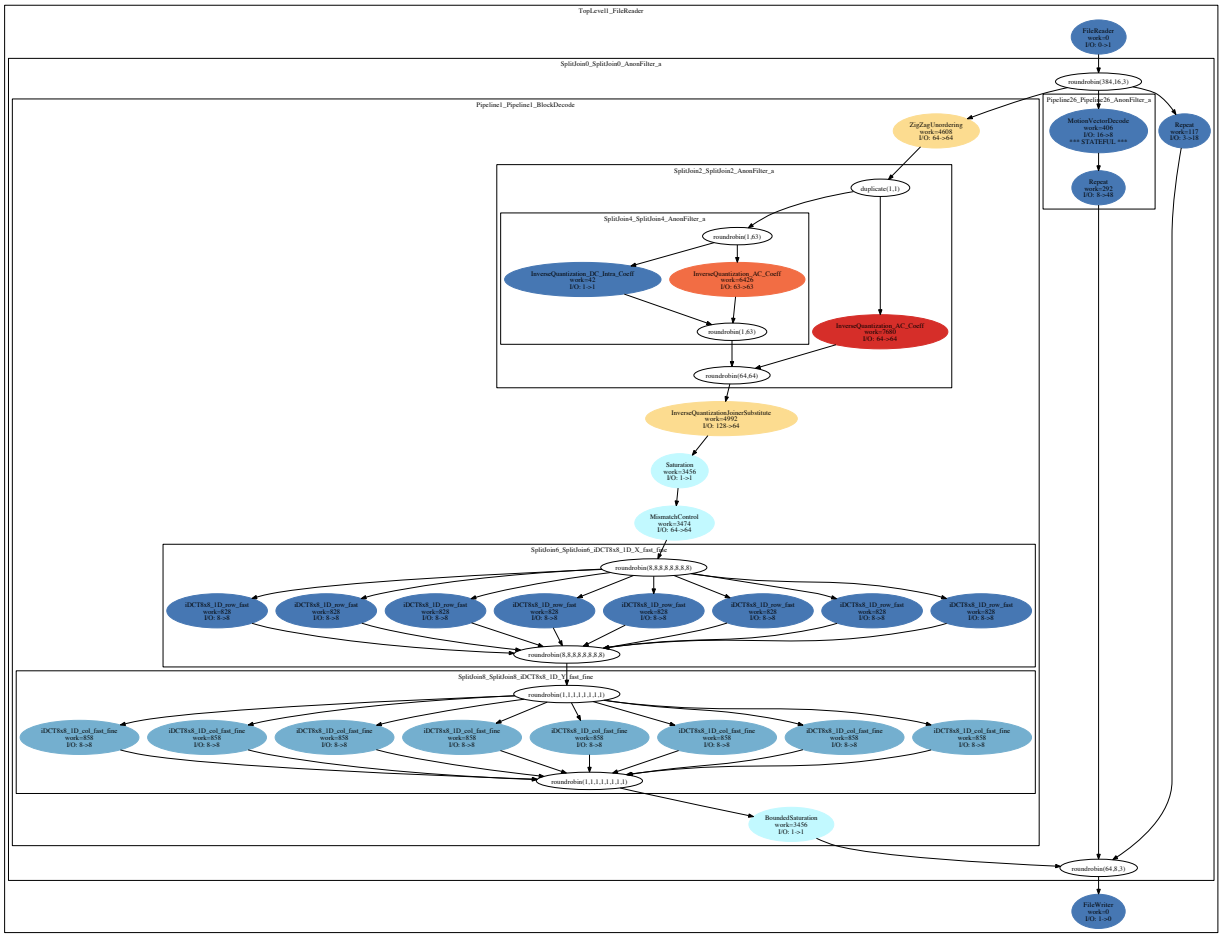


Figure 4-12: StreamIt graph for MPEG2Decoder.

The MPEG2Decoder benchmark (see Figure 4-12) implements the block decoding and motion vector decoding components of an MPEG-2 [MDH+06] decoder. These two components account for approximately one-third of the computation of the entire MPEG-2 decoder. The movie size for this benchmark is 352x240, the chroma format is 4:2:0. The input to the benchmark is the uncompressed stream of (i) picture types, (ii) motion vectors and (iii) zigzag-ordered, quantized, and frequency-transformed blocks.

The motion vector decoding and the block decoding are performed in parallel as streams in the outermost splitjoin. This splitjoin also includes a `Repeat` filter that passes picture type information through this stage (and duplicates it) for later processing.

The motion vector decoder reconstructors a motion vector from a variable length code vector, an integer vector, and the previous motion vector. The motion vector decoder in MPEG2decoder handles only progressive motion vectors (as opposed to interlaced). The motion decoder is stateful because it must remember the previously decoded motion vectors. After decoding the vectors are duplicated for processing in later stages.

The input to the block decoder is the zigzag ordered, quantized blocks of the frames. The matrices representing the blocks were encoding using a zigzag pattern. The block decoder first rebuilds the quantized blocks by reestablishing the original matrices in the filter `ZigZagUnordering`. The inverse quantization is handled by the next 4 filters.

In MPEG-2, inverse quantization of blocks is handled differently for I frames versus P and B frames. In MPEG2Decoder, the implementation performs both types of quantization in parallel on a block, and leaves it up to the filter `InverseQuantizationJoinerSubstitute` to choose the coefficients that were correctly calculated and drop the others. In this filter, MPEG2Decoder assumes that all frames are I frames. In the full MPEG-2 decoder implementation in StreamIt, this decision is sent via a message to the filter. The coefficients resulting from the inverse quantization are then saturated. Finally, the block is filtered through `MismatchControl` which sums the coefficients of the block and may add to the last coefficient (this is included to reduce the drift between different iDCT algorithms).

The next step for block decoding is an 8x8 2D inverse discrete cosine transform (iDCT). MPEG2Decoder does not use an IEEE compliant iDCT, instead it employs a fast, specialized 8-way iDCT implementation. The 2D iDCT is split into row and column iDCTs. The author of MPEG2Decoder introduced explicit data-parallelism at this stage by duplicating the row and column iDCTs 8 times and placing them in splitjoins. All the row DCT's are performed in parallel, and all of the column iDCTs are performed in parallel. The data is split and joined using a pattern similar to the DCT benchmark (Section 4.2.4). Finally, the inverse transformed values are saturated using a bounded saturation filter which employs a lookup table.

4.2.11 Vocoder

The Vocoder benchmark (see Figure 4-13) implements a phase vocoder which can scale both the frequency and time domains of audio signals. Vocoder was implemented with support from Seneff [Sen80]. A phase vocoder is employed by pitch-corrected software, widely used in musical production. Vocoder decodes a .wav file, and reduces the pitch of the signal by 0.6. The input signal is first passed to a filterbank containing 15 adaptive DFT filters to obtain a magnitude spectrum and a phase spectrum. In the adaptive DFT, each new output is obtained via incremental adjustments of the preceding output. The DFT coefficients are then converted from rectangular to polar coordinates.

The spectral envelop of the magnitude spectrum is estimated by an FIR smoothing filter. The excitation magnitude spectrum is found via a time-domain deconvolution. The spectral envelope is interpolated by a factor of 1/0.6. To prevent frequency aliasing, the unmodified spectral envelope is combined with the interpolated spectral envelope as phantom excitation samples.

The phase spectrum of each DFT filter is unwrapped in time (a stateful computation) and first-differenced to obtain the instantaneous frequency. The pitch is modified by modifying the first derivative of the unwrapped phases (ConstMultiplier and Accumulator filters). The Accumulator is stateful filter because of the granularity of the implementation.

Output is then re-encoded back into the .wav format in the resynthesis stage of the phase vocoder. This is achieved by performing a polar to rectangular conversion. Next the iDFT is calculated while unrotating the samples and applying a window to the resulting sample block. Finally, overlap-add consecutive frames. This is all performed at fine-granularity in the SumReals container.

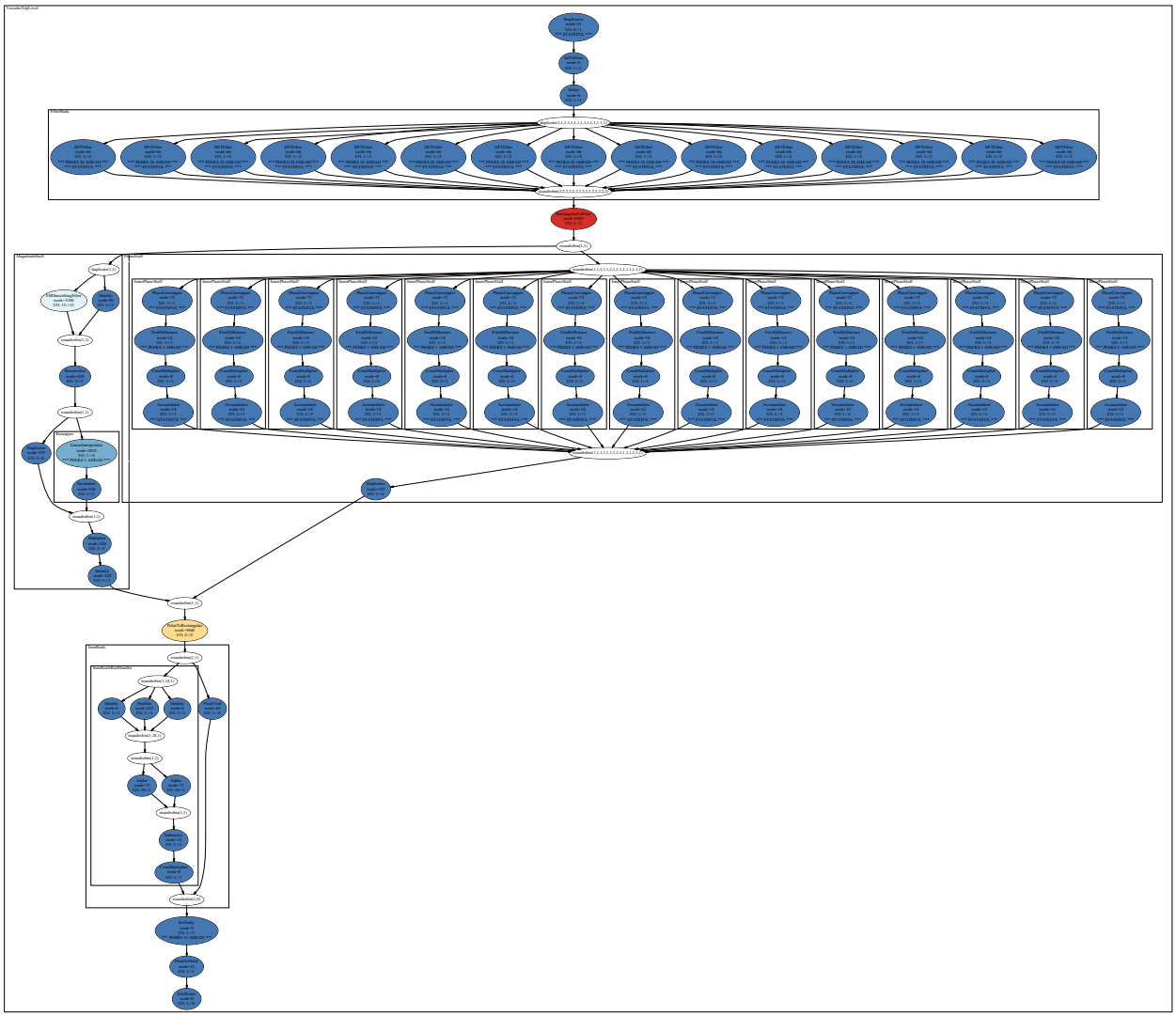


Figure 4-13: StreamIt graph for Vocoder.

4.2.12 Radar

The Radar benchmark (see Figure 4-14) implements a radar array frontend with separate components for input reduction, beamforming, and detection. This benchmark is an example application from MIT's Lincoln Laboratory's Polymorphous Computer Architecture (PCA) project [Leb01]. The Radar benchmark is instantiated with 12 input signals (channels) and 4 beams.

In Radar the input is not read from a file, instead it is created by starting from a known outcome and working backwards. The input stage is given the channel that includes the target and other parameters. Using these values it creates the data matrix for each channel. The input reduction stage is achieved by applying two different decimating FIR filter operations to the input data. Both FIR filters in this stage have 64 taps. The second FIR filter in this stage decimates its input by 2.

Next, the output of all the channels is collected and duplicated to the 4 beamform and detection stages. The beamforming filter organizes the input data into a set of beams; each beam represents

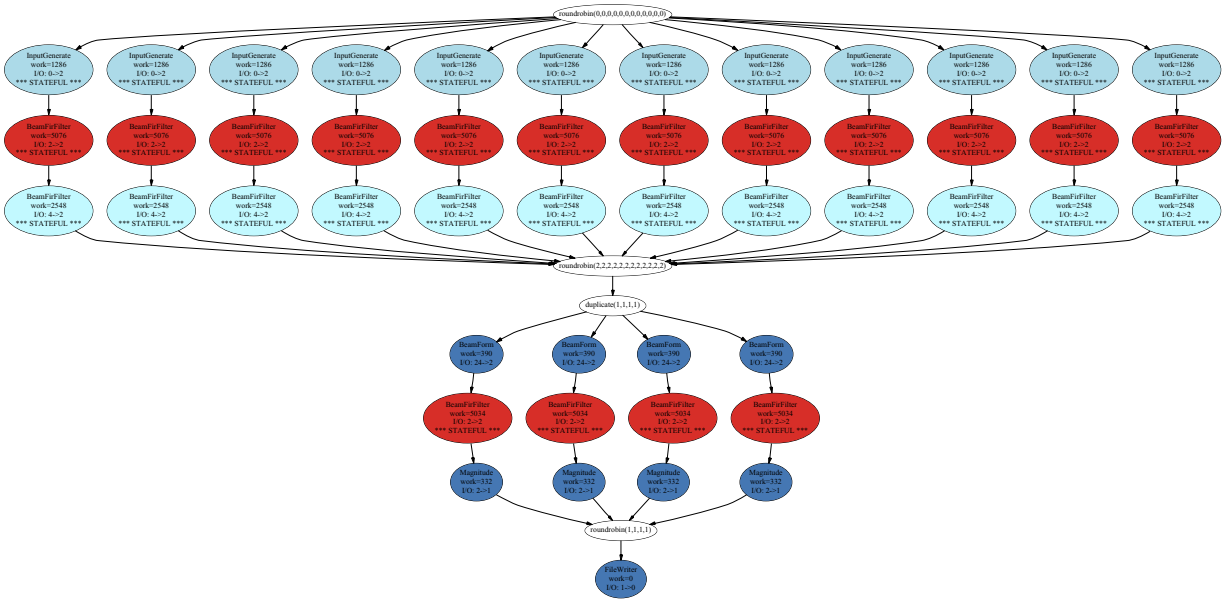


Figure 4-14: StreamIt graph for Radar.

a particular direction. The beamform filter multiplies the filtered input by a beamforming matrix (both real and imaginary components). The detection stage is implemented via an FIR filter and a magnitude filter. The response of the FIR filters match the expected response of the targets they are detecting. The implemented differs from the PCA specification which calls for a more sophisticated detector.

Each FIR filter in the Radar benchmark includes state and therefore cannot be data-parallelized. The state is introduced because Radar's author implemented the FIR filters at fine granularity. Each FIR filter has its own circular buffer for input data (real and imaginary components). The code maintains these circular buffers in order to simplify decimation and to reset the buffers between each input set. The state is not inherent to the computation being performed, in fact we have a version of Radar that is stateless. We include the stateful version here because Radar's author originally conceived the implementation in this form.

Benchmark	Parameters and default values	Parameterized:		Filter Execs per Steady State		
		Graph	I/O Rates	Min	Mode	Mode Freq.
Static apps (12):						
BitonicSort	number of values to sort (8)	✓		1	1	92%
ChannelVocoder	number of filters (16); pitch window (100); decimation (50)	✓	✓	1	50	66%
DCT	window size (16x16)	✓	✓	1	1	89%
DES	number of rounds (4)	✓		1	1	62%
FFT	window size (512)	✓	✓	1	1	12%
Filterbank	bands (8); window size (128)	✓	✓	1	8	64%
FMRadio	bands (7); window size (128); decimation (4)	✓	✓	1	1	97%
Serpent	number of rounds (32); length of text (128)	✓	✓	1	128	43%
TDE	number of samples (36); FFT size is next power of two	✓	✓	1	15	24%
MPEG2Decoder	MPEG-2 Block and Motion Vector Decoding	✓	✓	1	6	21%
Vocoder	pitch & speed adjustments, window sizes	✓	✓	1	1	88%
Radar	channels (12); beams (4); decimation rates; window sizes	✓	✓	1	1	49%

* Statistics represent properties of complete programs, in which libraries have been inlined into caller.

Table 4-15: Parametrization and scheduling statistics for StreamIt Core benchmark suite.

4.3 Analysis of Benchmarks

Each of the benchmarks represents a specific assignment of parameter values for the computation in which they implement. We include one assignment of those parameters in the suite. The previous section details the exact parameter assignments for each benchmark. Table 4-15 summarizes the parameter assignments and demonstrates that for all of the benchmarks, the parameter values affect the structure StreamIt graph, often by altering the length of pipelines or the width of splitjoins. For example, the length of the main pipeline in FFT is dictated by the window size of the FFT. For all but 2 of the benchmarks, the parameter assignment affects I/O rates. Changes to the rates of a benchmark may also affect the schedules and the balance of work across filters.

We have selected one assignment of parameters per benchmarks in order to employ the benchmarks to evaluate the techniques of this dissertation. Our goal in selecting parameter values for these benchmarks is to represent real-world applications of the benchmarks. However, for the two encryption benchmarks, DES and Serpent, we decreased the strength of the encryption algorithm (thus reducing the number of filters in the graph) because of memory footprint issues with an early published version of our compiler. The parameter values remain because of this historical reason. It is recommended that future work restore DES and Serpent to their prescribed encryption strengths.

Table 4-15 also defines statistics for the steady-state schedule of each benchmark. 6 of the 12 applications have steady-state schedules where greater than 50% of filters have multiplicity 1. This demonstrates that for many applications, rates are matches and steady-state multiplicities do not "blow up".

More detailed properties of the StreamIt graphs are given in Table 4-16. Looking more closely at the filters of the benchmark, on average each benchmark declares 9.25 filter types, and instantiates 50 filters in the StreamIt graph. Vocoder defines the most filter types with 30; Serpent instantiates the most filters with 135. Serpent also instantiates the most Identity filters (33); these identity filters are employed to pass each round's unaltered input to the XOR filter.

Table 4-17 provides load-related characteristics for the static benchmarks in the suite. "Comp / Comm" gives the static estimate of the computation to communication ratio of each benchmark for one steady-state execution. Section 4.1 describes how we calculate the static computation estimate for each filter. This is calculated by totaling the computation estimates across all filters

Benchmark	TOTAL FILTERS Types	TOTAL FILTERS Instances (non-Iden.)	TOTAL FILTERS Instances (Identity)	PEEKING FILTERS Types	PEEKING FILTERS Instances	STATEFUL FILTERS Types	STATEFUL FILTERS Instances	Splitjoins
Static apps (12):								
BitonicSort	1	26	-	-	-	-	-	4
ChannelVocoder	5	53	-	3	34	-	-	1
DCT	3	36	-	-	-	-	-	2
DES	13	33	4	-	-	-	-	8
FFT	4	17	-	-	-	-	-	-
Filterbank	9	67	-	2	32	-	-	9
FMRadio	7	29	-	2	14	-	-	7
Serpent	13	37	9	-	-	-	-	9
TDE	7	29	-	-	-	-	-	-
MPEG2Decoder	13	29	-	-	-	1	1	5
Vocoder	30	96	4	4	32	3	45	8
Radar	6	49	-	-	-	1	28	2

* Statistics represent properties of complete programs, in which libraries have been inlined into caller.

Table 4-16: Properties of filters and splitjoins for StreamIt Core benchmark suite.

and dividing by the number of dynamic `push` or `pop` statements executed in the steady-state (all items pushed and popped are 32 bits). "Max Work Single Filter" estimates the percentage of the total load that is contained in the most loaded filter. "Filters Max Work" gives the number of filters that are estimated to have the maximum single filter load.

"Task Parallel Critical Path" calculates, using static work estimates, the work that is on the critical path for a task parallel model, assuming infinite processors, as a percentage of the total work. The work for data parallel filters on the critical path is included without considering data parallelization (i.e., the steady-state work estimation for data parallel filters is included in the critical path). Smaller percentages indicate the presence of more task parallelism. "Max Work Peeking Filter" estimates the percentage of the total load that is contained in the most loaded peeking filter (if present). In Table 4-17, σ denotes the fraction of work (sequential execution time) that is spent within stateful filters. μ denotes the maximum work performed by any individual stateful filter.

The remainder of this section is organized into key characteristics of the benchmark suite that are important to appreciate the techniques detailed in subsequent chapters.

1. **Peeking is present in four benchmarks. Without peeking support, the peeking filters would often introduce a stateful bottleneck.** Peeking is present in 4 of the 12 static benchmarks (see Table 4-16). The peeking filters often implement a sliding widow operation, e.g., FIR LowPass and HighPass filters in ChannelVocoder, FilterBank, and FMRadio. Each of these filters peeks at N items, pops one item from the input and pushes a weighted sum to the output. The other pattern of peeking filters we see in the suite is when a filter peeks at exactly one item beyond its pop windows. The difference encoder filters in Vocoder follow this pattern. On its first execution, the filter performs the identity function without popping an item by specifying this behavior in its prework function. On subsequent firings, the output is the difference of the 2 items at the head of the input queue, popping one item. As illustrated in Figure 4-18(a), the difference encoder can be written as a stateless peeking filter with a prework. Without explicit peeking support, the filter is required to introduce internal state, as illustrated in Figure 4-

						STATEFUL FILTERS	
Benchmark	**Comp / Comm**	**Max Work Single Filter**	**Filters Max Work**	**Task Parallel Critical Path**	**Max Work Peeking Filter**	**Total Work (σ)**	**Max Work (μ)**
Static apps (12):							
BitonicSort	8	4%	24	25%	-	-	-
ChannelVocoder	22380	3%	33	9%	5.2%	-	-
DCT	191	17%	1	23%	-	-	-
DES	15	6%	2	89%	-	-	-
FFT	63	10%	8	100%	-	-	-
Filterbank	2962	3%	32	9%	3.1%	-	-
FMRadio	673	8%	14	17%	7.6%	-	-
Serpent	36	5%	7	52%	-	-	-
TDE	335	6%	6	100%	-	-	-
MPEG2Decoder	102	16%	1	59%	-	0.8%	0.8%
Vocoder	180	36%	1	85%	8.0%	16.0%	0.7%
Radar	1341	4%	12	11%	-	97.8%	3.9%

* Work estimations used for these characteristics were calculated by a static analysis. Actual runtimes may differ by 2x or more.

Table 4-17: Characteristics of the load and work distribution for the StreamIt Core benchmark suite.

18(b). The `FMDemodulator` filter in FMRadio also follows this pattern, however it performs is the non-linear function XOR.

The remaining peeking filters perform various other sliding window computations. ChannelVocoder's `CorrPeak` filter performs a sliding window autocorrelation and threshold across 50 items. The `InvDelay` filter of Vocoder uses peeking to skip 14 items in the stream. The `LinearInterpolator` filter of Vocoder creates linearly interpolated points between neighboring input items.

Without explicit peeking support in StreamIt, many of the computations described above would require filters with state, as the locations peeked by a filter would have to be converted to internal state. The presence of state would inhibit data-parallelization of these filters, as there would be a dependence between successive executions of the filters. To help appreciate the impact of this added state, Table 4-16 lists the estimated amount of work (load) in the most computationally-intensive peeking filter in each benchmark. For our benchmark suite, this single filter represents anywhere from 3.1% to 8.0%, a significant portion of the total load of the benchmark. Stateful implementations of these filters would inhibit data parallelism and require pipeline parallelism.

2. **Prework functions express startup behavior for peeking filters.** The Vocoder benchmark employs prework functions to describe the startup behavior of its difference filters and delay filters. For the delay filter, on the first execution it pushes 14 placeholder items; on subsequent firings it performs the identity function. Without prework, the delayed items would have to stored as internal state, prohibiting data parallelization. The difference filter's prework behavior is described above.

3. **Stateful filters, though not common, are present in three benchmarks. Mapping and scheduling techniques must parallelize stateful components.** MPEG2Decoder, Vocoder,

```
int->int filter DifferenceEncoder_Stateless {

    prework push 1 peek 1 {
        push(peek(0));
    }

    work pop 1 peek 2 push 1 {
        push(peek(1)-peek(0));
        pop();
    }
}
```

(a) Stateless version of a difference encoder, using peeking and prework.

```
int->int filter DifferenceEncoder_Stateful {
    int state = 0;

    work pop 1 push 1 {
        push(peek(0)-state);
        state = pop();
    }
}
```

(b) Stateful version of a difference encoder, using internal state.

Figure 4-18: Stateless and stateful versions of a difference encoder filter.

and Radar include stateful filters. In the case of Radar, this state is not inherent to the computation, but was included because the developer of the benchmark most easily comprehended the computation as having state. As described in Section 4.2.12, the author decided to maintain his own circular buffer that spanned multiple firing of the FIR filter because of special behavior at the start of each column of inputs. The state could be removed by coarsening the FIR filter.

The state in Vocoder and MPEG2Decoder is inherent to the underlying computation and follows the same pattern. The stateful filters in these benchmarks must store a calculated value for use in a later iteration's calculation. The motion vector decoder of MPEG2Decoder must store the calculated motion vectors of a macroblock because they are needed during the calculation of the next motion vector. The three stateful filters of Vocoder (`PhaseUnwrapper`, `Accumulator`, and `DFTFilter`) must also store calculated values for later use.

Table 4-17 presents the static estimates for the percentage of total of work for all stateful filters, and the percentage of total work for the single most computationally-intensive stateful filter for each benchmark. The percent of stateless work is 1%, 16%, and 98% for MPEG2Decoder, Vocoder, and Radar, respectively. The stateless work in Vocoder and Radar is significant, and any technique that cannot parallelize stateful work will be neglecting a significant bottleneck in the computation.

4. **10 of 12 benchmarks include splitjoins for data reorganization. Efficient implementations of scattering and gathering is important for good performance.** The benchmark suite includes 62 splitjoins (29 duplicate splitters and 32 round-robin splitters). The average width of a splitjoin is 4, the median width is 2, and the maximum width is 16. Due to the prevalence of splitjoins, it is imperative that the target architecture provide an efficient means of data-reorganization across cores, and that the compiler effectively targets this capability.

5. **7 of 12 benchmarks are estimated to have a computation to communication rate of under 200.** Obviously, the performance of a streaming application is affected the communication cost if compute/communication concurrency does not exist on the target. With this concurrency, as described Equation 3.1, the throughput of a streaming application is constrained by the max-

imum of the communication cost and the computation cost. In Table 4-17, we give estimates for the computation to communication rates in terms of:

$$\frac{\text{static work estimation}}{\text{total words pushed and popped}}$$

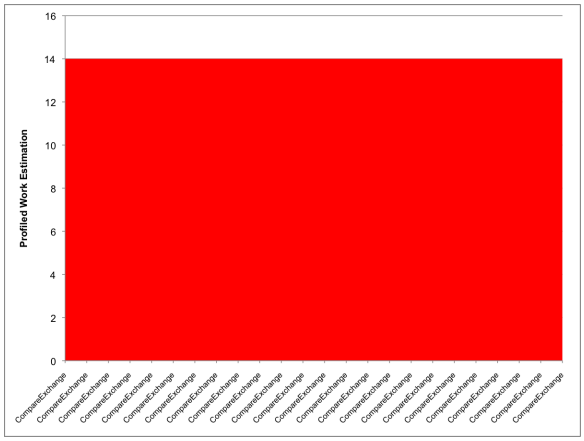
This quantity does attempt to estimate how the cost of communication will compare to the cost computation during execution. For example, communicating a single work across the cores of a SMP via the cache memory hierarchy may take hundreds of cycles. Some benchmarks have low computation to communication rates (e.g., BitonicSort, DES, and Serpent). For these benchmarks, depending on the parallelization strategy, communication cost might be the limiting factor for achieving scalability.

6. **Substantial amounts of task parallel work is found in 4 of the 12 static benchmarks.** Examining the "Task Parallel Critical Path" column of Table 4-17, we see that ChannelVocoder, FilterBank, FMRadio, and Radar each have a small percentage of work on the heaviest loaded path from the source to the sink. These benchmarks include significant amounts of task parallelism in the form of wide splitjoins. It is important to note that the splitjoins of these benchmarks are composed of well-balanced streams; often the parallel streams of the splitjoins are identical in terms of their filters, but are instantiated with unique parameters (e.g., weights, frequencies, etc.).

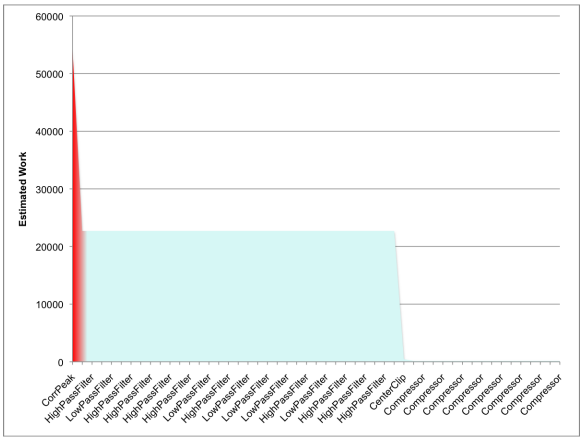
   Four benchmarks have little to no task parallelism. FFT and TDE are long pipelines with no task parallelism, DES has a single path that dominates its small fan-out, and the work in Vocoder is concentrated in 4 filters that are not task parallel. Partitioning and parallelization techniques cannot rely on the presence of task parallelism for all benchmarks; however, techniques should leverage task parallelism when available.

7. **The original, programmer-conceived versions of the StreamIt graphs have limited parallelism and thus limited scalability beyond 16 cores.** Figure 4-20 shows theoretical speedups for leveraging the programmer-introduced parallelism of the unmodified StreamIt graphs from 2 to 64 cores. The filters of the programmer-conceived StreamIt graph are assigned to cores in an optimal assignment strategy that minimizes the maximum work assigned to a core. The assignment algorithm uses a static work estimation of each filter that has been modeled after the Raw microprocessor (see Section 4.2). The algorithm disregards communication cost, synchronization costs, and dataflow dependencies of the benchmark. The theoretical speedup is calculated as the single core critical path estimate work (the total estimated work of all the filters in the graph) divided by the maximum estimated work assigned to a core by the optimal assignment.

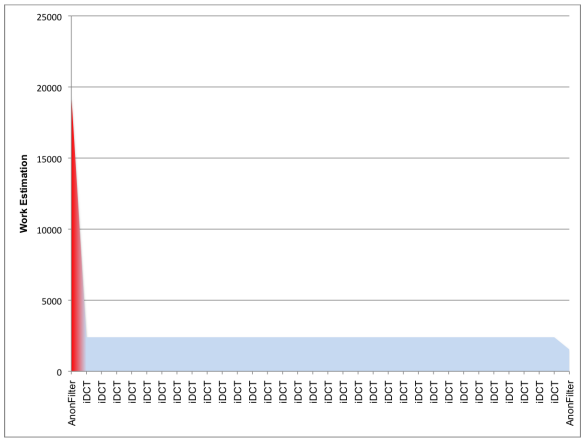
   The theoretical, unmodified-graph scalability analysis of Figure 4-20 considers only the programmer-introduced pipeline and task parallelism in the original benchmark. However, this analysis is enlightening because it allows us to quickly grasp characteristics of the benchmarks that will limit scalability depending on the partitioning and parallelism strategy. The theoretical scalability of the unmodified graph varies widely across benchmarks, from Vocoder which stops scaling at 4 cores, to FilterBank with scales to 32 cores. The average across the benchmark is 17x speedup at 64 cores. All of the benchmarks except for FilterBank and Vocoder



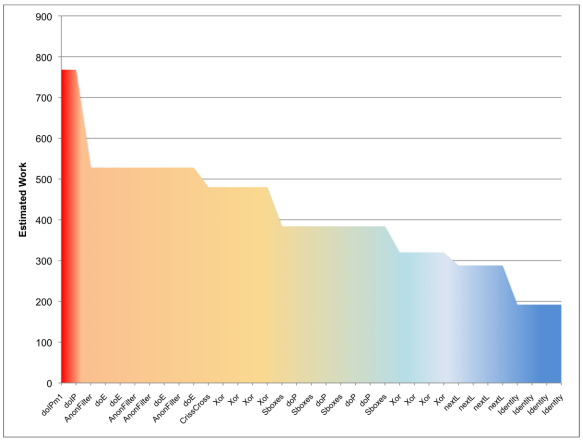
(a) BitonicSort



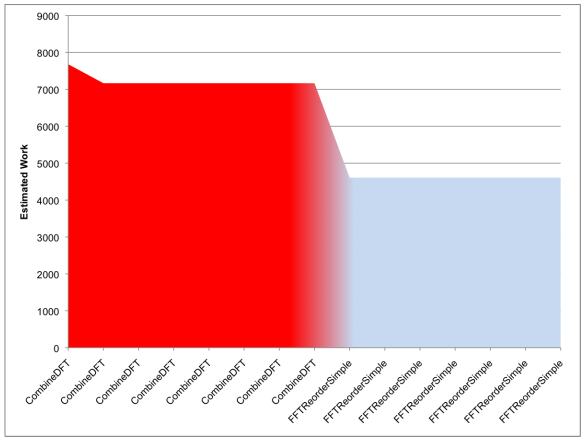
(b) ChannelVocoder



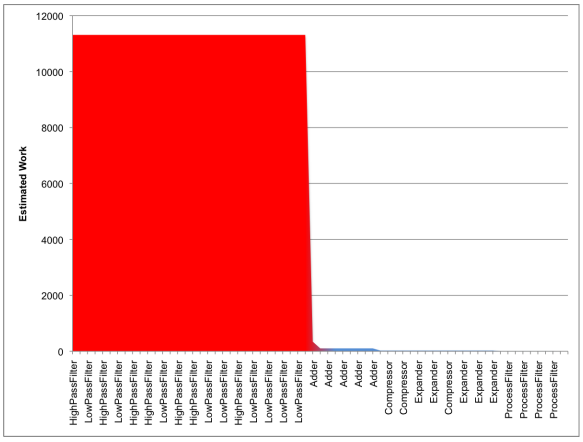
(c) DCT



(d) DES

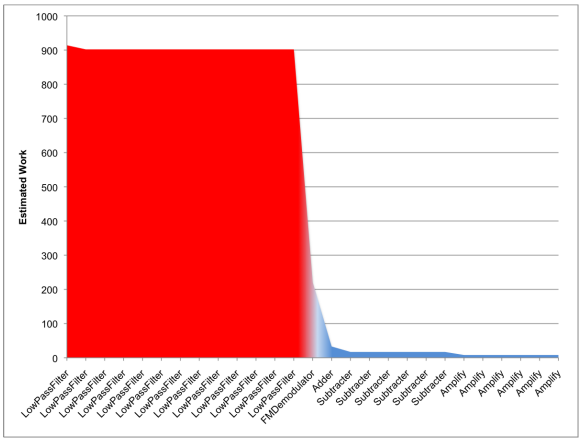


(e) FFT

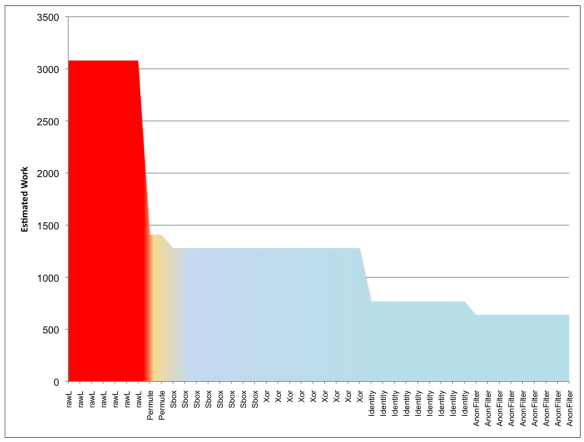


(f) FilterBank

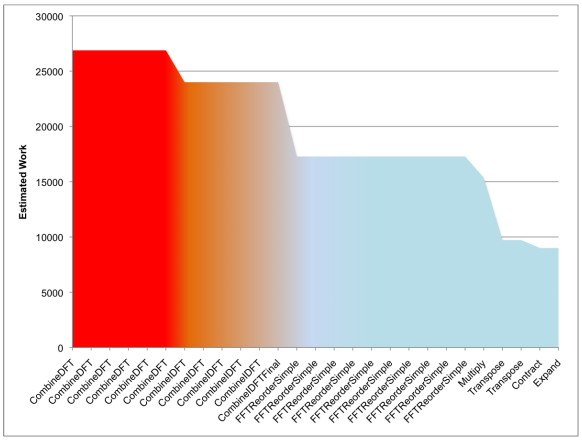
Figure 4-19: Static, estimated workload histograms for the filters of each benchmark.



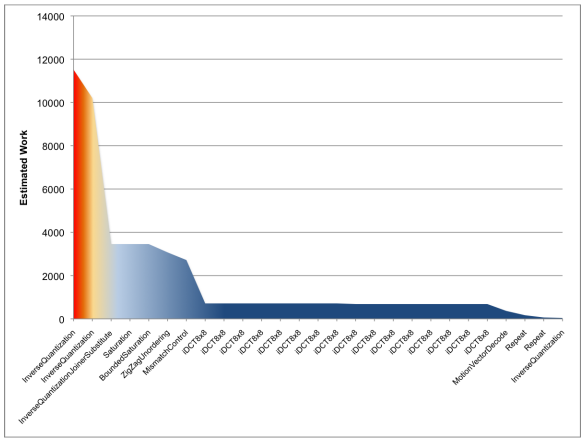
(g) FMRadio



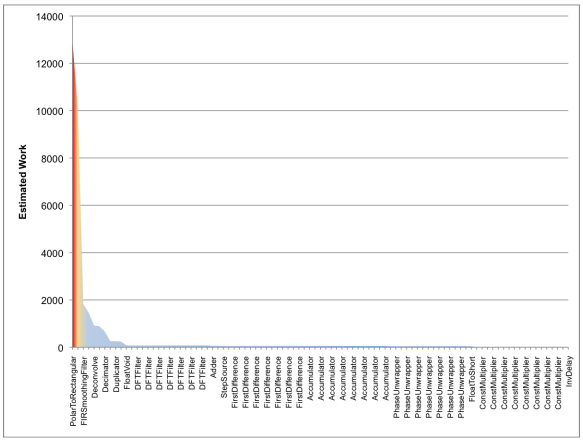
(h) Serpent



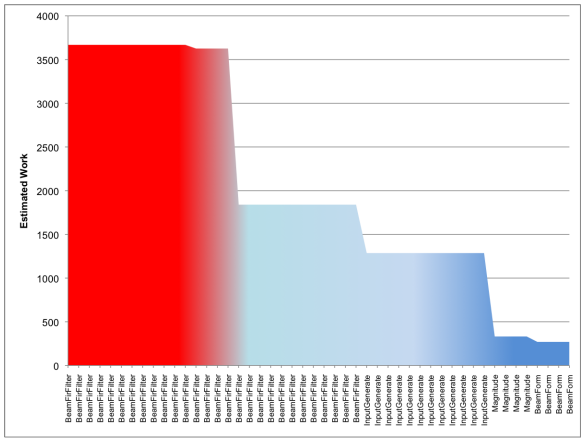
(i) TDE



(j) MPEG2Decoder



(k) Vocoder



(l) Radar

Figure 4-19: (Continued) Static, estimated workload histograms for the filters of each benchmark.

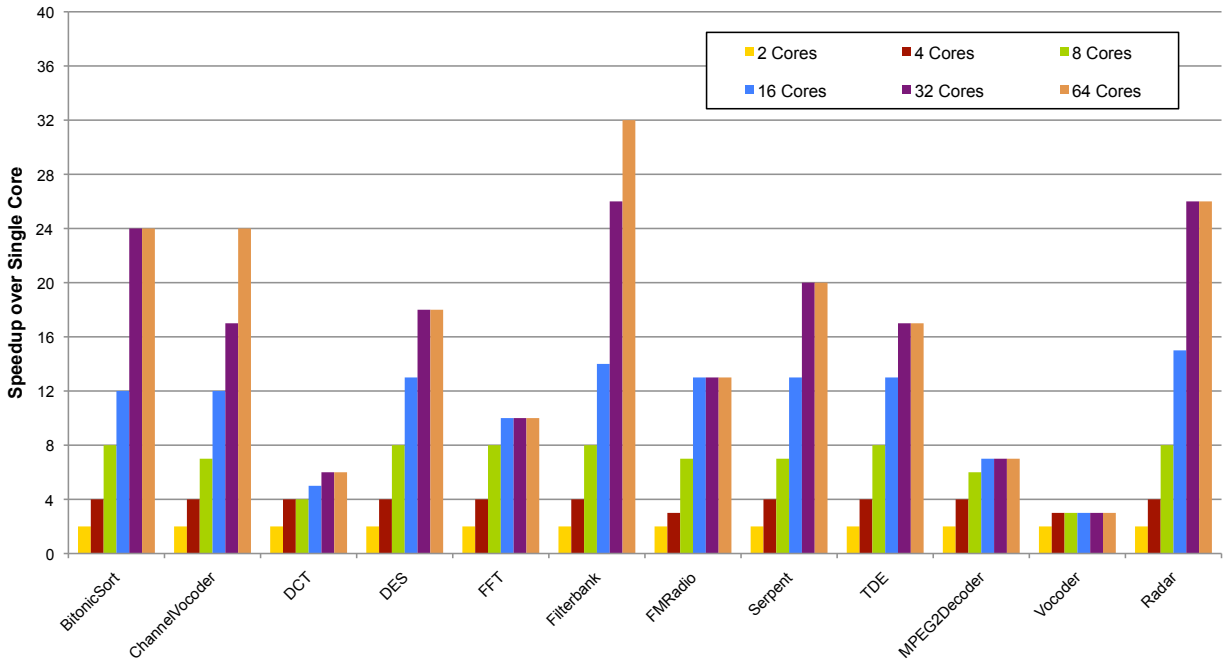


Figure 4-20: Theoretical speedup for optimal partitioning of programmer-conceived StreamIt graphs using static work estimation.

are limited by the number of filters in the benchmark. Since this analysis does not exploit data parallelism, the granularity of the graph is not altered, more filters cannot be created. FFT is the extreme example of this problem with only 17 filters.

Most of the benchmarks scale to 8 cores (the average speedup for 8 cores is 7x), however things start to fall off at 16 cores because of load balancing. Vocoder is estimated to have 36% of its total load in a single filter (see Table 4-17), hence it will only scale to 4 cores. DCT and MPEG2Decoder follow closely in terms of concentrated load with 17% and 16% of their load in a single filter respectively. The most scalable unmodified benchmark is FilterBank in which 32 of its filters performance the same function (FIR filtering) though they are instantiated with different weights. If the target includes more than 8 cores, parallelism and partitioning techniques must leverage the data parallelism inherent to the application in order to expose more parallelism and overcome load-balancing constraints of the original application.

4.4 Related Work

The StreamIt Core benchmark suite has gained wide acceptance since we first introduced the suite in [GTA06]. The suite has been employed by to evaluate streaming compilation techniques by outside research groups [KM08, UGT09b, HWBR09, HCK+09, NY03, UGT09a, ST09, HCW+10, CRA09, CRA10, HKM+08, KM08, CLC+09]. Many other accepted benchmark suites exist for specific domains [Dix93, BBB+91, Tai93]; the two most related suites are MediaBench [LPMS97] and MiBench [GRE+01]. These two benchmark suites include applications that fall into the streaming domain, including image processing (JPEG, TIFF), signal processing (FFT, GSM, IFFT,

and ADPCM), compression (PGP), graphics, and video (MPEG). The StreamIt Core benchmark suite does have some overlap with these suites (FFT and MPEG), and as StreamIt matures, we expect to add additional benchmarks to our suite. MiBench and MediaBench are distributed as C source code, so although many of the benchmarks are streaming, they are implemented in an imperative framework. It would be very difficult to automatically extract the streaming nature from these benchmarks, and use them to evaluate streaming compilation techniques, though work in this field has shown promise [TCA07] (although it relies on programmer annotations to extract some forms of parallelism).

Research evaluated in the context of other streaming languages employ their own suites of benchmarks. The SPUR programming language is evaluated with three image processing kernels [ZLSL05]. Research employing the Brook language is evaluated using linear algebra routines, FFT, ray tracing, lowpass filtering, and medical image processing [BFH+04, LDWL06]. Three benchmarks are employed to evaluate the Imagine Processor coded using StreamC/KernelC: a stereo depth extractor, MPEG-2 encode, a polygon rendering pipeline, FFT, DCT, a 3D perspective transform for images [KRD+03, KMD+01, RDK+98]. The StreamIt Core benchmark suite has some overlap with these suites. Furthermore, two of our applications include stateful computation, such computation cannot be represented by Brook, StreamC/KernelC nor SPUR.

4.5 Chapter Summary

This chapter describes the StreamIt Core benchmark suite of 12 streaming application employed to evaluate research undertaken in the context of the StreamIt programming language. The techniques presented in the remainder of this thesis are evaluated in the context of the suite. The development and analysis of the benchmark suite prompted many parallelization insights that inform the techniques of this dissertation. Language and execution model support for peeking enables the compiler to data-parallelize what would otherwise be a stateful computation. Communication during the initialization stage (prework) enabled many filters to be written in a stateless manner, exposing parallelism that would have been masked without these features.

We were surprised how few filters contain mutable state; this suggests that many programs can leverage data parallelism, rather than relying on task and pipeline parallelism, to achieve parallel performance. However, two programs contain significant amounts of state, and techniques that can parallelize stateful computation are required for robustness. Significant amounts of task parallelism can be found in only 4 of the 12 benchmarks. Fine-grained data reorganization is common across the benchmark suite. Finally, the programmer-conceived versions of the benchmarks have limited parallelism, and thus limited scalability beyond 16 cores.

Chapter 5

Space Multiplexing: Pipeline and Task Parallelism

In this chapter we review a mapping strategy that exploits hardware pipelining and task parallelism. Contiguous regions of the original StreamIt graph are merged into a unit that occupies its own core. The units mapped to different cores communicate via fine-grained communication primitives of the target architecture. We provide details covering the implementation of such a strategy, including the partitioning algorithm and the layout algorithm (for more information see [Gor02]). The technique achieves a 7.4x mean 16-core speedup as compared to single core performance across the StreamIt Core benchmark suite targeting the 16-core Raw microprocessor. We use the empirical results to elucidate the pros and cons of a hardware pipelining strategy.

5.1 Introduction

In the streaming domain, pipeline parallelism applies to chains of producers and consumers that are directly connected in the stream graph. In StreamIt, pipeline parallelism is directly exposed via the pipeline construct. Producer / consumer pairs are trivial to recognize as they are created by the programmer via the `add` statements of a pipeline container. The philosophy of StreamIt is that it should be easy to construct the stream graph unlike some languages (e.g., StreamC/KernelC and Brook with their separate data transfer functions that describe connections).

The compiler can exploit pipeline parallelism via space multiplexing, i.e., by mapping clusters of producers and consumers to different cores and using an on-chip network for direct communication between filters. At any given time, pipeline-parallel filters are executing different iterations from the original stream program in parallel. However, the distance between active iterations must be bounded, as otherwise the amount of buffering required would grow. To leverage pipeline parallelism, one needs to provide mechanisms for both decoupling the schedule of each filter, and for bounding the buffer sizes. This can be done in either hardware or software.

In *hardware pipelining*, groups of filters are assigned to independent cores that proceed at their own rate. As the cores have decoupled program counters, filters early in the pipeline can advance to a later iteration of the program. Buffer size is limited either by blocking FIFO communication, or by other synchronization primitives (e.g., a shared-memory data structure). However, hardware pipelining entails a performance trade-off: if each core executes its filters in a single repeating

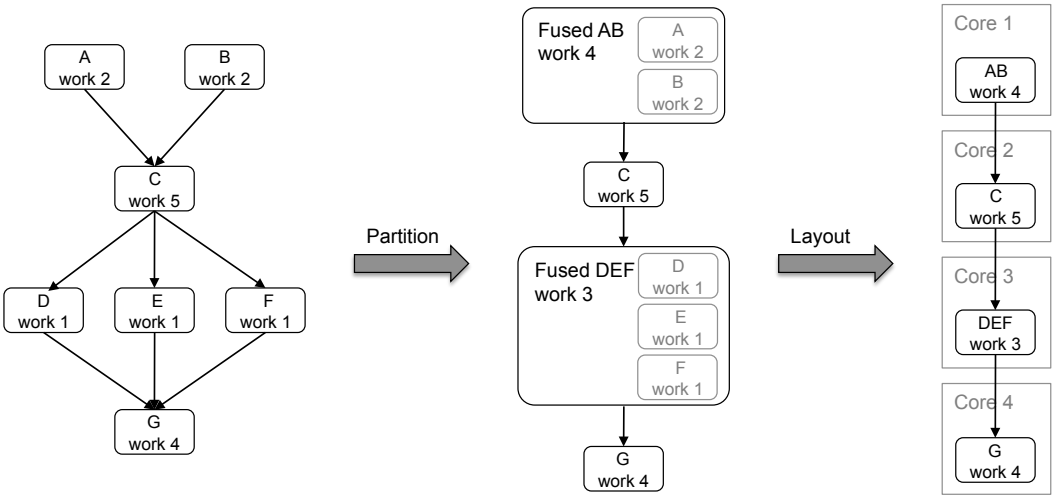


Figure 5-1: The steps of the hardware pipelined mapping strategy. The original stream graph is given on the left; the target has 4 cores. In the partition transformation, the filters are fused so that the graph contains 4 balanced filters. In this case, A and B are fused into a single filter, and D, E, and F are fused into a single filter. In the layout stage, the filters of the fused graph are mapped to cores. Notice that the mapping is not perfectly load-balanced. The steady-state execution time for the pipeline is 5, the work of C, the most loaded filter.

pattern, then it is only beneficial to map a contiguous[1] set of filters to a given core. Since filters on the core will always be at the same iteration of the steady-state, any filter missing from the contiguous group and executing at a remote location would only increase the latency of the core's schedule. As we will demonstrate, the requirement of contiguity can constrain the partitioning options and thereby worsen the load balancing.

StreamIt provides the filter construct as the basic abstract unit of autonomous stream computation. The programmer should decide the boundaries of each filter according to what is most natural for the algorithm under consideration. Thus, it is the responsibility of the compiler to adapt the granularity of the stream graph for efficient execution on a particular architecture. We use the word *partitioning* to refer to the process of dividing a stream program into a set of balanced computation units. Our partitioning algorithm operates on the StreamIt graph, fusing contiguous streams or fissing streams to produce a new valid StreamIt graph. The goal of the partitioning stage is produce a load-balanced set of filters ready to be mapped to the cores of the target. A separate phase, *layout*, is responsible for assigning the nodes of the partitioned StreamIt graph to cores. Partitioning and layout are highlighted in the example of Figure 5-1.

A pipelining strategy naturally supports task parallelism as contiguous units of the partitioned graph that are on parallel branches of a splitjoin are mapped to separate cores. Task parallelism maybe lost as splitjoins are fused in the process of load-balancing. Hardware pipelining introduces extra synchronization, as producers and consumers must stay tightly coupled in their execution. In addition, effective load balancing is critical, as the throughput of the stream graph is equal to the minimum throughput across all of the cores.

[1] In an acyclic stream graph, a set of filters is *contiguous* if, in traversing a directed path between any two filters in the set, the only filters encountered are also within the set.

Phase	Function
KOPI Front-end	Parses syntax into a Java-like abstract syntax tree.
SIR Conversion	Converts the AST to the StreamIt IR (SIR).
Graph Expansion	Expands all parameterized structures in the stream graph.
Scheduling	Calculates initialization and steady-state execution orderings for filter firings.
Partitioning	Performs fission and fusion transformations for load balancing.
Layout	Determines minimum-cost placement of filters on grid of Raw cores.
Communication Scheduling	Orchestrates fine-grained communication between cores via simulation of the stream graph.
Code generation	Generates code for the compute and switch processors.

Table 5-2: Phases of the StreamIt compiler.

The hardware pipelining strategy is portable across parallel architectures. The required communication and synchronization can be implemented via software or hardware mechanisms. For commodity SMP architectures, a software-synchronized approach is employed. Communication can be implemented via shared memory buffers that are synchronized via monitors or semaphores in the producer/consumer style [And99]. Also SMP architectures could leverage blocking interprocess communication primitives, e.g., sockets ([TKS+05] describes such an implementation of hardware pipelining). If the architecture supports computation and communication concurrency, the implementation can be double-buffered such that communication costs are hidden by computation.

In this chapter, we explore hardware pipelining implemented by means of hardware synchronization by targeting the Raw microprocessor (see Section 1.3.1). For communication, generated code leverages Raw's static network. Raw's static network provides low-latency, near-neighbor communication with hardware flow-control achieved via blocking hardware FIFO buffers. Raw's static network is controlled via separate switch processors that are coupled to each processing core. The switch processors include their own instructions and program counter. The compiler programs the switches to perform the communication of the mapped stream graph. The Raw architecture offers low overheads for the communication and synchronization requirements of a hardware-pipelined mapping. Software synchronization is not needed, and communication between neighboring cores is direct and fast. We consider Raw to be a ideal architectural target to test the hardware pipelining strategy. The hardware-synchronized hardware pipelining strategy achieves a 7.4x mean throughput speedup over single-core throughput on the StreamIt Core benchmark suite targeting Raw.

Previous work offers a thorough description of the hardware-synchronized, hardware pipelining StreamIt compiler [Gor02, GTK+02]. This chapter contributes a more in-depth analysis of the technique and offers conclusions valid across architectures.

5.2 The Flow of the Hardware-Pipelining Compiler

The phases of the StreamIt compiler are described in Table 5-2. The front end is built on top of KOPI, an open-source compiler infrastructure for Java [GPGLW01]; we use KOPI as our infrastructure because StreamIt has evolved from a Java-based syntax. We translate the StreamIt syntax into the KOPI syntax tree, and then construct the StreamIt IR (SIR) that encapsulates the hierarchical stream graph. Since the structure of the graph might be parameterized, we propagate constants and expand each stream construct to a static structure of known extent. At this point, we can calcu-

late an initialization and steady-state schedule for the nodes of the stream graph [Kar02]. Following the scheduler, the compiler has stages that are specific for communication-exposed architectures: partitioning, layout, and communication scheduling.

5.3 Partitioning

Given that a maximum of N computation units can be supported, the partitioning stage transforms a stream graph into a set of no more than N filters, the goal being each filter performs approximately the same amount of work during the execution of the program. Following this stage, each filter can be run on a separate core to obtain a load-balanced executable.

Load balancing is particularly important for pipeline parallelism, since the throughput of a stream graph is equal to the *minimum* throughput of each of its stages. This is in contrast to scientific programs, which often contain a number of stages which process a given data set; the running time is the *sum* of the running times of the phases, such that a high-performance, parallel phase can partially compensate for an inefficient phase. In mathematical terms, Amdahl's Law captures the maximum realizable speedup for scientific applications. However, for streaming programs, using the notation introduced in Section 2.1, the maximum improvement in throughput is given by the following expression:

$$\textit{Maximum speedup}(F, s, M) = \frac{\sum_{i=1}^{|F|} s(W_{F_i}, F_i) \cdot M(S, F_i)}{MAX_i(s(W_{F_i}, F_i) \cdot M(S, F_i))} \tag{5.1}$$

where F is the set of filters of the partitioned graph, s gives the work of a function of a filter (W is the work function), and M gives the multiplicity of a filter in a schedule (S is the steady-state). Thus, if we double the load of the heaviest node (*i.e.*, the node with the maximum $s(W_{F_i}, F_i) \cdot M(S, F_i)$), then the performance could suffer by as much as a factor of two. The impact of load balancing on performance places particular value on the partitioning phase of a stream compiler.

Our partitioner employs a set of fusion, fission, and reordering transformations to incrementally adjust the stream graph to the desired granularity (see [Gor02] for a detailed description of these transformations). To achieve load balancing, the compiler estimates the number of instructions that are executed by each filter in one steady-state cycle of the entire program; then, computationally intensive filters can be fissed, and less demanding filters can be fused. The work estimate is calculated via a pass over the statements and expressions of each filter or by profiling. Currently, a simple greedy algorithm is used to automatically select the targets of fusion and fission, based on the estimate of the work in each node. See Section 5.3.1 for more details of the partitioning algorithm.

For example, in the case of the Radar benchmark, the original stream graph (Figure 5-3) contains 49 filters. These filters have unbalanced amounts of computation, as evidenced by the colors of the graph (red filters are more loaded, see Section 4.2 for the work color legend). The partitioned graph targeting 16 cores is given in Figure 5-4. The partitioner vertically fuses the first two filters of the pipeline of the first splitjoin, then horizontally fuses pairs of those filters 12 filters down to 6. The 12 third filters of the original top pipelines are fused horizontally into 3 filters. The 4 filters of the first pipeline of the second splitjoin are fused horizontally into a single filter. Finally the 4 copies of the last 2 filters of the pipeline are fused horizontally into 2 filters. One can see by the prevalence of red in Figure 5-4 that the partitioned graph is estimated to be well load-balanced

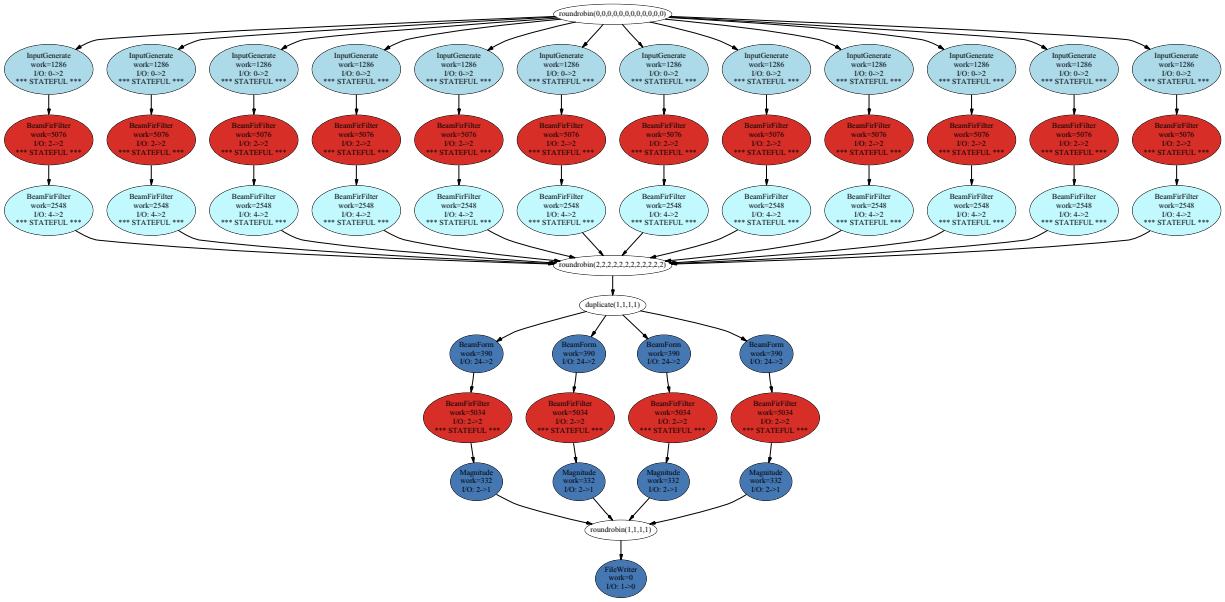


Figure 5-3: StreamIt graph of the original 12x4 Radar application with profiled work estimation encoded.

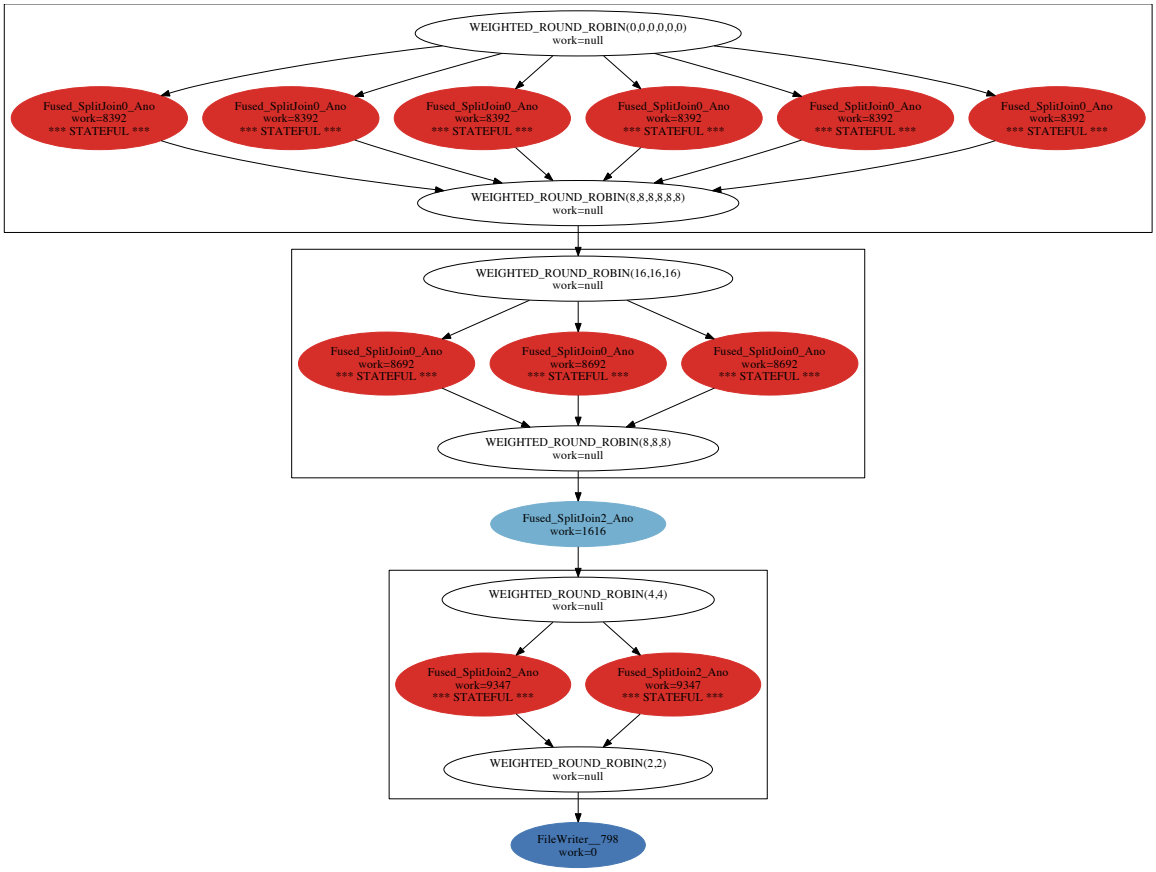


Figure 5-4: StreamIt graph of the partitioned, load-balanced Radar benchmark with profiled work estimation encoded. Our target has 16 cores. Note that joiner nodes occupy cores (see Section 5.4.1).

except for the one blue filter (ignore the last blue filter, it is a file output filter that is not mapped to a core).

5.3.1 Automatic Partitioning

In order to drive the partitioning process, we have implemented a simple greedy algorithm that performs well on most applications. The algorithm requires a means to calculate the work of a filter. We currently employ two methods: (i) a compiler pass that iterates over that statements and expressions of the `work` function (see Section 4.2) and (ii) a profile-driven framework that employs Raw's cycle-accurate simulator to calculate the number of cycles required for a filter's `work` function. As the partitioning algorithm proceeds, transformed filters are re-estimated.

In the case where there are fewer filters than cores, the partitioner considers the filters in decreasing order of their computational requirements and attempts to split them using filter fission. Fission proceeds until there are enough filters to occupy the available machine resources, or until the heaviest node in the graph is not amenable to a fission transformation. For pipeline parallelism, it is generally not beneficial to split nodes other than the heaviest one, as this would introduce more synchronization without alleviating the bottleneck in the graph.

If the stream graph contains more nodes than the target architecture, then the partitioner works in the opposite direction, and repeatedly fuses the least demanding stream construct until the graph will fit on the target. The work estimates of the filters are tabulated hierarchically and each construct (*i.e.,* pipeline, splitjoin, and feedbackloop) is ranked according to the sum of its children's computational requirements. At each step of the algorithm, an entire stream construct is collapsed into a single filter. The only exception is the final fusion operation, which only collapses to the extent necessary to fit on the target; for instance, a 4-element pipeline could be fused into two 2-element pipelines if no more collapsing was necessary.

We compared the greedy strategy to both an optimal ILP strategy and a hierarchical dynamic programming strategy, and despite its simplicity, this greedy strategy works well in practice. This is because most applications have many more filters than can fit on the target architecture; since there is a long sequence of fusion operations, it is easy to compensate from a short-sighted greedy decision. However, we can construct cases in which a greedy strategy will fail. For instance, graphs with wildly unbalanced filters will require fission of some components and fusion of others; also, some graphs have complex symmetries where fusion or fission will not be beneficial unless applied uniformly to each component of the graph.

A hardware pipelining strategy requires that contiguous units of the stream graph be fused when partitioning. How much does this constraint affect the final partitioning in terms of an effective load balancing? Figure 5-5 gives a theoretical comparison of unconstrained partitioning to contiguous partitioning. The unconstrained partitioning is an optimal partitioning where non-contiguous filter can be fused. Both are compared for 16 processing cores, i.e., producing load balanced graphs of 16 or fewer filters. Static work estimations are employed. The speedup is calculated as the inverse of the filter with the maximum workload divided by the total work of all the filters of the benchmark (to approximate single-core performance). This follows from Equation 5.1. The theoretical geometric mean speedup for the unconstrained partitioned scheme is 10.1x while the contiguous scheme has a 7.0x mean speedup. This demonstrates that hardware pipelining is paying a price for its contiguity constraint. The price is particularly heavy for DES as this benchmark has many unbalanced filters in contiguous pipelines with symmetries that do not match 16 cores. DCT

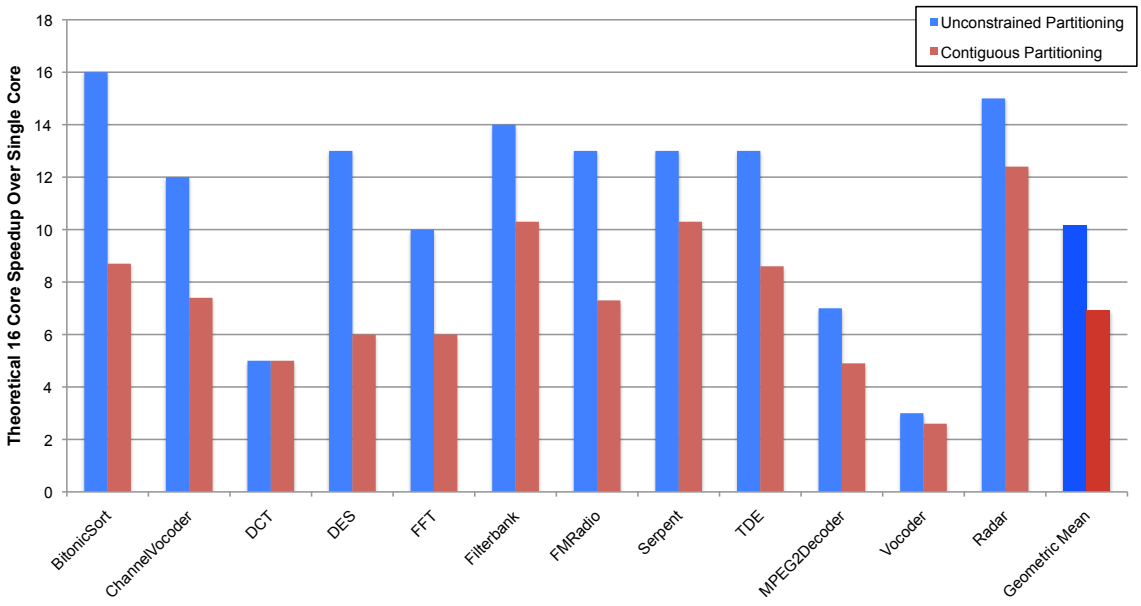


Figure 5-5: Speedup over single core performance for unconstrained partitioning and contiguous partitioning for 16 cores. Results based on statically-estimated work loads of maximally loaded filter.

does not scale for either technique because of an asymmetry with 16 cores. Furthermore, for DCT both techniques are equal because 32 of the 33 filters perform the same amount of work.

5.4 Layout

The goal of the layout phase is to assign nodes in the stream graph to computation nodes in the target architecture while minimizing the communication and synchronization present in the final layout. The layout assigns exactly one node in the stream graph to one computation node in the target. The layout phase assumes that the given stream graph will fit onto the computation fabric of the target and that the filters are load balanced. These requirements are satisfied by the partitioning phase described above.

Classically, layout (or placement) algorithms have fallen into two categories: constructive initial placement and iterative improvement [LaP96]. Both try to minimize a predetermined cost function. In constructive initial placement, the algorithm calculates a solution from scratch, using the first complete placement encountered. Iterative improvement starts with an initial random layout and repeatedly perturbs the placement in order to minimize the cost function. We chose the later approach because we found that constructing an initially good layout was very difficult.

The layout phase of the hardware pipelining StreamIt compiler is implemented using simulated annealing [KCGV83]. We choose simulated annealing for its combination of performance and flexibility. Simulated annealing is a form of stochastic hill-climbing. Unlike most other methods for cost function minimization, simulated annealing is suitable for problems where there are many local minima. Simulated annealing achieves its success by allowing the system to go uphill with some probability as it searches for the global minima. As the simulation proceeds, the prob-

ability of climbing uphill decreases. To adapt the layout phase for a given architecture, we supply the simulated annealing algorithm with three architecture-specific parameters: a cost function, a perturbation function, and the set of legal layouts. To retarget the compiler these parameters should require only minor modifications.

The cost function should accurately measure the added communication and synchronization generated by mapping the stream graph to the communication model of the target. Due to the static qualities of StreamIt, the compiler can provide the layout phase with exact knowledge of the communication properties of the stream graph. The terms of the cost function can include the counts of how many items travel over each channel during an execution of the steady-state. Furthermore, with knowledge of the routing algorithm, the cost function can infer the intermediate hops for each channel. For architectures with non-uniform communication, the cost of certain hops might be weighted more than others. In general, the cost function can be tailored to suit a given architecture.

5.4.1 Layout for Raw

In general, the layout phase maps nodes in the stream graph to cores. Each filter is assigned to exactly one core, and no core holds more than one filter. However, for Raw the ends of a splitjoin construct are treated differently; each splitter node is folded into its upstream neighbor, and neighboring joiner nodes are collapsed into a single core (see Section 5.5.1). Thus, joiners occupy their own core, but splitters are integrated into the core of another filter or joiner.

Due to the properties of the static network and the communication scheduler (Section 5.5.1), the layout phase does not have to worry about deadlock. All assignments of nodes to cores are legal. This gives simulated annealing the flexibility to search many possibilities and simplifies the layout phase. The perturbation function used in simulated annealing simply swaps the assignment of two randomly chosen cores.

After some experimentation, we arrived at the following cost function to guide the layout on Raw. We let $channels$ denote the pairs of nodes $\{(src_1, dst_1) \ldots (src_N, dst_N)\}$ that are connected by a channel in the stream graph; $layout(n)$ denote the placement of node n on the Raw grid; and $route(src, dst)$ denote the path of cores through which a data item is routed in traveling from core src to core dst. In our implementation, the $route$ function is a simple dimension-ordered router that traces the path from src to dst by first routing in the x dimension and then routing in the y dimension. Given fixed values of $channels$ and $route$, our cost function evaluates a given layout of the stream graph:

$$\mathbf{cost}(\textit{layout}) = \sum_{\textit{(src,dst)} \in \textit{ channels}} \mathbf{items}(\textit{src}, \textit{dst}) \cdot (\mathbf{hops}(\textit{path}) + 10 \cdot \mathbf{sync}(\textit{path}))$$

$$\text{where } \textit{path} = \mathbf{route}(\mathbf{layout}(\textit{src}), \mathbf{layout}(\textit{dst}))$$

In this equation, $\mathbf{items}(src, dst)$ gives the number of data words that are transfered from src to dst during each steady-state execution, $\mathbf{hops}(p)$ gives the number of intermediate cores traversed on the path p, and $\mathbf{sync}(p)$ estimates the cost of the synchronization imposed by the path p. We calculate $\mathbf{sync}(p)$ as the number of cores along the route that are assigned a stream node plus the number of cores along the route that are involved in routing *other* channels.

With the above cost function, we heavily weigh the added synchronization imposed by the layout. For Raw, this metric is far more important than the length of the route because neighbor communication over the static network is cheap. If a core that is assigned a filter must route data items through it, then it must synchronize the routing of these items with the execution of its `work` function. Also, a core that is involved in the routing of many channels must serialize the routes running through it. Both limit the amount of parallelism in the layout and need to be avoided.

5.4.2 Layout for SMP Architectures

The layout cost function for Raw is tailored to the specific topology of Raw's communication network. When considering an SMP target, the layout cost function will be quite different, and potentially less complex. SMPs normally offer non-uniform communication costs between cores, as cores shared memories at different levels of hierarchy and core may be located on separate dies. SMP communication is dynamic, i.e., the interleaving of items does not need to be decided by the compiler. Thus the cost of "synchronization", modeled in the Raw cost function, does not need to be considered for an SMP architecture. An effective layout cost function for such a topology should promote the placement of communication links with high bandwidth on cores that share the lowest memory at the lowest level in the hierarchy. This can be modeled as:

$$\mathbf{cost}(\mathit{layout}) = \sum_{(\mathit{src},\mathit{dst}) \in \mathit{channels}} \mathbf{items}(\mathit{src},\mathit{dst}) \cdot \mathbf{latency}(\mathbf{layout}(\mathit{src}), \mathbf{layout}(\mathit{dst}))$$

Where $\mathbf{latency}(\mathit{core}_1, \mathit{core}_2)$ returns the latency of communicating one item between the two argument cores.

5.5 Communication Scheduler

With the nodes of the stream graph assigned to computation nodes of the target, the next phase of the compiler must map the communication explicit in the stream graph to the interconnect of the target. This is the task of the communication scheduler. For architecture with hardware routing of communication, such as SMPs, the communication scheduling pass is null. For architectures with software-routed networks, the communication scheduler maps the infinite FIFO abstraction of the stream channels to the communication mechanisms and network topology of the target. Its goal is to avoid deadlock and starvation while utilizing the parallelism explicit in the stream graph.

Deadlock must be carefully avoided in the communication scheduler. Each architecture requires a different deadlock avoidance mechanism. In general, deadlock occurs when there is a circular dependence on resources. A circular dependence can surface in the stream graph or in the routing pattern of the layout. If the architecture does not provide sufficient buffering, the scheduler must serialize all potentially deadlocking dependencies. Alternatively, if hardware can recognize deadlocks, a slow path to alleviate the deadlock can be provided, though this is potentially expensive in implementation and execution time cost.

5.5.1 Communication Scheduler for Raw

The communication scheduling phase of the StreamIt compiler maps StreamIt's channel abstraction to Raw's static network. As mentioned in Section 1.3.1, Raw's static network provides optimized, nearest neighbor communication. Cores communicate using buffered, blocking sends and receives. It is the compiler's responsibility to statically orchestrate the explicit communication of the stream graph while preventing deadlock.

To statically orchestrate the communication of the stream graph, the communication scheduler simulates the firing of nodes in the stream graph, recording the communication as it simulates. The simulation does not model exactly the code inside each filter; instead it uses the work profile estimation for each filter and assumes all communication occurs at filter firing boundaries. This relaxation is possible because of the flow control of the static network. Since sends block when a channel is full and receives block when a channel is empty, the compiler needs only to determine the ordering of the sends and receives rather than arranging for a precise rendezvous between sender and receiver.

The communication scheduler is implemented as an event-based simulation incorporating the profiled work estimations for each filter. A filter will be scheduled to send its data only after it has finished firing in the simulation. We could have ignored the work estimates and scheduled the communication in any legal dataflow order. However, this scheme is not efficient. Imagine that we have a stream graph that includes a splitjoin with 2 filters, A and B. If A performs much more work than B, the communication scheduler schedules A's output communication first, and B's output items' route(s) cross A's output items' route(s), then B will have to wait for A to finish before B can finish sending its items. This is because the communication scheduler scheduled A's output items first at the intersection point of A and B. A communication scheduler that accounts for work estimation will schedule B's outputs first at the cross point, and not block B. A workload-conscious communication scheduler offers up to an 80% improvement over a communication scheduler that does not incorporate work estimates.

Special care is required in the communication scheduler to avoid deadlock in splitjoin constructs. Figure 5-6 illustrates a case where the naïve implementation of a splitjoin would cause deadlock in Raw's static network. The fundamental problem is that some splitjoins require a buffer of values at the joiner node–that is, the joiner outputs values in a different order than it receives them. This can cause deadlock on Raw because the buffers between channels can hold only four elements; once a channel is full, the sender will block when it tries to write to the channel. If this blocking propagates the whole way from the joiner to the splitter, then the entire splitjoin is blocked and can make no progress.

To avoid this problem, the communication scheduler implements internal buffers in the joiner node instead of exposing the buffers on the Raw network (see Figure 5-7). As the execution of the stream graph is simulated, the scheduler records the order in which items arrive at the joiner, and the joiner is programmed to fill its internal buffers accordingly. At the same time, the joiner outputs items according to the ordering given by the weights of the roundrobin. That is, the sending code is interleaved with the receiving code in the joiner; no additional items are input if a buffered item can be written to the output stream. To facilitate code generation (Section 5.6), the maximum buffer size of each internal buffer is recorded.

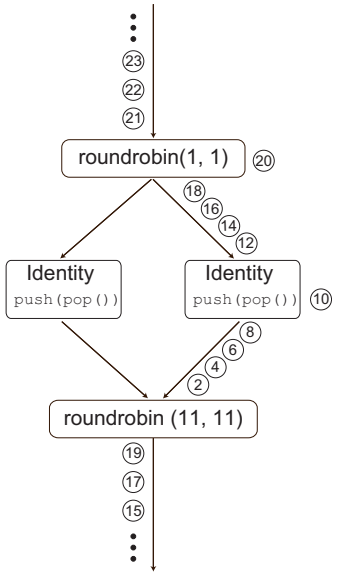


Figure 5-6: Example of deadlock in a splitjoin. As the joiner is reading items from the stream on the left, items accumulate in the channels on the right. On Raw, senders will block once a channel has four items in it. Thus, once 10 items have passed through the joiner, the system is deadlocked, as the joiner is trying to read from the left, but the stream on the right is blocked.

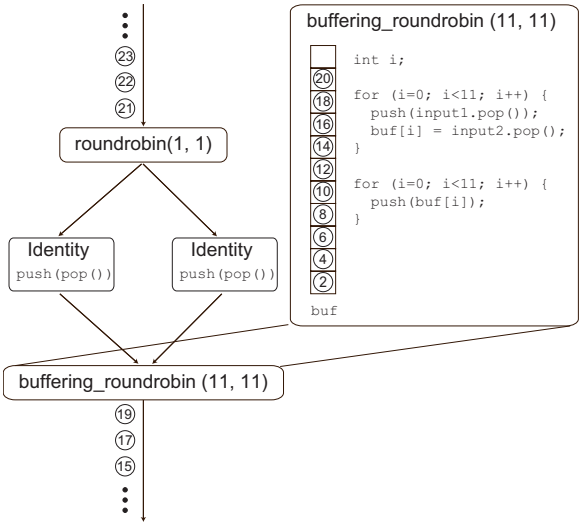


Figure 5-7: Fixing the deadlock with a buffering joiner. The buffering_roundrobin is an internal StreamIt construct (it is not part of the language) which reads items from its input channels in the order in which they arrive, rather than in the order specified by its weights. The order of arrival is determined by a simulation of the stream graph's execution; thus, the system is guaranteed to be deadlock-free, as the order given by the simulation is feasible for execution on Raw. To preserve the semantics of the joiner, the items are written to the output channel from the internal buffers in the order specified by the joiner's weights. The ordered items are sent to the output as soon as they become available.

5.6 Code Generation

The final phase in the flow of the StreamIt compiler is code generation. The code generation phase must use the results of each of the previous phases to generate the complete program text. The results of the partitioning and layout phases are used to generate the computation code that executes on a computation node of the target. The communication code of the program is generated from the schedules produced by the communication scheduler.

5.6.1 Code Generation for Raw

The code generation phase of the Raw backend generates code for both the compute processor and the switch processor of each core. For the switch processor, we generate assembly code directly. For the compute processor, we generate C code that is compiled using Raw's GCC port. First we will discuss the compute processor code generation. We can directly translate the intermediate representation of most StreamIt expressions into C code. Translations for the `push(value)`, `peek(index)`, and `pop()` expressions of StreamIt require more care.

In the translation, each filter collects the data necessary to fire in an internal buffer. Before each filter is allowed to fire, it must receive *pop* items from its switch processor (*peek* items for the initial firing). The buffer is managed circularly and the size of the buffer is equal to the number of items peeked by the filter. `peek(index)` and `pop()` are translated into accesses of the buffer, with `pop()` adjusting the end of the buffer, and `peek(index)` accessing the $index^{th}$ element from the end of the buffer. `push(value)` is translated directly into a send from the compute processor to the switch processor. The switch processors are then responsible for routing the data item.

As an optimization, if a filter does not contain any `peek` statements, and all `pop` statements occur before any `push` statement, a buffer is not needed. In this case, each `pop` statement is translated an instruction that directly receives a word from the switch processor (from the network). Each `push(value)` statement is translated into an instruction that injects `value` onto the network by sending it to the switch processor. This optimization is applicable to 27% of the filters of the benchmark suite, and it increases a benchmark's performance by anywhere from 18% to 3x.

As described in Section 5.5.1, the communication scheduler computes an internal buffer schedule for each collapsed joiner node. This schedule exactly describes the order in which to send and receive data items from within the joiner. The schedule is annotated with the destination buffer of the receive instruction and the source buffer of the send instruction. Also, the communication scheduler calculates the maximum size of each buffer. With this information the code generation phase can produce the code necessary to realize the internal buffer schedule on the compute processor.

Lastly, to generate the instructions for the switch processor, we directly translate the switch schedules computed by the communication scheduler. The initialization switch schedule is followed by the steady state switch schedule, with the steady-state schedule looping infinitely.

5.6.2 Code Generation for SMP Architectures

When targeting an SMP architecture, the code generation phase will be very similar to Raw's code generation phase. However, the SMP code must also include the software producer/consumer synchronization. The implementation would depend on the mechanism utilized. Furthermore, the

				450 MHz 16 Core Raw Microprocessor		
Benchmark	**Work Est. Accuracy**	**Cores Used**	**Predicted Speedup**	**Actual Speedup**	**Utilization**	**MFLOPS**
Static apps (12):						
BitonicSort	87.4%	12	8.7	2.9	37.0%	-
ChannelVocoder	84.6%	14	7.4	4.9	31.0%	366
DCT	93.3%	14	5.6	4.0	24.2%	362
DES	90.4%	14	6.0	9.7	37.6%	-
FFT	76.2%	14	6.0	9.1	26.5%	424
Filterbank	90.7%	14	10.3	10.4	35.6%	742
FMRadio	99.0%	11	7.3	4.3	44.2%	580
Serpent	36.3%	14	10.3	35.0	48.3%	-
TDE	81.2%	14	8.6	19.1	27.5%	626
MPEG2Decoder	27.8%	10	4.9	3.7	20.1%	-
Vocoder	97.4%	15	2.6	6.0	29.9%	202
Radar	92.7%	12	12.4	7.9	80.9%	1277
Geo Mean			**7.0**	**7.4**		

Table 5-8: Performance results for 16 core Raw multicore processor. Speedup results compared to single core performance of StreamIt code. Works estimation accuracy compares the profiled work estimate to the static work estimation pass. Predicted speedup employs profiled work estimation to predict the 16 core speedup.

compute code would have to implement the data distribution described by the input and output distributions (joiners and splitters) of the graph. Destinations for data items can be calculated statically from the distribution sequences of each filter. The distribution can be implemented via store instructions transferring items between memory buffers, or via pop, peek, and push implementations that read from or write to the correct sequence of locations. See [Tan09] for a description of an implementation of a StreamIt code generation pass for SMP architectures.

5.7 Results

The hardware pipelined flow of the StreamIt compiler infrastructure supports fully automatic compilation from StreamIt source to output code for Raw. The output code includes a mix of C and assembly for the compute processors of the cores, and assembly code for the switch processors. The output code is compiled with GCC version 3.3 at optimization level 3. The code is executed on Raw's cycle accurate simulator. The code also executes on the physical Raw microprocessor. For the results of this section we employed a profiled work estimator and optimization level 2 for the StreamIt compiler.

We evaluate the effectiveness of the techniques on the static benchmarks of the StreamIt Core benchmark suite. This evaluation differs from the evaluation in [GTK+02] and cannot be directly compared because the results presented here were gathered using different, more accurate measurements. Furthermore, some benchmarks and many of the compiler techniques have evolved since [GTK+02].

Table 5-8 presents the main evaluation characteristics for the hardware pipelined compiler on the static StreamIt Core benchmark suite. The column "Work Est. Accuracy" quantifies the accuracy of the static work estimation pass by comparing its estimates to that of profiled work estimation. The profiled work estimation scheme employs Raw's cycle accurate simulator during the compilation process. The single result for each benchmark is an average over all filters of the

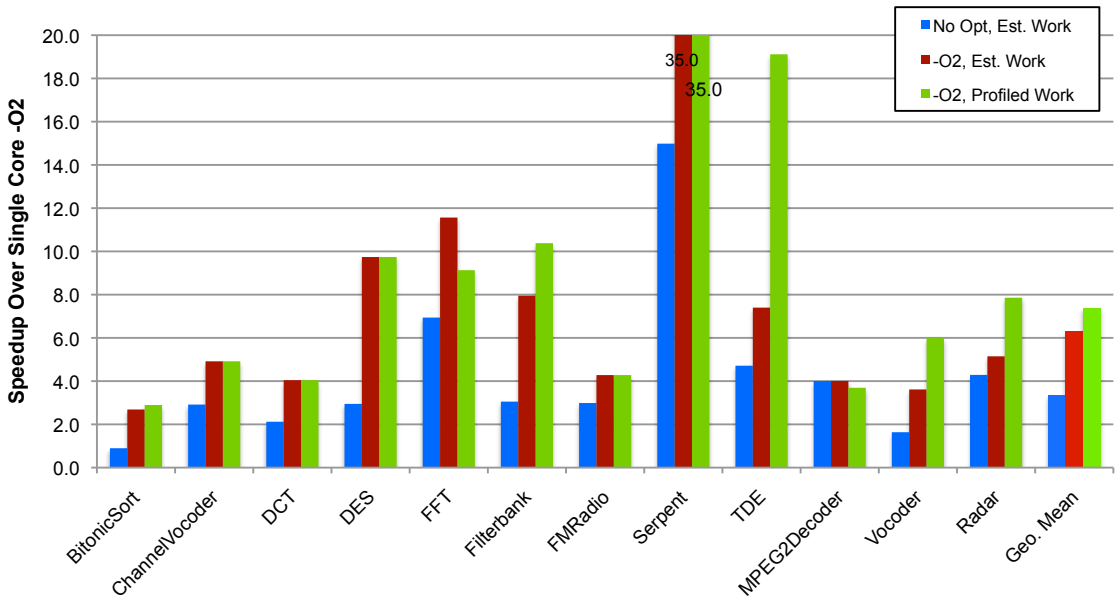


Figure 5-9: Actual speedup results for 3 different configurations of the StreamIt compiler targeting 16 cores.

difference between the ratio of a filter's work estimate compared to the filter with the maximum load for both estimation schemes:

$$= 1 - \text{avg}_{f \in F} \left(\left| \frac{s_s(W_f, f) \cdot M(S, f)}{s_s(W_{f\text{max}}, f\text{max}) \cdot M(S, f\text{max})} - \frac{s_p(W_f, f) \cdot M(S, f)}{s_p(W_{f\text{max}}, f\text{max}) \cdot M(S, f\text{max})} \right| \right)$$

where F is the set of all filters of the benchmark F_{max} is the filter with the max load based on profiling, $s_s(W, f)$ is the static work estimation for f's `work` function, $s_p(W, f)$ is the profiled work estimation for f's `work` function, and as always, $M(S, f)$ denotes the multiplicity of f in the steady-state. 100% represents a static work estimation that closely matches a profiled work estimation for every filter.

The purpose of including the work estimation accuracy result is to demonstrate the accuracy of our tuned static work estimation pass as compared to profiled work estimation. Accurate work estimate is vital to effectively guide the partitioning pass. In Table 5-8 the accuracy ranges from 27%-99% with 9 of 12 benchmarks above 80%. In MPEG2Decoder, the static pass mis-estimates the work of small filters (saturation filters) that are executed many times. The small mis-prediction is compounded by the multiplicity of the filters. A similar problem affects Serpent, where filters with little work per firing are mis-estimated as compared to the profiler. Although we did not employ our static work estimation pass for the results presented in this section, we included "Work Est. Accuracy" to demonstrate that an accurate, fast static work estimation pass can be built for an architecture.

The next column, "Cores Used", gives the number of cores that are occupied by StreamIt filters for the mapping. Since the partitioner is hierarchical, it partitions stream containers (pipelines, splitjoin, etc.). In all cases, fewer than 16 cores are mapped. In the case of MPEG2Decoder only 10 cores are mapped because a long pipeline of relatively light-load filters are fused.

"Predicted Speedup" divides the filter with the maximum work in the partitioned graph by the total work in the unpartitioned graph for each benchmark based on profiled work estimation. It is a crude estimation of the predicted speedup that only predicts the parallelism (both pipeline and task) of the partitioning. Communication and synchronization are ignored. This simple prediction is not accurate for most benchmarks (exceptions are Filterbank, and possibly DCT and MPEG2Decoder). The inaccuracy demonstrates that much more complex issues affect the resulting performance of the partitioned, hardware pipelined benchmarks than just the extracted parallelism of the partitioned mapping to cores.

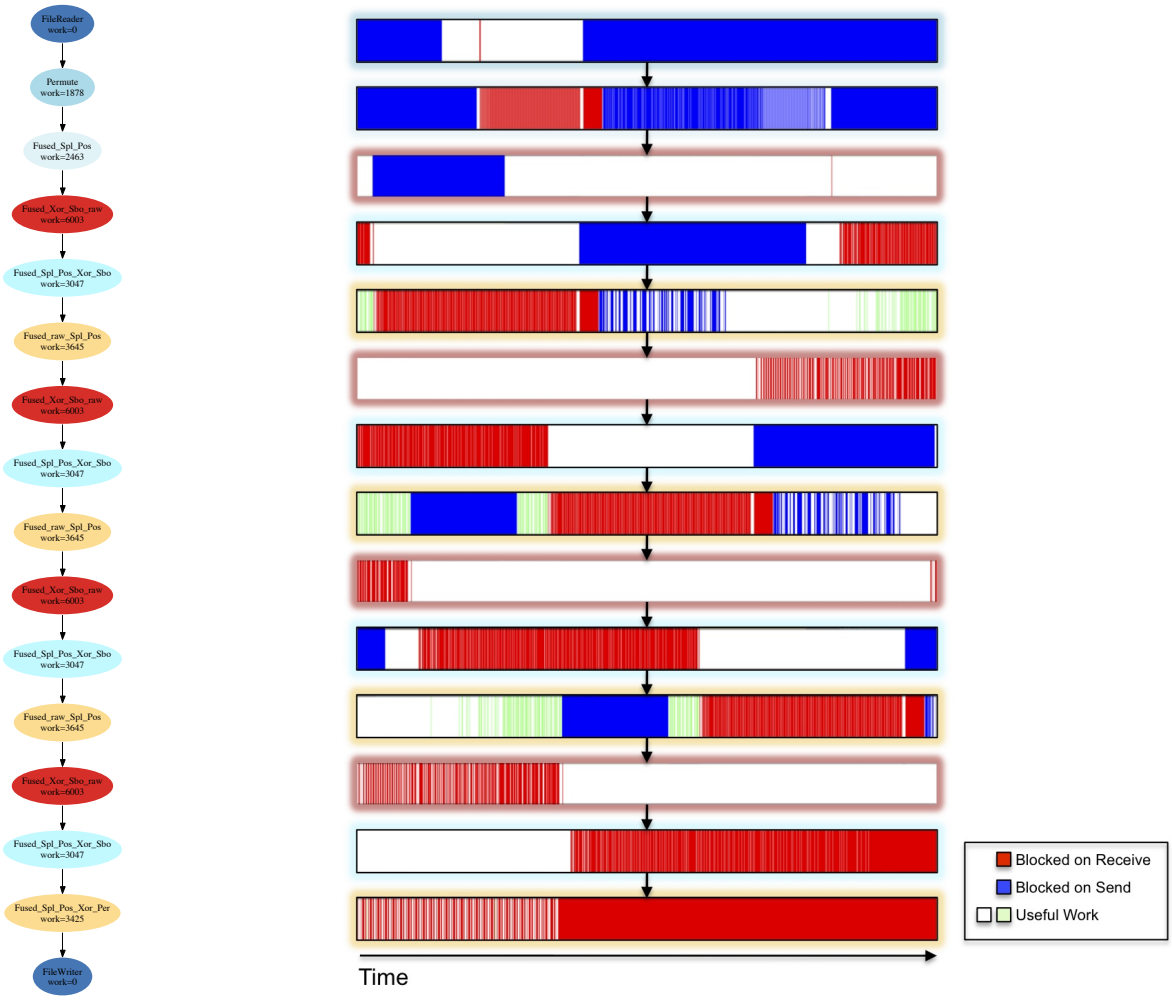
The last 3 columns give results for the benchmarks running on the 16 core Raw microprocessor. "Actual Speedup" is calculated as the throughput of the 16 core mapping over the single core throughput. "Utilization" gives compute processor utilization (the percentage of time that an *occupied core* is not blocked on a send or receive). Finally, we show MFLOPS (which is not applicable for integer applications).

The geometric mean speedup over the 12 benchmarks is 7.7x targeting 16 cores. This number is encouraging, however it falls short of a scalable result. The individual benchmarks vary from 2.9x to 35x, with 7 benchmarks below 9x. The results demonstrate that some benchmarks are more suited to hardware pipelining than others. Utilization ranges from 20% to 81% but generally does not correlate well with speedup.

Figure 5-9 gives the actual speedup over single core for 3 different configurations of the StreamIt compiler. The first configuration uses optimization level 0 of the StreamIt compiler and the static work estimation pass. The second used optimization level 2 and the static work estimation pass. Comparing the results of these 2 configurations, we see that our optimization suite improves speedup from 3.4x to 6.3x, an 85% speedup. Optimization level 2 runs an aggressive unrolling pass (which is guided by the cache behavior of the resulting code), array scalarization, and the work-estimate-guided communication scheduler (see Section 5.5.1).

The third configuration of the StreamIt compiler in Figure 5-9 uses optimization level 2 but switches to profiled work estimation instead of the static work estimation pass. Profiled work estimation improves geometric mean performance by 17% although it hurts performance for FFT and MPEG2Decoder. The performance of FFT is diminished because with static work estimation, the partitioner results with a partitioned graph that is a pipeline. The partitioner, guided by profiled work estimation, settles on a graph that includes task parallelism and the added communication and synchronization of the crossed routes[2] of the splitjoin. The partitioner does not account for the communication and synchronization consequences of its decisions.

There are many issues that affect the resulting performance of a hardware-pipelined mapping for a benchmark: work estimation accuracy, computation to communication ratio of benchmark, load distribution, symmetries between stream graph and number of cores of architecture, communication pattern of mapped partitioned graph, and instruction and data cache effects. For the remainder of this section we will explore the causes for the results in detail, including analysis of three benchmarks.



(a) (b)

Figure 5-10: Serpent: (a) Profiled work for partitioned graph and (b) steady-state Raw execution visualization.

5.7.1 Serpent

The Serpent benchmark (see Section 4.2.8) is partitioned down to a 14 filter pipeline (see Figure 5-10(a)). The load in the partitioned graph is concentrated in 4 filters because of the uneven work distribution of the original graph and asymmetries between the original stream graph and the target of 16 cores. The rates between the filters is *matched* across all producers and consumer, meaning that one firing of a producer outputs exactly the number of items consumed by one firing of its consumer, for all producer/consumer pairs.

Serpent achieves our highest 16 core speedup at 35x over single core performance. There are multiple reasons for this:

[2]A route crosses another route when the communication scheduler decides that the path between a producer/consumer includes an intermediate hop (core) that is also a hop for a different producer/consumer pair. This forces the switch processor of the core to serialize the routes.

1. The partitioned graph is a pipeline, meaning that there is no synchronization other than between producers and consumers (no crossed routes).

2. Serpent has a large code footprint. The single core C/ASM file produced by the StreamIt compiler has approximately 150K lines. This leads to instruction cache misses at the start of each filter's firing for the single core baseline because the entire working set of filter instructions cannot fit in the cache. This problem is alleviated in the 16 core mapping as code is partitioned across 16 cores.

3. Serpent has a large data footprint. Serpent has a fusion buffering requirement that outpaces the data cache of a single Raw core. In the 16 core mapping, the buffers are distributed across the 16 cores, leading a reduced data working set size for any one core.

4. Serpent includes many parameter arrays in its filters. When compiling to 16 cores, we unroll and fully scalarize these array. We cannot scalarize the array in the single core case because unrolling by the factor required to scalarize (required because loops are required to be completely unrolled) decreases performance because of cache effects. We can see this effect in Figure 5-9. Without unrolling, Serpent achieves only a 14x speedup over single core; with optimization level 2, it achieves 35x.

5. Figure 5-10(b) provides a graphical representation of a single steady-state execution of Serpent's on 16 cores. In the figure, there is one "box" for each core. Time flows from left to right for each core, and the color denotes the state of the compute processor. White means the processor is executing instructions and is not blocked. We can see from the figure that Serpent's 4 bottlenecks (the boxes highlighted red in the graph on the right) spend a very large percentage of their execution issuing instructions and not blocked.

   The blocking that is present is because of the small computation to communication ratio of Serpent (see Table 4-17). It takes some cycles for each of the bottlenecks to send and/or receive items. Since we rely on hardware flow control (and Raw's hardware buffer are quite small), for a pipeline, each producer and consumer has to synchronize to transfer data. This synchronization bubbles up the pipeline, and affects all filters.

5.7.2 Radar

Radar (see Section 4.2.12) is partitioned down to a graph with 12 filters and 3 joiners (see Figure 5-11(a)). The 3 joiners each occupy a core to perform joining and prevent deadlock. We achieve a 7.9x 16-core speedup for Radar and 80% utilization. The Radar benchmark illustrates a somewhat different set of causes for its performance compared to Serpent:

1. The instruction cache behavior of the single core mapping of Radar is good. This because the filters of Radar have relatively little code (many loops). Since the single core performance is good, we cannot expect super-linear speedups because of cache effects as in Serpent.

2. Consulting the Raw execution visualization given in Figure 5-11(c), we achieve high utilization for the 11 bottleneck filters of the partitioned graph. The Radar benchmark includes floating point operations; the pink of the execution trace denotes the compute processor has issued a floating point operation. In the figure, the cores that have the bottleneck filters

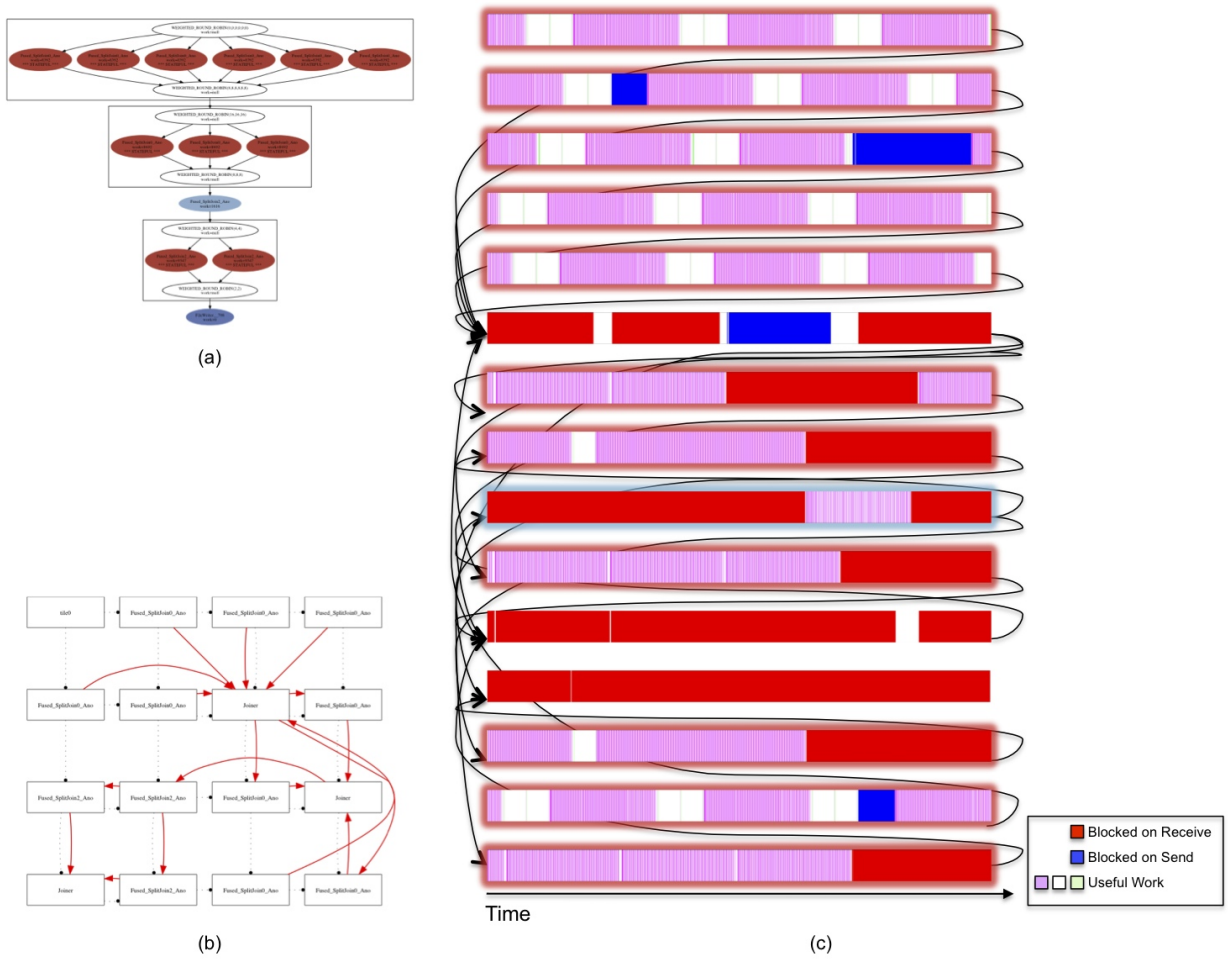


Figure 5-11: Radar: (a) Profiled work for partitioned graph, (b) layout for the partitioned graph to 16 cores, and (c) steady-state Raw execution visualization.

mapped to them are highlighted in red. We can see that 3 of the filters execute at near 100% utilization. We can explain the 7.9x speedup for Radar by correlating the utilization of the 11 bottleneck cores, as they are utilized 74%.

3. The 11 bottleneck cores include some blocking because of overlapping routes caused by mapping the partitioned graph to the constrained topology of Raw's near-neighbor communication network. Figure 5-11(b) gives the layout of the graph, note that items are routed in a dimension-ordered routing scheme, which is not shown on the layout figure. Even with a large computation to communication rate (see Table 4-17), Radar is still affected by this type of synchronization.

5.7.3 Filterbank

The hardware-pipelined mapping achieves a 16 core speedup of 10.4x and a utilization of 36% on Filterbank. This discrepancy is due to the poor single core performance combined with the high cost of synchronization of the partitioned graph. Figure 5-12(a) gives the partitioned graph. This

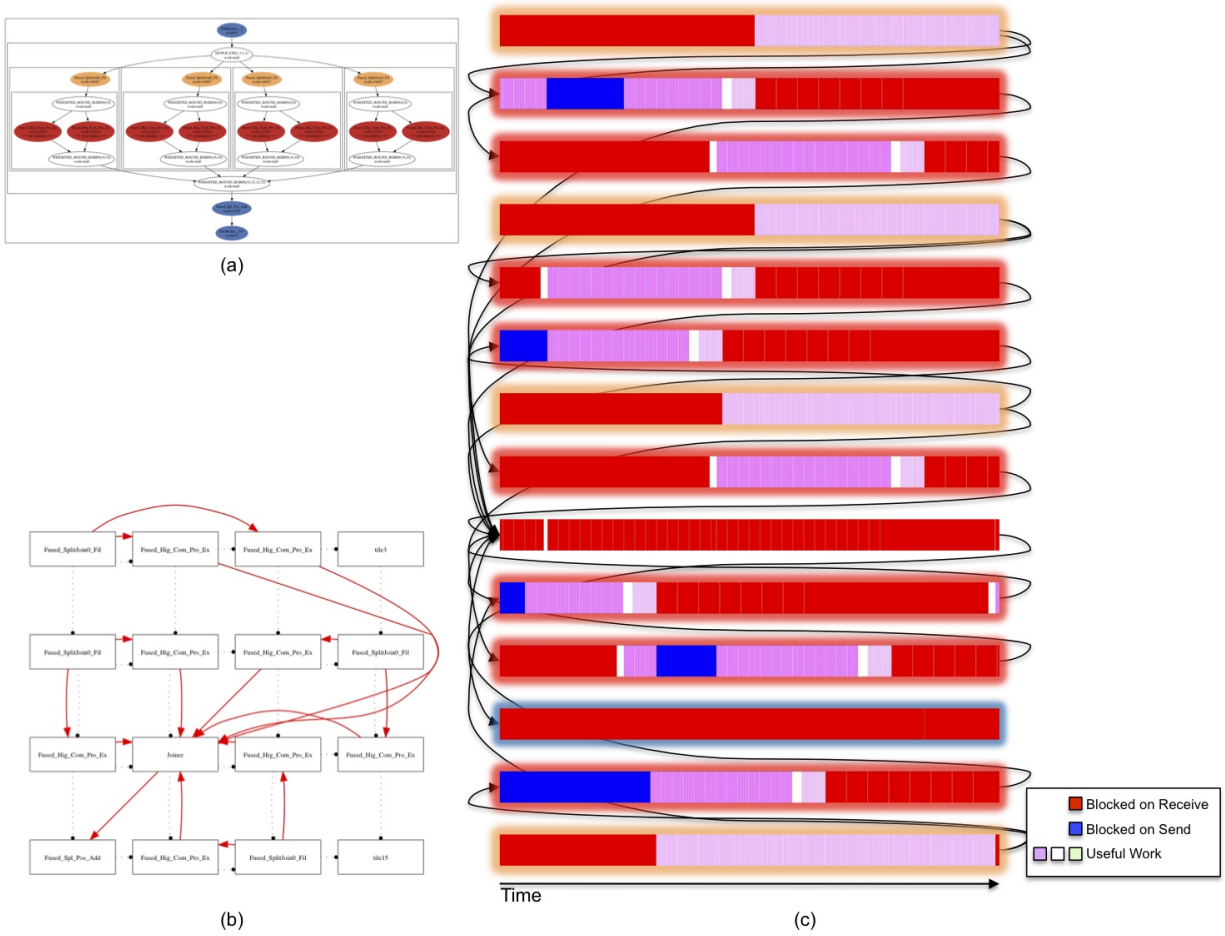


Figure 5-12: Filterbank: (a) Profiled work for partitioned graph, (b) layout for the partitioned graph to 16 cores, and (c) steady-state Raw execution visualization.

graph includes 6 splitjoins, two of which have width 4. The layout of Figure 5-12(b) shows the many crossed routes that introduce artificial synchronization points not present in the original partitioned graph. Studying the execution visualization given in Figure 5-12(c), the synchronization accounts for approximately 50% of the steady-state for the bottleneck filters. Finally, the profiler mis-calculated the load of the graph. Empirically, via the execution visualization, the orange filters operate on useful work for longer than the calculated bottleneck filters in red. It is unclear why the profiler mis-calculated the load for these filters; this may have affected the partitioning scheme.

5.7.4 Other Benchmarks

In this section we highlight key causes for the performance characteristics of the benchmarks not covered above:

- BitonicSort: the low 16 core speedup (2.9x) results from a very low computation to communication ratio, good single core performance (good cache behavior because of loops), and synchronization of the 16 core mapping of splitjoins.

- MPEG2Decoder: the workload of this benchmark is concentrated in two filters (see Figure 4-19(j)). The cores to which those 2 bottleneck cores are mapped achieve near full utilization. All other cores however, perform little work.

- TDE: the super-linear speedup achieved on this benchmark result from bad single core cache behavior. The partitioned graph is a pipeline that is effectively mapped to Raw's communication network. TDE's low utilization is due to the partitioning. In the partitioning, 2 filters (of 14) dominated the work. The cores to which these filters are mapped have near 100% utilization.

- ChannelVocoder: this benchmark suffers from a horrible mis-prediction of work. One `LowPassFilter` includes a small mis-prediction, however, this filter executes many times in the steady-state, hence the mis-prediction is magnified. The workload for this filter is underestimated and in the final execution the core to which it is mapped is utilized 100% while other cores are mostly blocked.

- FMRadio: the partitioned graph includes an 8 way splitjoin resulting in excessive synchronization of the crossed routes that result from mapping the graph to Raw's near-neighbor network.

5.8 Analysis

This section consolidates the lessons learned from developing and evaluating the hardware pipelining strategy:

1. **Partitioning of data and instructions leads to improved cache effects, greatly affecting performance.** As described in Section 3.6, a space multiplexed approach such as hardware pipelining divides the instruction and data footprint across cores. This improved cache performance accounts for much of the speedups we see, especially the super-linear speedups of TDE and Serpent.

2. **Constraining partitioning to contiguous regions of a StreamIt graph often prevents partitioning from achieving a load-balanced partitioned graph.** As evidenced by Figure 5-5, constraining partitioning to contiguous regions hurts its effectiveness for real-world applications as represented by the StreamIt Core benchmark suite. For many benchmarks, a scalable speedup for 16 cores was a non-starter because of the load imbalance of the partitioned graph (e.g., MPEG2Decoder, Vocoder, DCT, DES, and FFT).

3. **The effectiveness of the partitioning stage is often affected by symmetries, or lack thereof, between the original StreamIt graph and topology of the target architecture.** Unfortunately, it is often the case that parameter values greatly affect the efficiency of a hardware pipelined mapping because parameter values affect the stream graph. If an application has symmetries with the target architecture (e.g., splitjoin widths that divide evenly the number of cores) an effective partitioning can often be found. However, the inverse is also true; asymmetries often prevent the effective partitionings. For example, our ChannelVocoder benchmark (Section 4.2.2) includes a large splitjoin that is has a width of 17 (passing through cascading

splitters). If we reduce this width to 16 by altering the size of the bank of bandpass filters, we achieve a speedup of 10.4x over single core performance; compare that to a 4.9x speedup for the original version. For some benchmarks we find symmetries to 16 cores: the width 8 splitjoin of Filterbank (Section 4.2.6) and FMRadio (Section 4.2.7), and the widths of the splitjoins of Radar (Section 4.2.12).

4. **Accurate work estimation is vital to the effectiveness of the partitioning stage, and thus the effectiveness of a hardware pipelined mapping.** As covered in Section 3.7 and demonstrated in Figure 5-9, the accuracy of the work estimation strategy will greatly affect the effectiveness of the partitioning algorithm. In the streaming domain, control flow is infrequently data-dependent, however, it is still difficult to perform accurate work estimation statically. Profiling does help, but is not perfect (e.g., Filterbank and ChannelVocoder). Our profiler does not attempt to model the expected input for a filter. Because of this, it will mis-predict code that has data-dependent workload (e.g., standard library implementations of trigonometric functions).

5. **The simulated annealing layout algorithm works quickly and effectively.** For none of the benchmarks could the author devise a better assignment of filters of the partitioned graph to cores than the layout calculated by the simulated annealing pass. Furthermore, on a older version of the compiler (though a version with the same layout algorithm), the layout algorithm was 3.7x better than random layouts for a slightly different benchmark set.

In summary, a hardware pipelining strategy makes sense if an architecture is constrained by small data and/or instruction caches, work estimation can be accurately modeled, and if the intended application domain can be effectively load balanced by a contiguous partitioner. The next section analyzes the appropriateness of the Raw microprocessor for a hardware pipelined strategy.

5.8.1 The Raw Microprocessor for Streaming

The Raw microprocessor was designed as a research vehicle to explore solutions to the rapidly worsening issues of wire delay and design complexity. Wire delay is addressed by the inclusion of replicated cores; a signal never traverses a path longer than a core. The designers of Raw eschewed shared memory and cache coherence in favor of a scalable near-neighbor mesh network. Each core was a simple inorder 8-stage pipeline so that new ways of mapping ILP could be explored, for example [LBF+98]. The following explores how these design decisions affected the performance of the StreamIt compiler's hardware pipelining path.

1. **Too many or too wide splitjoins introduces synchronization from the target architecture not present in the partitioned stream graph.** Oftentimes mapping the communication of the partitioned graph to the near-neighbor mesh network of Raw introduces synchronization because of crossed routes. A large splitjoin in a partitioned graph (e.g., FMRadio) or many splitjoins (e.g., Filterbank) will cause crossed routes. Two or more routes that cross at a core force the switch processor of the core to serial the routes, servicing one route at a time. The pattern of serialization is decided by the work-estimation-based simulation of the communication scheduler and encoded in the routing instructions executed by the core. Due to Raw's flow control, the blocked routes will often back up at the producer core, and block the core on sending. This blocking is present in Filterbank's execution visualization in Figure 5-12.

A solution to this problem is difficult. Raw's interconnect was purposefully designed to be simple and scalable. A crossbar would not suffice for these properties. One option would be to extend the network topology to additional dimensions to increase connectivity. Another option would be to increase buffering in the static network. The switch processors could use cache memory to buffer items injected into the network so the computer processors are free to continue execution. This would prevent send instructions from blocking. Finally, we could target Raw's dynamic network (see Section 1.3.1) that does not require routes to be laid out by the compiler statically. However, this network is susceptible to deadlock [T+02]. Furthermore, joiners would require a throttling mechanism plus incoming data to be tagged with its source. We implemented a prototype dynamic network option in the compiler, and for a subset of our benchmarks, performance hardly improved due to the overhead of implementation (maximum improvement was 17% for Radar).

2. **The compute processor of each core is required to actively fetch data from the static network and move it to memory.** Each core on Raw has a single-ported data cache. This port is connected to the compute processor. To receive an item from the static network and place it in memory, the compute processor must issue a store instruction where the source operand is the incoming static network port. This design forces the compute processor to actively issue instructions to control the flow of items from the network to memory (cache). If the item is not ready in the network, the compute processor is forced to block waiting for the item. Furthermore, if the consumer core is not actively receiving, the producer core will block once in-network buffering is overwhelmed.

   This design choice forces fine synchronization between producers and consumers, effectively forcing a *rendezvous* between them if the buffering in the static network is overwhelmed. If data cache memory were multi-ported[3] and the switch code could directly access the cache, then we could implement a double-buffering scheme that hides communication cost by parallelizing it with computation.

3. **Each core's simple 8-stage inorder pipeline significantly limits utilization and MFLOPS.** The absence of any ILP structures (VLIW or superscalar) means that pipeline stalls are frequent. In [Gor02], a limit study was performed on a similar suite of benchmarks that showed an average increase of only 41% in MFLOPS if communication blocking and synchronization are disregarded. Furthermore, we amended the Raw simulator so that it can model a VLIW processor with up to four lanes operating in lock-step. Compiling our output code using a compiler that extracts superword-level parallelism [Lar06] achieves over a 2x speedup for some benchmarks in our suite (FFT, FMRadio, and Radar).

5.9 Dynamic Rates

The hardware pipelining path of the StreamIt compiler is actually a sub-component of a larger compilation framework. This framework supports automatic compilation of static rate StreamIt programs, *and* near-fully automatic compilation of dynamic rate StreamIt programs. The compilation flow proceeds as follows. The StreamIt source program's stream graph is divided into

[3]The addition of another port on the data cache memory is not a simple change and would affect other design choices such as cache size.

static-rate subgraphs. Each subgraph represents a stream in which child filters have static data rates for internal communication. A variable data rate can appear only between subgraphs. The phases covered above (partitioning, layout, communication scheduling, and code generation) are performed independently for each static-rate subgraph with some addition code inserted to perform the dynamic rate communication between static-rate subgraphs. The static benchmarks that comprise the StreamIt Core benchmark suite do not include dynamic rates, and thus the phases of the compiler are only performed once, on the complete graph.

For applications that include dynamic rates, the first stage of the compiler (after parsing and semantic analysis) cuts the graph into static rate subgraphs. Essentially, the process cuts that graph at all dynamic rate communication channels, meaning a channel where either the producer has a dynamic push rate and/or the consumer has a dynamic rate pop and/or peek rate. Not all dynamic rate graphs are supported however. Dynamic rates for round robin splitters and joiners are not supported. Furthermore, filters that have dynamic push rates cannot directly feed a joiner because our of deadlock avoid scheme covered in Section 5.5.1.

The partitioning stage of each static subgraph is detailed in Section 5.3. Since the rates are statically unknown (dynamic) between subgraphs, the compiler cannot compare work estimations for filters in different subgraphs. The compiler requires that the programmer assign a fixed number of cores to each static subgraph. The total number of cores assigned to subgraphs must be less than or equal to the number of cores of the target. Then the partitioning stage operates on each subgraph, partitioning the subgraph so that it executes efficiently on the number of cores the programmer assigned to it.

Variable data rates impose two layout constraints not covered in Section 5.4. First, a switch processor must not interleave routing operations for distinct static subgraphs. Because the relative execution rates of subgraphs are unknown at compile time, it is impossible to generate a static schedule that interleaves operations from two subgraphs without risking deadlock. Second, there is a constraint on the links between subgraphs: variable-rate communication channels that are running in parallel and have downstream synchronization must not cross on the chip. Even when such channels are mapped to the dynamic network, deadlock can result if such channels share a junction, since a high-traffic channel can block another. In our implementation, these constraints are incorporated into the cost function in the form of large penalties for illegal layouts.

Channels with variable data rates are mapped to Raw's general dynamic network (rather than the static network, which requires a fixed communication pattern). Within each subgraph, the static network is used. Our implementation avoids the cost of constructing the dynamic network header for every packet; instead, we construct the header once at initialization time. Even though the rate of communication is variable, the endpoints of each communication channel are determined at compile time.

Note that while the static network can also be used for variable data rates when the filters in question are on adjacent cores and there are no overlapping communication routes, forcing all variable-rate pairs to be adjacent would greatly constrain the set of legal layouts. Using the dynamic network allows subgraphs to be separated on chip, as channels between them can overlap many static routes (and also certain dynamic routes, subject to the layout constraints described previously).

Dynamic rate support in the StreamIt compiler is essential to [CGT+05]. In this work, we explore compile-time resource allocation techniques to solve load-balancing problems for graphics rendering workloads. Full graphics pipelines are expressed in StreamIt and compiled by the

StreamIt compiler. Throughput of the specialized pipelines are 55–157% better than a non-specialized allocation of resources. Finally, the compiler supports compilation of large dynamic rate applications such as the full MPEG2 decoder and JPEG transcoding.

5.10 Related Work

Carpenter et al. map StreamIt to heterogeneous multicore architectures leveraging hardware pipelining and task parallelism [CRA10]. Their techniques differ from ours in that they calculate an initial partitioning, then iteratively apply various heuristics to better the partition. They do not compare to our work, even though we are closely related, and they cite us.

The Transputer architecture [tra88] is an array of processors, where neighboring processors are connected with unbuffered point-to-point channels. The Transputer does not include a separate communication switch, and the processor must get involved to route messages. The programming language used for the Transputer is Occam [Inm88]: a streaming language similar to CSP [Hoa78]. However, unlike StreamIt filters, Occam concurrent processes are not statically load-balanced, scheduled and bound to a processor. Occam processes are run off a very efficient runtime scheduler implemented in microcode [MSK87].

DSPL is a language with simple filters interconnected in a flat acyclic graph using unbuffered channels [MT93]. Unlike the Occam compiler for the Transputer, the DSPL compiler automatically maps the graph into the available resources of the Transputer. The DSPL language does not expose a cyclic schedule, thus the compiler models the possible executions of each filter to determine the possible cost of execution and the volume of communication. It uses a search technique to map multiple filters onto a single processor for load balancing and communication reduction.

The iWarp system [BCC+88] is a scalable multiprocessor with configurable communication between nodes. In iWarp, one can set up a few FIFO channels for communicating between non-neighboring nodes. However, reconfiguring the communication channels is more coarse-grained and has a higher cost than on Raw, where the network routing patterns can be reconfigured on a cycle-by-cycle basis [TLAA03]. ASSIGN [O'H91] is a tool for building large-scale applications on multiprocessors, especially iWarp. ASSIGN starts with a coarse-grained flow graph that is written as fragments of C code. Like StreamIt, it performs partitioning, placement, and routing of the nodes in the graph. However, ASSIGN is implemented as a runtime system instead of a full language and compiler such as StreamIt. Consequently, it has fewer opportunities for global transformations such as fission and reordering.

SCORE (Stream Computations Organized for Reconfigurable Execution) is a stream-oriented computational model for virtualizing the resources of a reconfigurable architecture [CCH+00]. Like StreamIt, SCORE aims to improve portability across reconfigurable machines, but it takes a dynamic approach of time-multiplexing computations (divided into "compute pages") from within the operating system, rather than statically scheduling a program within the compiler.

Ptolemy [Lee01] is a simulation environment for heterogeneous embedded systems, including the domain of Synchronous Dataflow (SDF). In all, the StreamIt compiler differs from Ptolemy in its focus on optimized code generation for the nodes in the graph, rather than high-level modeling and design.

5.11 Chapter Summary

This chapter described and evaluated a hardware pipelining strategy for mapping stream programs. In this strategy, groups of filters are assigned to independent cores that proceed at their own rate. The compiler can exploit pipeline parallelism via space multiplexing, i.e., by mapping clusters of producers and consumers to different cores and using an on-chip network for direct communication between filters. The phases of hardware pipelining are described in detail: partitioning, layout, communication scheduling, and code generation. We give particular attention to the partitioning stage, and demonstrate that hardware pipelining's reliance on contiguous partition hampers load-balancing of the StreamIt Core benchmark suite.

The hardware pipelining strategy is evaluated in the context of the Raw microprocessor. The implementation utilizes Raw's static network for hardware synchronization between producers and consumers. Software synchronization is not needed, and communication between neighboring cores is direct and fast. Hardware pipelining achieves a mean 7.4x 16-core throughput speedup over single core throughput for the StreamIt Core benchmark suite. We offer the following insights regarding hardware pipelining:

1. Partitioning of data and instructions leads to improved cache effects, greatly affecting performance.

2. The effectiveness of the partitioning stage is often affected by symmetries, or lack thereof, between the original StreamIt graph and topology of the target architecture.

3. Accurate work estimation is vital to the effectiveness of the partitioning stage, and thus the effectiveness of a hardware pipelined mapping.

Certain properties of the streaming domain enable hardware pipelining. Of particular importance are the following features of the execution model:

1. Exposing producer-consumer relationships between filters. This enables us to easily discover pipeline parallelism.

2. Exposing the outer loop around the entire stream graph. This enables producers to get ahead of each other and achieve pipeline parallelism.

3. Static rates enable efficient filter fusion, static work estimation, and accurate calculation of a cost function for the simulated annealing solution to the layout phase.

Chapter 6

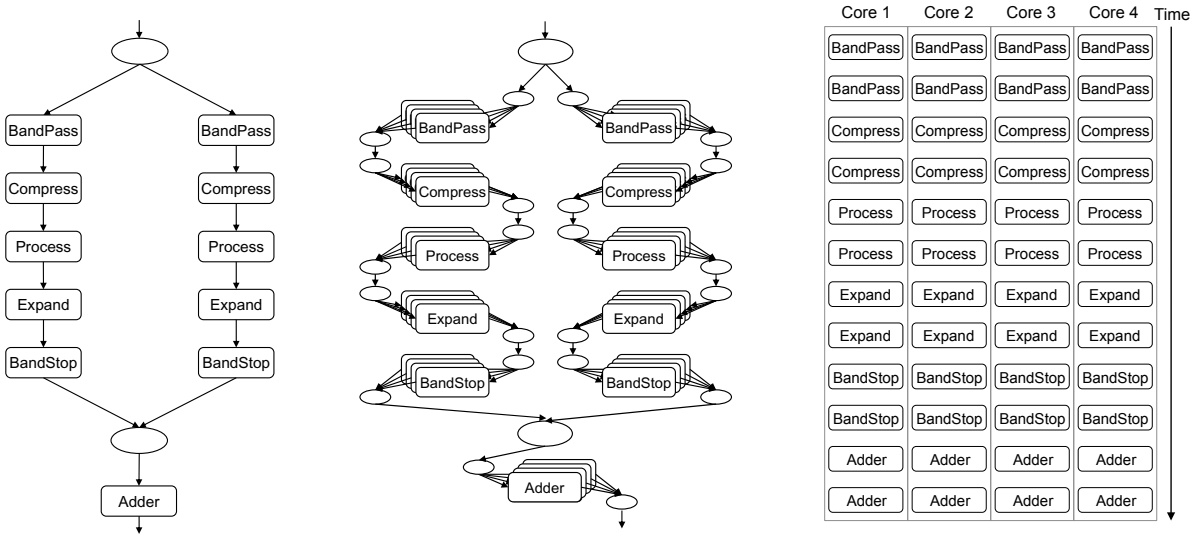
Time Multiplexing: Coarse-Grained Data and Task Parallelism

In this chapter we cover data parallelization techniques that reduce the amount of inter-core communication as compared to the traditional solution. In order to reduce the inter-core communication requirement for data distribution, we coarsen the stream graph by fusing filters, being careful not to obscure data parallelism. Fusing filters keeps communication between filters within a core, implemented via in-core memory buffers. After the graph has been coarsened, filters are data-parallelized, but the transformation is conscious of task parallelism so that the communication required for data-parallelizing peeking filters is reduced. A detailed evaluation is presented for three architectures: Raw, TILE64, and an SMP Xeon system. The Raw architecture achieves a 12.2x throughput speedup for 16 cores over single-core performance for the techniques. Tilera achieves a 37.6x 64-core speedup; SMP achieves an 8.2x 16-core speedup.

6.1 Introduction

In the streaming domain, data parallelism can be leveraged on any filter that does not have dependences between one execution and the next. As mentioned in Section 2.3, such filters are termed *stateless* filters. Stateless filters offer seemingly unlimited data parallelism because they can be spread across any number of cores (because of the infinite nature of the inputs to stream programs). As we will see, however, data parallelism must be leveraged appropriately or else a mapping will be ineffective. There is typically widespread data parallelism in a stream graph, as stateless filters can be applied in parallel to different parts of the data stream. Across the StreamIt Core benchmark suite, 95% of static filter implementations are stateless, while 85% of filter instantiations are stateless. Examples of stateless filters include FIR filters, FFTs, DCTs, color conversions, decimators, or any arithmetic. The peeking idiom enables the stateless representation of sliding window filters such as FIRs, lowpass filters, and highpass filters. The compiler detects stateless filters using a simple program analysis that tests whether there are any filter fields (state variables) that are written during one iteration of the work function and read during another

StreamIt's philosophy is that the programmer should not have to introduce data parallelism. An early compiler pass recognizes programmer introduced data parallelism, and collapses multiple data parallel filters (that perform the same computation) into a single filter. A StreamIt programmer has no control on the amount of data parallelism exploited. It is the compiler's decision to duplicate stateless filters to introduce data parallelism.



(a) Original (b) Fine-grained data parallelism (c) Steady-state execution

Figure 6-1: Mapping a simplified version of the Filterbank benchmark for execution on four cores: (a) the original stream graph (b) the fine-grained data parallel stream graph, and (c) the steady-state execution on 4 cores. Communication is not shown in (c).

To leverage the implicit data parallelism of a stateless filter, the compiler: (i) duplicates the filter, (ii) modifies the filter's source so that it splits its output among the fission products, and (iii) joins the output of the duplicated filters in a round robin at the consumer of the original filter's output. This process, which is called filter fission (see Section 2.4), causes the steady-state work of the original filter to be evenly split across the duplicated filters. Each fission product executes less frequently than the original filter. The computation of the `work` function is unchanged, though as we will see, the rates and multiplicities of the product filters must be altered. The fission transformation can be applied at the StreamIt source level, or on the more general representation of Section 2.1. Chapter 8 demonstrates that certain optimized implementations of fission require the more general model.

Chapter 5 covers hardware pipelining and task parallelism, detailing the problems encountered with such a strategy, including the reliance on work estimation accuracy and effective load balancing of contiguous regions. In contrast, exploiting data parallelism does not entail problems with load estimation and distribution as a single filter is fissed to occupy all the cores of the processor. These fission products execute the same code, so load is balanced among them.

However, achieving an efficient data parallel mapping is not straightforward. If we attempt to exploit data parallelism at a fine granularity, by simply replicating each stateless filter across the cores of the architecture we run the risk of overwhelming the communication substrate of the target architecture. To study this, we implemented a simple algorithm for exposing data parallelism: replicate each filter by the number of cores, mapping each fission product to its own core. We call this fine-grained data parallelism. Figure 6-1(b) gives the fine-grained data parallel stream graph for a simplified version of the Filterbank benchmark shown in Figure 6-1(a). Figure 6-1(c) gives the mapping and scheduling for one steady-state execution of the fine-grained data parallel graph.

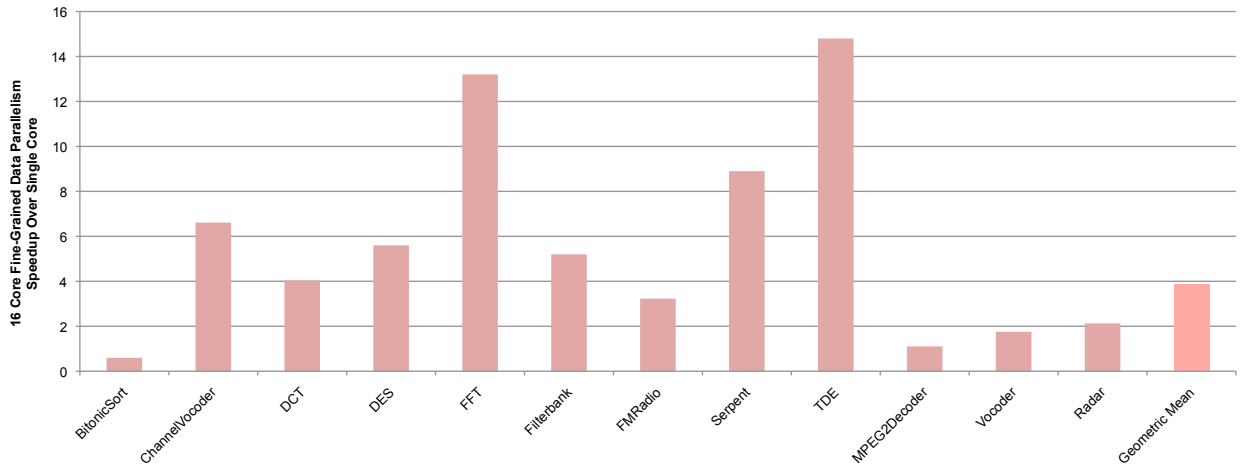
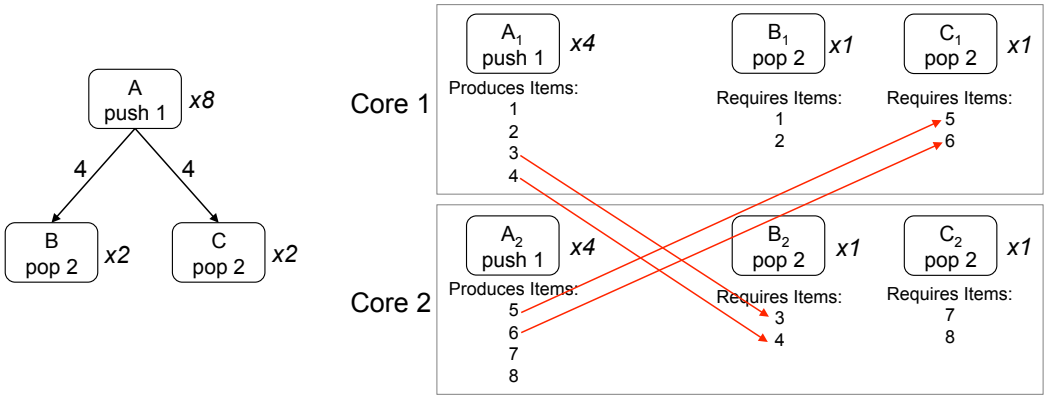


Figure 6-2: Fine-Grained Data Parallelism normalized to single core. Results gathered on the Raw microprocessor.

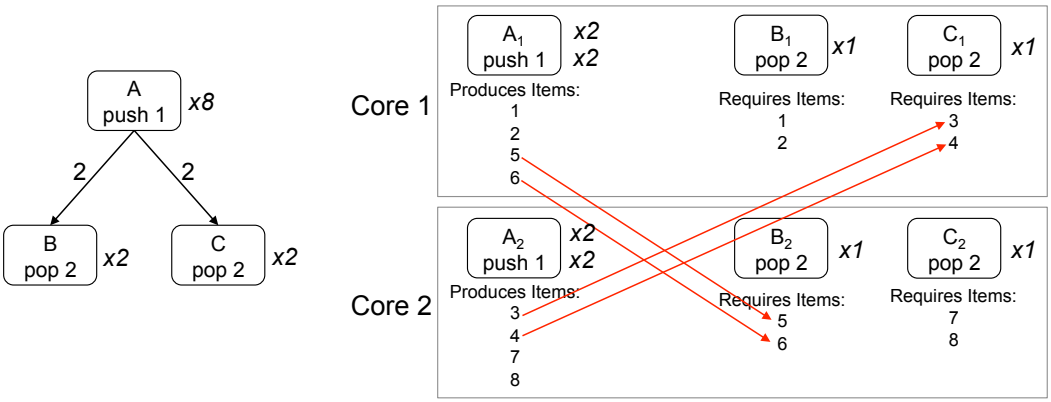
In Figure 6-2, we show this technique normalized to single-core performance for the StreamIt Core benchmark suite. Fine-grained data parallelism achieves a mean speedup of only 3.9x over sequential StreamIt on the Raw microprocessor. For the BitonicSort benchmark, fine-grained duplication on 16 cores has lower throughput than single core. The reason for the overall inadequate parallelization speedups for fine-grained data parallelism is that the technique produces too much inter-core communication. There are two scenarios that produce inter-core communication for fine-grained data parallelism: (i) data distribution and filter rates that are not matched to the mapping and (ii) communication caused by the sharing of items required by the fission of peeking filters.

Notice in Figure 6-2 that the fine-grained data parallel technique is doing well for FFT, Serpent, and TDE. These happen to be the benchmarks that do not include any StreamIt splitjoins, and are thus straight pipelines of filters. The remaining 9 benchmarks include data distribution, and the distribution interacts with the data parallelization to require inter-core communication in many instances. This occurs when filters are data parallelized, and because of the data distribution, items must be communicated between cores, as the fission products split the production and consumption of items. Figure 6-3 illustrates two examples of the need for inter-core communication. In the figure, the need for inter-core communication occurs when the firings of the fission products of the producer filter are interleaved such that the fission products of the destination filters must read items that are produced on a remote core.

Vocoder, Filterbank, ChannelVocoder, and FMRadio include peeking filters. The mean 16 core speedup for fine-grained data parallelism for these applications is 4.1x. As covered in Section 2.4, fission of a peeking filter requires sharing of input items between the fission products of the filter. The sharing is implemented via inter-core communication with the duplication of items over the communication substrate. Figure 6-4 gives an example of the inter-core communication requirement for the fission of peeking filters using the Filterbank example. In the example. the BandStop filter peeks at 126 items past last item dequeued for each firing. Given the small pop rate of the filter, this means that each fission product needs to read all items. This is confirmed by using



(a)



(b)

Figure 6-3: Two example of inter-core communication generated by fine-grained data parallelism. For each example, the original stream graph is shown on the left, annotated with rates, output distribution for A, and steady-state multiplicities. On the right of each example, the mapping for 2 cores is given. The index of the items that the fission products of A is given, plus the index of the items read by the products of B and C. Red arrows denote inter-core communication requirements. In (a), the fission products of A are blocked, such that each fires 4 time consecutively. Because of the weights of the output distribution of A, the products of B and C require items that are produced remotely. In (b) the firings of A in the products are interleaved, but because of the weights of the output distribution, inter-core communication is required. The two examples demonstrate that the general problem cannot be solved with a single strategy of interleaving of firings of the source filter.

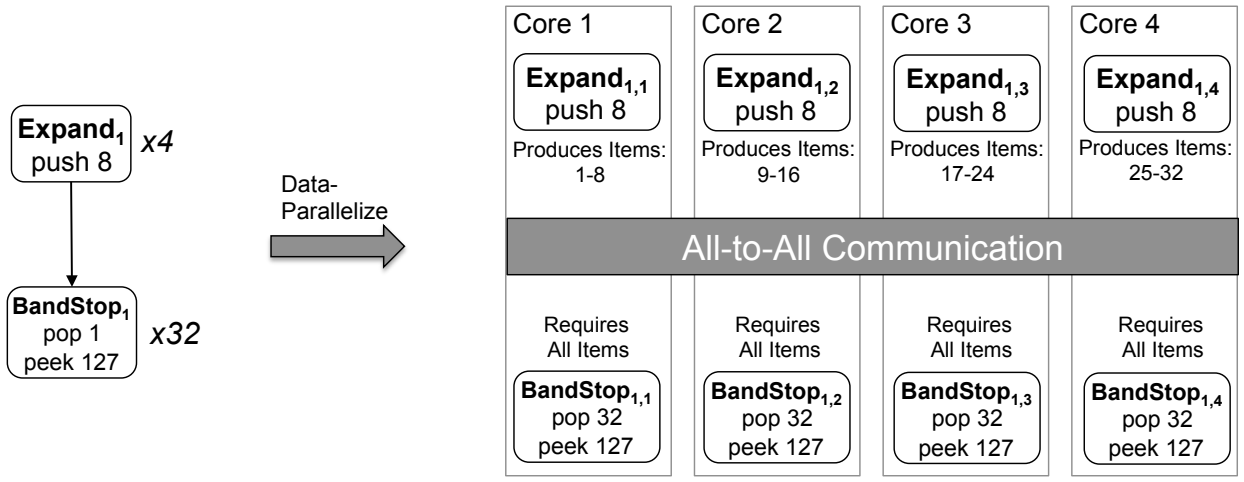


Figure 6-4: An example of inter-core communication generated by the fission of a peeking filter. This example is taken from the Filterbank benchmark. In the benchmark, the two filters on the left communicate and have the rates and multiplicities given. On the right, we show the fine-grained data parallel mapping for 4 cores for the filters. Because of the rates of the BandStop filter, the fission products of this filter must read all items produced by the products of the Expand filter. This requirement is implemented by communication all items produced by the Expand products to all cores, a very expensive operation for some architectures.

Equation 2.1, the formula for the average sharing requirement for each input item as a result of fission a peeking filter. The example in Figure 6-4 requires an all-to-all communication, each item produced has to be communicated to all cores. This communication cost overwhelms the benefits of data parallelizing the BandStop filter.

These two issues motivate the need for a more intelligent approach for exploiting data parallelism in streaming applications when targeting multicore architectures. Towards this end, we have developed new compiler techniques that are generally applicable to any coarse-grained multicore architecture for leveraging data parallelism in stream programs. The technique leverages data parallelism, but avoids the communication overhead of data distribution by first coarsening the granularity of the stream graph. Using a program analysis, we fuse actors in the graph as much as possible so long as the result is stateless. Figure 6-5 shows the results of granularity coarsening on the example of Figure 6-3. Since we fuse the filters of the example, all communication between them remains within a core. The fusion transformation alleviates the need to compute an interleaving of firings of the products of the consumer filter that minimizes inter-core communication. Furthermore, the fusion transformation enables powerful inter-node optimizations such as replacing the memory buffer with registers for communication within a fused filter.

To further reduce the communication costs, we introduce a new data parallelization transformation termed *judicious fission* that also leverages task parallelism; for example, two balanced task-parallel actors need only be split across half of the cores in order to obtain high utilization. Judicious fission reduces the inter-core communication requirement for the fission of a peeking filter if task parallelism is present when fissing the filter. Figure 6-6 gives an example of the inter-

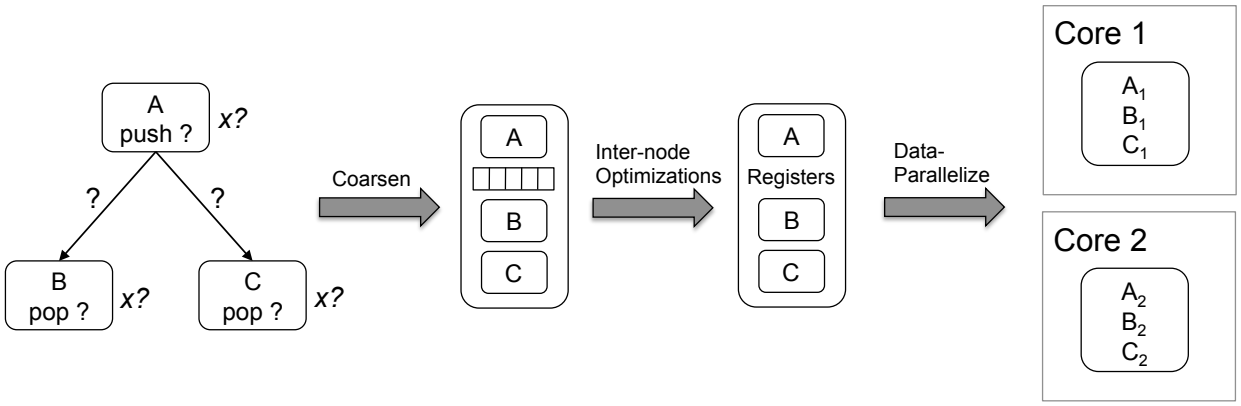


Figure 6-5: An example of coarsening the graph via filter fusion to remove the requirement of inter-core communication for splitjoins. This example is motivated by Figure 6-3. On the far left, we show the original stream graph. When employing coarsening, the rates, the output distribution, and the multiplicities do not affect the inter-core communication requirement. If we assume filters A, B, and C are stateless and are not peeking, we can fuse them into a single filter (the coarsen step). When data-parallelized, the coarsened graph does not require inter-core communication. The coarsen step may increase the latency of the mapping because it is not interleaving the firings of the original graph, but parallelizing the fused filter.

core communication requirement of judicious fission. The example uses the filters from Figure 6-4, but when employing judicious fission, each filter is fissed by only 2 ways because the inherent task parallelism of the benchmark is leveraged. This reduces the requirement of fine-grained data parallelism that each item be communicated to all 4 cores, instead requiring each item be communicated to 2 cores.

The combination of the two techniques to produce a load-balanced, parallel mapping is termed *coarse-grained data and task parallelism* (CGDTP). We evaluate the effectiveness of CGDTP in the context of the Raw, TILE64, and Xeon multicore systems. Targeting Raw, CGDTP achieves a mean performance gain of 12.2x over a sequential single-core baseline, and 1.7x speedup over hardware pipelining. This also represents a 3.2x speedup over a fine-grained data parallel approach. For the TILE64, CGDTP achieves a 10.7x 16 core speedup and a 37.6x 64-core speedup, both are normalized to single core TILE64 throughput. Finally, for the SMP Xeon system, CGDTP achieves an 8.2x 16-core speedup over single-core performance.

While these techniques exploits data parallelism, they also rely on the producer-consumer relationships evident in stream programs for increasing the granularity of the graph. In StreamIt, producer / consumer relationships are trivial to recognize as they are created by the programmer via the `add` statement of the pipeline container.

6.1.1 The Flow of the StreamIt Compiler

The compiler flow for CGDTP is given in Table 6-7. The next sections detail our techniques for intelligently exploiting data and task parallelism. These techniques are applied after the scheduling pass of the compiler. The discussion begins with the algorithm that coarsens the granularity of the StreamIt graph. This algorithm is formulated as a source-to-source translation and is applied to the

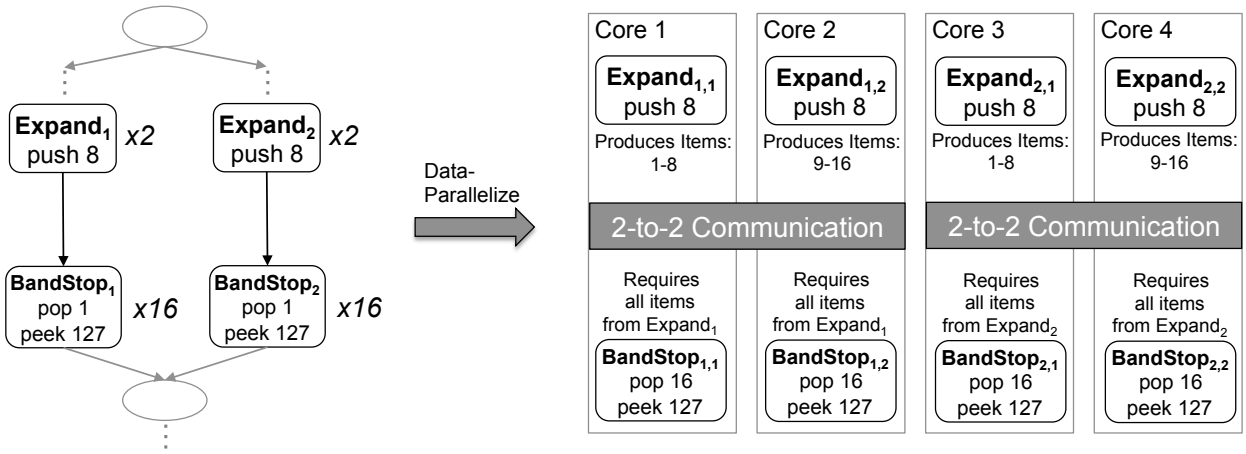


Figure 6-6: An example of the inter-core communication savings provided by judicious fission when fissing a peeking filter. This example uses the same filters as Figure 6-4, however judicious realizes that there is inherent task parallelism for the two filters (see Figure 6-1(a) for the complete stream graph). Since we are targeting 4 cores, and there exists 2 way task parallelism, each filter needs only to be fissed by 2 ways. This is shown on the right. Now the communication requirement each fission application dictates that each items be communicated to 2 cores (instead of 4 with fine-grained data parallelism).

StreamIt graph for all targets. Judicious fission can be applied to either the StreamIt graph or the more general stream graph detailed in Section 2.1.

6.2 Coarsening the Granularity

Fine-grained data parallelism, the traditional technique for leveraging data parallelism, may require inter-core communication depending on the interaction between the rates, data distributions, execution interleavings, and core assignments of the fissed filters. We instead aim to preclude inter-core communication by fusing multiple filters into a single filter, then data parallelizing the fused filter, such that there is local communication within the fused filter. Also, fusion enables powerful inter-node optimizations such as scalar replacement [STRA05] and algebraic simplification [ATA05, LTA03]. This technique may increase latency, as we may need to increase the steady-state multiplicity of the fused filter in order to enable fission across cores.

Some filters in the application cannot be fully fused without introducing internal state, thereby eliminating the data parallelism. Fusion may introduce state even for previously stateless filters due to items buffered on the intervening data channel. For example, in Figure 6-1(a), the BandStop filters perform peeking (a sliding window computation) and always require a number of data items to be present on the input channel. If the BandStop filter is fused with filters above it, the data items will become state of the fused filter, thereby prohibiting data parallelism. Due to StreamIt's support for peeking (sliding window computations) as well as an optional `prework` function (performing distinct computation on the first firing of a filter), an initialization schedule may be needed to prime the buffers prior to steady-state execution [Kar02]. If this schedule deposits items between two filters, then those items introduce state into the fused filter and prohibit data parallelism (even

Phase	Function
KOPI Front-end	Parses syntax into a Java-like abstract syntax tree.
SIR Conversion	Converts the AST to the StreamIt IR (SIR).
Graph Expansion	Expands all parameterized structures in the stream graph.
Collapse Data Parallelism	Remove programmer-introduced data parallelism in graph.
Scheduling	Calculates initialization and steady-state execution multiplicities.
Coarsen Granularity	Fuse pipelines of stateless filters to reduce communication.
Judicious Fission	Fiss stateless filters while conscious of task parallelism.
Layout	Assign filters to cores minimizing critical path workload.
Communication Scheduling	Orchestrates streaming communication between cores and off-chip memory.
Code generation	Generates and optimizes computation and communication code.

Table 6-7: Phases of the StreamIt compiler. The phases beginning with "Coarsen Granularity" are covered in this chapter.

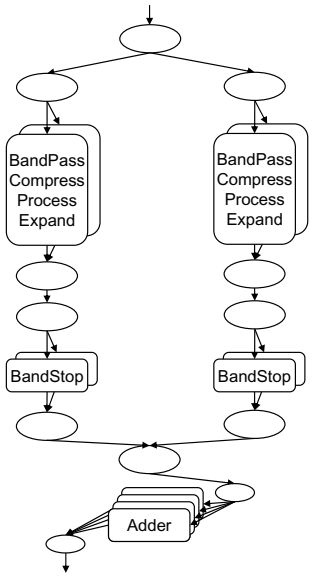


Figure 6-8: The simple Filterbank of Figure 6-1(a) after coarsening.

if the original filters were stateless). In addition, our fusion algorithm introduces state when fusing such filters with neighbors in a roundrobin splitjoin (to optimize access to the input buffers).

Thus, our algorithm for coarsening the granularity of data-parallel regions operates by fusing pipeline segments as much as possible so long as the result of each fusion is stateless. For every pipeline in the application, the algorithm identifies the largest sub-segments that contain neither stateful filters nor buffered items and fuses the sub-segments into single filters. It is important to note that pipelines may contain splitjoins in addition to filters, and thus stateless splitjoins may be fused during this process. While such fusion temporarily removes task parallelism, this parallelism will be restored in the form of data parallelism once the resulting filter is fissed. The output of the algorithm on the FilterBank benchmark is illustrated in Figure 6-8.

Algorithm 1 gives pseudocode for our granularity coarsening algorithm. The code repeatedly considers pairs of stateless filters that are adjacent in the stream graph. If the filters are contained

Algorithm 1 Granularity coarsening algorithm to expose coarse-grained data parallelism.

COARSENGRANULARITY(G)

```
repeat
    for all (f1, f2) pairs of adjacent stateless filters in G do
        if f1 and f2 in a splitjoin then
            Speculatively fuse f1 and f2
        else if f1 and f2 in a pipeline and
                no initial buffering between f1 and f2 then
            Finalize any speculative fusion within f1 and f2
            Fuse f1 and f2
        end if
    end for
until nothing is fused
Undo remaining speculative fuses
```

in a splitjoin, they are "speculatively" fused by the algorithm[1]. The fusion is speculative because it will only eliminate communication if it enables later fusion within a pipeline construct. If the filters are in a pipeline and do not require a buffer at initialization time, then they are permanently fused, and any speculative fuses performed in building the filters are finalized. At the end of the algorithm, all remaining speculative fuses are cancelled. Note that all fused filters produced by the algorithm are stateless and, due to an increased computation to communication ratio, are likely to benefit from filter fission.

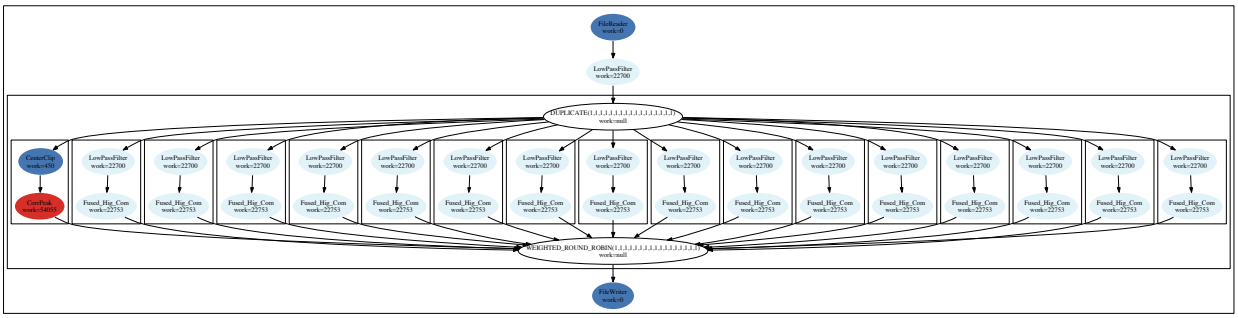
6.2.1 Coarsening the Granularity of the StreamIt Core Benchmark Suite

Six of the twelve benchmarks in the StreamIt Core benchmark suite are completely stateless, do not include a filter that defines a prework, and do not include any filters with a peek rate greater than their pop rate (BitonicSort, DCT, DES, FFT, Serpent, and TDE). For these benchmarks, COARSENGRANULARITY of Algorithm 1 fuses each to a single stateless filter that can be then data parallelized. The only communication present in the coarsen graph is input and output. Conversely, the Radar benchmark is unaltered by COARSENGRANULARITY because of the placement of stateful filters (there does not exist any adjacent pairs of stateless filters). The effect of COARSENGRANULARITY on the remaining benchmarks is illustrated in Figure 6-9(a)-(e).

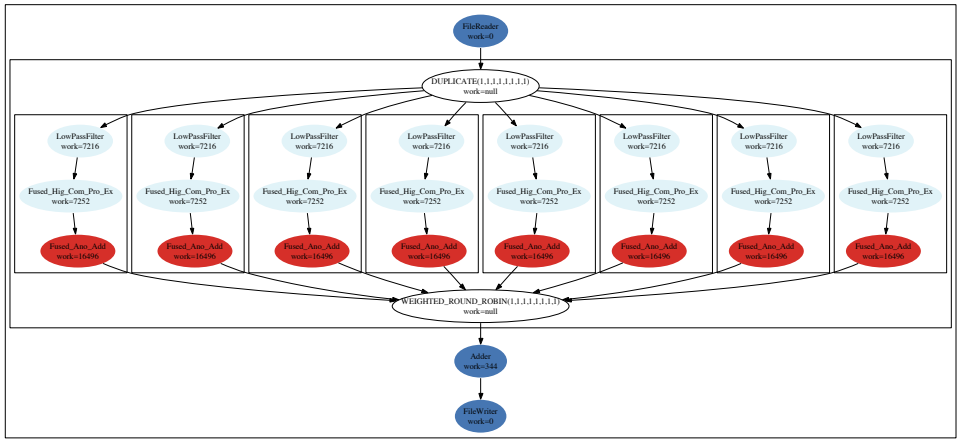
6.3 Judicious Fission: Complementing Task Parallelism

Even if every filter in an application is data-parallel, it may not be desirable to fiss each filter across all of the cores. Doing so would eliminate all task parallelism from the execution schedule, as only one filter from the original application could execute at a given time. An alternate approach is to preserve the task parallelism in the original application, and only introduce enough data parallelism to fill any idle processors. This serves to reduce the synchronization imposed by fission of peeking filters, as filters are fissed to a smaller extent and will span a more local area of the chip. Also,

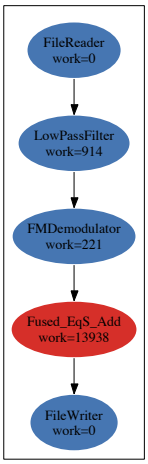
[1] Our compiler performs speculative fusion symbolically rather than actually fusing the filters.



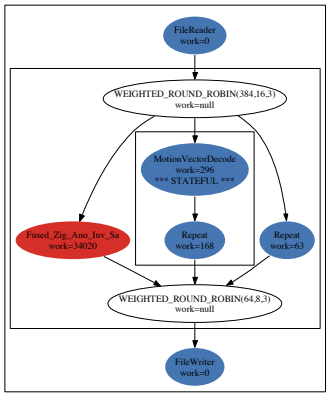
(a) ChannelVocoder



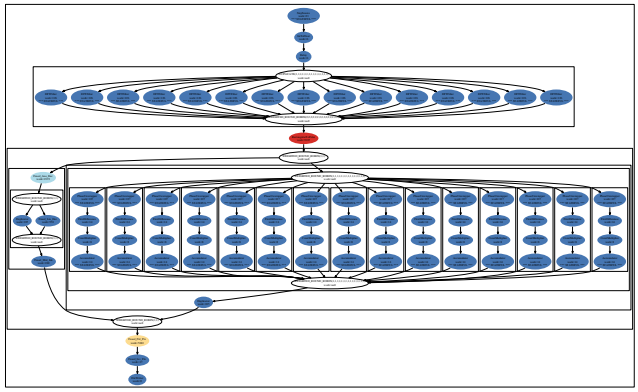
(b) Filterbank



(c) FMRadio



(d) MPEG2Decoder



(e) Vocoder

Figure 6-9: The coarsened StreamIt graphs for (a) ChannelVocoder, (b) Filterbank, (c) FMRadio, (d), MPEG2Decoder, and (e) Vocoder.

Algorithm 2 Heuristic algorithm for calculating the number of ways a filter should be fissed to fill all cores with task or data-parallel work.

```
▷ F is the filter to fiss; N is the total number of cores
int JUDICIOUSFISSION(filter F, int N)
  ▷ Estimate work done by F as fraction of
  ▷ everyone running task-parallel to F
  fraction = 1.0
  Stream parent ← GETPARENT(F)
  Stream child ← F
  while parent ≠ null do
    if parent is splitjoin then
      total-work ←
          ∑_{c∈CHILDREN(parent)} AVGWORKPERFILTER(c)
      my-work ← AVGWORKPERFILTER(child)
      fraction ← fraction * my-work/total-work
    end if
    child ← parent
    parent ← GETPARENT(parent)
  end while
  ▷ Return fission factor influenced by F's task-parallel work
  return CEIL(fraction * N)
```

the task-parallel filters are a natural part of the algorithm and avoid any computational overhead imposed by filter fission (e.g., fission of peeking filters introduces a decimation stage on each fission product).

In order to balance task and data parallelism, we employ a "judicious fission" heuristic that estimates the amount of work that is task-parallel to a given filter and fisses the filter accordingly. Depicted in Algorithm 2, this algorithm works by ascending through the hierarchy of the stream graph. Whenever it reaches a splitjoin, it calculates the ratio of work done by the stream containing the filter of interest to the work done by the entire splitjoin (per steady-state execution). Rather than summing the work within a stream, it considers the average work per filter in each stream so as to mitigate the effects of imbalanced pipelines.

After estimating a filter's work as a fraction of those running in parallel, the algorithm attempts to fiss the filter the minimum number of times needed to ensure that none of the fission products contains more than $1/N$ of the total task-parallel work (where N is the total number of cores). Note that if several filters are being fissed, the fission transformations must be applied after JUDICIOUSFISSION is called on each filter to be fissed. After judicious fission, we fuse all adjacent filters directly contained in a pipeline. This is done to force the fused filter communication to occur within a single core.

The driver of the judicious fission pass only fisses filters in which the estimated computation to communication ratio is above a given threshold for an architecture. This threshold is determined empirically. For example, we use a threshold of 10 compute instructions to 1 item of communication for the Raw microprocessor.

6.3.1 Judicious Fission Applied to the StreamIt Core Benchmark Suite

Judicious fission applied to the six benchmarks that do not include state, prework, or peeking (BitonicSort, DCT, DES, FFT, Serpent, and TDE) will result in the single filter of the coarsened graph being fissed by a factor equal to the number of cores. The judiciously fissed versions of these benchmarks include no communication other than application input and output. Figure 6-10 gives the result of applying JUDICIOUSFISSION with $N = 16$ to ChannelVocoder, Filterbank, FMRadio, and MPEG2Decoder. The Vocoder benchmark is not shown because of the large amount of stateful computation in the benchmarks; Vocoder is better served by the techniques of Chapter 7.

- **ChannelVocoder**: See Figure 6-10(a). The coarsened graph (see Figure 6-9(a)) includes one LowPass filter with no task parallel work. JUDICIOUSFISSION calculates that this filter should be fissed 16 ways to occupy all the cores of the chip. The remaining filters are unfused as each filter is a member of a 17-way task parallel unit. The fusion is applied to each pipeline of the 17-way splitjoin to reduce communication as described above.

- **Filterbank**: See Figure 6-10(b). The coarsened graph includes 8-way its main splitjoin (see Figure 6-9(b)). The filters of this splitjoin are fissed 2-ways. The final filter (before the file output filter) is not fissed because its computation to communication ratio is too low.

- **FMRadio**: See Figure 6-10(c). Each filter of the coarsened graph (see Figure 6-9(c)) is fissed 16 ways because there is no task parallelism.

- **MPEG2Decoder**: See Figure 6-10(d). The coarsened graph (see Figure 6-9(d)) includes 3-way task parallelism. However, the load is concentrated in the left filter. This filter is fissed 14-ways to create 16-way parallelism.

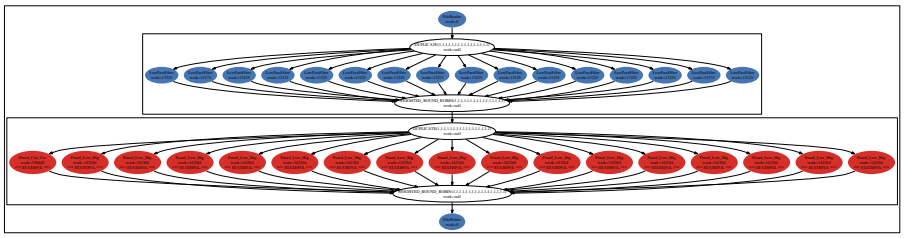
6.3.2 Alternative Method for Applying Judicious Fission

Algorithm 2 examines the StreamIt graph in order to calculate the judicious fission factor for each filter. The algorithm ascends the hierarchy of the graph and makes the assumption that task parallelism is symmetric in the graph. By calling AVGWORKPERFILTER for each child stream of a splitjoin, the algorithm is assuming that task parallel streams have the same distribution of work as the parent stream of the filter F. For the StreamIt Core benchmark suite, task parallel symmetry exists for all benchmarks, and it is a general pattern in the streaming domain that task parallel computation is symmetric. However, this is not a general rule. For example, Figure 6-11 demonstrates an example where Algorithm 2 does not calculate the optimal mapping for 4 cores. This lack of generality motivates an alternative method for calculating judicious fission by utilizing the general stream graph.

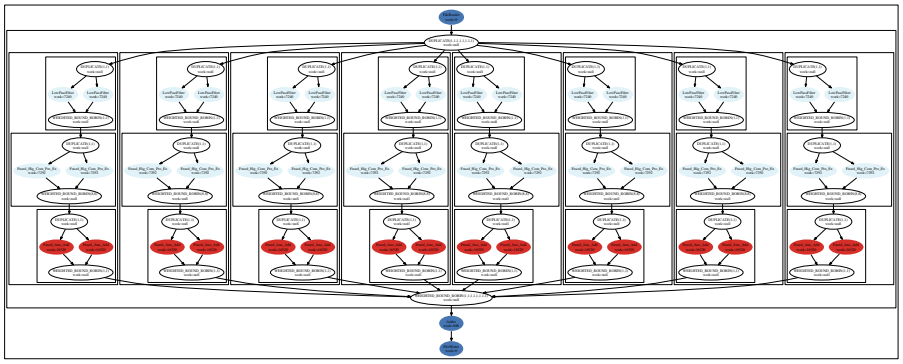
For an acyclic stream graph, we can assign *levels* to each filter in the graph. The first level l_0 consists of the source filters. Each successive level l_i is the set of filters whose input dependences are satisfied by previous levels l_J where $j < i$. After we have assigned levels to the filters of the graph, we can calculate the judicious fission factor for filter f as:

$$\textit{JudiciousFission}(f, G, N) = \left\lceil N \cdot \frac{s(W_f, f) \cdot M(S, f)}{\sum_{t \in l_f} s(W_t, t) \cdot M(S, t)} \right\rceil$$

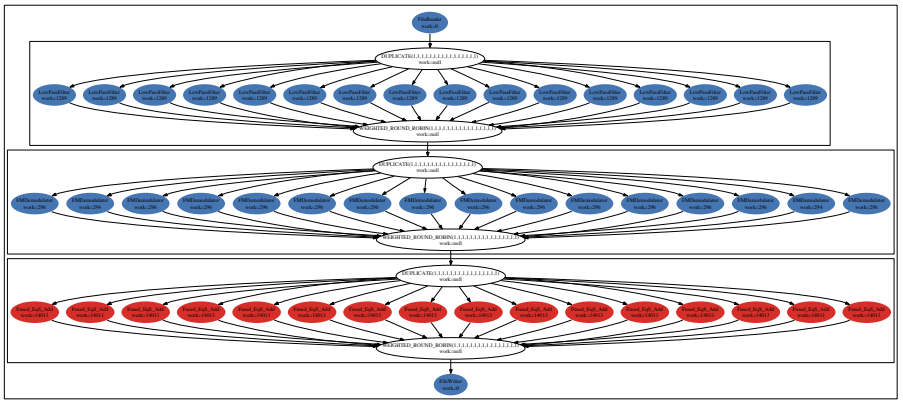
where N is the number of cores to target and l_f is the set of filters in f's level.



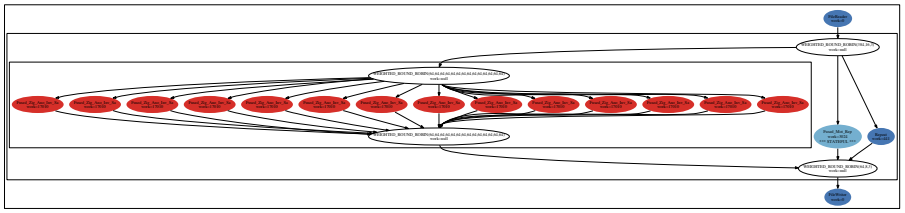
(a) ChannelVocoder



(b) Filterbank



(c) FMRadio



(d) MPEG2Decoder

Figure 6-10: The judiciously fissed StreamIt graphs for (a) ChannelVocoder, (b) Filterbank, (c) FMRadio, and (d), MPEG2Decoder.

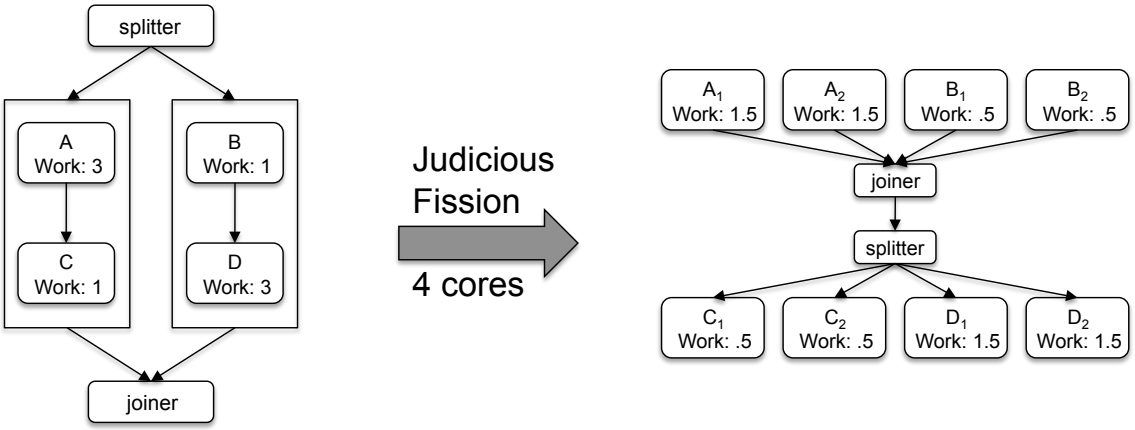


Figure 6-11: Example of asymmetry problems with the Judicious Fission formulation on the StreamIt graph. The original stream graph is given on the right, and the results of judicious fissed graph is given on the left. There exists an asymmetry between the workloads of the filters in the pipelines of the splitjoin; the workload is concentrated at different levels. The results of judicious fission will average the workloads for each pipeline, missing the fact the work is concentrated at different levels. The result will fiss each filter by 2 to create a 4 core mapping. This is not the optimal result: A and D should be fissed 3 times, and C and B not fissed at all.

This method is more precise with calculating task parallel work as it does not average the work in task parallel containers as Algorithm 2 does. For the StreamIt Core benchmark suite, this method calculates identical factors as Algorithm 2. This method is used in the Tilera path (Section 8.8) and SMP path (Section 8.9) of the StreamIt compiler, as well as [Tan09].

6.3.3 The Fission Transformation on the StreamIt Graph

Section 6.3 details the JUDICIOUSFISSION algorithm employed by the CGDTP path of the compiler. This algorithm calculates a fission factor for each filter of the stream graph. Once a factor has been calculated for all filters, the actual fission transformation is performed. For this chapter, our fission transformations operate on the StreamIt graph. In Chapter 8, we develop optimized fission techniques that require the more expressive data distribution of the general stream graph.

In the simple case, for a filter f where the peek rate is equal to the pop rate, i.e., $e(W_f, f) = o(W_f, f)$ and $e(W_f^p, f) = o(W_f^p, f)$, the fission transformation duplicates the filter and places it in a splitjoin with a roundrobin splitter and a roundrobin joiner. The weights of the splitter are equal to the pop rate of the original filter, and the weights of the joiner are equal to the push rate of the original filter. The constituent filters are exact copies of f.

For f that peeks in the steady-state, i.e., $e(W_f, f) > o(W_f, f)$ and $e(W_f^p, f) = o(W_f^p, f)$, a different transformation is applied (see Figure 6-12). In this case, the splitter is a duplicate, since the component filters need to examine overlapping parts of the input stream. The i'th component has a steady-state work function that begins with the work function of f, but appends a series of $(K-1) * o(W_f, f)$ pop statements in order to account for the data that is consumed by the other components. Also, the i'th filter has a prework function that pops $(i-1) * e(W_f, f)$ items from the input stream, to account for the consumption of previous filters on the first iteration of the splitjoin. As before, the joiner is a roundrobin that has a weight of $o(W_f, f)$ for each stream.

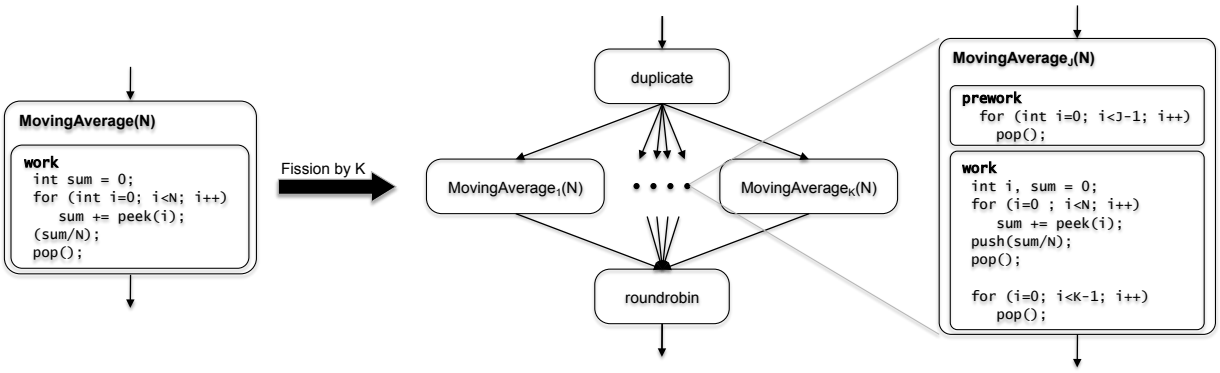


Figure 6-12: Fission of a filter that peeks.

6.4 Evaluation: The Raw Microprocessor

This section details our evaluation of the coarse-grained data and task parallelism (CGDTP) path of the StreamIt compiler in the context of the Raw 16-core microprocessor. The configuration used for this evaluation differs slightly from the configuration used in Section 5.7 in that we simulate a memory module attached to each I/O port of the Raw microprocessor. We employ a simulation of a CL2 PC 3500 DDR DRAM, which provides enough bandwidth to saturate both directions of a Raw port [TLM+04]. Additionally, each chipset contains a streaming memory controller that supports a number of simple streaming memory requests. In our configuration, 16 such DRAMs are attached to the 16 logical ports of the chip. The chipset receives request messages over the dynamic network for bulk transfers to and from the DRAMs. The transfers themselves can use either the static network or the general dynamic network (the desired network is encoded in the request).

The Raw microprocessor is rare in that it is often more efficient to stream data to / from offchip memory versus streaming data to / from another core. In the compiler, we elected to buffer all streaming data off-chip. Given the Raw configuration we are simulating and for the regular bulk memory traffic we generate, it is more expensive to stream data onto the network from a core's local data cache than to stream the data from the streaming memory controllers. A load hit in the data cache incurs a 3-cycle latency. So although the networks are register-mapped, two instructions must be performed to hide the latency of the load, implying a maximum bandwidth of 1/2 word per cycle, while each streaming memory controller has a load bandwidth of 1 word per cycle for unit-stride memory accesses. When targeting an architecture with more modest off-chip memory bandwidth, the stream buffers could reside completely in on-chip memory. For example, the Tilera compiler path described in Section 8.8 allocates all streaming buffers to core data cache.

6.4.1 Layout: Minimizing Critical Path Work

The layout phase of the CGDTP flow for Raw assigns filters of the coarsened, judiciously fissed graph to cores. Layout in the data and task parallelism path of the StreamIt compiler is equivalent to static scheduling of a coarse-grained dataflow DAG to a multiprocessor which has been a well-studied problem over the last 40 years (see [KA99b] for a good review). We leverage sim-

ulated annealing as randomized solutions to this static scheduling problem are superior to other heuristics [KA99a].

Briefly, we generate an initial layout by assigning the filters of the stream graph to processors in dataflow order, assigning a random processing core to each filter. The simulated annealing perturbation function randomly selects a new core for a filter, inserting the filter at the correct slot in the core's schedule of filters based on the dataflow dependencies of the graph. The cost function of the annealer uses our static work estimation to calculate the maximum critical path length (measured in cycles) from a source to a sink in the graph. After the annealer is finished, we use the configuration that achieved the minimum critical path length over the course of the search. Communication and synchronization costs of a layout are ignored.

6.4.2 Mapping to the Raw Microprocessor

Once the graph has been coarsened, judiciously fissed, and layout has assigned filters to cores, the StreamIt graph is ready for mapping to the cores of the Raw microprocessor. The StreamIt graph is first converted to an unstructured graph where nodes can have multiple inputs and multiple outputs. This representation is covered in Section 2.1. Splitters and joiners do not appear in this graph; the unstructured graph folds splitters into their upstream filter, and joiners are folded into their downstream filters. A synchronization removal pass to convert the graph into the unstructured form finds the final filter destinations for all output items, and encodes this in the distribution schedules for each node in the unstructured graph. Section 8.3 covers this translation in more detail.

Since the unstructured graph supports multiple input and multiple output nodes, a joiner followed by a duplicate splitter (as appears in the judiciously fissed ChannelVocoder in Figure 6-10(a)) will be converted into all-to-all communication. Data does not have to be collected at a single joiner and then redistributed.

Across the cores, filters execute in dataflow order, with each filter reading its input from and writing its output to banked offchip DRAM memory. Each core executes a schedule of filters assigned to it and each filter executes on a single processing core for the entire execution. Each core executes independently until a split or join point is reached. As opposed to the hardware pipelining path of Chapter 5 joiners do not occupy a core. Splitting and joining is performed by the static network as programmed by the compiler. The static network routes items from their source to destination, and performs duplication in network. Figure 6-13 details the execution steps for the ChannelVocoder benchmark when mapped to Raw using the techniques covered in this chapter.

As mentioned above, all filters read their input from and write their input to off-chip memory configured with a streaming controller. Each core is assigned a DRAM bank that it always reads from or write to. If a producer and consumer pair are mapped to different cores, the items must be moved from the producer's bank to the consumer's bank. The edges between nodes of the unstructured graph are mapped to buffers allocated in off-chip memory banks. As an optimization, not all buffers edges are represented with buffers. For example, if a producer and consumer occupy the same core, the producer is single output, and the consumer is single input, the three buffers (output of the producer, edge between the producer and consumer, and input to the consumer) are folded into a single buffer allocated in the DRAM mapped to the core. Folding also occurs whenever a node is single input or single output.

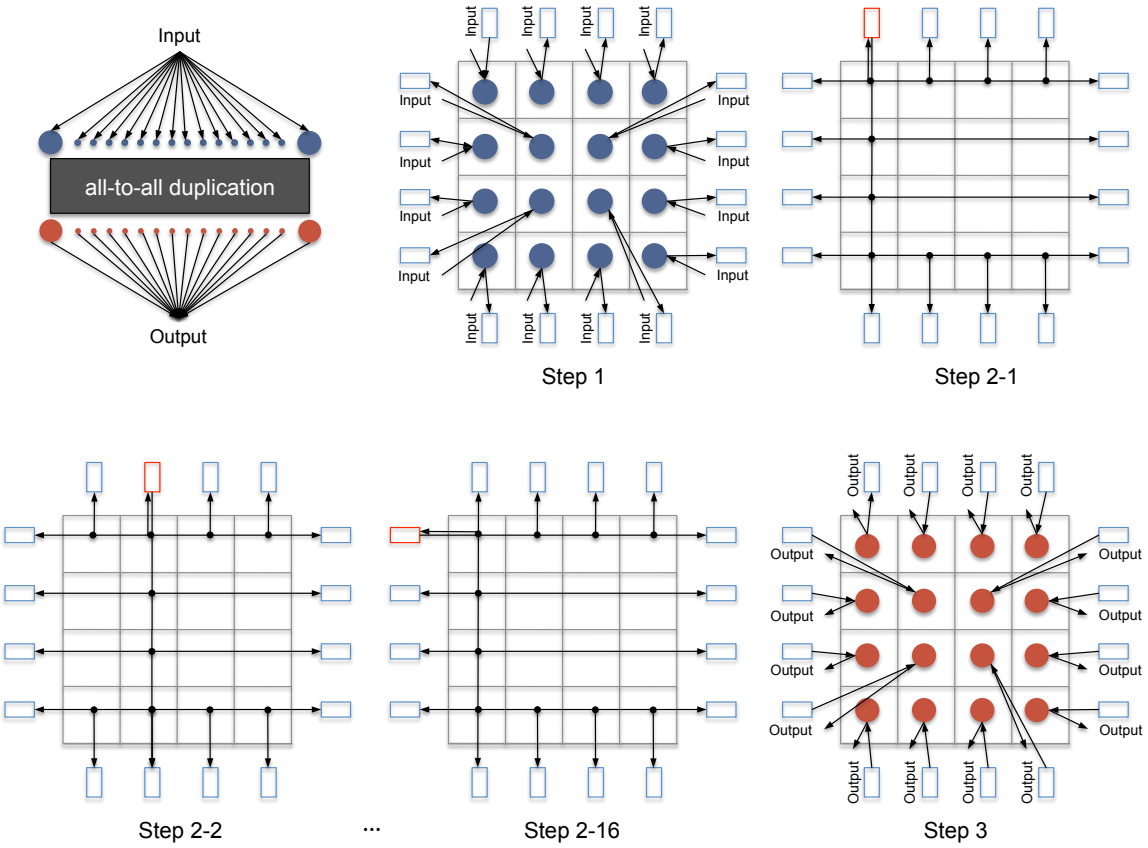


Figure 6-13: ChannelVocoder execution steps for CGDTP on Raw. The StreamIt graph (see Figure 6-10(a)) is de-structured into the graph shown top-left. Step 1: the fissed LowPass filter executes on the 16 cores, reading their input from the 16 I/O ports of the Raw processor and writing to DRAM. Steps 2-1 through 2-16 perform the all-to-all communication. During each sub-step, items are read from a source memory bank (highlighted in red), and the items are duplicated to all memory banks. The compiler programs the static network to achieve the duplication of all items in the banks. This process repeats 16 times for all 16 memory banks. Step 3: the fissed fused (red) filters executes, reading their input from the duplicated items that were written into their memory bank in step 2. The filters write their output to the 16 I/O ports of the Raw processor.

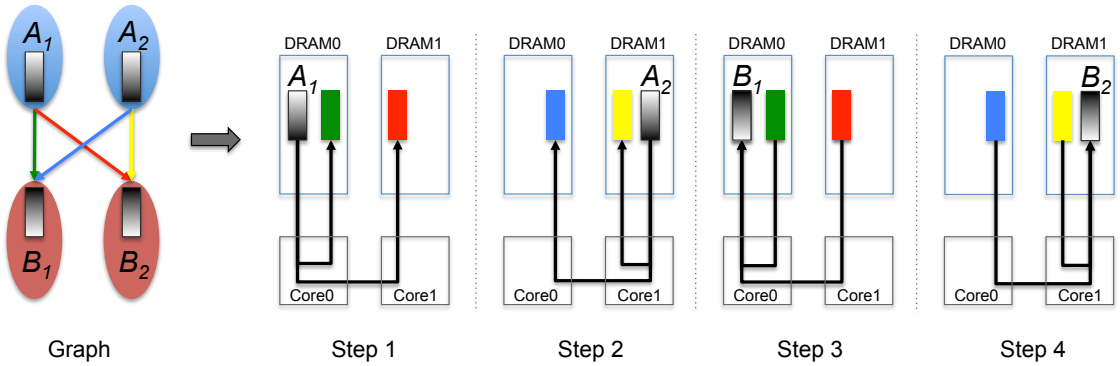


Figure 6-14: DRAM transfer steps for a 2x2 all-to-all communication for the graph on the left. Each colored edge corresponds to a channel that is buffered off-chip in DRAM. In Step 1, A_1's output is transferred from DRAM0 to the green and red buffers in DRAM0 and DRAM1, respectively. The items flow onto the chip, where the static network implements the splitting pattern as encoded by the compiler. In Step 2, A_2's output is split to the blue and yellow buffers. In Step 3, the green and red buffers are joined into B_1's input buffer. The joining pattern in implemented by the static network. In Step 4, B_2's input is joined from the blue and yellow buffers. Eight transfers were required for this 2x2, all-to-all distribution.

For splitting, data is moved from the output buffer of the filter to intermediate buffers that represent the edges of the graph. For joining, data is moved from buffers that represent the incoming edges to the input of the filter. Figure 6-14 gives an example of the buffers required and the transfers required to perform a 2x2, all-to-all distribution. Note that each edge buffer for a split of the output of a node must be assigned to a distinct DRAM. The same is true for the buffers edges of a join for filter. The compiler includes a pass that assigns channel buffers (the colored buffers in Figure 6-14) to DRAMs. This pass uses a greedy algorithm that for each splitting and joining at a node, orders the edges by the number of items that flow over the edge in the steady-state, and assigns the most trafficked buffer to the unassigned dram closest to the source (for splitting) or destination (for joining). This greedy pass improves spatial locality and leads to a 7% throughput improvement for 16 cores versus a random assignment of edge buffers to DRAM banks.

As mentioned above, for fission of peeking filters, the Raw path implements a duplication and decimation approach to data distribution. This is because duplication is inexpensive on Raw. An item can be injected onto the network once, and the switch processors can perform duplication by routing a single source to multiple destinations. In this way, we can duplicate an item from one core to all 16 cores in the worst case 6 cycles, the time to route one time from one corner of the chip to the opposite corner. The compute processor will then store all the duplicated items in memory, but ignore the items that it does not require. The explicit decimation in Figure 6-12, the `pops` performed in the `for` loops, are not performed and are translated into a single index increment.

Not all multicore architectures support such efficient duplication; this technique is specific to Raw. Chapter 8 explores optimizations of the duplication and decimation technique for communication mechanisms without support for efficient duplication.

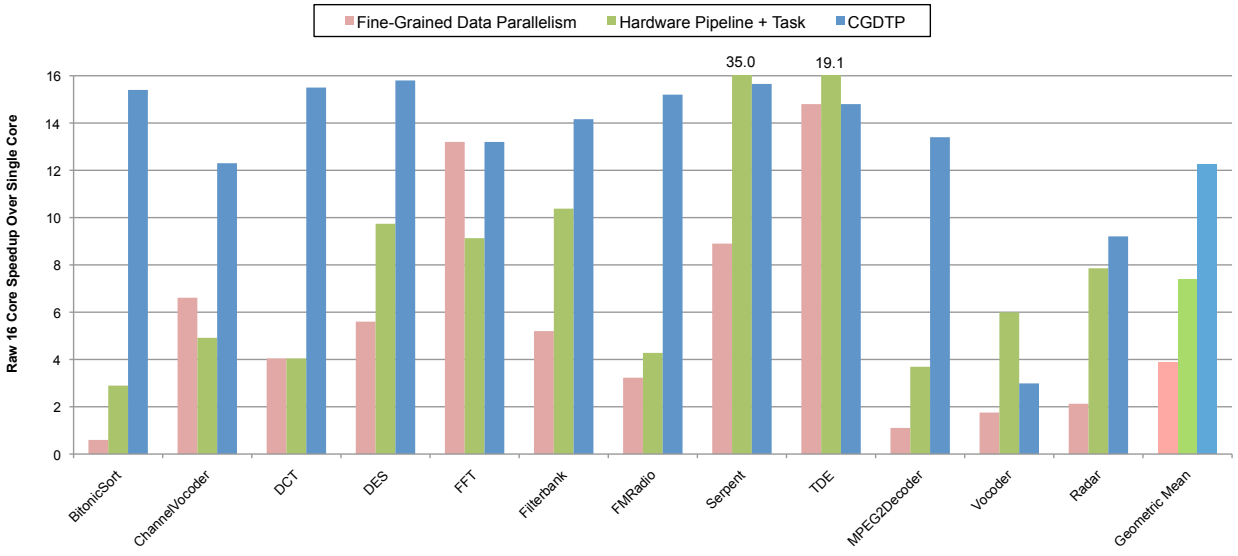


Figure 6-15: 16-core Raw speedup normalized to single core for "Hardware Pipeline + Task" and "Coarse-grained Data + Task".

6.4.3 Results

We evaluate the techniques of this chapter on the StreamIt Core benchmark suite introduced in Chapter 4. Figure 6-15 presents the 16 core speedup over single core sequential throughput for fine-grained data parallelism, the hardware pipelining techniques of Chapter 5 and the techniques of this chapter (CGDTP). Our technique for exploiting coarse-grained data and task parallelism achieves a mean performance gain of 12.2x over a sequential single-core baseline, and 1.7x speedup over hardware pipelining. This also represents a 3.2x speedup over a fine-grained data parallel approach (see Section 6.1).

9 of the 12 applications have higher 16 core throughput using the techniques in this chapter versus hardware pipelining. BitonicSort, ChannelVocoder, DCT, DES, Filterbank, FMRadio, and MPEG are stateless benchmarks that benefit from the more effective load balancing of the techniques in this chapter. Hardware pipelining struggled to find load-balanced contiguous partitionings for these benchmark. CGDTP, on the other hand, achieves a load balanced parallel mapping by judiciously data-parallelizing to 16 core units, and swapping between these units. For benchmarks without task parallelism in the coarsened graph (BitonicSort, DCT, DES, and FMRadio) each fissed stage achieves load balancing because the fissed filters are executing the same work function (with the exception of decimation).

The benchmarks with that include task parallelism in their coarsened graph (ChannelVocoder, Filterbank, and MPEG2Decoder) rely on work estimation to incorporate task parallelism into exploited data parallelism via the JUDICIOUSFISSION algorithm of Algorithm 2. Filterbank includes 8-way task parallelism where each level of the coarsened graph performs the same operation but with different coefficients. It is not required that work estimation be highly accurate in this case. This is a property we see often in benchmarks, task parallel filters performing the same operation only with different coefficients. This mitigates the need for accurate work estimation. The execution visualization for Filterbank on a 16 core Raw microprocessor is given in Figure 6-16. We can

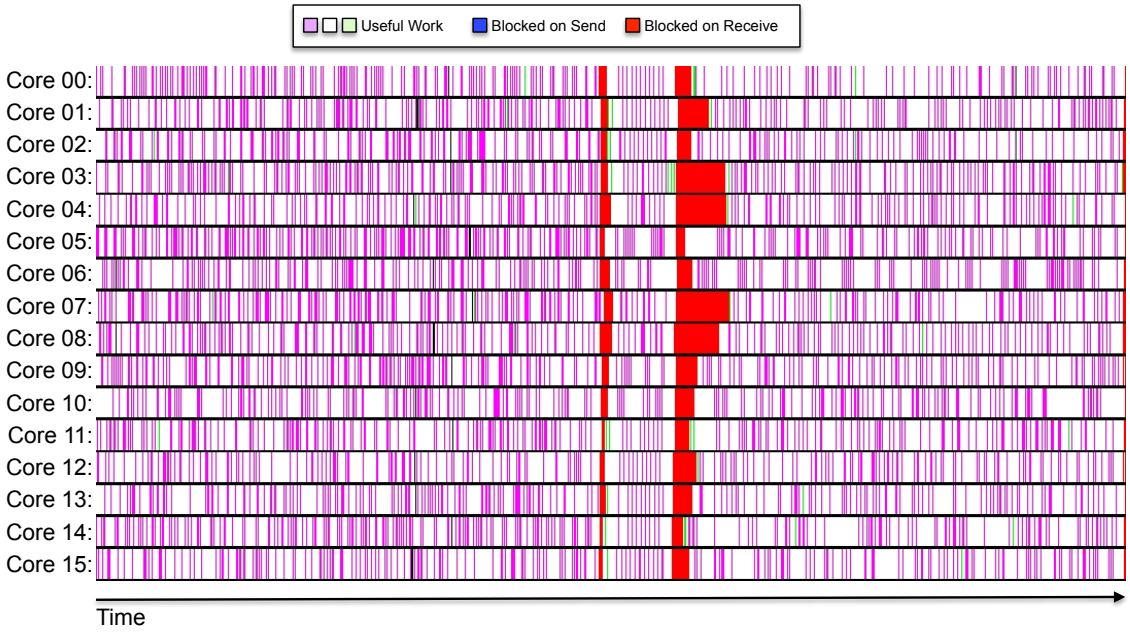


Figure 6-16: Raw execution visualization for Filterbank. From figure 6-10(b), we see we see that the judiciously fissed Filterbank graph includes 3 levels of 16-way parallelism. One can clearly see the 3 stages of computation. Each stage is demarcated by a period of network activity that corresponds to the data distribution between levels.

see that the mapping offers high utilization with periods of compute inactivity during data distribution. The techniques in this chapter achieves a 14.1x speedup for Filterbank. Another quality of Filterbank that accounts for the excellent speedup is its large computation to communication ratio (see Table 4-17).

Interestingly, for Filterbank we see that the static work estimation has mis-estimated the relatively work load of the different levels, as the judicious fissed graph (Figure 6-10(b)) shows one stage that dominates the load. However, the execution visualization does not reflect this. CGDTP still achieves an effective mapping because the techniques does not require accurate work estimation between filters of different levels.

MPEG2Decoder includes a filter with state. However, this stateful filter is not the bottleneck of the coarsened, judiciously fissed graph (see Figure 6-10(d)). Just as important, this filter is task parallel to the stateless bottleneck of the graph. Fissing the stateless bottleneck 14-ways, such that the single level of the graph possesses 16-way parallelism, achieves a very efficient mapping leading to a 13.4x speedup. The techniques in this chapter do not explicitly attempt to parallelize stateful computations, however, in the case of MPEG2Decoder we are fortunate that the stateless filter is task parallel to the stateless bottleneck.

The Radar benchmark shows little improvement over the hardware pipelined mapping because of the preponderance of stateful filters. The techniques in this chapter do not extract any additional parallelism from this benchmark. The 12 way task parallelism of the top splitjoin and the 4-way task parallelism of the bottom splitjoin (see Figure 4-14) offer enough parallelism to achieve a 9.2x speedup over single core.

3 of the 12 applications have lower 16 core throughput for the CGDTP approach versus the hardware pipelined approach. For Vocoder, the presence of numerous stateful filters limits what can be data parallelized. Figure 6-9(e) gives the coarsened graph for Vocoder. Although there is ample task parallelism in the graph, relying only on the techniques of this chapter leads to too much inter-core communication and synchronization for the Vocoder mapping. The techniques presented in Chapter 7 extend the techniques of this chapter to add pipeline parallelism to parallelize stateful components and additional fusion techniques to reduce inter-core communication and synchronization between stateful components.

Hardware pipelining performs well for TDE and Serpent. This is because these applications contain long pipelines that can be load-balanced, and their rate-matched pipelined communication is effectively mapped to the static network of Raw. For these benchmarks, the hardware pipelined mapping does not require full fusion of filters, so this small benchmark executes without fusion overhead. For the hardware pipelined versions of these benchmarks, the data and code footprint of the is spread across the 16 cores of the chip, leading to super-linear speedups over single core performance. Conversely, the CGDTP mapping fuses the pipeline down to a single filter; each fission product executes fusion code, and the computation of the entire benchmark.

6.5 Evaluation: Tilera TILE64

This section details the evaluation of CGDTP for the Tilera TILE64 architecture. The communication and synchronization scheme for the TILE64 is quite different from the scheme employed from Raw detailed above. The main difference is that items are directly transferred between cores, and are not stored in offchip memory. The code generated by the StreamIt compiler for the TILE64 processor follows the remote store programming (RSP) model [HWA10]. The filters mapped to a single core execute in a single process that is mapped to the core for the duration of execution. Each process has a private address space, but each process can award remote processes write access to their local memory. When a producer process has write access to a consumer process's memory, the producer communicates directly with the consumer via store instructions whose destination is an address in the consumer's shared memory. Communication is initiated by the producer, and is fine-grained. The consumer reads directly from it's local memory (L2) when accessing input. Each remote producer and consumer pair are mapped to physically proximal cores so that communication latency is minimized.

The TILE64 architecture allows a process to allocate shared memory that is *homed* on the allocating core. When a process grants write access to homed memory to a remote process, the remote process can write into the allocated block directly. For a remote write, no cache line is allocated on the producer core; writes stream out of the core without generating coherence traffic. After each filter executes, remote store statements are executed that transfer the items that are required to the consumers on remote cores.

In the case where producer f and consumer g are mapped to the same core, and f writes to a contiguous region of g's input, f and g have a single buffer between them, and data does not need to be reordered after between f's output and g's input. Otherwise, 2 buffers are present between f and g, a buffer for f's output and a buffer for g's input. Data is written into g's input buffer at the appropriate indices given g's input distribution using a combination of remote stores for producers mapped to a different core than g, and local stores for the producer mapped to the same core as g.

The RSP model provides for computation and communication concurrency. Each word communicated between producer and consumer requires occupancy in the consumer's computational pipeline in the form of the remote store instruction. However, once this instruction is executed, the underlying cache coherence mechanism is responsible for forwarding the word to the consumer, and the producer core is free to do other work. The communication requires no occupancy on the consumer core; however, L2 memory bandwidth on the consumer core is utilized by the remote write.

To take advantage of this concurrency, and to coarsen the granularity of synchronization, the StreamIt compiler double-buffers producers and consumers. During execution, each producer and consumer are executing at different steady-state iterations of the application. Buffering is introduces so the producer can "get ahead" of the consumer. During execution, the producer writes into a buffer that the consumer will read on the the consumer's next iteration (as opposed to the consumer's current iteration). Execution is enabled via the same buffering and scheduling techniques used to enable coarse-grained software pipelining in Chapter 7. We do not refer to the execution here as "software-pipelined" because a producer and consumer are never scheduled execute in parallel in time. Only data and task parallelism are exploited in the mapping.

Synchronization is handled directly between each producer and consumer pair mapped to different cores. At the end of each steady-state, each process executes a memory fence operation; then each process notifies it's producer processes that it is ready for more data.[2] Synchronization within the double-buffered steady-state is not necessary since the producer is writing to and the consumer is reading from distinct buffers.

The layout algorithm considers each level of the judiciously fissed graph in turn. The number of filters in each level is less than or equal to the number of cores. If this is not the case for a level, we successively fuse the 2 neighbors with the least amount of combined work in the level until this precondition is achieved.[3] Each filter is greedily mapped to the core that will minimize the number of items that will have to be communicated inter-core.

I/O is handled via 4 interfaces that provide approximately 20GB/s of bandwidth. Input is streamed onto the chip from these 4 interfaces and forwarded to each tile via the I/O network. Output is handled similarly, and as with Raw, the cost of constituting the output into a single stream is not measured. The I/O bandwidth is not a bottleneck for any of our benchmarks.

6.5.1 Results

The graph is Figure 6-17 gives the throughput speedup results for CGDTP over single core throughput for 4, 16, 36, and 64 tile configurations. The mean speedup over single core for 16 cores is 10.7x and 37.6x for 64 cores. A 10.7x 16 core speedup is lower than Raw's 12.2x 16 core speedup. This is because the communication mechanism we are using on Tilera, remote writes implemented via distributed shared memory, is not as efficient as Raw's static network for fine-grained communication of this nature. Also, Raw includes 14 hardware I/O ports that are able to keep up with the I/O requirements of the data parallelized filters of a CGDTP mapping. 7 of the 12 benchmarks scale well with 64 core speedups over 51x (BitonicSort, DCT, DES, FFT, Serpent, TDE,

[2]The notification from the consumer core to the producer core is communicated via TILE64's user-level streaming network.

[3]JUDICIOUSFISSION will never create a level with more filters than cores, however, the coarsened graph may have levels with more filters than cores for small configurations with low core counts (e.g., 4 cores and 16 cores).

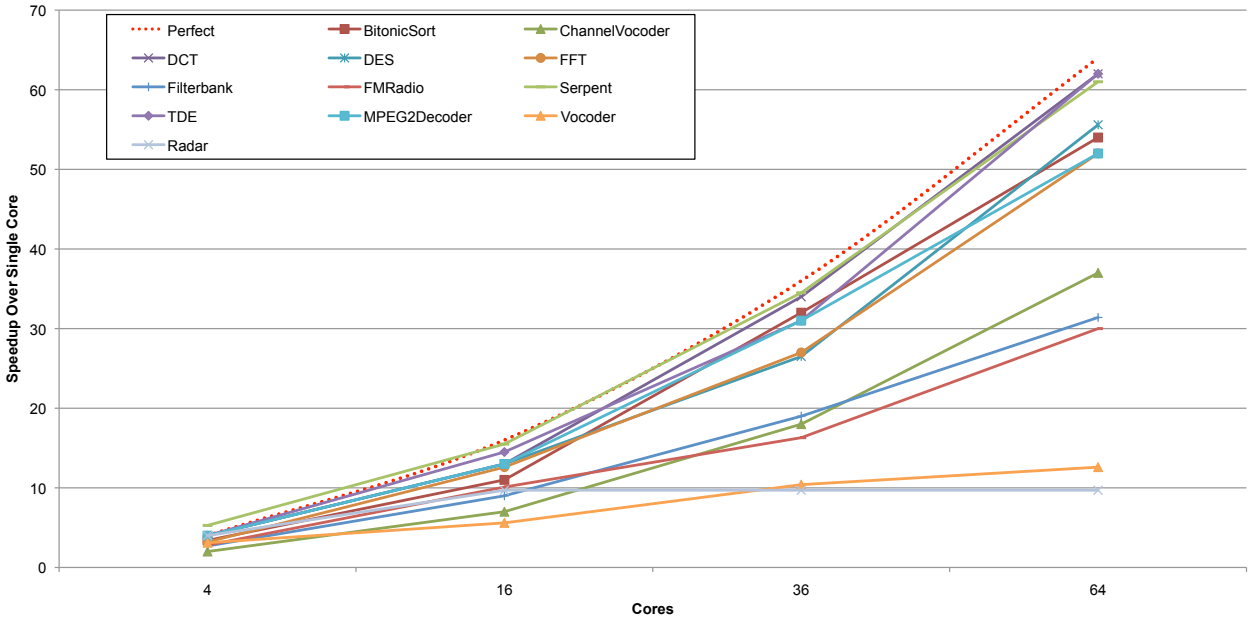


Figure 6-17: The throughput speedup results for CGDTP over single core performance for different configurations of the TILE64.

and MPEG2Decoder). These benchmarks contain little or no stateful computation, and do not include peeking. For these benchmarks, CGDTP is effective at determining a load-balanced parallel mapping, and keeping low the communication requirement of the data parallel mapping.

For the remaining 5 benchmarks, 64 core speedups are quite poor, and fall short of a scalable result (under 38x for 64 cores). There are two reasons for this. ChannelVocoder, FilterBank, FMRadio, and Vocoder include peeking filters. Fissing the peeking filters of these benchmarks requires inter-core communication in order to share the items required by multiple fission products. The cost of the inter-core communication requirement for these benchmarks reduces the effectiveness of data parallelization. The problem is that duplication is not efficiently supported in Tilera's network as is the case with Raw's static network. This shortcoming motivates the techniques for optimized fission of peeking filters covered in Chapter 8.

The reason for the lack of performance scaling for Vocoder (in addition to fission of peeking filters) and Radar is that they include significant amounts of stateful computation. CGDTP does not explicitly parallelize stateful filters. It must instead rely on the presence of task parallelism for each stateful filter. For Radar, there exists some task parallelism (see Figure 4-14), but it is not enough to continue scalable parallel performance past 16 cores. The cut off in scaling for Vocoder is similar (see Figure 4-13). This shortcoming of CGDTP motivates the techniques of Chapter 7 that leverage pipeline parallelism to parallelize stateful filters.

6.6 Evaluation: Xeon 16 Core SMP System

The compiler remains largely unmodified for the SMP path as compared to the TILE64 path, demonstrating the portability of our techniques. We will highlight the main differences; for more details on the implementation of the SMP path, see [Tan09]. The SMP path utilizes the cache

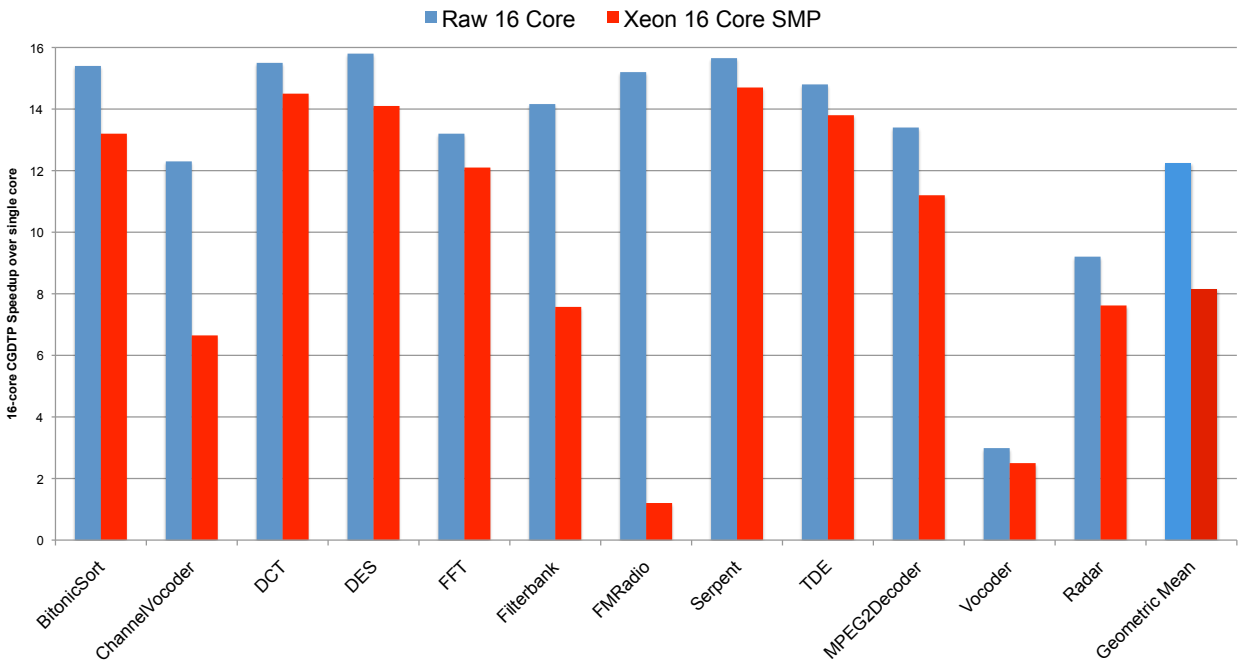


Figure 6-18: 16-core throughput speedup results for CGDTP over single core performance for the Xeon system. Each set of results is normalized to the single-core performance of the respective architecture.

coherency mechanism of the architecture in order to communicate items from producers to consumers. The code generation pass for SMP allocates a single buffer for the input of all products of a single fission application. Each fission product reads from the single shared input buffer at an appropriate offset. Each filter in the judiciously fissed graph has its own output buffer. Items are transferred from output buffers to input buffers using store instructions. For our purposes, the stores are functionally equivalent to the remote writes of Tilera, although for an SMP architecture each store generates much more coherence traffic than a remote write. As in the TILE64 case, fission products are mapped to distinct cores. The cache-coherence mechanism is responsible for sharing the input buffer across the cores of the products, moving locations back and forth between the storing producer core and the loading consumer core. A global barrier instruction is included at the end of each steady-state to synchronize producers and consumers. Producers and consumers are again double-buffered so that synchronization is not needed within a steady-state.

6.6.1 Results

Figure 6-18 provides the 16-core throughput speedup results for the 16 core Xeon configuration. The results for the Raw processor as duplicated in Figure 6-18 for reference. Each set of 16 core results are normalized to single core performance on the respective architecture. The mean 16-core speedup for the Xeon is 8.2x for the StreamIt Core benchmark suite. The results are scalable for the 7 benchmarks that do not include peeking or significant amounts of state (each is over 11x). We are very happy with these results considering the SMP communication mechanism is not optimized for streaming communication. The same problems appear on the SMP as in Tilera. CGDSP cannot

parallelize stateful computation (Vocoder and Radar). Also, the SMP communication mechanism does not have support for efficient duplication between cores, accounting for inefficient fission of peeking filters (ChannelVocoder, Filterbank, FMRadio, and Vocoder).

6.7 Analysis

This section collects some of the conclusions that we reached in this chapter regarding coarse-grained data and task parallelism. Exploiting data parallelism is the accepted technique for mapping streaming application to parallel targets. However, for achieving portability of source code, application graphs must be transformed into a form appropriate for exploiting data parallelism, and other forms of parallelism should not be ignored:

1. **While there is ample data parallelism in most streaming applications, data parallelism must be exploited at the appropriate granularity.** Section 6.1 demonstrates that leveraging data parallelism naïvely (for each filter of the original application) produces communication and synchronization that overwhelms the benefits of parallelism. The granularity of the source application must be adjusted such the unit of data parallelism is appropriate for the source. COARSENGRANULARITY adjusts the granularity by fusing filters, making larger data parallel units that require less communication and synchronization versus smaller units. Also, the algorithm is careful not to destroy data parallelism. COARSENGRANULARITY assumes that there is always a benefit to fusing, however this may not always be the case because as we fuse we create filters with larger instruction and data footprints. We never experienced a limit to the fusion benefit, however the case might be different for embedded processors with small caches. It is trivial to extend the algorithm to limit the size of the coarsened filters.

2. **StreamIt's support for explicit producer/consumer relationships enables the fusion transformations that are employed by COARSENGRANULARITY.** Producer / consumer pairs are trivial to recognize as they are created by the programmer via the `add` statements of a pipeline container. Furthermore, single input and single output filters simplify the fusion transformations, as they require only local analysis and transformations.

3. **Task parallelism can complement data parallelism to reduce the amount of data parallelism that is required to be exploited, thus reducing communication and synchronization.** Many streaming applications include task parallelism. JUDICIOUSFISSION seeks to incorporate task parallelism as it attempts to create load-balanced sections (levels) of work from the coarsened application. This reduces the communication and synchronization requirement for the fission of peeking filters as filters are fissed at reduced factors.

4. **While JUDICIOUSFISSION still relies on accurate work estimation when task parallelism is present, it is typically less demanding versus hardware pipelining for accurate work estimation.** As demonstrated in Chapter 5, hardware pipelining requires accurate work estimation in order to partition the graph into load balanced units. JUDICIOUSFISSION incorporates work estimation while leveraging task parallelism, however, it is often the case that task parallel filters perform the same function.

5. **Data parallelism is not the most effective mapping for all benchmark and architecture pairs.** For infinite streaming applications, data parallelism seems to offer the perfect form of parallelism. However, for multicore architectures it often has heavy communication and synchronization requirements. For an application that originally is (or can be transformed into) a load-balanced pipeline, a pipelined mapping strategy can offer more a efficient execution for architectures that offer efficient mechanisms for hardware pipelining (e.g., Raw, Tilera, FPGAs, and graphics architectures). Furthermore, although we did not quantify this, data parallelism does not benefit as much as pipelining from the positive caching effects of partitioning the code and data of an application.

6. **It is often the case that CGDTP requires many more I/O ports than hardware pipelining.** In the common case, the CGDTP mapping requires the input to be split to multiple filters that are data-parallelized. For Raw, we utilize all 16 I/O ports (2 are virtualized) on the Raw processor. The input file is automatically partitioned by the compiler via a file reader that feeds the I/O ports. Output is written to the 16 I/O ports, and the cost of composing the multiple output into a single stream is not included. If we include the cost of splitting a single input stream and constructing a single output stream, the mean 16 core speedup for CGDTP falls to 10.3x. Conversely, the hardware pipelining backend does not exploit data parallelism, and thus the input is split to fewer destinations. For our benchmark suite, on average the input is split to 4.3 filters for hardware pipelining, compared to 14.75 for CGDTP. For example, Serpent and TDE require only a single input port and a single output port. The load-balancing flexibility of data parallelism comes at the cost of higher I/O bandwidth (and more ports) required to feed the data-parallel units. Of course, if targeting an architecture with DMA, one may be able to hide the cost of distributing the input and constructing the output.

7. **Overall, CGDTP is effective for mapping the StreamIt Core benchmark suite to multicore architectures.** The geometric mean speedup for CGDTP on our benchmark suite targeting Raw is 12.2x. We feel this represents a positive efficient result given that our source applications are written in a high-level language without regard for the target architecture and without explicit parallelism. The results are not as scalable for the SMP Xeon system and the TILE64, 8.2x for 16 cores and 38x for 64 cores, respectively. These two architectures motivate the need for more efficient fission of peeking filters; the topic of Chapter 8.

6.8 Related Work

In this section we present closely related work that leverages data parallelism to map coarse-grained dataflow graphs to parallel targets. We delay the discussion of other related work until our entire scheduling framework is covered (see Section 7.7).

The Imagine architecture is specifically designed for the data parallelism in the streaming application domain [RDK+98]. It operates on streams by applying a computation kernel to multiple data items off the stream register file. The compute kernels are written in Kernel-C while the applications stitching the kernels are written in Stream-C. Unlike StreamIt, with Imagine the user has to manually extract the computation kernels that fit the machine resources in order to get good steady-state performance for the execution of the kernel [KMD+01]. On the other hand, StreamIt uses fission and fusion transformations to create load-balanced computation units and filters are

replicated to create more data parallelism when needed. Furthermore, the StreamIt compiler is able to use global knowledge of the program for layout and transformations at compile-time while Stream-C interprets each basic block at runtime and performs local optimizations such as stream register allocation in order to map the current set of stream computations onto Imagine.

Liao et al. map the Brook programming language to multicore processors by leveraging the affine partitioning model [LDWL06]. While affine partitioning is a powerful technique for parameterized loop-based programs, in StreamIt we simplify the problem by fully resolving the program structure at compile time. This allows us to schedule a single steady-state using flexible, non-affine techniques (e.g., simulated annealing) and to repeat the found schedule for an indefinite period at runtime. Gummaraju and Rosenblum map stream programs to a general-purpose hyperthreaded processor [GR05]. Such techniques could be integrated with our spatial partitioning to optimize per-core performance. Gu et al. expose data and pipeline parallelism in a Java-like language and use a compiler analysis to efficiently extract coarse-grained filter boundaries [DFA05].

Previous work in scheduling computation graphs to parallel targets has focused on partitioning and scheduling techniques that exploit task and pipeline parallelism [PL95, PBL95, MSK87, KA99b, EM87]. Application of loop-conscious transformations to coarse-grained dataflow graphs has been investigated. Unrolling (or "unfolding" in this domain) is employed for synchronous dataflow (SDF) graphs to reduce the initiation interval but they do not evaluate mappings to actual architectures [CS97, PM91]. Furthermore, these systems do not provide a robust end-to-end path for application parallelization from a high-level, portable programming language.

OpenMP [Cha01] is a popular standard for adding programmer annotations to imperative languages to enable data parallelism and task parallelism. The parallelism exposed by OpenMP is scheduled via a runtime scheduler. Data parallelism is exploited between iterations of DOALL loops. Also, one can define task parallel sections of the graph [NIO+03]. Initial work has been published on automatically adjusting the granularity of data and task parallelism in OpenMP; however it requires programmer annotations and results have not been published [KP08]. We differ in that we support automatic granularity adjustment and have a holistic approach to leveraging task and data parallelism in a program.

6.9 Chapter Summary

In this chapter we developed effective and portable techniques for leveraging data parallelism in stream programs. The techniques are motivated by the large inter-core communication costs of the tradition data parallelism technique, costs that often surpass the parallelization benefits. Our techniques address two problems: (i) the inter-core communication requirement that occurs because of data distribution between producers and consumers and (ii) the duplication requirement engendered by fission of peeking filters. To address (i), we coarsen the stream graph by fusing filters, being careful not to obscure data parallelism. Fusing filters keeps communication between filters within a core, implemented via in-core memory buffers. To address (ii), data parallelism is applied conscious of task parallelism, so that the communication required for data-parallelizing peeking filters is reduced. Our techniques are presented as source-to-source transformations on the StreamIt graph.

Certain properties of the streaming domain enable these transformations. Of particular importance are the following features of the execution model:

1. Exposing producer-consumer relationships between filters. This enables us to efficiently coarsen filters via filter fusion to remove inter-core caused by data distribution.

2. Exposing sliding window computation to the compiler via the peek idiom allows the compiler to data parallel these types of filters.

3. Exposing the outer loop around the entire stream graph. This enables coarse-grained data parallelism, as the products of filter fission may span multiple steady-state iterations of the original stream graph.

A detailed evaluation is presented for three architectures: Raw, TILE64, and an SMP Xeon system. The Raw architecture achieves a 12.2x throughput speedup for 16 cores over single-core performance for the techniques. Tilera achieves a 37.6x 64-core speedup; SMP achieves an 8.2x 16-core speedup. Coarse-grained data parallelism is very effective across architectures for applications that do not include significant amounts of state and do not include peeking filters. For benchmarks that include peeking filters, Raw's network is able to efficiently duplicate the sharing requirement of fission of these filters; however, the TILE64 and Xeon do not scale for these benchmarks. This motivates the techniques covered in Chapter 8. Finally, the techniques covered in this chapter do not explicitly parallelize stateful filters, thus they do not produce scalable mapping for such benchmarks. Chapter 7 covers complementary techniques that parallelize state.

Chapter 7

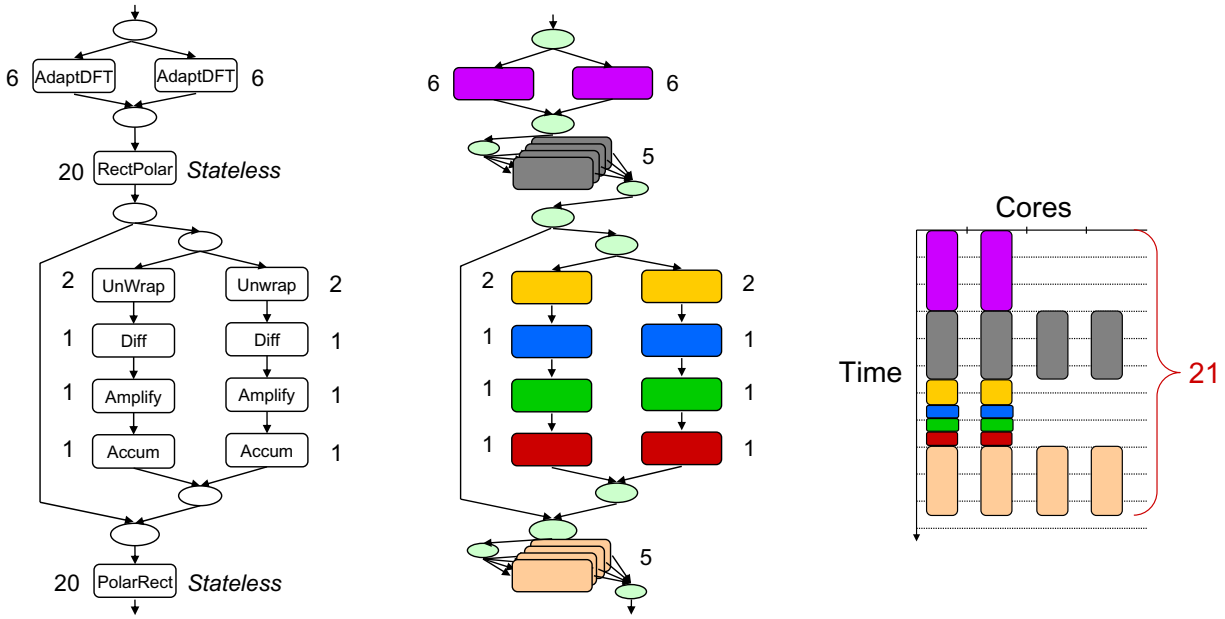
Space-Time Multiplexing: Coarse-Grained Software Pipelining

The data parallelism techniques of Chapter 6 require inherent algorithmic task parallelism to parallelize stateful filters. This chapter details complementary techniques that explicitly parallelize stateful filters. The overall technique presented in this chapter, termed coarse-grained software pipelining (CGSP), is akin to traditional software pipelining of instructions nested in a loop, though here it is applied to coarse-grained filters. CGSP allows a space-multiplexing strategy to schedule the steady-state without regard for the data dependences of the stream graph. For the StreamIt Core benchmark suite, coarse-grained software pipelining combined with coarse grained data parallelism achieves a mean 14.9x 16-core throughput speedup over single core for the Raw microprocessor. The combined techniques offer robust scalable parallelization across all of the benchmarks for the Raw processor. We delay evaluation of CGSP for the TILE64 and Xeon system until Chapter 8 because the techniques presented in the chapter are required for scalability.

7.1 Introduction

Three of the benchmarks in the StreamIt Core benchmark suite (MPEG2Decoder, Vocoder, and Radar) include stateful filters, i.e., filters that maintain state across firings. The techniques covered in Chapter 6, coarse-grained data and task parallelism (CGDTP), do not explicitly attempt to parallelize stateful filters. Considering the CGDTP results for Raw (see Section 6.4.3), the Vocoder benchmark achieves a paltry 2.4x speedup from CGTDP targeting 16 cores. MPEG2Decoder and Radar fare better (13.4x and 9.2x, respectively), however this is by construction of the original application. CGDTP may get lucky if a stateful filter has task parallelism in the coarsened graph, and this parallelism matches the target architecture. This chapter introduces complementary techniques for *explicitly* parallelizing stateful filters of a streaming application.

In this chapter we employ software pipelining techniques to execute filters from different iterations in parallel. While software pipelining is traditionally applied at the instruction level, we leverage powerful properties of the stream programming model to apply the same technique at a coarse level of granularity. This effectively removes all dependences between filters scheduled in a steady-state iteration of the stream graph, greatly increasing the scheduling freedom. Like hardware-based pipelining, software pipelining allows stateful filters to execute in parallel. However unlike hardware pipelining, software pipelining does not require contiguous partitioning. We



(a) (b) (c)

Figure 7-1: (a) Simplified subset of the Vocoder benchmark. Nodes are annotated with the amount of work that they perform per steady-state. (b) Simplified vocoder mapped with data parallelism for 4 cores. In the stream graph, stateless nodes are replicated but stateful nodes remain untouched. (c) An execution trace requires 21 units per steady-state.

call the techniques *coarse-grained software pipelining* (CGSP). CGSP is implemented by adding additional buffering between filters, such that filters from distinct steady-states can be executing in parallel. Typically one steady-state's worth of items is buffered on a channel between a producer and consumer. A *prologue* schedule runs before the steady-state schedule (but after the initialization) to fill buffers.

CGSP schedules the new dependence-free steady-state by mapping the problem to bin-packing. The cost of the schedule is the bin with the maximum workload (the sum of the static work estimations for the filters mapped to it). The bin packing algorithm is wrapped in a pass termed *selective fusion* that is applied to the StreamIt graph. Selective fusion seeks to minimize inter-core communication and synchronization by fusing adjacent filters so long as it does not affect the critical path of the schedule that will be executed. Adjacent filters are fused such that their communication remains within a core, and buffering of items between the fused filters by the prologue schedule is not required.

As mentioned above, the Vocoder example includes a significant number of stateful filters that account for approximately 16% of total load. Figure 7-1(a) gives a simplified version of Vocoder. We can see that there are only 2 stateless filters in the graph. Figure 7-1(b) shows the result of applying CGDTP to the stream graph targeting 4 cores. While two of the filters are data-parallelized, there remains large gaps in the execution schedule (see Figure 7-1(c)). The task parallelism of the original stream graph is not enough to occupy the cores for the levels that contain stateful filters.

As illustrated in Figure 7-2, CGSP involves unrolling the execution of the CGDTP stream graph into two stages. The first stage, executes prologue schedule to establish the appropriate buffering

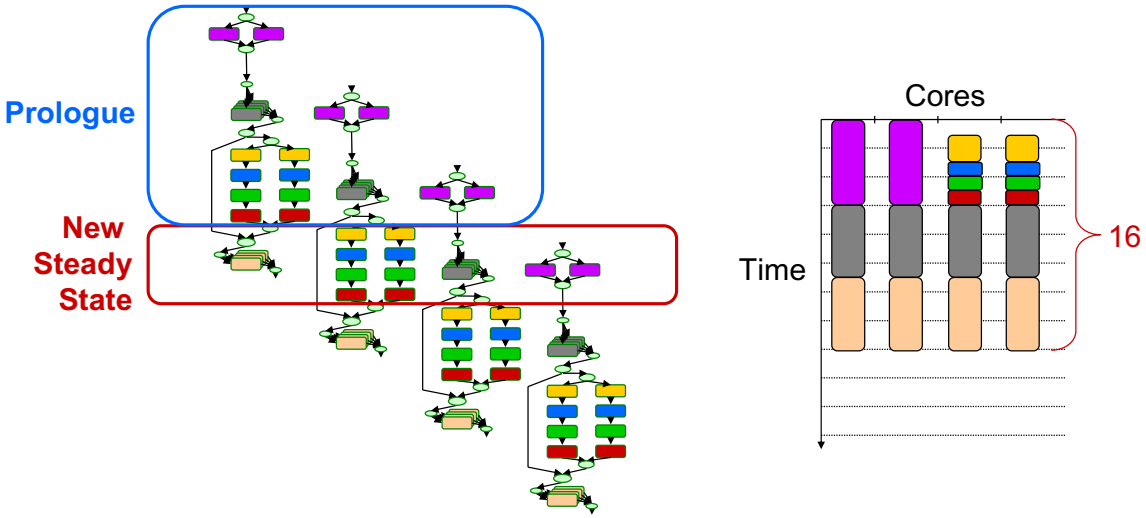


Figure 7-2: Simplified vocoder mapped with coarse-grained software pipelining. By unrolling multiple executions of the stream graph (left), stateful nodes can run in parallel with other nodes during the steady-state. An execution trace (right) requires 16 units per steady-state, an improvement over plain data parallelism.

in data channels. Then in the CGSP steady-state, the filters are decoupled and can execute in any order, writing intermediate results to the buffers. Compared to CGDTP alone, adding CGSP results in 4.2x performance gain for the full Vocoder benchmark for the 16 cores Raw microprocessor.

CGSP a global transformation that is beyond the reach of traditional compilers. Rather than pipelining individual instructions, it pipelines what can be thought of as loop nests, the scheduling loop of each filter. This entails the global reordering of large pieces of the program. The stream programming model makes such transformations feasible because of the regular and exposed flows of data between filters.

CGSP is combined with the CGDTP to achieve a load-balanced mapping that is resilient to the original formulation of the application. Combining the techniques yields the most general results, as data parallelism offers good load balancing for stateless filters while software pipelining enables stateful filters to execute in parallel. Any task parallelism in the application is also naturally utilized, or judiciously collapsed during granularity adjustment. This integrated treatment of coarse-grained parallelism leads to a geometric mean throughput speedup of 14.9x over a single core for the StreamIt Core benchmark suite for the Raw 16-core multicore.[1]

7.1.1 Benefits of Pipeline Parallelism

Pipeline parallelism is an important mechanism for parallelizing stateful filters. Stateful filters are not data parallel and do not benefit directly from the techniques described in Chapter 6. While many streaming applications have abundant data parallelism, even a small number of stateful filters

[1]The techniques in this chapter are not evaluated on the TILE64 nor the Xeon system because these architecture require the techniques covered in Chapter 8 for robust scalability.

can greatly limit the performance of a purely data-parallel approach on a large multicore architecture.

The potential benefits of pipeline parallelism are straightforward to quantify. Consider that the sequential execution of an application requires unit time, and let σ denote the fraction of work (sequential execution time) that is spent within stateful filters Also let μ denote the maximum work performed by any individual stateful filter. See Table 4-17 for statically estimated σ and μ for the StreamIt Core benchmark suite. Given N processing cores, we model the execution time achieved by two scheduling techniques: 1) data parallelism, and 2) data parallelism plus pipeline parallelism. In this exercise, we assume that execution time is purely a function of load balancing; we do not model the costs of communication, synchronization, locality, etc. We also do not model the impact of task parallelism.

1. Using data parallelism, $1 - \sigma$ parts of the work are data-parallel and can be spread across all N cores, yielding a parallel execution time of $(1 - \sigma)/N$. The stateful work must be run as a separate stage on a single core, adding σ units to the overall execution. The total execution time is $\sigma + (1 - \sigma)/N$.

2. Using data and pipeline parallelism, any set of filters can execute in parallel during the steady-state. (That is, each stateful filter can now execute in parallel with others; the stateful filter itself is not parallelized.) The stateful filters can be assigned to the processors, minimizing the maximum amount of work allocated to any processor. Even a greedy assignment (filling up one processor at a time) guarantees that no processor exceeds the lower-bound work balance of σ/N by more than μ, the heaviest stateful filter. Thus, the stateful work can always complete in $\sigma/N + \mu$ time. Stateless work can be spread across all cores, yielding a parallel execution time of $(1 - \sigma)/N$. Adding these two components of the execution yields $1/N + \mu$.

Using these modeled runtimes, Figure 7-3 illustrates the potential speedup of adding pipeline parallelism to a data-parallel execution model for various values of μ/σ on a 16-core architecture. The speedup is calculated as:

$$\textit{PipelineSpeedup} = \frac{\sigma + \frac{(1-\sigma)}{n}}{\frac{1}{N} + \mu} \tag{7.1}$$

In the best case, μ approaches 0 and the speedup is $(\sigma + (1 - \sigma)/N)/(1/n) = 1 + \sigma * (N - 1)$. For example, if there are 16 cores and even as little as $1/15$th of the work is stateful, then pipeline parallelism offers potential gains of 2x. For these parameters, the worst-case gain ($\mu = \sigma$) is 1.8x. The best and worst cases diverge further for larger values of σ. Considering Vocoder and consulting Table 4-17, $\sigma = 16\%$, and $\mu = 0.7\%$. Using the Equation 7.1, we expect a 3.1x speedup for 16 cores (though this assumes no task parallelism exists in Vocoder).

7.1.2 Exploiting Pipeline Parallelism: Hardware or Software

At any given time, pipeline-parallel filters are executing different iterations from the original stream program. However, the distance between active iterations must be bounded, as otherwise the amount of buffering required would grow toward infinity. To leverage pipeline parallelism, one

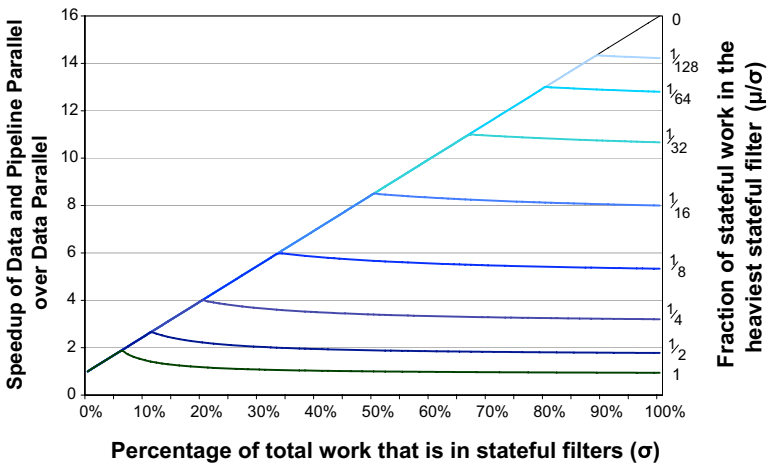


Figure 7-3: Potential speedups of pure pipeline parallelism over pure data parallelism for varying amounts of stateful work in the application. Each line represents a different amount of work in the heaviest stateful filter. The graph assumes 16 cores and does not consider task parallelism or communication costs.

needs to provide mechanisms for both decoupling the schedule of each filter, and for bounding the buffer sizes. This can be done in either hardware or software.

For hardware pipelining, groups of filters are assigned to independent processors that proceed at their own rate (see Figure 7-4(a)). As the processors have decoupled program counters, filters early in the pipeline can advance to a later iteration of the program. Buffer size is limited either by blocking FIFO communication, or by other synchronization primitives (e.g., a shared-memory data structure). However, hardware pipelining entails a performance tradeoff:

- If each processor executes its filters in a single repeating pattern, then it is only beneficial to map a contiguous set of filters to a given processor. Since filters on the processor will always be at the same iteration of the steady-state, any filter missing from the contiguous group and executing at a remote location would only increase the latency of the processor's schedule. The requirement of contiguity can greatly constrain the partitioning options and thereby worsen the load balancing.

- To avoid the constraints of a contiguous mapping, processors could execute filters in a dynamic, data-driven manner. Each processor monitors several filters and fires any who has data available. This allows filters to advance to different iterations of the original stream graph even if they are assigned to the same processing node. However, because filters are executing out-of-order, the communication pattern is no longer static and a more complex flow-control mechanism (e.g., using credits) may be needed. There is also some overhead due to the dynamic dispatching step.

Coarse-grained software pipelining offers an alternative that does not have the drawbacks of either of the above approaches (see Figure 7-4b). Software pipelining provides decoupling by executing two distinct schedules: a loop prologue and a steady-state loop. The prologue serves to advance each filter to a different iteration of the stream graph, even if those filters are mapped

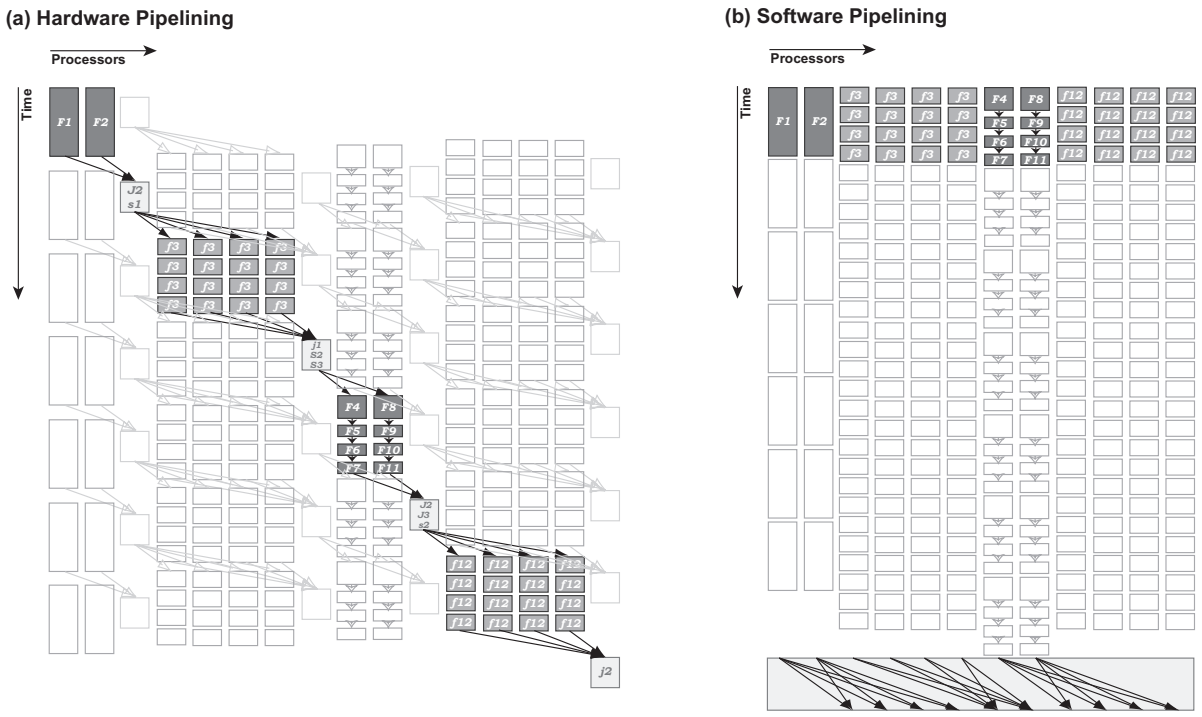


Figure 7-4: Comparison of hardware pipelining and software pipelining for the Vocoder example (see Figure 7-2), though in this case for 16 cores. For clarity, the same assignment of filters to processors is used in both cases, though software pipelining admits a more flexible set of assignments than hardware pipelining. In software pipelining, filters read and write directly into buffers and communication is done at steady-state boundaries. The prologue schedule for software pipelining is not shown.

to the same core. Because there are no dependences between filters within an iteration of the steady-state loop, any set of filters (contiguous or non-contiguous) can be assigned to a core. This offers a new degree of freedom to the partitioner, thereby enhancing the load balancing. Also, software pipelining avoids the overhead of the demand-driven model by executing filters in a fixed and repeatable pattern on each core. Buffering can be bounded by the on-chip communication networks, without needing to resort to software-based flow control.

7.1.3 Flow of the Compiler

The flow of the compiler is given in Table 7-5. This sequence describes the combined techniques of CGTDP and CGSP. Chapter 6 describes the phases through "Judicious Fission". This chapter describes "Selective Fusion" and beyond. For the scheduling techniques of this chapter, filters are considered by the aggregate of their steady-state behavior. If we remember the notation from Section 2.1, we define peek, pop, and push rates of a filter F per firing of the work function as $e(W, F)$, $o(W, F)$, and $u(W, F)$ respectively, where W is the work function of F. For this chapter, we define the following shorthand to denote rates of a filter over the entire steady-state:

- $o(F) \equiv o(W, F) \cdot M(S, F)$

Phase	Function
KOPI Front-end	Parses syntax into a Java-like abstract syntax tree.
SIR Conversion	Converts the AST to the StreamIt IR (SIR).
Graph Expansion	Expands all parameterized structures in the stream graph.
Collapse Data Parallelism	Remove programmer-introduced data parallelism in graph.
Scheduling	Calculates initialization and steady-state execution multiplicities.
Coarsen Granularity	Fuse pipelines of stateless filters to reduce communication.
Judicious Fission	Fiss stateless filters while conscious of task parallelism.
Selective Fusion	Fuse adjacent filters without affecting the critical path to reduce communication.
Bin-packing	Assign filters to cores minimizing critical path workload.
Prologue scheduling	Generate the prologue schedule to fill buffers.
Communication Scheduling	Orchestrates streaming communication between cores and off-chip memory.
Code generation	Generates and optimizes computation and communication code.

Table 7-5: Phases of the StreamIt compiler. The phases beginning with "Selective Fusion" are covered in this chapter.

- $e(F) \equiv (e(W,F) - o(W,F)) + o(F)$
- $u(F) \equiv u(W,F) \cdot M(S,F)$

where $M(S,F)$ defines the multiplicity of F for the steady-state schedule S. We also use the shorthand $s(F)$ to denote the total execution time of all the work function firings of F in the steady-state (i.e., $s(F) \equiv s(W,F) \cdot M(S,F)$, where $s(W,F)$ defines the work estimation of a firing of F's work function W). Selective fusion is applied to the StreamIt graph, after data parallelism has been exposed in the graph. It is the topic of the next section.

7.2 Selective Fusion

The goal of the Selective Fusion pass is to reduce inter-core communication and buffering without negatively impacting the effectiveness of the software-pipelined schedule for the steady-state. Algorithm 3 gives pseudocode for the driver of the Selective Fusion pass. SELECTIVEFUSION iteratively applies fusion to neighboring filters until fusion begins to negatively impact the flexibility of the scheduling heuristic. We employ a greedy bin-packing algorithm as a heuristic to assign filters to cores (space scheduling); it is described in more detail below. The cost of a schedule is the bin (core) with the most work assigned to it. In Algorithm 3 an initial packing is calculated, and the cost of this initial schedule is recorded. Successive applications of fusion are attempted in the loop. For each fusion application, a new schedule is calculated via bin-packing. If this new schedule is worse than the initial schedule by some threshold, the loop is exited and the fusion is undone. The compiler sets the threshold at 10%.

The ADJACENTGREEDYFUSION algorithm is displayed in Algorithm 4. This fusion technique is different than the hierarchical greedy technique discussed in Section 5.3.1 in that it ignores hierarchy. The first phase of the algorithm records the combined computation cost for each pair of filters in the graph. Pairs are considered if they are a producer / consumer pair of a pipeline, or if the two filters are adjacent in a splitjoin, where each parallel stream of the splitjoin is a single filter (this is termed a *simple* splitter in Algorithm 4). The pair with the minimum combined work is then fused.

Algorithm 3 Selective Fusion

SELECTFUSION($G = (V, E), w, P$)

```
1: (A, coreWeight) ← BINPACK(G, w, P)
2: initialCP ← MAXCORE(coreWeight)
3: repeat
4:     G_prev ← G
5:     G ← ADJGREEDYFUSION(G, w)
6:     w ← UPDATEWORKESTIMATES(G)
7:     (A, coreWeight) ← BINPACK(G, w, P)
8:     newMax ← MAXCORE(coreWeight)
9:     change ← initialCP/newMax
10: until change < threshold
11: return G_prev
```

As an example of the effectiveness of the Selective Fusion pass, let's consider the full Vocoder benchmark. Figure 7-6(a) shows the Vocoder graph after granularity adjustment and judicious fission. The graph contains 133 filters. Without the Selective Fusion pass, coarse-grained software pipelining would require at least a steady-state's worth of input items to be buffered on each channel. Furthermore, all of the communication between filters mapped to distinct cores would entail inter-core communication. After Selective Fusion completes, the transformed graph has less than half the number of filters at 64. However, the flexibility of the bin-packing algorithm to find a load-balanced schedule has not been sacrificed by the fusion, as the core assigned the most work has only increased by 8%. The Selective Fusion pass achieves a 2.6x speedup for Vocoder targeting Raw's 16 cores using the full compilation path (CGDTP + CGSP). The Selective Fusion pass improves performance by 2.1x on the Radar benchmark (CGDTP + CGSP).

7.3 Scheduling in Space: Bin Packing

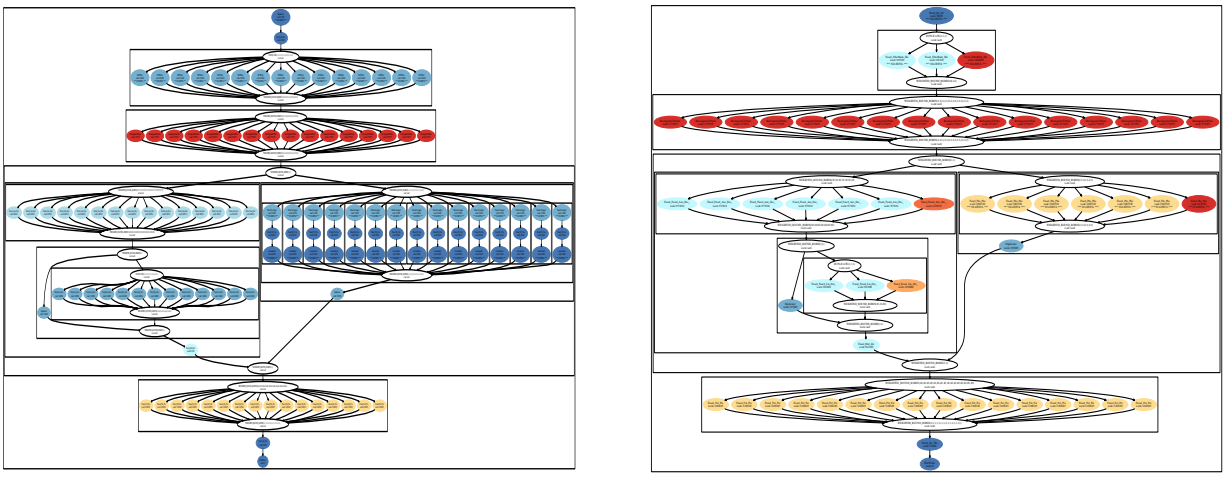
For the bin-packing algorithm, we found that a simple greedy heuristic works fine. As the load-balancing problem is NP-complete (by reduction from SUBSET-SUM [Mic97]), we use a greedy partitioning heuristic that assigns each filter to one of N processors. The algorithm considers filters in order of decreasing work, assigning each one to the processor that has the least amount of work so far. As described in Section 7.1.1, this heuristic ensures that the bottleneck processor does not exceed the optimum by more than the amount of work in the heaviest filter. We also experimented with a simulated annealing packing and an optimal packing, neither bettered the greedy heuristic by more than 11% for the full compilation (CGDTP + CGSP) of Radar and Vocoder, the two stateful benchmarks helped most by the techniques in this chapter.

When Selective Fusion is performed, the final bin packing (of G_{prev} in Algorithm 3) is recorded as the assignment of filters to cores. The core with the maximum workload is considered the critical path, and it's workload defines the inverse of the computational throughput of the graph. This bin packing ignores the cost of communication and synchronization required to communicate item between procedures and consumers on distinct cores, and the cost to split and join data. The bin-packer ignores the data dependences between filters so that it can have complete freedom in

Algorithm 4 Adjacent Greedy Fusion

ADJGREEDYFUSION($G = (V, E), w$)

for all $v \in V$ **do**
  for all $u \in V$ **do**
    pairWeight$[u, v] \leftarrow \infty$
  end for
end for
for all $v \in V$ **do**
  ▷ For a filter v with downstream filter u, record the combined work of $u + v$
  if FILTER(v) $\wedge$ FILTER(OUT(v)[0]) **then**
    $u \leftarrow$ OUT(v)[0]
    pairWeight$[v, u] \leftarrow w(v) + w(u)$
  else if SPLITTER(v) **then**
    ▷ decide if this is a simple splitjoin
    simple $\leftarrow true$
    for all $d \in$OUT(v) **do**
      ▷ check if edge connected to filter that is connected to a joiner (single filter stream)
      if $\neg$FILTER(d) $\vee\neg$JOINER(OUT(d)[0]) **then**
        simple $\leftarrow false$
        break
      end if
    end for
    ▷ record the combined work of each pair neighboring parallel filters of the splitjoin
    if *simple* **then**
      for $i \leftarrow 0, |\text{OUT}(v)| - 1$ **do**
        $x \leftarrow$OUT(v)[i]
        $y \leftarrow$OUT(v)[$i + 1$]
        pairWeight$[x, y] \leftarrow w(x) + w(y)$
      end for
    end if
  end if
end for
▷ Find the min adjacent pair and fuse them
$(p, q) \leftarrow$ MINPAIR(*pairWeight*)
$G \leftarrow$ FUSE(G, p, q)
▷ Delete any splitJoins that contain only a single filter
REMOVEDEADSPLITJOINS(G)
return G



(a) (b)

Figure 7-6: The Vocoder benchmark after coarse-grained data parallelism has been exploited: (a) before Selective Fusion; and (b) after Selective Fusion.

scheduling the filters in space to achieve a load balanced mapping. The bin-packer does not calculate how the filters should be ordered in time. Coarse-grained software pipelining is required to realize the schedule calculated by the bin-packer.

In general, the scheduler should consider all computation nodes of the graph. If splitters and joiners require computation resources, they should be included in the bin packing. For Raw, we do not devote computation resources to splitting and joining, the static network is responsible for data distribution, thus the cost of splitting and joining is not considered by the bin-packing scheduler.

7.4 Coarse-Grained Software Pipelining

The data dependence constraints encoded in the channels of the stream graph must be honored in a valid schedule. In order to realize the throughput of the packed schedule, multiple iterations of the stream graph are active during execution. Across all cores, iterations of a single steady-state execute sequentially, Within a single core, multiple iterations are active concurrently. The new steady-state of the schedule becomes a cross-section of the primed loop, where filters are executing from different iterations of the original stream graph. We call this technique coarse-grained software pipelining as it has parallels to traditional software pipelining of loops of processor instructions [Lam88]. CGSP is enabled by an important property of the StreamIt programming model, namely that the entire stream graph is wrapped with an implicit outer loop. As the granularity of software pipelining increases from instructions to filters, one needs to consider the implications for managing buffers and scheduling communication.

CGSP does not have to account for many of the complexities of traditional instruction level software pipelining. In traditional software pipelining terminology the *initiation interval* (II) defines the number of cycles between initiations of new loop iterations. It can be thought of as inverse throughput; akin to the core with the maximum load as assigned by the bin-packer. The II is limited by machine resources (functional units and registers) and dependence constraints [Lam88].

Algorithm 5 CalcIteration

CALCITERATION($G = (V, E), f$)

```
1: maxIteration ← 0
2: anyRemote ← false
3: splittingProducer ← false
4: ▷ Find max iteration of upstream producers, also remember if any are not mapped same core
5: for all (p, f) ∈ E do
6:     if ITERATION(p) > maxIteration then
7:         maxIteration ←ITERATION(p)
8:     end if
9:     if CORE(p) ≠ CORE(f) then
10:        anyRemote ← true
11:    end if
12:    if |OUT(p)| > 1 then
13:        splittingProducer ← true
14:    end if
15: end for
16: iteration ← maxIteration
17: ▷ If any producer is mapped to different core, or we need to distribute data incoming to f
18: ▷ then create the iteration distance
19: if anyRemote ∨ |IN(f)| > 1 ∨ splittingProducer then
20:    iteration ← iteration + 1
21: end if
22: return iteration
```

For CSGP, functional units are analogous to cores, and the bin-packer handles the mapping of filters to cores. For CGSP, buffering resources are effectively infinite (unlike registers). CGSP can always add buffering to realize the II as calculated by the bin-packer. Furthermore, filters only have true dependences in the stream graph, meaning that as long as there is adequate buffering, there are no inter-iteration dependences, and all filters can execute in parallel in the CGSP schedule.

The technique for calculating the space schedule is currently implemented using greedy bin-packing. The techniques for *realizing* the space schedule are decoupled from the algorithm that calculates it. In this section we describe how we can realize a space schedule calculated via any means.

7.4.1 Calculating Iteration Number

Each filter in the graph[2] is assigned an iteration number relative to the input node(s) of the graph. This corresponds to the iteration of the steady-state the filter is executing in the CGSP steady-state. For each producer / consumer pair, the producer filter can be executing at an iteration ahead of the consumer. Algorithm 5 gives means of calculating the iteration for a node in the general

[2]This graph is the coarsened, judiciously fissed, and selectively fused StreamIt graph converted to the general graph of Section 2.1

stream graph. As an optimization, if f is mapped to the same core as its producer p, and no data reorganization happens between f and p, i.e., f is single input and p is single output, then we do not need to create an iteration distance between f and the producer with the maximum iteration number. Because filters are executed in dataflow order within a core, the producers will all fire before f in a legal ordering. No additional buffering is required between f and its producers.

For architectures that support computation and communication concurrency (e.g., DMA), the memory transfers between filters assigned to distinct cores also need to be assigned iteration numbers. This is because the transfers have to be scheduled concurrently with filter work in order to hide their cost. This is analogous to double-buffering. To account for this, the graph should be modified to include explicit nodes for DMA transfers. The DMA nodes should not be assigned to a core so that line 9 of CALCITERATION evaluates to false, and the iteration distance will be created.

For Raw, the static network is targeted to implement splitters and joiners, so they do not occupy computation resources. For an architecture that must devote computation resources to splitting and joining, the splitting and joining nodes must be included in the calculation of the iteration numbers above if the throughput of the bin-packed schedule is to be realized. This can be accomplished by keeping the splitting and joining nodes and keeping all filter nodes single input / single output in the general graph during conversion from the StreamIt graph (see Section 6.4.2). For architectures that must devote computational resources to data distribution, the clause of line 19 of CALCITERATION might be too restrictive. A positive iteration distance would not be required if splitting and joining is accomplished by compute resources, all filters involved are on the same core, and it was accounted for by the bin-packer when calculating the critical path. Raw requires the iteration distance to implement the data distribution in the network separate from the computation.

7.4.2 Prologue Schedule

As in traditional software pipelining, we construct a loop prologue schedule so as to buffer at least one steady-state of data items between each pair of dependent filters. This allows each filter to execute completely independently during each subsequent iteration of the stream graph, as they are reading and writing to buffers rather than communicating directly. The prologue schedule is executed after the initialization schedule. The prologue schedule is encoded in the compute code generated for each core; it is discussed in more detail below.

7.4.3 Buffering

Buffering between producers and consumers is required to store the items that remain after the prologue schedule. This buffering allows filters from the CGSP steady-state to execute in any order, without respect to the data-dependences of the original steady-state. Each edge in the graph requires additional buffering (over its original steady-state requirement) if the consumer has a greater iteration number than the producer in the CGSP steady-state.

Remember that the CGSP steady-state is concerned with the aggregate steady-state behavior of each filter. First let us define the steady-state buffer requirement for an edge without CGSP (for the original stream graph). The buffering requirement for an edge (u, v) connecting producer u and consumer v equals:

$$\textbf{\textit{BufReq}}(u, v) = u(W_u, U) * M(u, S) * \textbf{\textit{RO}}(u, v, S) = o(W_v, U) * M(v, S) * \textbf{\textit{RI}}(u, v, S)$$

Remember that ***RO***(u, v, S) and ***RI***(u, v, S) defines the ratio of items that flow over edge (u, v) to the total number of items for the producer and consumer in the steady-state (S), respectively (see Section 2.1). For any valid steady-state, $u(W_u, U) * M(u, S) = o(W_v, U) * M(v, S)$, hence the additional equality above. Note that the buffer requirement for the initialization schedule is usually less than the steady-state for an edge. However, this is not a guarantee, and ***BufReq*** should calculate the maximum of the initialization and the steady-state requirement.

For CGSP, we require addition buffering for any producer and consumer pair that are at different iteration numbers in the CGSP steady-state. We use the iteration number of each filter to calculate the buffering requirement for an edge (u, v):

$$\textit{BufReqCGSP}(u, v) = (I_v - I_u + 1) \cdot \textit{BufReq}(u, v)$$

where I_u and I_v are the iteration numbers of u and v, respectively, as calculated by Algorithm 5.

The buffering requirement could be satisfied by implementing a circular structure with multiple buffers ($I_v - I_u + 1$) of size ***BufReq***. Or the buffering requirement could be satisfied by distinct buffers located at the producer's local memory and consumer's local memory. For distributed memory machines, the implementation has to decide on the memory bank of each buffer, i.e., should the buffer component be co-located with the producer or consumer. See [KM08] for an implementation of circular buffers for a DMA architecture.

7.4.4 Synchronization

The final requirement for the CGSP state-state is a means of synchronization. The synchronization implementation must synchronize producers and consumers and not allow one to "get ahead" of the other. Without synchronization, for example, since producers and consumers are decoupled, a producer could overwrite items yet to be read by the consumer. The synchronization could be achieved via a global barrier between steady-states to assure all consumers have consumed their data before beginning the next CGSP steady-state. For Raw, we take a different approach that is quite specific to the Raw architecture; it is discussed below.

The remainder of the compiler flow for CGSP is architecture specific. We will discuss the remaining concerns in the context of the Raw microprocessor.

7.5 Code Generation for Raw

This section covers the concrete implementation details of CGSP for the Raw microprocessor. We will describe the buffering and data distribution scheme with enough detail to allow the reader to grasp the concepts, for more information consult the documentation of the StreamIt compiler. The configuration used for this evaluation is the same as in Section 6.4. We simulate a memory module attached to each I/O port of the Raw microprocessor. We employ a simulation of a CL2 PC 3500 DDR DRAM, which provides enough bandwidth to saturate both directions of a Raw port [TLM+04]. Additionally, each chipset contains a streaming memory controller that supports a number of simple streaming memory requests. In our configuration, 16 such DRAMs are attached to the 16 logical ports of the chip. The chipset receives request messages over the dynamic network for bulk transfers to and from the DRAMs. The transfers themselves use the static network.

The Raw microprocessor does not support computation and communication concurrency. A compute processor, once it issues a transfer command for the streaming DRAM must actively grab items from or inject item to the network. Because of the lack of computation and communication concurrency we cannot hide the cost of the communication, we can only seek to make it efficient. With this particular Raw configuration, it is more efficient for filters to read their input from and write their output to off-chip memory. Even a producer and consumer pair mapped to the same core will communicate via off-chip streaming memory. Every core has a bank of memory. During execution, a filter will read input and write output only to the bank that is assigned to the core. Separate transfers are required to move data between banks for producers and consumers on different cores.

The implementation details are complicated by the fact that we use the network for splitting and joining. Data distribution is conflated with buffering for the Raw implementation. As we will see, additional buffers may be needed to implement the data distribution of a node's multiple outputs or multiple inputs. The Raw implementation passes operate on the general graph of Section 2.1 where nodes with multiple inputs and multiple outputs are supported. In fact, the distribution mechanism of the execution model was designed to match the distribution that can be achieved via the static network, without compute processor involvement. Notice that output splitting in the general graph supports both duplication and round-robin distribution.

To aid in the presentation of the implementation we present Figure 7-7, an example CGSP mapping for Raw of a simple stream graph given in Figure 7-7(a). The bin-packed schedule is given in Figure 7-7(b); the iteration number assignment is given in Figure 7-7(c). Notice that since the producer C and consumer E are mapped to the same core, and because C is the only producer for E, they can be assigned the same iteration number (see Algorithm 5) in order to reduce the buffering requirement.

The implementation first must calculate required buffering. All buffers are allocated to off-chip memory. Between any producer and consumer pair, there may exist up to 3 buffers. One represents the output of the producer, one the input of the consumer, and one the edge between two a producer and consumer. The output buffer stores the output produced by a filter, in the order it was produced. The size is equal to the number of items the filter produces in the steady-state ($u(W_f, f) * M(f, S)$ for filter f). Every filter has an output buffer.

A filter may also require an input buffer if it has multiple inputs. This buffer allows the multiple inputs to be arranged in the order stipulated by the input distribution pattern, so that the input can be streamed to the filter in the correct order and pattern. A filter with a single input does not require an input buffer. If a filter requires an input buffer, the size of the buffer is equal to the number of items consumer in the steady-state ($o(W_f, f) * M(f, S)$ for filter f)

Edges in the graph may also need to be represented by a buffer in the implementation, and furthermore, an edge buffer may include multiple constituent buffers to realize the prologue schedule. The size of each constituent buffer for edge (u, v) is given by ***BufReqCGSP***(u, v) above. The constituents are organized in a circular linked list, with producers and consumers having separate pointers to the constituents. This structure is similar to a rotating register file. The number of constituent buffers required is equal to the difference of the iteration numbers between the producer and consumer, i.e., for edge (u, v) the required number of constituent buffer is $I_v - I_u$.

For examples of these buffering calculations consult Figure 7-7(d). In the example, C and E are mapped to the same core, have equal iteration number, and there is no distribution between them, thus only a single buffer is required, E's output buffer. Between A and B, we require 2

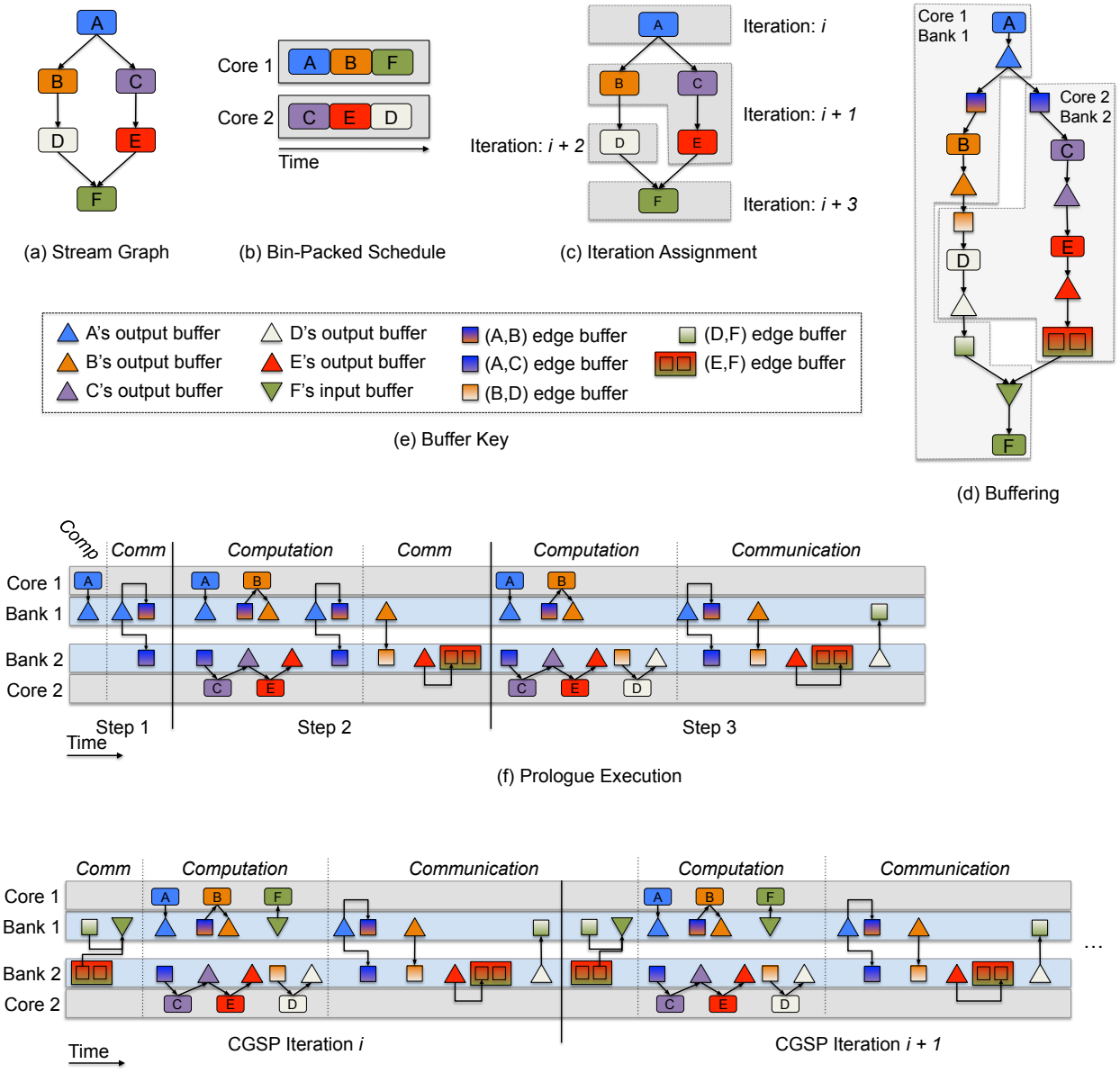


Figure 7-7: An extended example of the scheduling, mapping, buffering and execution of a simple stream graph for the CGSP techniques. The work of the filters has not been represented to scale, and the work of the Prologue and CGSP steady-state has not been represented to scale.

buffers, A's output buffer, and the edge buffer of (A, B) because of the iteration distance. The (A, B) edge buffer has only one constituent. Finally, we require 3 buffers between E and F: E's output buffer, the edge buffer for (E, F) with two constituent buffers, and since F has multiple inputs, F's input buffer.

Notice that we attempt to fold some of the buffering requirement for the iteration distance into the buffers required for data distribution. We know that every filter has an output buffer and its presence is accounted for in the calculation of the number of constituent buffers for an edge. Figure 7-7(d) shows that between B and D we only require a single constituent from the edge buffer because B writes into its output buffer. The edge buffer makes up for the requirement of the iteration difference between B and D, and it is required because B and D are mapped to different cores. Without explicit output buffering, the required constituent buffers for an edge buffer would be $I_v - I_u + 1$.

The static network performs all necessary data distribution. Items are streamed from off-chip streaming memory onto the static, on-chip network. The network executes the compiler generated sequence of routing instructions to realize the data distribution. Data distribution is required whenever a producer and consumer are mapped to distinct cores, or whenever we have a multiple input or multiple output node in the graph. In order to perform a join, the buffers representing the inputs to the joining operation must be mapped to distinct banks by the compiler. The same is true for splitting, the buffers to split into need to be mapped to distinct banks. The streaming DRAM controllers are limited in the type of reorganization they can accomplish (they are limited to simple strided loads and stores). If two buffers required for data distribution were mapped to the same banks, the load or store distribution may be too complex for the streaming DRAM controller. So we do not rely on the controller to perform reorganization, it only reads and writes at unit stride. As mentioned in Section 6.4.2, the compiler includes a stage that maps buffers to banks, making sure all buffers for each distribution are mapped to distinct banks. If a split or join is wider than the number of banks, a pass breaks it down into multiple levels that can be supported.

For Raw, producers and consumers synchronize via the static network and the single owner of each memory bank. No barrier is needed between steady-states of the CGSP steady-state. Each core is assigned ownership of a memory bank. All transfers to/from this bank are initiated on this core. The program executed by the static network synchronizes each producer and consumer by the sequence of routing instructions. One cannot get ahead of the other because the static network will not let it happen. One issue that can happen unless we are careful is if a compute processor that owns a DRAM is not synchronized via the graph to any of the producers and/or consumers of any buffers of the DRAM, the owner core may get behind or ahead (and overwhelm the transfer command buffer) of the producers and/or consumers of the DRAM. In this rare case, we add token synchronization between the cores at the boundary of a CGSP steady-state.

Execution for Raw begins with the initialization stage. The initialization stage is required to fill buffers with at least $e(W_F, f) - o(W_f, F)$ items (see Section 2.1). The initialization schedule is executed in data flow order, and the rotating buffers are not rotated. The items remaining on each buffer is transferred into the filter state for storage. The un-popped items from the previous execution are shifted onto the beginning of a filter's in-core input buffer before the filter receives new items in the prologue and steady-state.

After the initialization schedule, the prologue schedule is executed. This schedule is encoded in the execution code for each core; its implementation is shown in Figure 7-8(a). This code has the result of executing the graph in a staged fashion, leaving enough buffering between producer u and

```
void CGSPPrologue() {
  char iteration[N] = {0};
  iteration[0] = 1;

  repeat {
    if (iteration[0]) {
      // work functions
      // splitting transfer commands
    }
    if (iteration[1]) {
      // joining transfer commands
      // work functions
      // splitting transfer commands
    }
    ...
    if (iteration[N-2]) {
      // joining transfer commands
      // work functions
      // splitting transfer commands
    }

   // Shift-left iteration predicate
   for(j=N-1; j>=1; j--)
      iteration[j] = iteration[j-1];

  } until (iteration[N-1]);
}
```

(a) CGSP Prologue code.

```
void CGSPSteadyState() {
  while (true) {
    // joining commands for all bank's buffers

    //call the work functions for each iteration
    workIteration1();
    workIteration2();
    ...
    workIterationN();

    // splitting commands for all bank's buffers
  }
}
```

(b) CGSP Steady-state code.

Figure 7-8: Generated execution code for CGSP. (a) lists the code to implement the prologue schedule to enable the software pipeline. (b) lists the code for the software-pipelined steady-state.

consumer v for $I_v - I_u$ executions of v to occur before u is required to execute. The CGSP steady-state is ready to execute after the prologue is complete, see Figure 7-8(b) for its implementation.

For the prologue and CGSP steady-state, data reorganization and bank transfers are not interleaved with computation. We separate their execution so that all bank transfers and switch reordering occurs in a separate *communication phase*. This allows the entire chip to be used for data reorganization, and does not require synchronization between computation and communication. All computation occurs without inter-core synchronization during the *computation phase*. During the computation phase, each core reads/writes only to its bank through a direct pipe via the static or dynamic network. During both the computation and communication phase, after a buffer or filter writes to or read from an edge buffer, its pointer to the constituents of the buffer is advanced.

To achieve the separate computation and communication phases, for each prologue stage or for each execution of the entire CGSP steady-state, first all joining is scheduled, followed by the computation stage of schedule filters, finally all splitting is scheduled. *Joining* denotes that for all input buffers, the reorganization described by all the filters' input distributions are executed. *Splitting* denotes that all output buffers perform the reorganization described by the output dis-

tribution patterns of the filters of the graph.[3] As successive executions of prologue stages or the CGSP steady-state are executed, splitting and joining are executed in sequence. This combination is the communication stage: spitting of iteration i and joining for iteration $i + 1$. The joining and splitting scheduling is evident in the code in Figure 7-8.

Figure 7-7(f) demonstrates the prologue schedule execution for the stream graph of the example for 2 cores. The prologue is composed of 3 stages, one less than the maximum iteration number of the graph. For each stage, communication is scheduled only for filters that will execute (for joining) or have executed (for splitting) in the stage. The edge buffer between (E, F) is rotated twice by the output, but zero times by the input after the prologue is complete. Each filter has items buffered so that it is ready to fire (a join is required first before F can fire, this is scheduled first in the CGSP steady-state).

Figure 7-7(f) demonstrates two consecutive iterations of the CGSP steady-state. The joining for F is scheduled first from the edge buffer (D, E) and F's input buffer's pointer into the rotated edge buffer (E, F). The computation stage executes all filters; filters within a core execute in dataflow order. C and E, have equal iteration number, and communicate directly during the computation stage. Splitting occurs last during each CGSP iteration; output buffers for A, B, D, and E are split. The splits for B's and D's output implement a direct transfer between banks because the consumer filter for each is mapped to a different core.

7.6 Evaluation: The Raw Microprocessor

In this section we evaluate the scheduling and implementation techniques for coarse-grained software pipelining in the context of the 16 core Raw architecture. As we will demonstrate, our greedy technique for scheduling filters in space, coupled with Selective Fusion for communication and synchronization reduction, is effective for exploiting coarse-grained pipelined parallelism. We refer to the combined techniques as "CGSP." More importantly, leveraging data parallelism by coarsening and judiciously fissing before applying CGSP provides a cumulative performance gain, especially for applications with stateful computation. We refer to this sequence as "CGDTP + CGSP". We will first examine the two benchmarks in the StreamIt Core benchmark suite that include significant amounts of stateful computation: Radar and Vocoder.

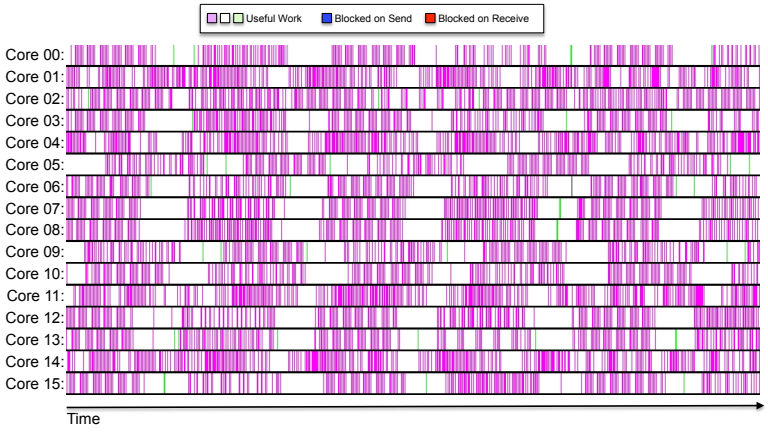
7.6.1 Radar

Stateful computation accounts for nearly 98% of the statically estimated workload for the Radar benchmark; however, a maximum of only 4% of this load is contained in any one filter (see Table 4-17). Due to this evenly distributed load, we expect coarse-grained software pipelining to perform well for Radar. Figure 7-9(a) gives the selectively fused StreamIt graph for Radar. The graph is composed of 16 filters, with all filters performing about the same amount of work; each filter is assigned to its own core. This partitioning is excellent, and leads to the near 100% utilization that is demonstrated in Radar's 16 core Raw execution visualization in Figure 7-9(b).

CGSP on Radar achieves a 19.6x throughput speedup over single core performance. The super-linear speedup over single core is explained by improved caching behavior, and more effective

[3] An output buffer that is directly connected to a consumer is not scheduled during for splitting, e.g., the buffer between C and E in Figure 7-7(d).

(a)



(b)

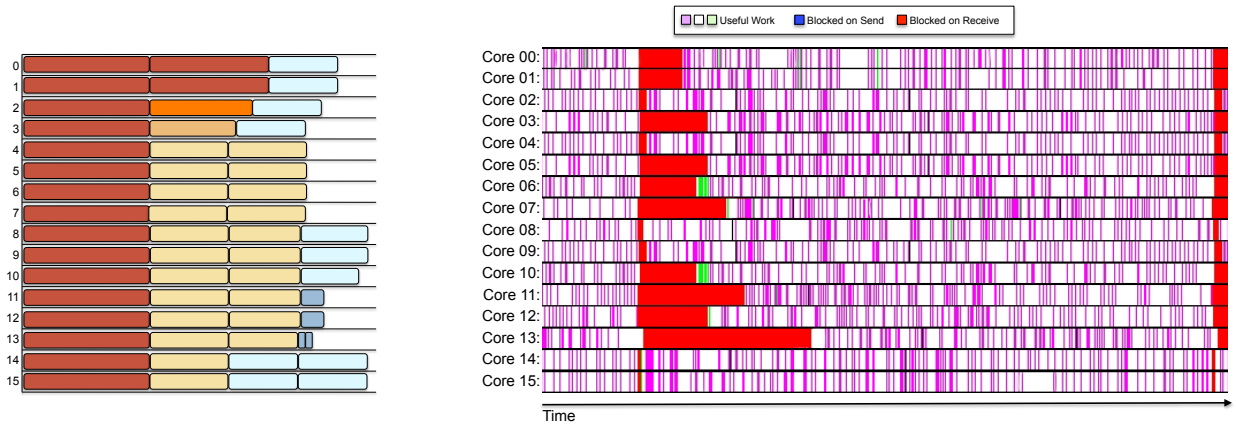
Figure 7-9: The Radar benchmark: (a) The results of selective fusion on the CGDTP graph, and (b) the 16 core Raw execution visualization.

unrolling and array scalarization. The CGSP mapping achieves a 2.1x speedup over CGDTP. As mentioned in Section 6.4, CGDTP extracts no additional parallelism over a task parallel mapping because of the presence of so much stateful computation. For CGDTP Radar's graph is essentially unaltered with periods of idleness while the task parallelism does not adequately span all cores. Converse, CGSP is able to pack a balanced CGSP steady-state for Radar.

CGSP achieves a 2.5x speedup over hardware pipelining and task parallelism (HPTP). This mainly due to the increased scheduling flexibility and the more efficient data distribution afforded by CGSP. The hardware pipelining partitioner is forced to devote cores to joiners, so it cannot settle on the same load-balanced graph as CGSP. Furthermore, the splitjoins of the hardware pipelined graph (Figure 5-4) require inter-core synchronization in the steady-state. The CGSP mapping has no synchronization during the computation phase, and because of Radar's large computation to communication ratio, the communication phase is very fast (almost unseen in Figure 7-9(b)).

7.6.2 Vocoder

Stateful computation accounts for 16% of the statically estimated workload for Vocoder, however at most only 0.7% of the total load is concentrated in a single filter (see Table 4-17). Due to this



(a) (b)

Figure 7-10: The Vocoder benchmark: (a) bin-packed CGSP steady-state, and (b) 16 core Raw execution visualization for CGDTP + CGSP.

evenly distributed load, we expect a hefty speedup from CGSP coupled with CGDTP. Figure 7-6(b) gives the selective fused StreamIt graph. In Figure 7-10(a) we present the bin-packed 16 core space schedule based on static work estimation. We can see that the schedule is quite well-packed, with little waste. Figure 7-10(b) provides the 16 core Raw execution visualization. From the figure, we can see the waste in the schedule as cores waiting to receive data at a given time as represented by red. The amount of blocking is small, and Vocoder achieves a 12.6x speedup over single core throughput.

CGDTP + CGSP achieves a 4.2x speedup over CGDTP for Vocoder. This is because for CGDTP, the state quickly becomes a bottleneck for the mapping. Even though there is some task parallelism in Vocoder, the task parallelism is not at the appropriate symmetries nor granularity for 16 cores, so CGTDP loses out. CGDTP + CGSP achieves a 2.1x speedup over HPTP for Vocoder because the hardware pipelined partitioning is not well-load balanced, and there is much synchronization during the steady-state. The CGSP + CGDTP mapping is able to effectively parallelize and load balance both stateful and stateless computation, while implementing the data-distribution efficiently.

7.6.3 The Complete Evaluation

Figure 7-11 presents the results of (i) HPTP, (ii) CGDTP, (iii) CGSP, and (iv) CGDTP + CGSP on the entire StreamIt Core benchmark suite. The chart presents the speedup as 16-core throughput over single core throughput for each technique. CGSP coupled with CGDPT achieves a geometric mean speedup of 14.9x across the benchmark suite. This is quite remarkable considering that the Raw architecture does not include computation and communication concurrency. CGSP contributes to a 1.2x speedup over CGDTP alone. For some stateless benchmarks such as BitonicSort, ChannelVocoder, and FMRadio we see a benefit from CGSP added to CGDTP because of selective fusion and the efficient data distribution. The Selective Fusion pass increases performance by a geometric mean of 16% across all benchmarks for CGDTP + CGSP.

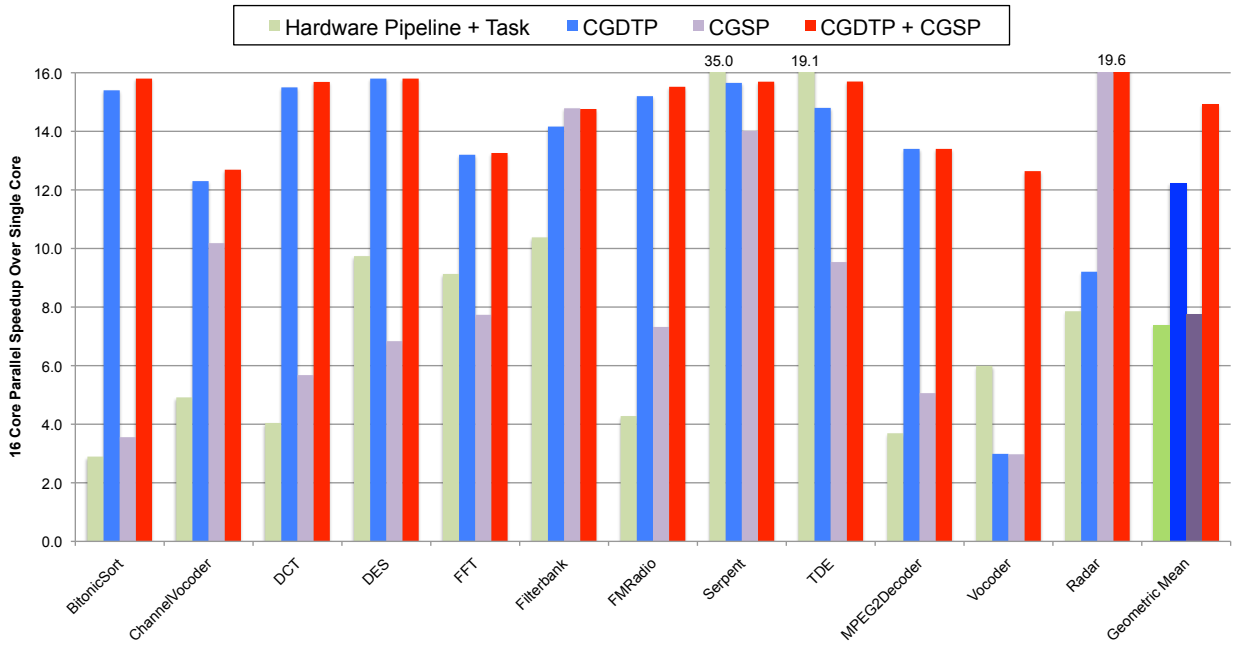


Figure 7-11: Raw 16 core speedup over single core throughput for (i) Hardware Pipelining + Task, (ii) CGDTP, (iii) CGSP, and (iv) CGDTP + CGSP.

The third bar of Figure 7-11 gives the speedup for CGSP alone. In this experiment we do not exploit data parallelism. On average, software pipelining has a speedup of 7.7x over single core (compare to 12.2x for CGDTP). Alone, CGSP performs well when it can effectively load-balance the packing of the dependence-free steady-state. In the case of Radar and Filterbank, software pipelining achieves comparable or better performance compared to data parallelism. For these applications, the workload is not dominated by a single filter and the resultant schedules are statically load-balanced across cores. However, when compared to data parallelism, CGSP alone is hampered by its inability to reduce the bottleneck filter when the bottleneck filter contains stateless work (e.g., DCT, MPEGDecoder). Also, our data parallelism techniques tend to coarsen the stream graph more than the selective fusion stage of software pipelining, removing more synchronization. For example, in DES, selective fusion makes a greedy decision that it cannot remove communication without affecting the critical path workload. CGSP performs poorly for this application when compared to CGDTP, 6.9x versus 15.8x over single core, although CGSP calculates a load-balanced mapping. Another consideration when comparing CGSP alone to CGDTP is that the software pipelining techniques rely more heavily on the accuracy of the static work estimation strategy, although it is difficult to quantify this effect.

Finally, CGSP alone and HPTP achieve a very similar speedup, 7.7x and 7.4x respectively, but for different reasons. CGSP bests HPTP for benchmarks that are not partitioned into load-balanced pipelines (BitonicSort, ChannelVocoder, DCT, FilterBank, FMRadio, MPEG2Decoder, and Radar). This is because CGSP has more flexibility when space scheduling and can more efficiently implement large or numerous splitters and joiners. CGSP cannot best HPTP for the benchmarks partitioned to a load-balanced pipeline because the pipelined communication on Raw is so efficient.

7.7 Related Work

Printz also compiles synchronous dataflow graphs (including support for sliding window operations and stateful filters) to the distributed-memory Warp machine [Pri91]. Printz's solution includes some of the same basic principles as ours, namely, to favor full data parallelization of filters, and to use pipeline and task parallelism to parallelize stateful computation. However, Printz does not attempt to minimize communication when assigning tasks to processors; data-parallel actors are assigned to processors in any topological order and are always spread across as many processors as possible. We instead fuse the stream graph first, and then incorporate task parallelism to reduce the communication requirements for data parallelization. In order to support applications with hard real-time constraints, he minimizes the latency of a single steady state rather than maximizing the overall throughput. Both coarse-grained data parallelism and coarse-grained software pipelining have the potential to increase the latency of the program, and thus fall outside the scope of his model. Still, it is difficult to compare our techniques directly because they focus on different things. Our focus is on stateless actors with limitless data parallelism, while Printz focuses on actors with a fixed amount of internal data parallelism. He also has built-in support for systolic computations (pipelines of balanced filters) which we do not model explicitly.

Previous work on software pipelining has focused on scheduling machine instructions in a loop via modulo scheduling [RG81, Lam88]. The algorithms devised must account for tight resource constraints and complex instruction dependencies. Our software-pipelining problem is much less constrained, enabling us to employ a simple greedy heuristic. Furthermore, a traditional modulo scheduling algorithm is not needed because we have an implicit loop barrier at the end of each steady-state. ILP compilers for clustered VLIW architectures [Ell85, LFK+93, LBF+98, QCS02] must partition instructions and assign them to clusters as part of the instruction scheduling. Clustering is analogous to our application of filter fusion in our software pipelining algorithm.

Before our publication [GTA06], software pipelining techniques had been applied to SDF graphs onto various embedded and DSP targets [BG99, CV02], but had required much programmer intervention and programmer knowledge of both the application and the architecture. The programmer was required to manually assign filters to processors, and manually schedule the software pipelined loop. Furthermore, the solutions targeted heterogeneous parallel processors with few processing elements. To the best of our knowledge, we were the first to automatically leverage coarse-grained software pipelining in the SDF domain.

Das et al. employed software pipelining techniques to parallelize computation with communication when compiling the StreamC/KernelC language to the Imagine architecture [DDM06]. We differ in that we use software pipelining techniques to parallelize stateful computation between different cores.

Since [GTA06] others have tried various solutions to the problem of mapping filters to cores exploiting data, task, and software pipeline parallelism. Kudlur and Mahlke solved the scheduling problem by mapping it to an integer linear programming formulation [KM08]. Whereas our methods rely on heuristics, their method calculates an optimal partitioning that utilizes data, task, and pipeline parallelism via a translation to an integer linear programming formulation. We have tried a similar approach in the past, and the problem with this method is that it relies heavily on accurate work estimation. Furthermore, their formulation does not attempt to minimize inter-core communication. The later is fine because their target architecture (Cell [Hof05]) includes computation/communication concurrency. On the other hand, our heuristic approach only relies on work

estimation for parallelizing stateful computation and for reducing the communication requirement of the fission of peeking filters. Our heuristics specifically attempt to minimize communication via fusion of contiguous sections of the graph and via incorporating task parallelism to reduce the width of data parallelism. Kudlur and Mahlke use the StreamIt Core benchmark suite for evaluation and do try to compare to our work, however, they did not replicate our methods across all benchmarks. For the benchmarks that do replicate our methods (6), our techniques achieve higher throughput.

Hormati et al. describe a framework solving our scheduling and mapping problem in [HCK+09]. In this case there is a prepass data-parallelization stage that data parallelizes filters with heavy workload. Next an ILP formulation assigns filters in the transformed graph to cores. Software pipelining is utilized to enable pipeline parallelism between non-contiguous sections of the graph. Also, their partitioning algorithm supports heterogeneous architectures. Finally, the system also incorporates online adaptation and migration of filters if, for instance, the number of cores assigned to the application changes. They do not directly compare to our techniques, but they do include the same baseline as [KM08], which they fail to match in terms of throughput; as we mentioned above, we beat this baseline. The techniques in the research are formulated to enable online adaptation. However, our techniques have been extended to support online adaption in [Tan09].

Udupa et al. describe an ILP formulation similar to [KM08], though they include additional steps to choose the best static assignment of resources for a partition to promote simultaneous multi-threading on a GPU [UGT09a]. They extend their work in [UGT09b] to partition a stream program between a multicore CPU and a GPU.

To summarize the work that has come after [GTA06], the initial publication of the work in this chapter, more sophisticated techniques have been attempted and more complex architecture targets considered. Regarding the former, we still believe that our techniques are superior because they do not rely as heavily on work estimation as subsequent techniques. Finally, we were the first to "make the leap" and automatically parallelize filter iterations from multiple steady-states via coarse-grained software pipelining.

7.8 Chapter Summary

For many applications that include stateful filters, data parallel techniques are not adequate to achieve an effective parallel mapping. Pipeline parallelism must be leveraged to parallelize the stateful filters of an application. The overall technique presented in this chapter, termed coarse-grained software pipelining (CGSP), is akin to tradition software pipelining of instructions nested in a loop. Conceptually, the outer scheduling loop that surrounds a steam graph is recognized, and leveraged to "unroll" the stream graph. CGSP allows a space scheduling technique to schedule the steady-state without regard for the data dependences of the stream graph. This flexibility is achieved with a combination of buffering and a prologue schedule. This implementation enables us to employ a highly-effective greedy bin-packing space scheduler for the steady-state. Furthermore, we presented a technique for reducing inter-core communication without significantly affecting the flexibility of the bin packing scheduler. CGSP combined with data parallelism (CGDTP) achieves a geometric mean throughput speedup of 14.9x over single core throughput for the StreamIt Core benchmark suite targeting the 16 core Raw architecture.

Many of the same properties of stream programs enable CGSP as CGDTP (Chapter 6). Of particular importance are the following features of the execution model:

1. Exposing producer-consumer relationships between filters. This enables us discover pipeline parallelism.

2. Exposing the outer loop around the entire stream graph. This enables coarse-grained software pipelining, filters in the software-pipelined steady-state may span multiple steady-state iterations of the original stream graph.

3. Static rates of filters enables the compiler to schedule the steady-state of the application, calculate work estimations, and calculate buffer requirements for CGSP.

Chapter 8

Optimizations for Data-Parallelizing Sliding Windows

In this chapter we introduce methods that further reduce the inter-core communication requirement of the fission of peeking filters. These techniques do not alter the stream graph, but instead route shared data more precisely and alter the steady-state to reduce sharing requirements between fission products. Across the StreamIt Core benchmark suite, CGDTP employing these new techniques combined with CGSP attain a 14.0x mean parallelization speedup for the 16 core SMP and a 51.7x mean parallelization speedup for the 64 core TILE64.

8.1 Introduction

Chapter 6 covers effective techniques for exploiting coarse-grained data parallelism in stream graphs. COARSENGRANULARITY allows the compiler to explicitly alter the granularity of the programmer-conceived graph to reduce communication and synchronization incurred by data distribution between producers and consumers. Furthermore, JUDICIOUSFISSION leverages data parallelism conscious of task parallelism such that fission of peeking filters requires less communication and synchronization to span the cores of target. Coarsening the granularity and judicious fission are adaptable with minor changes across multicore architectures. The techniques of Chapter 6 achieved a 12.2x 16-core parallelization speedup over single core for the Raw microprocessor. However, the speedup was not as impressive for TILE64 and the Xeon system, 37.2x to 64 cores and 8.2x to 16 cores, respectively. As highlighted in Section 6.6 and Section 6.5, the reason for the diminished performance of SMP and Tilera, respectively, as compared to Raw, is that these architectures do not support efficient duplication of data items across cores in their network. Fission of a peeking filters requires sharing, and thus duplication, of items between fission products of the filter. Interestingly, the performance model of Chapter 3 predicts that the efficiency of communication for data-parallelization of peeking filters is important for realizing effective mappings

A peeking filter is defined as a filter f with $e(S, f) > o(S, f)$, that is, the filter inspects items on its input buffer beyond what it dequeues. When a peeking filter is fissed, input items must be shared across the fission products. One product filter will read without dequeuing an item, but a product filter that executes a later iteration will read and dequeue it from the input. If these two filters are mapped to separate cores, this item must be replicated on both cores. Section 6.3.3 introduces the fission transformation for peeking filters as it applies to the StreamIt graph. The effect of this

transformation is that each fission product receives *every* input data item. For each filter, items that are not read are dequeued from the input queue. If each product is mapped to a distinct core, the replication of items is achieved via the communication network. Items are duplicated to each core. We call this technique *duplication and decimation* or *dupdec*.

The efficiency of this technique depends on the application being mapped and the communication mechanism of the target. In our application suite, ChannelVocoder, FilterBank, and FMRadio include stateless peeking filters. Examining each benchmark, we can determine what percentage of total steady-state communication is unnecessarily duplicated by a coarsened, judiciously fissed (CGDTP) graph where fission is implemented by dupdec:

- ChannelVocoder: 38% unnecessarily duplicated (5,200 of 13,336 total items)
- FMRadio: 61% unnecessarily duplicated (1,108 of 1,808 total items)
- FilterBank: 0% duplicated (912 total items)

It is clear that the number of items unnecessarily duplicated and communicated by dupdec can vary widely by application. Given that we are targeting the Raw microprocessor in Chapter 6, 61% and 38% of unnecessary duplicated communication for ChannelVocoder and FMRadio, respectively, does not greatly negatively affect performance. This is because, as Section 6.4.2 covers, duplication is essentially free on Raw's static network. However, if we are targeting an architecture where duplication requires occupancy of compute and/or communication resources, dupdec may be ineffective.

In this chapter, we present techniques that seek to reduce the communication and synchronization overhead for exploiting data parallelism via fission. Figure 8-1 gives examples of the techniques. The techniques are motivated by the requirements of scalable mappings for architecture that do not include mechanisms for efficient duplication of items in their network. The Tilera TILE64 processor represents a distributed memory machine with more traditional communication mechanisms (DMA and distributed shared memory) versus Raw, while the Xeon system represents a commodity shared-memory architecture. For these architectures, duplication is not free in the network and more efficient techniques are required for fission of peeking filters.

The techniques covered in this chapter are enabled by the more expressive data distribution of the general stream graph of Section 2.1. We first cover this representation in more detail, including the translation from StreamIt graph to this representation, including a technique for reducing the synchronization requirement of the input StreamIt graph. Next, fission of peeking nodes in the general stream graph is presented (see Figure 8-1(b)). This type of fission does not unnecessarily duplicate items, and thus leads to savings for ChannelVocoder and FMRadio.

Even without unnecessarily duplication of items, inter-core communication as a result of fission accounts for a large percentage of the total items communicated between filters of an application. ChannelVocoder has 48% of total communication as inter-core communication caused by fission, Filterbank 50%, and FMRadio 93%.[1] Recognizing that the compiler has control over the multiplicities of nodes in the steady-state enables the final technique, *sharing reduction optimization* (see Figure 8-1(c)). The sharing reduction optimization seeks to reduce the amount of data shared

[1]Total communication includes items that are "communicated" between filters mapped to the same core through core memory. The percentages are based on the dupdec steady-state multiplicities.

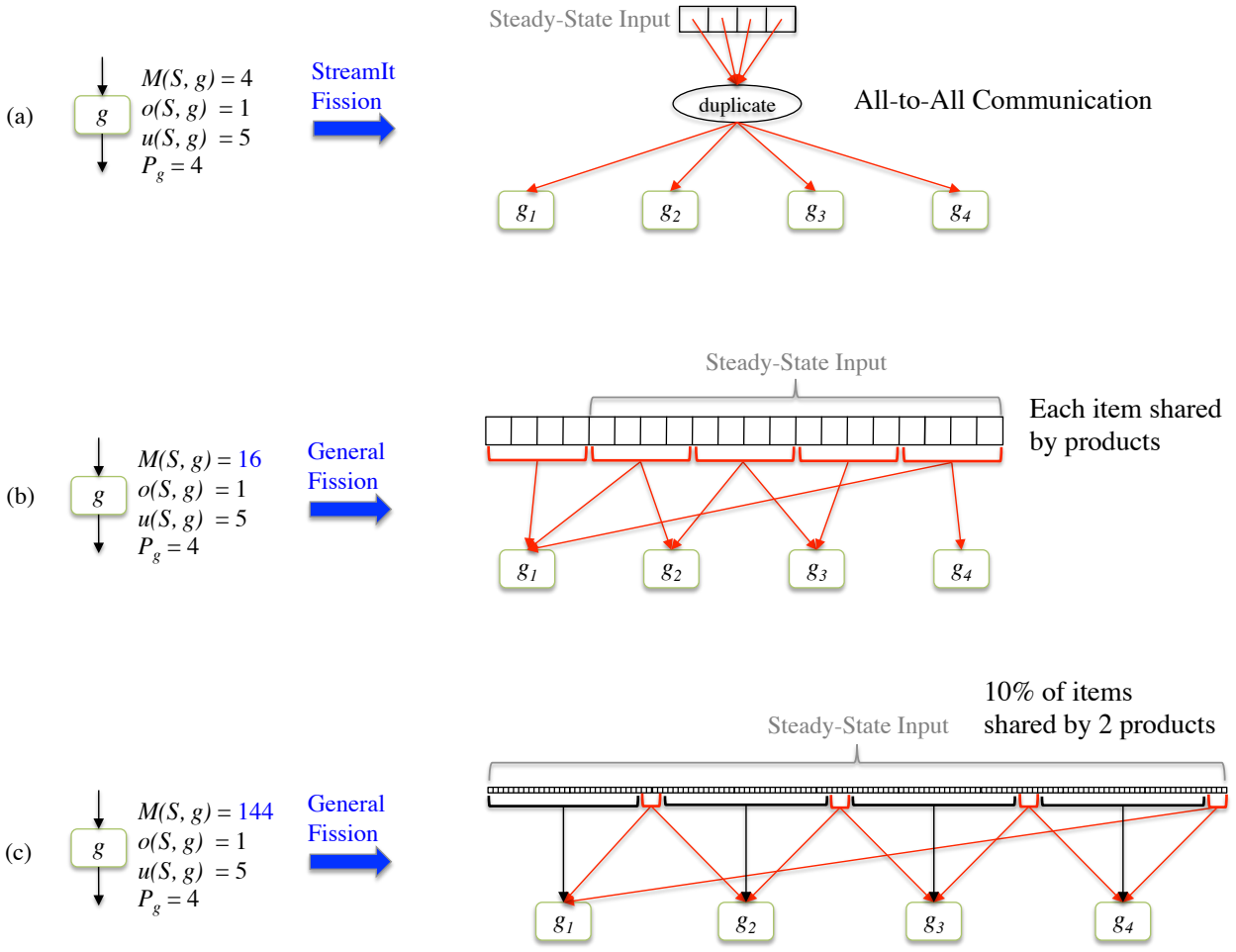


Figure 8-1: An overview of the fission techniques introduced in this chapter. The goal of each is to fiss filter g to 4 cores. Sharing that requires inter-core communication is denoted by red arrows. g has a peek rate of 5 and a pop rate of 1. In (a) we show fission using the StreamIt graph (see Section 6.3.3). In this technique, the fission products of g are placed in a splitjoin with a duplicate splitter, and every input item is duplicated to all of the g_is. This requires all-to-all communication. We call this technique duplication and decimation because each input item is duplicated to all fission products, and a fission product will decimate the item if it does not require it. In (b) we demonstrate fission on the general graph, with its required steady-state alteration ($M(S, g) = 16$). In this case the communication pattern is more efficient, and each item is only shared by 2 fission products. (c) demonstrates sharing reduction. By further increasing the steady-state multiplicity of g ($M(S, g) = 144$), we can reduce the sharing requirement such that only 10% of input items need to be shared by 2 fission products and 90% input items are required by one product.

across fission products when fissing peeking filters, while keeping communication within cores. The techniques effectively applies to a common case across our benchmarks, when a producer and consumer are fissed by the same width. When employing sharing reduction, the compiler is able to *control* the percentage of items that are communicated between cores as a result of fission by altering the multiplicities of nodes in the steady-state. We can bring the percentages stated in beginning of this paragraph for FMRadio and Filterbank down to near zero. However, as we will show, we have settled on 10% as a percentage of inter-core fission communication as a sweet spot.

The combination of CGDTP (Chapter 6) employing the techniques in this chapter and CGSP (Chapter 7) achieves robust scalable performance for TILE64 and the Xeon system across the StreamIt Core benchmark suite. For TILE64, the combined techniques of this dissertation realize a 51.7x 64-core geometric mean speedup over single-core performance for the 12 benchmarks of our suite (59x speedup only considering the 9 fully-stateless benchmarks). The techniques covered in this chapter lead to a 1.8x mean speedup over duplication and decimation for the four benchmarks with peeking when scheduling via CGDTP + CGSP for the TILE64. For the Xeon system, the combined techniques of this dissertation achieve a 14x 16-core geometric mean speedup over single-core performance for our suite. The techniques of this chapter lead to a 5.5x mean speedup over dupdec for the four benchmarks that include peeking when scheduling via CGDTP + CGSP for the Xeon system. These results, coupled with the results of the Chapter 7 for Raw, demonstrate that the techniques covered in this dissertation realize portable, robust, and scalable parallelism for stream programs targeting varying multicore architectures.

The techniques in this chapter are enabled by the rate declarations of static filters; the regular, repeating nature of communication in StreamIt; and the implicit outer scheduling loop of an application. These properties allow the StreamIt compiler to redistribute computation, precisely schedule communication, and alter the steady-state for the purposes of locality and reduced steady-state communication.

8.2 The General Stream Graph

The techniques covered in this chapter are enabled by the expressiveness of the general stream graph. The general stream graph dispenses with the hierarchy and structure found in the StreamIt graph, and adds the ability to coalesce duplication and round-robin distribution into the output of a single node.

The StreamIt graph is a direct representation of StreamIt source code, and it is useful as a representation of the original application and for transformations early in the flow of the StreamIt compiler. Structure and hierarchy are important properties of the StreamIt language for increasing programmer productivity and attracting programmers to the StreamIt language [Thi09]. Also, high-level transformations and calculations, such as scheduling [KTA03], partitioning, fusion [Gor02], and granularity coarsening (Chapter 6) are more naturally expressed because of hierarchy and structure. These techniques rely on the structure and hierarchy to simplify and localize reasoning.

Most networks are not constrained by structure and hierarchy. When calculating communication patterns for a mapping, requiring hierarchy and structure can hinder the expressiveness of the desired communication. For example, requiring single input and single output filters with separate nodes for splitting and joining that are each constrained, limits the types of communication patterns that can be expressed. Synchronization points are present at each splitter and joiner in the

StreamIt graph. The remainder of this section will cover some of notation and representation for general graphs employed in this chapter.

The general graph G is composed of nodes[2] V and edges, E, where $e \in E$, $e = \{(u, v) | u, v \in V\}$. Nodes in the general graph are multiple input and multiple output. The input joining pattern is termed the *input distribution* and is represented as:

$$\textit{ID}(\Sigma, F) \in (\mathbb{N} \times E)^n = ((w_1, e_1), (w_2, e_2), ..., (w_n, e_n))$$

Where n is the width of the input distribution pattern. The input distribution describes the round robin joining pattern for organizing the input data into a single stream, where w_i items are received from edge e_i before proceeded to the next edge, e_{i+i}. The output distribution pattern describes both round-robin splitting and duplication in a single structure:

$$\textit{OD}(\Sigma, F) \in (\mathbb{N} \times (P(E) - \emptyset))^m = ((w_1, d_1), (w_2, d_2), ..., (w_n, d_n))$$

Each d_i is called the *dupset* of weight i. The dupset d_i specifies that the w_i items be duplicated to the edges of the dupset. For both the *OD* and the *ID*, the variable Σ specifies the schedule to which the distribution pattern describes, either I for initialization, or S for steady-state.

Another quantity that is important to the discussion in this chapter is a node f's *copydown*, $C(f)$. This is equal to the number of items that remain in f's input buffer after the initialization schedule has executed. For a peeking filter, $C(f) \geq e(S, f) - o(S, f)$. The general calculation of $C(f)$ subtracts the number of items f consumes in initialization schedule from the number of items produced by the producers of f that are forwarded to f:

$$C(f) = \left[\sum_{p \in \text{OUT(F)}} \textit{RO}(p, f, I) \cdot \left(u(W^P, p) + (M(I, p) - 1) \cdot u(W, p) \right) \right] - \left[o(W^P, f) + (M(I, f) - 1) * o(W, f) \right]$$

8.2.1 Conversion from StreamIt Graph to General Stream Graph

The general stream graph has expressiveness that is a superset of StreamIt graph. It is straightforward to convert the StreamIt graph to the general stream graph, by converting all filters, splitters and joiners to nodes in the general stream graph. Edges are created to represents StreamIt's channels within a container. To establish the edges of the general graph across StreamIt containers, edges are created from the last filter or joiner of the upstream container to the first filter or splitter of the downstream container. Since all nodes in the general stream graph are computation nodes, after the conversion, nodes of the general graph that represent joiners or splitters perform the identity function in their work function.

The above conversion is a first pass at converting to the general stream graph. The conversion replicates all of the synchronization of the original StreamIt graph, e.g., splitters and joiners are explicitly represented. In the next section we describe the process for removing synchronization from the general stream graph by removing all nodes that perform the identity function. After synchronization removal is applied to all identity filters, we achieve a conversion such as the example in Figure 8-2.

[2] In this chapter we refer to nodes of the general graph as either "nodes" or "filters". We use these terms interchangeably.

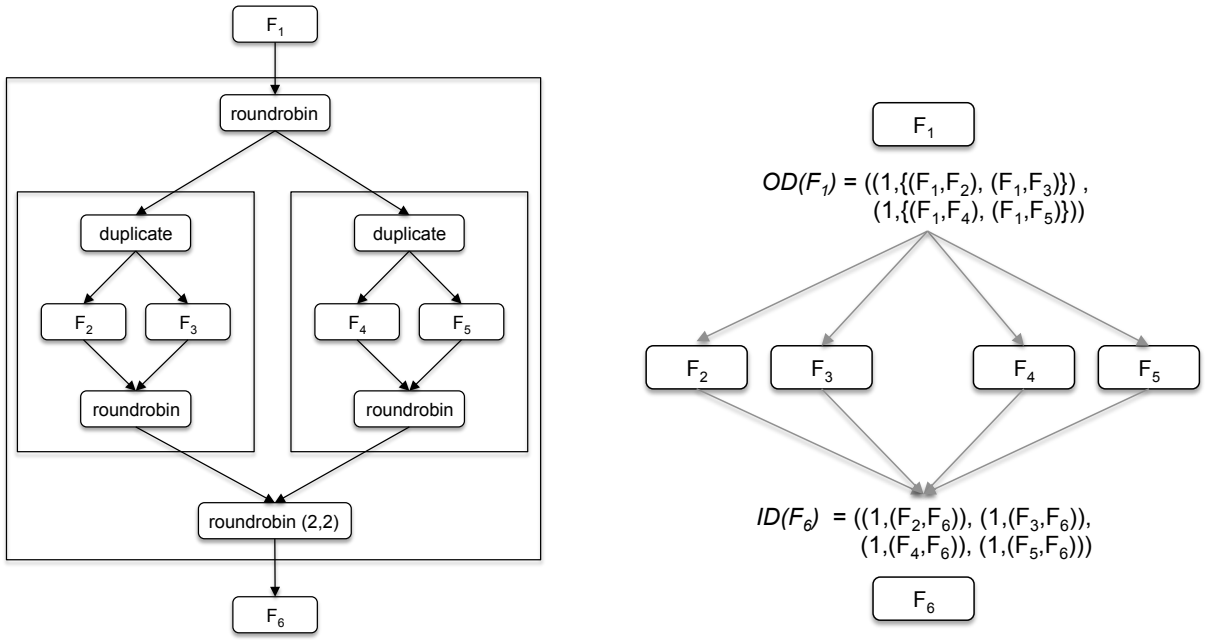


Figure 8-2: An example of the conversion from (a) StreamIt graph to (b) general stream graph. Synchronization removal has been applied to all nodes that represented splitters and joiners in the general stream graph.

8.3 Synchronization Removal for the General Stream Graph

Synchronization removal is a transformation that is applied to nodes calculating the identity function in the general graph such that they can be removed, but their communication patterns remain embedded in the general graph. After conversion to the general graph from the StreamIt graph, synchronization removal is applied to all nodes that represented identity filters[3], splitters, or joiners. Synchronization removal is also applied as a step during fission of nodes in the general stream graph (Section 8.4).

Algorithm 6 gives the synchronization removal algorithm as applied to a single identity node f. The algorithm applies to a single schedule of execution, and should be called separately for both steady-state and initialization (though the algorithm is not parametrized by schedule for clarity's sake). After SYNCHREMOVE has been applied to f for both the initialization and steady-state, f can be removed from the set of nodes, and all edges with f as a source or destination can be removed from the set of edges. At the high-level, SYNCHREMOVE replaces all edges that reference f by determining edges' endpoints *through* f. For output distributions for producers of f, the process is complicated because f's output distribution may duplicate or distribute the item in a complex manner. This added distribution may not "fit" in the number of edges to f that a producer's output distribution originally contained. Because of this, the output distribution of each producer must be matched in size to the distribution pattern that f applies to the producer's

[3]StreamIt includes a special predefined filter for identity filters. This filter is useful for describing data-reordering such as transpose and bit-reversals [Thi09].

outputs. The same complexities affect the input distributions of the consumers of f. Their original distribution may not be have room to fit the pattern that f applied to it's input items.

We describe the algorithm employing an example in Figure 8-3. Figure 8-3(a) gives the partial general graph for our example. We seek to remove the synchronization of f. In Figure 8-3(a) the *OD*'s and the *ID*'s for each filter is given, where it is necessary for the removal of f. SYNCHREMOVE operates by unrolling the input and output distribution pattern of f such that they "align". Once they align, for item i, it is simple to determine the producer and consumer(s) of i by inspecting the unrolled *ID* and *OD* at position i. The algorithm begins by calculating the LCM of the *ID* and *OD* such that they can be unrolled and aligned to the same length (lines 1 to 13). UNROLL unrolls a distribution pattern such that all weights are 1; REPEAT repeats a distribution pattern a number of times by concatenation. Figure 8-3(b) demonstrates this phase. The *OD* (*outMatched*) and *ID* (*inMatched*) of f are unrolled and repeated such that they are of the same length. f's unrolled *ID* has length 3 and f's unrolled *OD* has length 5 for an LCM of 15; the unrolled *ID* is repeated 5 times, and the unrolled *OD* is repeated 3 times.

The next step in SYNCHREMOVE loops over all the producers feeding f and calculates new output distributions so that no dupsets remain that reference f. For each producer p, p's *OD* is unrolled and repeated such that the number of items p sends to f equals the number of items f receives from p's *inMatched* (lines 15 to 27). p's new output distribution is *unrolledDist*. Next, the output dupsets in *unrolledDist* are looped over. For each dupset that includes f as a destination in an edge of the dupset, the destination that f sends the item to is calculated by consulting f's *inMatched* at the index of the next item received from p. Then f's *outMatched* is consulted at this index to find the new destinations. New edges (more than one if f duplicated the item) are created that replace the edge to f. These edges are then installed in p's *unrolledDist*. The next index of p in f's *inMatched* is then calculated, restarting the linear search if we reach the end of the *inMatched*. After all output dupsets of p that reference f are replaced, p's output distribution is re-rolled using run-length encoding.

Figure 8-3(c) shows an example of this process for node I_1. I_1's output distribution is unrolled to match the number of times I_1 is referenced in f input distribution. I_1 sends 10 items to f, and f receives 5 items in it's *inMatched*. The lowest common multiple of 10 and 5 is 10, so I_1's output distribution is not repeated. The new destinations for the dupsets of I_1 are calculated by looping over I_1's unrolled dupset, and for each dupset that included f (blank in the *unrolledDist* of Figure 8-3(c)), the aligned distributions of f are consulted at points that received from I_1. Note that the dupset that includes only an edge to v is not affected. Figure 8-3(d) gives the re-encoded *OD* of I_1.

The final step in SYNCHREMOVE is remove all the edges that reference f of the input distributions of the consumers of f. This begins on line 48, and is similar to replacing the edges of producers of f described above. See Figure 8-3(c) and (d) for an example of this on node O_2 of the example.

The worst-case time complexity of one application of SYNCHREMOVE of Algorithm 6 on f occurs when f is connected to all nodes of the general graph. We cannot bound the input and output distribution patterns for a node, however, N is a safe worst-case average. The time complexity is $O(N^3)$ as there are N nodes with distributions to patch, and for each node, we must search over the number of edges or dupsets in the distribution (worst-case average N), and for each edge that must be replaced, we must search f's (unrolled and repeated worst-case average some constant of N) input or output distribution.

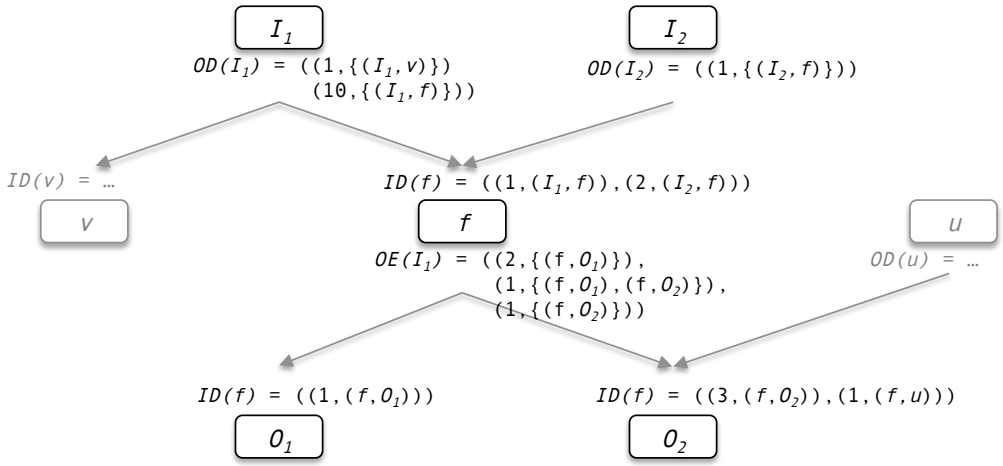
Algorithm 6 SynchRemove

SYNCHREMOVE($G = (V, E)$, f)

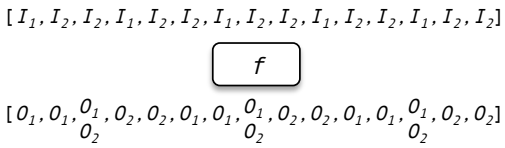
▷ Find the LCM of the number of items joined and split for f
▷ So we can unroll them to same number of items
totalItems ←LCM($\sum_{w \in ID(f)} w$, $\sum_{w \in OD(f)} w$)
▷ Calculate number of times to repeat in (join) and out (split) distributions for f
inRepeat ← *totalItems* / $\sum_{w \in ID(f)} w$
outRepeat ← *totalItems* / $\sum_{w \in OD(f)} w$
▷ Unroll the input and output patterns to all weight 1,
▷ explicitly repeating for > 1 weight in original
inUnrolled ← UNROLL(*ID(f)*)
outUnrolled ← UNROLL(*OD(f)*)
▷ Repeat the dist. patterns so they are equal length
inMatched ← REPEAT(*inUnrolled*, *inRepeat*)
outMatched ← REPEAT(*outUnrolled*, *outRepeat*)
▷ Replace all edges to f in the producers
for all $p \in$ IN(f) **do**
    ▷ Sum weights of p's output dist that send to f
    sumOutWeightToF ← SUMOUTWEIGHT(p, f)
    ▷ Sum weights of f's input dist that receive from p
    sumInWeightFromP ← SUMINWEIGHT(p, f)
    ▷ Unroll and repeat output dist of p
    ▷ so that it can be synced with occurances of p in input dist of f
    repeatOutP ←
        LCM(*sumOutWeightToF*, *sumInWeightFromP* $*$ *inRepeat*) / *sumOutWeightToF*
    ▷ Create an uncompressed list of dests for p (each has weight 1) from output dist
    unrolledDist ← UNROLL(*OD(p)*)
    ▷ Repeat the unrolled list *repeatOutP* times
    unrolledDist ← REPEAT(*unrolledDist*, *repeatOutP*)
    ▷ Loop over edges of p, replacing edges with dest f based on aligned in and out distribution
    inIndex ← -1
    for $i \leftarrow 0$, |*unrolledDist*| **do**
        dupSet ← *unrolledDist*[i]
        if $(p, f) \notin$ *dupSet* **then**
            continue
        end if
        ▷ Starting at *inIndex*$+1$, wrapping linear search for edge with p as source in *inMatched*
        inIndex ← FINDNEXTSOURCEWITH(p, *inMatched*, *inIndex*)
        ▷ The new dests are found in the aligned out dist of f at *inIndex*
        newDests ← *outMatched*[*inIndex*]
        ▷ Replace f with p as the source for edges in the dup set *newDests*
        newDests ← REPLACESOURCE(*newDests*, f, p)
        ▷ Replace old dup set with the new aligned, substituted dup set
        unrolledDist[i] ← *newDests*
    end for

Algorithm 7 SynchRemove Continued

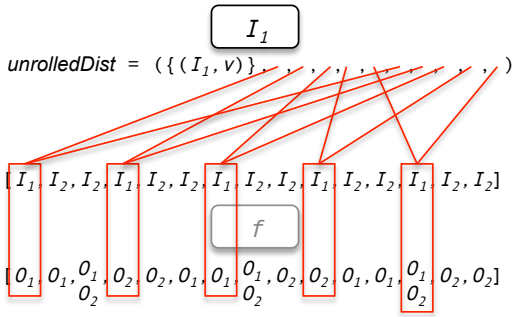
```
    ▷ Re-roll the out dist. of p using a run-length encoding
    OD(p) ← ReRoll(unrolledDist)
end for
▷ Replace all edges from f in the consumers
for all c ∈ Out(f) do
    ▷ Sum weights of f's output dist that send to c
    sumOutWeightToC ← SumOutWeight(f, c)
    ▷ Sum weights of c's input dist that receive from f
    sumInWeightFromF ← SumInWeight(f, c)
    ▷ Unroll and repeat input dist of c
    ▷ so that it can be synced with occurances of c in output dist of f
    repeatInC ←
        LCM(sumInWeightFromF, sumOutWeightToC ∗ outRepeat) / sumInWeightFromF
    unrolledDist ← UnRoll(ID(c))
    unrolledDist ← Repeat(unrolledDist, repeatInC)
    ▷ Loop over input dist of c,
    ▷ Replacing edges with source f based on aligned in and out distribution
    outIndex ← −1
    for i ← 0, |unrolledDist| do
        e ← unrolledDist[i]
        if e ≠ (f, c) then
            continue
        end if
        ▷ Starting at outIndex + 1, wrapping linear search for edge with c as dest in outMatched
        outIndex ← FindNextDestWith(c, outMatched, outIndex)
        ▷ New edge has c as dest and the source from the aligned input dist. inMatched
        newSource ← Source(inMatched[outIndex])
        unrolledDist[i] ← (newSource, c)
    end for
    ▷ Re-roll the in dist. of c using a run-length encoding
    ID(c) ← ReRoll(unrolledDist)
end for
```



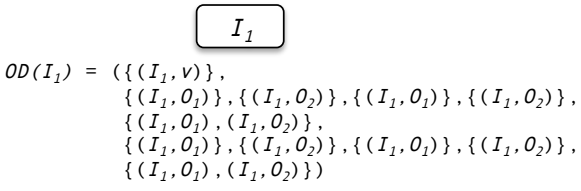
(a)



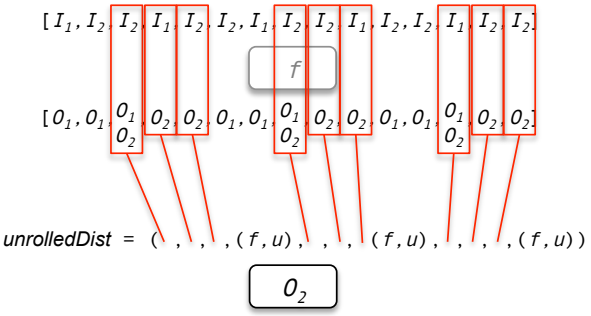
(b)



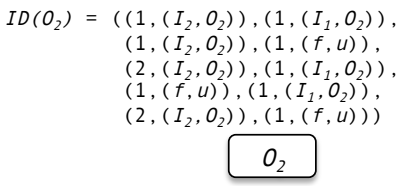
(c)



(d)

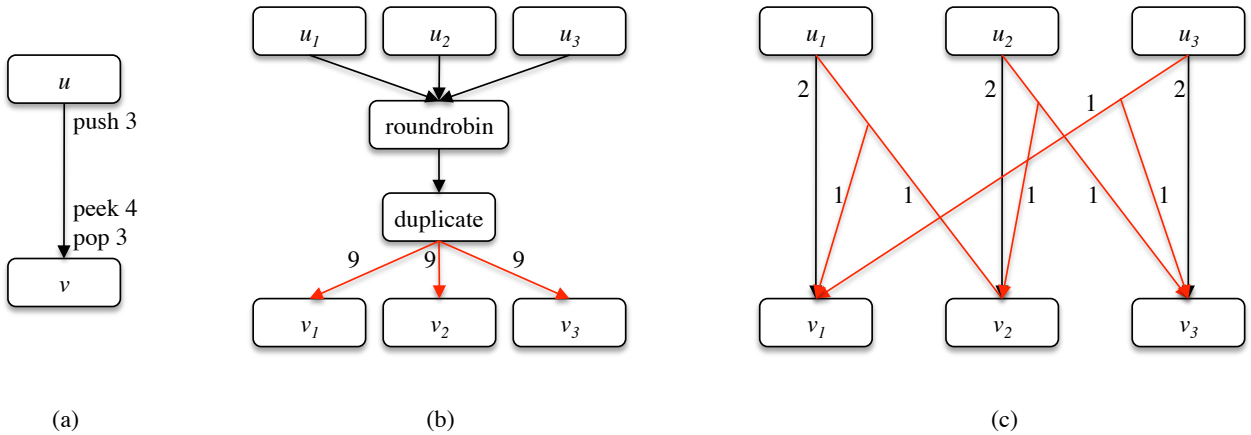


(e)



(f)

Figure 8-3: Synchronization removal example. Remove the synchronization caused by filter f assuming it performs the identity function. Details are covered in the text.



(a) (b) (c)

Figure 8-4: An example of the savings in communication that fission on the general graph can provide. (a) gives two filters u and v that are to be fissed 3 ways. (b) demonstrates StreamIt graph fission of u and v. 27 items are communicated by the duplicate splitjoin to the fission products of v per steady-state. (c) demonstrates the resulting communication pattern when general graph fission of u and v is performed. Items are duplicated only when necessary (as shown by red arrows). Only 12 items are communicated per steady state. Note that the item communicated from u_3 to v_1 is not consumed until the next steady-state iteration.

8.4 Fission on the General Stream Graph

Section 6.3.3 gives the fission transformation as applied to a filter in the StreamIt graph. This implementation duplicates all input data items to all fission products. If a product does not need the input item, it decimates it. This scheme leads to much unnecessary communication, as duplication is implemented via communicating items to multiple cores. This may be a problem depending on the architecture or application.

This section covers an improved form of fission applied to nodes of the general graph. Applying fission to nodes of the general graph allows the compiler to express more precise communication patterns for the distribution of input items to the fission products. This allows the fission technique to avoid unnecessary duplication (and thus communication) of input items. Figure 8-4 demonstrates an example of the savings in communication for fission on the general graph versus fission on the StreamIt graph.

A key design goal for the implementation of fission on the general graph was that communication between levels of the stream graph be efficient (see Section 6.3.2 for the description of a level). Often when data parallelism is applied, the producer filters in level l are fissed at the same width as the consumer filters in level $l + 1$. To make this common case efficient, general graph fission communicates a large number of items between disjoint producer-consumer pairs from the levels. Next, a small number of items (the number of items inspected by the consumer filter) is duplicated and communicated from consumer to a producer in a different pair. Figure 8-4(c), demonstrates this pattern, with u_i and v_i forming the pairs. Each pair should be mapped to the same core, so that the bulk of communication for the fission distribution pattern is intra-core. For Figure 8-4(c), if the pairs are mapped to same core, only 3 of 12 items are communicated inter-core.

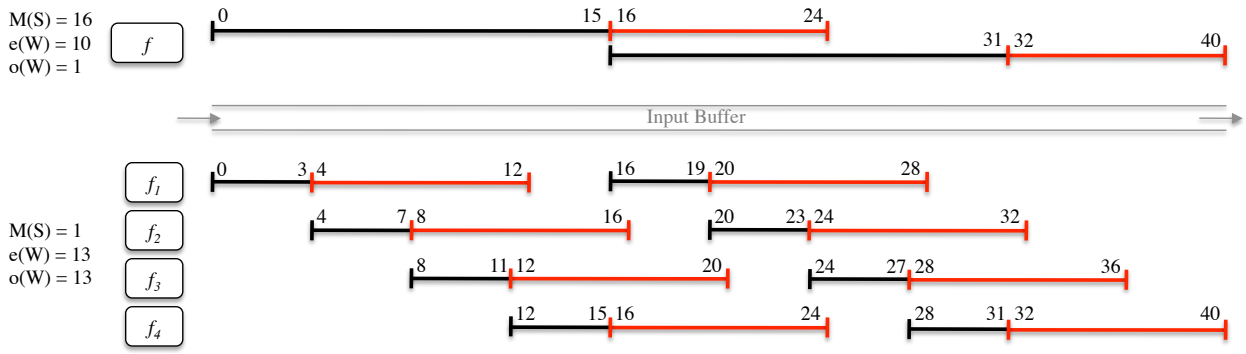


Figure 8-5: An example of the duplication of items required by fission. Filter f is fissed by 4 into f_1 - f_4. Two steady-states of the item indices required for both f and the fission products are shown. Item indices that are inspected but not dequeued are in red. For the fission products, it is required to duplicate 1 out of 4 items to all 4 filters, and the remaining 3 items are duplicated to 3 filters. Notice that the number of items inspected by the fission products is the same as f, but the total number of items dequeued per steady-state is split across the f_is.

In the general case, when fissing a filter that peeks, i.e., a filter f with $e(W, f) - o(W, f) = dup_f < C(f) > 0$, by P, the producers of f need to duplicate output items to an average of:

$$\max\left(1 + \frac{C(f)}{M(S, f) \cdot o(W, f)/P}, P\right)$$

fission products of f. Figure 8-5 gives an example of the required sharing for a fission application. The filter f is duplicated 4 ways, has $C(f) = 9$, $o(W, F) = 4$, and $M(S, f) = 16$. From the above formula, each item is duplicated to an average of 3.25 fission products.

Before we describe the fission transformation, we need to list the preconditions that must hold before fission of f by P in the general graph can be performed:

- $C(f) < (M(S, f)/P) \cdot o(W, f)$. The items remaining of the input buffer of the original filter after initialization must be less than the number of items dequeued by each fission product. This implies $dup_f < (M(S, f)/P) \cdot o(W, f)$ since $dup_f \leq C(f)$. If dup_f for the steady-state is greater than the number of items that each fission product pops, then some items will be duplicated to more than 2 fission products. We do not support this case, as it can be avoided, and we want to limit inter-core communication.

- $M(S, f) \mod P = 0$. In this version of the algorithm we create fission products with equal work, an equal division of the original steady-state multiplicity of f. This restriction can be relaxed via simple modifications to the algorithm.

All the multiplicities of the steady-state can be multiplied by the same constant c, and the result will still be a valid steady-state. We call this process *increasing* the steady-state of the graph by c. Each of these preconditions can be enforced by increasing the steady-state of the graph.

The details of fission on a filter of the general stream graph are given in Figure 8-6. The process includes the following steps (Figure 8-6 illustrates steps 1-9):

1. Create P copies of f and set their rates and work functions according to Figure 8-6.
2. Create two identity nodes, ID_I and ID_O, that will encode the distribution for the fission.
3. Move the initialization stage computation of f to f_1 according to Figure 8-6.
4. Move input distribution of f to ID_I replacing occurrences of f with ID_I in edges.
5. Move output distribution of f to ID_O, replacing occurrences of f with ID_O in edges.
6. Create the fission duplication pattern in the output distribution of ID_I.
7. Create a round robin joining pattern for the output identity filter ID_O to receive from each fission product.
8. For each node p that is a producer of f, replace the occurrences of f with O_I in the edges of the dupsets of p's output distribution.
9. For each node c that is a consumer of f, replace the occurrences of f with O_O in incoming edges c's input distribution.
10. SYNCHREMOVE(ID_I)
11. SYNCHREMOVE(ID_O)

To understand the transformation, we first need to understand the item distribution and sharing that is required by fission on a filter f that adheres to the preconditions above. Figure 8-7 gives another example of the input items required by fission products. In this example, both:

$$C(f) = \textit{dup}_f < (M(S,f)/p) \cdot o(W,f)$$
$$M(S,f) \mod P = 0$$

so f adheres to the preconditions stipulated above for general fission. In the example, the items read for f plus its fission products for $P = 4$ are shown for the initialization plus two steady-states. After the initialization stage, $C(f) = 2$ items are enqueued to the input buffer by the producer(s) to f, and $M(S,f) \cdot o(W,f) = 16$ items are enqueued by the producer(s) for each steady-state. Examining the sharing requirement for fission products of the second steady-state, we can see a pattern emerge with the following features:

- No item is read by more than 2 fission products.
- A fission product does not need to remember items across steady-state executions of itself.
- Only the first fission product f_1 is required to receive the $C(f)$ initialization items because $C(F) < (M(S,f)/p) \cdot o(W,f)$, and it will consume the $C(f)$ items on its first invocation.
- The presence of the $C(f)$ items in the input buffer after initialization must be accounted for by shifting the read pattern for the fission products. The first fission product f_1 is offset by $C(f)$ items in that it reads its first $C(f)$ items from the previous execution stage. In the steady-state, f_1 executing at steady-state iteration i shares items with f_P executing at steady-state iteration $i - 1$.

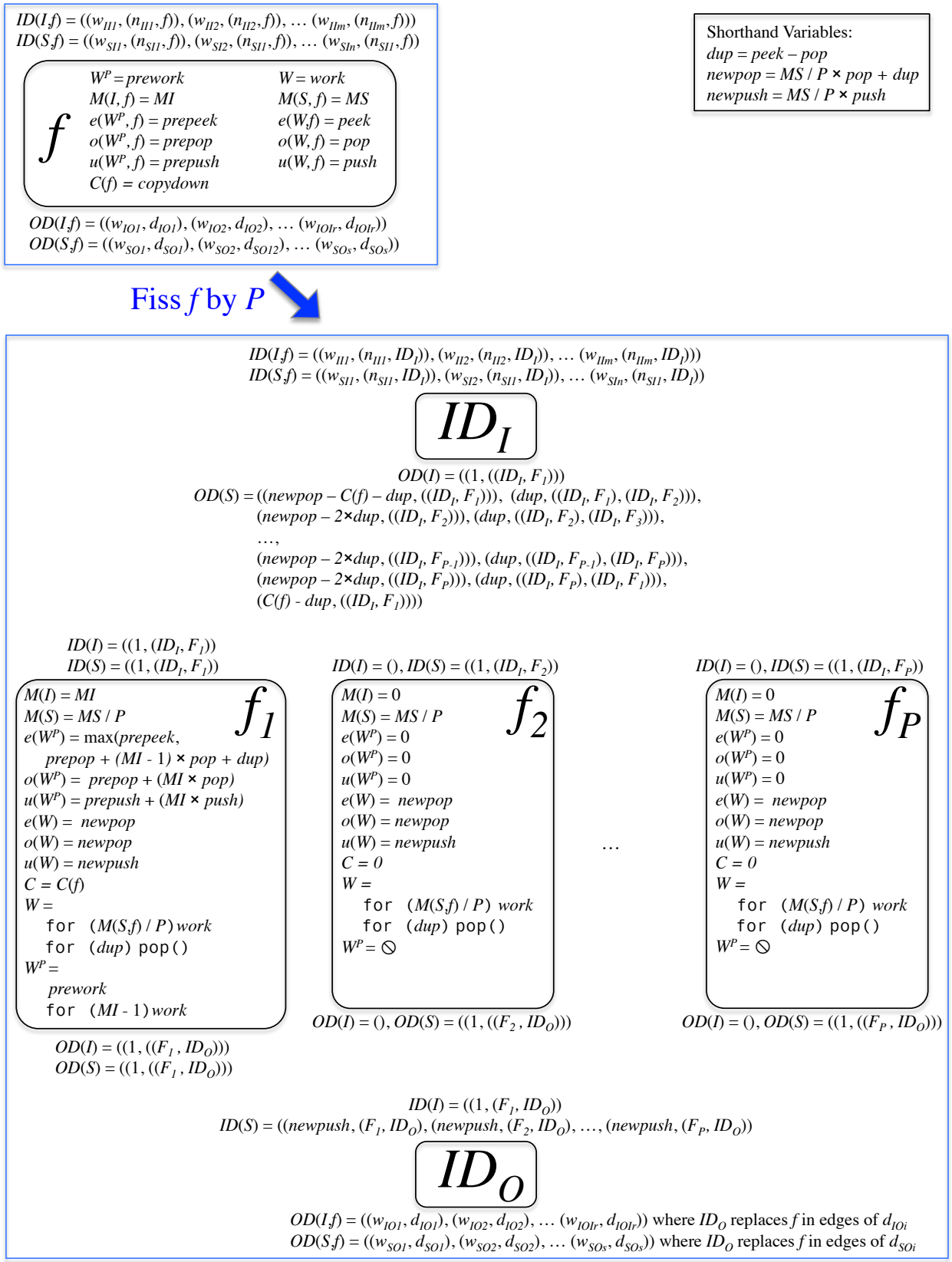


Figure 8-6: Fission of a node f by P in the general stream graph.

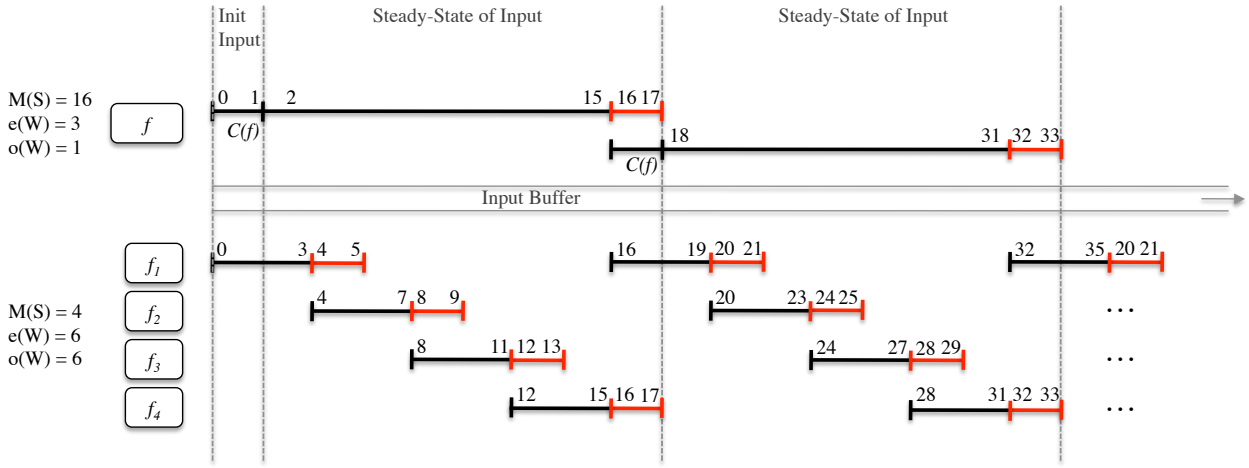


Figure 8-7: An example of the sharing of items required by fission for a filter f that adheres to the preconditions for general fission. Filter f is fissed by 4 into f_1 - f_4. $C(f)$ items remain on the input buffer after initialization but before steady-state commences. Two steady-states of the item indices required for both f and the fission products are shown. Item indices that are inspected but not dequeued are in red. For the fission products, it is required to duplicate all items to 2 filters.

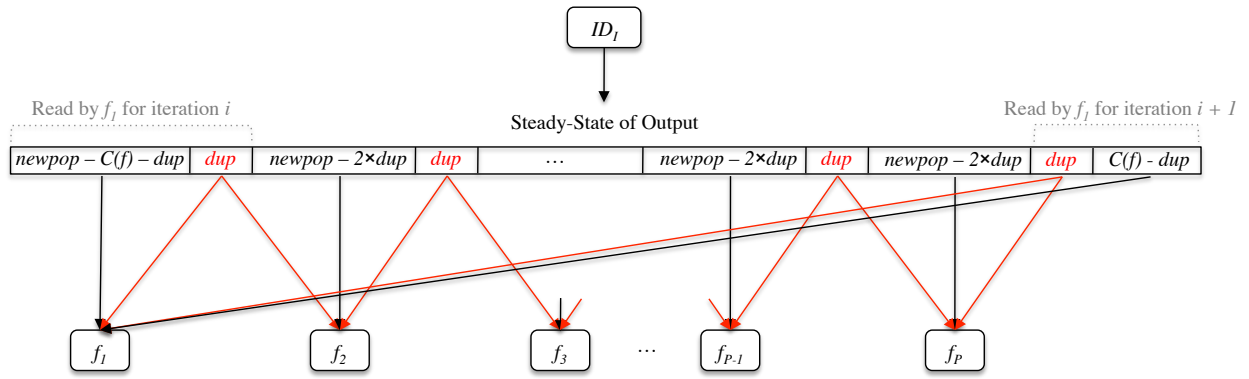


Figure 8-8: The steady-state output distribution installed for identity node ID_I by general fission. Red edges denote items that are shared across fission products via duplication. Though not demonstrated in the example of Figure 8-7, in the general case, if $C(f) > dup$, $C(f) - dup$ items at the end of the steady-state input are distributed to f_1.

The general fission transformation creates two identity nodes (ID_I and ID_O) that are encoded to implemented the data distribution for the fission products. The pattern seen in Figure 8-7 and described above is common to the transformation for all filters we seek to fiss that meet the preconditions of the transformation. This sharing pattern is encoded in the output distribution pattern for the identity filter ID_I. Figure 8-8 shows the weights for the output distribution and how these weights are distributed to the fission products.

The input distribution of ID_I is set to the input distribution of f. Thus, ID_I joins data as described by f's input distribution, and splits data according to the fission output pattern for sharing between at most 2 filters. The output identity ID_O collects the output items of the fission products in a weighted round robin, and the distributes them according to f's output distribution.

Since a fission product, does not need to remember items across firing, a fission product's peek rate equals its pop rate in the steady-state. This rate equals the division of f's dequeued items plus the number of items inspected but not dequeued by f for each firing:

$$o(S, f_i) = e(S, f_i) = \frac{M(S, f)}{P} \cdot o(W, f) + \textit{dup}$$

The peeking of the original filter f is now encoded in the sharing across fission products achieved via the duplication pattern.

The computation and communication performed by f during the initialization stage is transferred completely to the first fission product, f_1. Since, by construction, only f_1 requires the items remaining after the initialization stage. The other fission products are idle during this stage.

The final steps of the general fission transformation applies SYNCHREMOVE to remove the identity filters, and stitch the communication directly between the fission products and f's producer(s) and consumer(s).

8.5 The Common Case Application of General Fission

The techniques covered above are employed during the compiler flow outlined in Chapter 6 to effectively leverage data and task parallelism. The coarsened StreamIt graph is converted into a general by employing SYNCHREMOVE (see Section 8.3 for details). JUDICIOUSFISSION is performed on the coarsened general graph using the technique covered in Section 6.3.2. Fission of a node in the general graph replaces fission of a filter in the StreamIt graph (see Section 6.3.3) in the process of judicious fission. Conversion to the general graph with synchronization removal, accompanied by employing general fission, greatly reduces the inter-core communication requirement of the resulting mapping when fissing filters that peek. Figure 8-4 gives a simple example of the potential savings.

The design of the general fission algorithm was informed by the fact that for most fission applications of judicious fission, a producer and consumer are fissed by an equal width. This is because in most of our benchmarks, a level's width of task parallelism is equal to the width of task parallelism for it's consuming level. In this case, fission on the general graph will map the bulk of communication to intra-core resources. Furthermore, in the next section we demonstrate that the ratio of inter-core communication to total communication is parametrizable for this case.

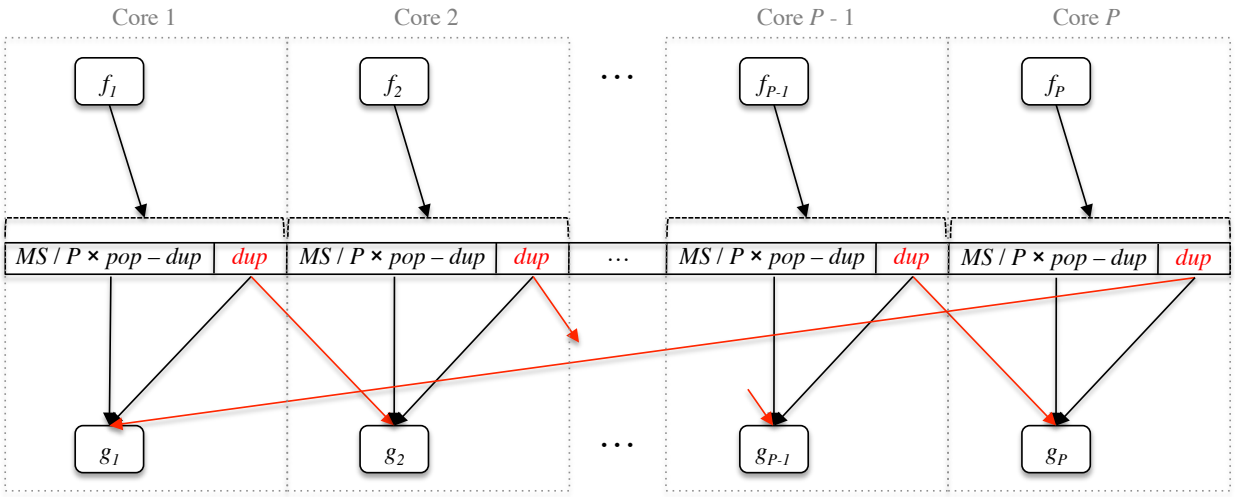


Figure 8-9: Communication details for fission products of consumer f and producer g each fissed by P. f was a single output node, and g was a single input node. Furthermore, $C(g) = \textit{dup}_g$. Each f_i, g_i pair are mapped to the same core. The sizes of the buffer sections are in terms of g. Red arrows denote inter-core communication.

To understand why general fission achieves the reduction of inter-core communication for this case, we must remember a few properties of the steady-state schedule of the stream graph. For single output producer f and single input consumer g, where $(f, g) \in E$:

$$M(S, f) \cdot u(W, f) = M(S, g) \cdot o(W, g)$$

When f and g are fissed (employing general fission) by the same amount P, the above equality no longer holds across all the fission products if g peeks. This is because the peeking of g is converted into duplication of items to the products of g's fission application. The fission products of g each consumes more items than each fission product of f produces. However, we can make it the case that *most* of the outputs of a fission product f_i are directed to g_i, for each $1 \leq i \leq P$. When each f_i and g_i are mapped to the same core, the bulk of communication is intra-core.

Figure 8-9 illustrates the details of communication between f and g when each is fissed by P. In the figure it is assumed that $C(g) = \textit{dup}_g$, meaning the number of items remaining after the initialization schedule equals the number of items that g inspected but did not pop. Each f_i produces $M(S, f)/p \cdot u(S, f)$ items, while each g_i must consume the original pop rate multiplied by it's slice of f's multiplicity plus the number of inspected items of g:

$$M(S, g)/P \cdot o(W, g) + C(g)$$

Since $M(S, g)/P \cdot o(W, g) = M(S, f)/p \cdot u(S, f)$, the number of items each f_i produces and the number of items each g_i consumes differs by $C(g)$. As Figure 8-9 demonstrates, each g_i receives $C(g)$ duplicated items from g_{i-1} (with g_1 receiving from f_p). When the fission products are assigned to cores as given in the figure, the percentage of inter-core communication to total communication can be reduced to:

$$\textit{InterCore}(g) = \frac{C(g)}{M(S,g)/P * o(W,g)} \tag{8.1}$$

This percentage corresponds to the best case, in which a producer and consume are fissed by the same amount and $C(g) = \textit{dup}_g$. This case is common in our benchmarks.

The next section covers a technique that seeks to decrease the percentage of total communication that must be communicated inter-core. The technique directly decreases the number of items that are shared, while also increasing the alignment of communication to minimize inter-core communication.

8.6 Reducing Sharing Between Fission Products

Since streaming applications typically execute for many iterations of the steady-state, it is legal to increase the steady state by multiplying each filter's steady-state multiplicity by the same constant. As long as this constant c is less than the number of steady-state iterations I of the application, and c is a factor of I, then the transformation is legal. As mentioned above, we call this *increasing* the steady-state by c. This transformation increases the latency of the steady-state.

By recognizing this property, we can directly reduce the amount of inter-core communication that occurs between the fission products of a fissed peeking filter for certain cases of general fission, including the common case. As we have seen above, the duplicated shared items translate directly into inter-core communication for the common case of fissing a producer and consumer by the same width. For producer f and consumer g, fissed by P, where each f_i is mapped the to same core as g_i, Equation 8.1 gives the formula for calculating the percentage of the total items that are duplicated for a fission application of g. In the equation we can directly control the steady-state multiplicity of g, $M(S,g)$. We can calculate a constant c_g such that the percentage is less than a threshold $T_{\textit{sharing}}$:

$$c_g = \frac{1}{T_{\textit{sharing}}} \cdot \frac{C(g) \cdot P}{M(S,g) \cdot o(W,g)} \tag{8.2}$$

where $0.0 < T_{\textit{sharing}} \leq \textit{InterCore}$. Increasing the steady-state of the graph by c_g before general fission is applied will assure that the percentage of items duplicated will be equal to $T_{\textit{sharing}}$. For each producer f and consumer g that is duplicated by the same amount, and mapped to cores such that f_i is mapped to same core as g_i, this will ensure that at most $T_{\textit{sharing}}$ fraction of total items are communicated inter-core.

The analysis of the inter-core communication requirements of the fission of producer f and consumer g has so far assumed that $C(g) = \textit{dup}_g$. This is not always the case however. If $C(g) > \textit{dup}_g$, the communication between the fission products of f and g will be unaligned. Figure 8-10 illustrates this case. In the graph before fission is applied, $C(g)$ items will remain in the input buffer between steady-state iterations. After fission by P, for each steady-state, g_1 will first consume the $C(f)$ items that f_p sent it from the previous steady-state iteration. g_1 will then require:

$$M(S,g)/P \cdot o(W,g) + \textit{dup} - C(g)$$

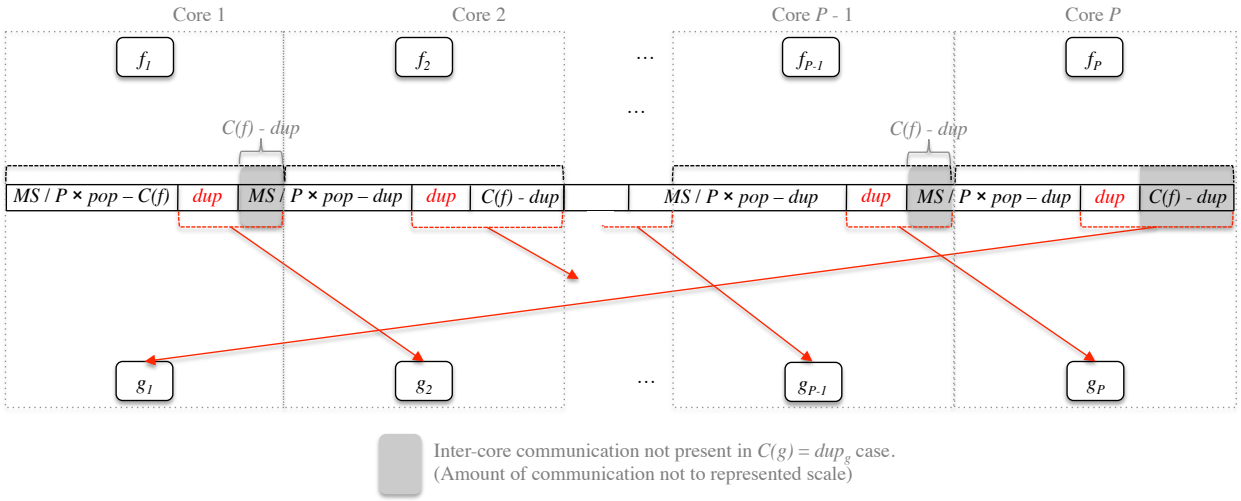


Figure 8-10: Inter-core communication details for fission products of consumer f and producer g each fissed by P. f was a single output node, and g was a single input node. For g, $C(g) > \textit{dup}_g$. Each f_i, g_i pair are mapped to the same core. The sizes of the buffer sections are in terms of g. Red arrows denote inter-core communication.

additional items from f_1.[4] This requirement is less than the number of items that f_1 produces $M(S,g)/P \cdot o(W,g)$ since $C(g) > \textit{dup}$. So, $C(g) - \textit{dup}$ items must be transferred from f_1 to g_2 in addition to the shared and duplicated *dup*. In all, $C(g)$ items must be transferred from f_i to g_{i+1}. Thus, even in the general case where $C(g) > \textit{dup}_g$, Equation 8.1 calculates the percentage of total communication that is inter-core for the fission of g by P given the original steady-state. Furthermore, Equation 8.2 calculates the constant c_g by which to increase the steady-state so that inter-core communication resulting from the fission of g is equal to $T_{\textit{sharing}}$ fraction of total communication.

8.6.1 Sharing Reduction for Other Fission Cases

The sharing reduction optimization does not apply as simply to all cases of fission of a peeking filter. Consider the case where we have single output producer f and single input consumer g with $(f,g) \in E$, but f and g are fissed by differing widths, i.e., $P_f \neq P_g$. This case engenders inter-core communication not only because of sharing but because of the communication required by the differing fission widths. For simplicity of analysis, let us assume that $C(g) = 0$, and P_f is a multiple of P_g. Figure 8-11 presents an example with these assumptions. In the example, f and g are fissed by 8 and 4 respectively. Since we want to maintain data parallelism, each f_i, $1 \leq i \leq P_f$, is mapped to a distinct core, as is each g_j, $1 \leq j \leq P_g$. Since each f_i produces half the number of items required by each g_j, half the communication in this case is inter-core, caused by the misalignment of the rates of the producer and consumer fission products.

Going forward, let us assume that $P_f > P_g$. If P_f is a multiple of P_g, the analysis is straightforward: the percentage of inter-core communication to total communication is $(1 - \frac{P_g}{P_f})$. If P_f is

[4] It is guaranteed that $M(S,g)/P \cdot o(W,g) > C(g)$ by the precondition of general fission.

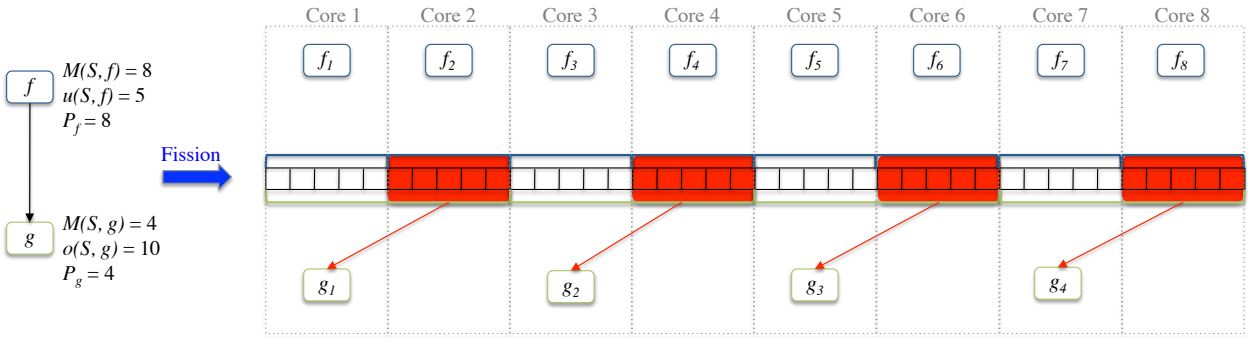


Figure 8-11: An example where the producer filter f is fissed by a greater width than the consumer filter g. Assume $C(g) = 0$. Red denotes inter-core communication. Since each f_i produces half the required input items for each g_j, half the communication in the steady-state is inter-core. The percentage equals the ratio of $\frac{P_g}{P_f}$.

not a multiple of P_g misalignment also occurs because for one or more g_j, it's input does not begin in alignment with any f_i's. The analysis of the percent of inter-core communication for this latter case is complex, but it suffices for our purposes to bound it by $(1 - \frac{P_g}{P_f})$.

Additionally, if $C(g) > 0$, it is required to share $C(g)$ items between two cores. There are P_g sections of $C(g)$ items, and this data has to be communicated inter-core to one other core. Given this analysis, we can quantify an *approximation* of the percentage of inter-core communication for the fission application of g when $P_f > P_g$:

$$\textit{InterCore}(g) = \frac{(1 - \frac{P_g}{P_f}) \cdot M(S,g) \cdot o(W,g) + P_g \cdot C(g)}{M(S,g) \cdot o(W,g)} \tag{8.3}$$

$$\textit{InterCore}(g) = (1 - \frac{P_g}{P_f}) + \frac{P_g \cdot C(g)}{M(S,g) \cdot o(W,g)} \tag{8.4}$$

The first term on the RHS of Equation 8.4 gives the percentage of inter-core communication produced by the fission width inequality. The second term quantifies the percentage of inter-core communication caused by the sharing due to peeking.

Consider the case where g has multiple inputs. In this case g receives $\textit{RI}(f,g,S) \cdot M(S,g) \cdot o(W,g)$ items from f during the steady-state, and $(1 - \textit{RI}(f,g,S) \cdot M(S,g) \cdot o(W,g))$ from its other producers. There is a choice: for which producer of g to optimize the communication of g? If f is chosen, then the assignment of filters to cores should minimize the inter-core communication by co-locating the fission products of f and g just as the assignment would when g is single input. However, the calculation must now account for the inter-core communication of the other producers sending to the fission products of g. We define $\textit{InterCore}(g,f)$ to approximate the percentage of inter-core communication to total communication for the input of the fission products of g optimizing for the placement of the fission products of f:

$$\textit{InterCore}(g,f) = (1 - \textit{RI}(f,g,S)) + (1 - \frac{P_g}{P_f}) \cdot \textit{RI}(f,g,S) + \frac{P_g \cdot C(g)}{M(S,g) \cdot o(W,g)} \tag{8.5}$$

The first term accounts for the percentage of items that g's fission products receive from producers that are not products of f. The second term, the percentage items that are communicated inter-core because of the fission width mis-alignment, must now account for the fact that not all items are from f. Equation 8.5 can be simplified to:

$$\textit{InterCore}(g, f) = (1 - \textit{RI}(f, g, S) \cdot \frac{P_g}{P_f}) + \frac{P_g \cdot C(g)}{M(S, g) \cdot o(W, g)} \tag{8.6}$$

How do we decide whether it is worthwhile to apply the sharing reduction optimization to g given Equation 8.6? When the sharing reduction optimization is applied, the steady-state multiplicity of g is increased by a constant c. If this is applied to Equation 8.6, the first term will be unaffected, however the second term can be reduced because a smaller percentage of steady-state items needs to be duplicated. By Amdahl's law, sharing reduction can only decrease $\textit{InterCore}(g, f)$ to under $T_{\textit{sharing}}$ if the second term of Equation 8.6 accounts for more than $(1 - T_{\textit{sharing}})$ of the duplication. We define a constant $T_{\textit{apply}}$ such that sharing reduction should only be applied to g optimizing for f if:

$$T_{\textit{sharing}} > T_{\textit{apply}} \geq (1 - \textit{RI}(f, g, S) \cdot \frac{P_g}{P_f}) \tag{8.7}$$

Finally, since it was assumed that $P_f > P_g$, in order to generalize the above equation to account for the opposite case, we must choose the appropriate fraction of local communication by taking the minimum of the two fission width ratios:

$$T_{\textit{sharing}} > T_{\textit{apply}} \geq (1 - \textit{RI}(f, g, S) \cdot \min(\frac{P_g}{P_f}, \frac{P_f}{P_g})) \tag{8.8}$$

As $T_{\textit{apply}} \rightarrow 0.0$ the application of sharing reduction to g will approach $T_{\textit{sharing}}$. Notice in the common case where f and g are fissed by the same width, and f has single output and g has single input, the RHS of Equation 8.8 evaluates to 0, indicating to always apply sharing reduction for this case.

8.6.2 Sharing Reduction Applied to the Stream Graph

So far we have considered the application of the sharing reduction optimization to a single filter g in the stream graph. Now we will cover how to apply sharing reduction across all the filters of the stream graph for which it is appropriate. The goal is reduce the percentage of inter-core communication due to the sharing between fission products of a peeking filter to *approximately* $T_{\textit{sharing}}$. Sharing reduction may not achieve $T_{\textit{sharing}}$ because there may exist peeking filters that are to be fissed for which Equation 8.8 cannot be satisfied. For these filters, we do not apply the sharing reduction optimization.

The process of applying sharing reduction to the entire stream graph consists of determining which filters are appropriate and determining a steady-state multiplier by which to increase the graph. To determine the steady-state graph multiplier, the compiler calculates the percentage of

sharing over all of the filters which adhere to Equation 8.8. Let Φ denote the set of filters for which Equation 8.8 holds and that we seek to fiss:

$$c = \frac{1}{T_{sharing}} \cdot \frac{\sum_{g \in \Phi} P_g \cdot C(g)}{\sum_{g \in \Phi} M(S,g) \cdot o(W,g)} \tag{8.9}$$

Increasing the steady-state by c for all filters of the graph will reduce the total sharing for the filters of Φ to $T_{sharing}$.

8.7 Data Parallelization: General Fission and Sharing Reduction

General fission and sharing reduction are employed after JUDICIOUSFISSION (see Algorithm 2) calculates a fission width for each filter of the graph. The formulation of general fission in Section 8.4 includes two preconditions for filter g:

$$C(g) < (M(S,g)/P) \cdot o(W,g) \tag{8.10}$$

$$M(S,g) \mod P = 0 \tag{8.11}$$

The compiler is required to calculate a multiplication factor c for the entire graph that satisfies the preconditions for every filter. At the same time, c applies sharing reduction to the graph by increasing the graph such that Equation 8.9 is satisfied by the constant.

DATAPARALLELIZE of Algorithm 8 highlights the steps for data parallelizing the general graph representation of the application. This sequence is applied after the StreamIt graph has been coarsened and converted to a general graph. The algorithm first calculates the judicious fission widths for each filter of the general graph. For each peeking filter g that will be fissed, line 11 finds the producer of g that minimizes Equation 8.8. If this value is below T_{apply}, g is added to the sharing reduction calculation by adding g's shared items and g's total items to the running totals of each quantity. Sharing reduction is incorporated into the steady-state multiplier *minMult* in line 22 employing Equation 8.9. The algorithm applies Equation 8.10 to each peeking filter that will be fissed by maintain a running value of the minimum multiplier that enforce the precondition (line 18). The sharing reduction multiplicity (*minMult*) is required to be greater than the precondition multiplier. Because of the precondition of Equation 8.11, the final multiplication factor for increasing the steady-state must be a multiple of all the fission widths calculated by JUDICIOUSFISSION. The algorithm assures this by finding the least common multiple of all the P_f's greater than *minMult* (line 26). The steady-state multiplicity of the graph is then increased by multiplying all of the multiplicities by c. Finally, each filter f can be fissed by P_f using general fission.

Through empirical experimentation on FMRadio, Filterbank, and ChannelVocoder, we have settled on $T_{sharing} = .10$ and $T_{apply} = 0.05$. These constants are the sweet stop for the two architectures employed in the experimentation, being a good compromise between buffer size and inter-core communication. Figure 8-12 demonstrates the efficiency of general fission for the FMRadio benchmark when judiciously fissed to 4 cores. In the example, the coarsened version of FMRadio is given on the left. The steady-state of FMRadio is increased by 2560 so that the total sharing is under 10%. The last filter in the graph, the *Equalizer* has a large *dup*, and this had to be overcome for each core.

Algorithm 8 Exploit Data Parallelism in the General Graph for N Cores

DATAPARALLELIZE($G = (V, E), N, T_{\textit{sharing}}, T_{\textit{apply}}$)

preCondMult $\leftarrow 1$, *sharingItems* $\leftarrow 0$, *totalItems* $\leftarrow 0$
▷ For each filter in the graph find the fiss factor
for all $g \in V$ **do**
    $P_g \leftarrow$ JUDICIOUSFISSION(g, N)
end for

▷ Calculate multiplier for fission preconditions and for sharing reduction
for all $g \in V$ **do**
    ▷ If this is a peeking filter we are fissing:
    if $P_g > 1 \wedge C(g) > 0$ **then**
        ▷ Find the producer of g with the lowest value for Eq. 8.8
        $f \leftarrow \min_{f \in \textit{In}(g)}(1 - \textit{RI}(f, g, S) \cdot \min(\frac{P_g}{P_f}, \frac{P_f}{P_g}))$
        ▷ If the min value is below $T_{\textit{apply}}$, then record the sharing and total communication
        if $T_{\textit{apply}} \geq (1 - \textit{RI}(f, g, S) \cdot \min(\frac{P_g}{P_f}, \frac{P_f}{P_g}))$ **then**
            sharingItems $\leftarrow$ *sharingItems* $+ P_g \cdot C(g)$
            totalItems $\leftarrow$ *totalItems* $+ M(S, g) \cdot o(W, g)$
        end if
        ▷ Assure that fissing peeking filters adhere to Eq. 8.10
        preCondMult $\leftarrow \max($*preContMult*, $\frac{C(g) \cdot P_g}{M(S,g) \cdot o(W,g)})$
    end if
end for

▷ Find the multiplier for sharing reduction
minMult $\leftarrow T_{\textit{sharing}} \cdot \frac{\textit{sharingItems}}{\textit{totalItems}}$
▷ Make sure that multiplier assures all fissing peeking filters adhere to Eq. 8.10
minMult $\leftarrow \max($*minMult*, *preCondMult*$)$
▷ Find the least common multiple of the P_f's greater than *minMult* to adhere to Eq. 8.11
$c \leftarrow$LCM($\forall P_f | f \in V) >$ *minMult*

▷ Increase the steady-state by c
for all $f \in V$ **do**
    $M(S, f) \leftarrow c \cdot M(S, f)$
end for

▷ Apply general fission to all nodes in the graph
for all $f \in V$ **do**
    GENERALFISS(f, P_f)
end for

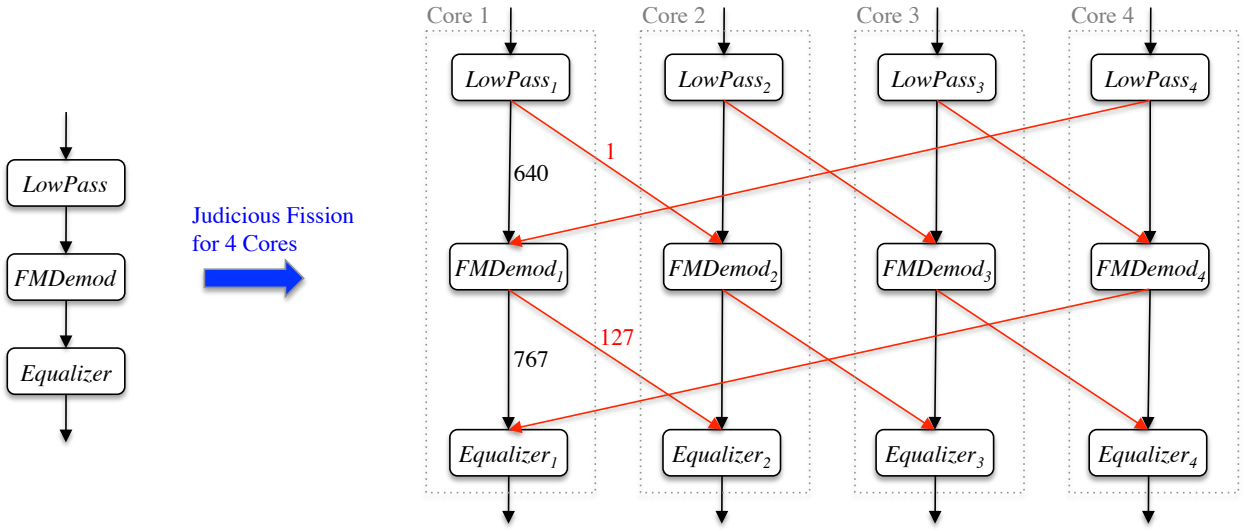


Figure 8-12: Judicious general fission as applied to the FMRadio benchmark. FMRadio's coarsened graph is given on the left. On the right is the judiciously fissed result for 4 cores employing general fission. The core assignment is also given. $C(f) = dup_f$ for all filters f of the coarsened graph. Edges for the first core are annotated with the number of items communicated per steady-state. The numbers are the same for the remaining cores. Only 9% of total non-I/O items are communicated inter-core.

Figure 8-13(a) gives the constant c calculated for ChannelVocoder, Filterbank, and FMRadio for 4, 16, 36, and 64 cores with: $T_{sharing} = 0.10$ and $T_{apply} = 0.05$. The factor is larger for FMRadio because one filter has $C(f) \gg o(S, f)$. The multiplication factor affects both latency and buffer sizes adversely. The application designer will have to decide if the latency of these techniques can be borne given the application criteria. The total buffering requirement is increased when the steady-state is increased. However, since we are then fissing, the buffer is divided amongst the fission products, and the *per-core* buffering requirement is unaffected by the increase. For example, FMRadio, has a per-core 18 KB buffering requirement across all configurations (4, 16, 36, and 64 cores). This requirement fits in the per-core L2 size of 64 KB for the Tile64.

Figure 8-13(b) compares the steady-state with and without sharing reduction for a 64-core mapping. For ChannelVocoder, sharing reduction has no effect because most of the peeking filters do not satisfy $T_{apply} = 0.05$. For the peeking filters that do, the steady-state multiplier required for legal general fission for the graph is enough to assure $T_{sharing}$ is met. Even though sharing reduction has no effect for ChannelVocoder, general fission avoids the 38% of total items that were unnecessary duplicated by dupdec.

For FMRadio and Filterbank, sharing reduction leads to significant decreases in the percentage of total items communicated inter-core for each steady-state. The buffer requirement is increased an average of 5.2x for these benchmarks. The total number of words communicated inter-core during each steady-state is the same, with and without sharing reduction. However, the steady-state is greater in the sharing reduction case, thus producing more outputs.

Benchmark	Steady-State Multiplication Factor			
	4 Cores	16 Cores	36 Cores	64 Cores
ChannelVocoder	8	32	72	192
FMRadio	2560	10240	23040	40960
Filterbank	1	318	636	1272

(a)

	64 Cores						
Benchmark	No Sharing Reduction				Sharing Reduction		
	Steady-State Multiplier	Buffering Per Core	% Comm. Inter-Core	Total Words Inter-Core	Steady-State Multipler	Buffering Per Core	% Comm. Inter-Core
ChannelVocoder	192	21.1 KB	48.0%	159648	192	21.1 KB	48.0%
FMRadio	8192	4.4 KB	33.3%	8192	40960	18.5 KB	9.1%
Filterbank	128	3.4 KB	39.8%	16256	1272	21.4 KB	6.2%

(b)

Figure 8-13: Communication, multiplier and buffering characteristics for benchmarks: (a) gives the steady-state multipliers calculated for sharing reduction, (b) compares the steady-state with and without sharing reduction.

8.8 Evaluation: Tilera TILE64

This section evaluates fission of peeking filters on the general stream graph for the Tilera TILE64 64-core processor. This section also presents results for the combined techniques of this thesis for the TILE64. For the evaluation, the benchmarks are mapped by applying coarse-grained data parallelism followed by coarse-grained software pipelining (CGDTP + CGSP). The scheduling algorithms and code generation strategies are covered in Chapters 6 and 7, respectively. The specifics of code generation, communication and synchronization for the TILE64 for CGDTP are detailed in Section 6.5. Section 6.5 describes that for CGDTP on the TILE64 producers and consumers are double-buffered. Adding CGSP does not change the mapping significantly, the only difference is that a producer and consumer may be executing in the CGSP steady-state with a difference of more than 1 iteration of the original steady-state. Thus, we execute the prologue schedule, plus add additional buffering as described as needed by the CGSP mapping described in Chapter 7.

As Section 6.5 describes, the layout algorithm greedily assigns filters of a level sequentially. Each filter is assigned to the core that will minimize the number of input items that will be communicated inter-core. In the common case outlined in Section 8.5, each producer fission product f_i communicates to exactly two consumers, g_i and g_{i+1}. The communication between f_i and g_i equals 10 times the communication between f_i and g_{i+1} because $T_{sharing} = 0.1$. The layout algorithm will map g_i to the core that f_i was mapped to, minimizing inter-core communication. The filters of the first level of the graph f_1 through f_n are mapped to the cores in a "snake" fashion, such that f_i is neighbors with f_{i+1}, and f_1 is neighbors with f_N. This predisposes the layout to map common-case inter-core communication to neighboring tiles.

8.8.1 All Benchmarks

Figure 8-14 gives the parallelization speedups for multiple configurations of the TILE64 for all benchmarks of the StreamIt Core benchmark suite. The benchmarks are compiled using CGDTP + CGSP. CGDTP employs optimized fission on the general graph with sharing reduction. Across

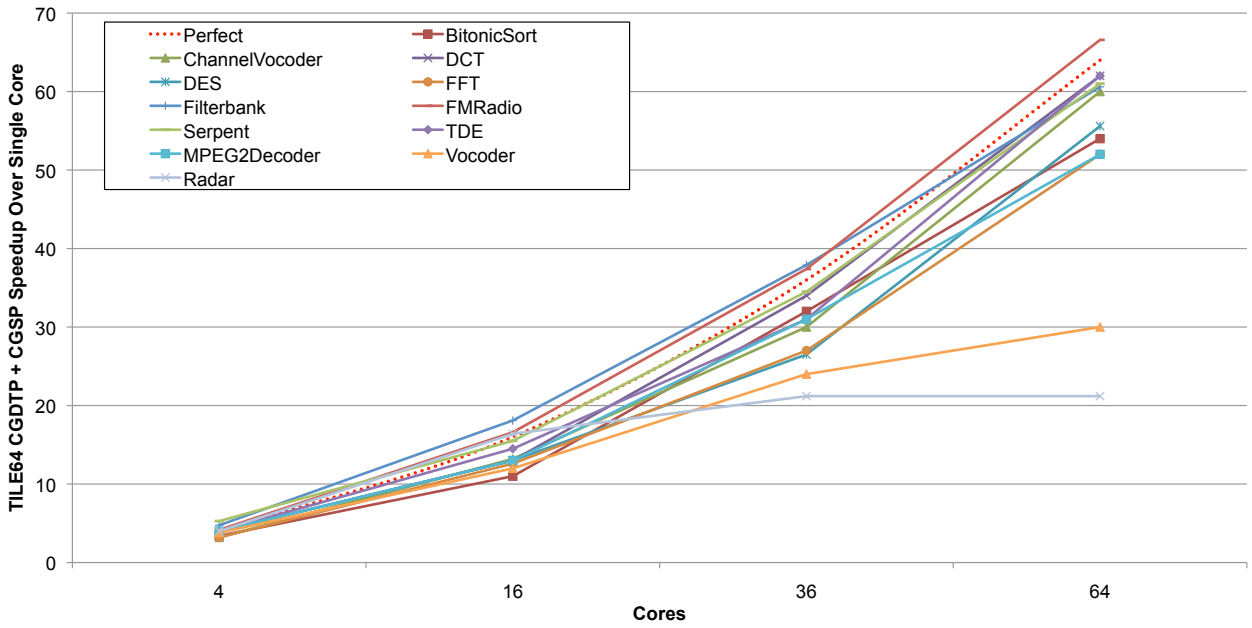


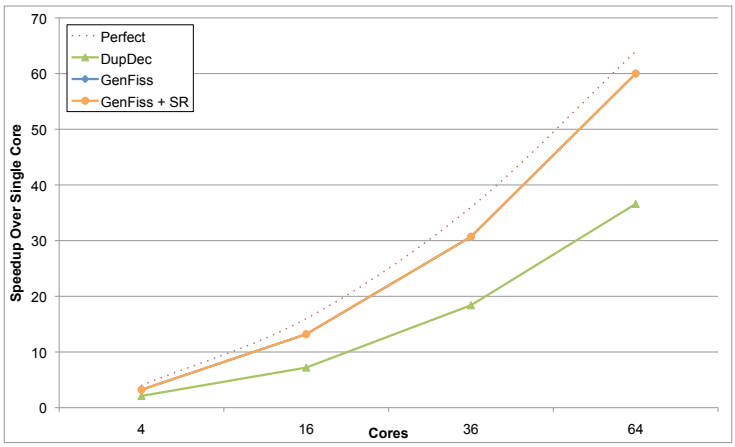
Figure 8-14: Parallelization speedup for the benchmarks of the StreamIt Core benchmark suite targeting multiple configurations of the TILE64. The benchmarks are compiled using CGDTP + CGSP. CGDTP employs optimized fission on the general graph with sharing reduction.

the benchmark suite, the combined techniques achieve a 51.7x speedup for 64 cores as compared to the performance of a single TILE64 core. Only considering the 9 completely stateless benchmarks plus MPEG2Decoder (which includes little state), a 59x 64-core parallelization speedup is achieved. The combined techniques effectively (i) leverage data parallelism, (ii) remove the inter-core communication requirement of data parallelism when data distribution is present, (iii) employ task parallelism to reduce the span of data parallelism of peeking filter to reduce the inter-core communication requirement, and (iv) employ general fission to further reduce the inter-core communication requirement of fission of peeking filters by reducing sharing and more accurately distributing data.

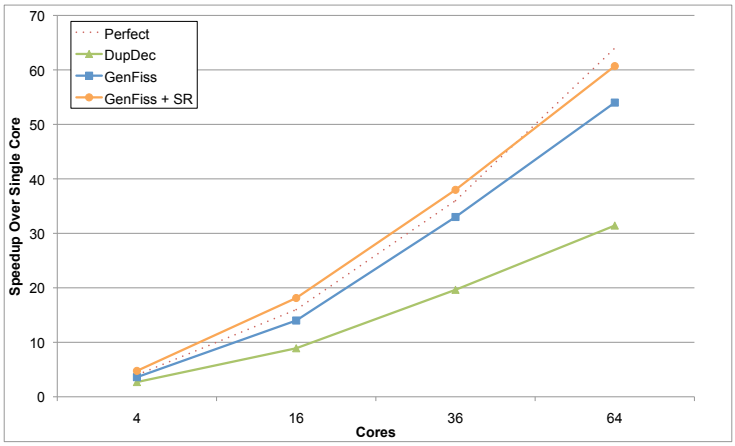
For the two benchmarks that include significant amounts of state, the limits to scalable parallelism is reached before 64 cores. This is because the applications do not include enough stateful pipeline parallelism to scale to 64 cores. This occurs because either the load is concentrated in a single stateful filter that has greater workload than all other filters, or there are just not enough stateful filters to spread across 64 cores. The former is true in the case of Vocoder, and the latter is true for Radar.

8.8.2 Stateless Benchmarks with Peeking Filters

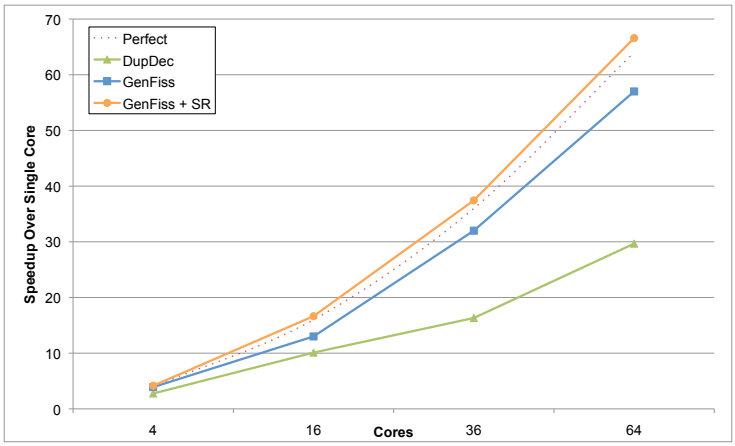
The techniques introduced in this chapter greatly reduce the inter-core communication requirement for the fission of peeking filters as compared to previous techniques. We evaluate the effectiveness of general fission for peeking filters by comparing them to fission of peeking filters in the StreamIt graph (see Section 6.3.3). Sharing reduction is also toggled in order to quantify its effect. In the case of fission of peeking filters in the StreamIt graph, sharing between fission products is



(a) ChannelVocoder



(b) Filterbank



(c) FMRadio

Figure 8-15: Experimental evaluation for duplication and decimation (dupdec), general fission (GenFiss), and general fission with sharing reduction (GenFiss + SR) in the context of the Tilera TILE64 processor utilizing 4, 16, 36, and 64 cores.

achieved via duplication of input to all products and decimation (dupdec). In our comparison for the TILE64, the compiler flow is the exactly the same for dupdec and general fission except that for dupdec, JUDICIOUSFISSION employs fission on the StreamIt graph with the conversion from StreamIt graph to general graph occurring after JUDICIOUSFISSION.

The evaluation is performed on three benchmarks: ChannelVocoder, Filterbank, and FMRadio. The entire benchmark suite is not included because the general fission techniques only affect benchmarks with peeking. We did not include Vocoder because it includes a large amount of stateful computation, and most of it's peeking filters are stateful, and thus not fissed.

Figure 8-15 gives the results of the comparison utilizing the three benchmarks targeting 4, 16, 36, and 64 cores of the TILE64. It is clear from the graphs that general fission plus sharing reduction enable scalable parallelization for the TILE64 architecture. General fission with sharing reduction outperforms dupdec by an average of 1.9x for the three benchmarks when targeting 64 cores. The average 64-core speedup over single core is 61x for the general fission plus sharing reduction for these three benchmarks.

FMRadio experiences the most significant gain from general fission plus sharing reduction over dupdec. FMRadio has the lowest computation to communication ratio of the 3 benchmarks. Furthermore, each filter of the coarsened graph is fissed by the number of cores targeted. For 64 cores, each filter is fissed 64 ways. Dupdec must perform a global all-to-all communication involving all 64 cores between each level of the graph! Conversely, general fission results in near-neighbor communication in a "snake" across the 64 cores. Dupdec achieves only a 30x speedup for 64 cores while general fission plus sharing reduction achieves a 67x speedup for 64 cores.

The coarsened graph for ChannelVocoder is given in Figure 6-10(a). When targeting 64 cores, the single filter of the first level (we do not include the input filter as a level) is judiciously fissed 64 ways, while the 16 of the 17 filters of the level 2 levels are judiciously fissed 3 ways. Level 3 is the same as level 2. Thus level 2 and 3 do not possess 64-way parallelism (instead they possess 49-way parallelism) because of asymmetries of the graph with the target. We achieve a 60x speedup for general fission over a single core because the bottleneck for the application is the single filter of the first level. Since the width of many of the fission applications is 3, dupdec is duplicating this input data to groups of 3 filters. Thus the speedup for general fission over dupdec for ChannelVocoder (1.62x) is not as great as compared to FMRadio when dupdec duplicates to all 64 cores. Filterbank is similar, the width of fission is 4 for all filters when targeting 64 cores.

Sharing reduction is required to achieve scalable speedups for both FMRadio and Filterbank. For FMRadio, sharing reductions leads to a 17% speedup increase for 64 cores. This because sharing reduction significantly reduces the number of remote write store instructions required per output. This affects FMRadio because of its low computation to communication ratio. Furthermore, the communication in FMRadio is completely hidden by computation once sharing reduction is enabled. Sharing reduction sees a modest 12% increase on Filterbank, as Filterbank has a larger computation to communication ratio.

8.9 Evaluation: Xeon 16 Core SMP System

This section demonstrates that the techniques covered in this chapter are essential in order to achieve scalable parallel performance when targeting a commodity symmetric multiprocessor architecture (SMP). This section presents results for the combined techniques of this thesis for the

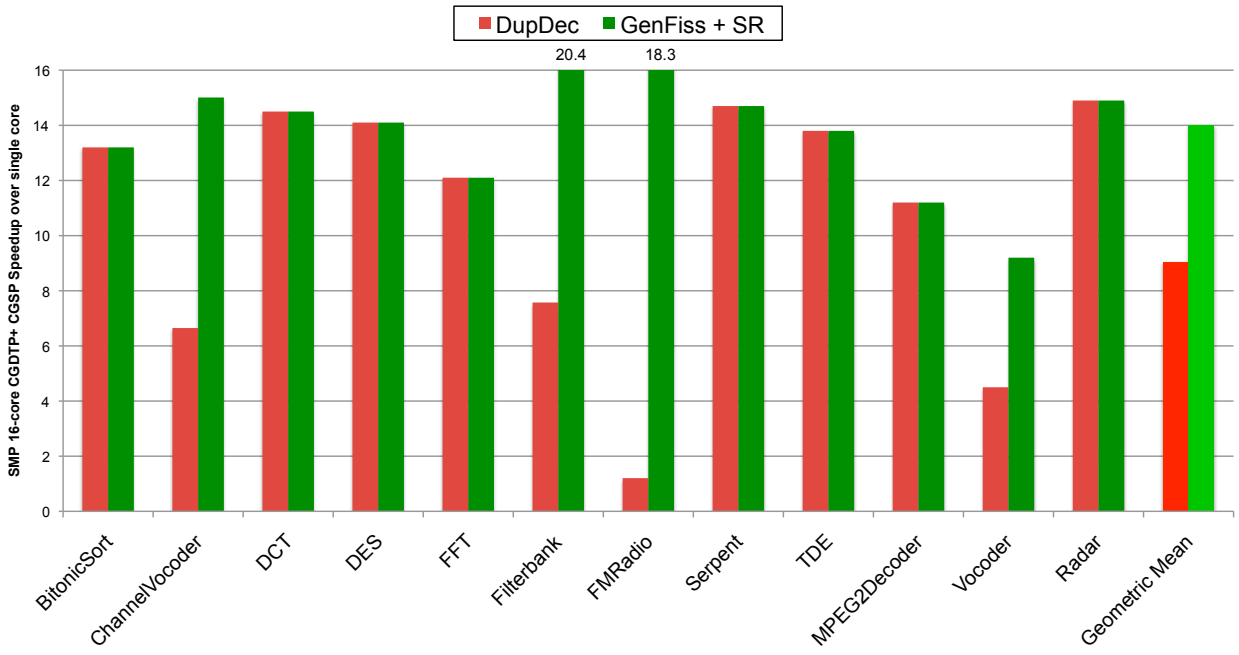


Figure 8-16: Evaluation for dupdec versus general fission with sharing reduction for the 16-core SMP architecture.

Xeon system. For the evaluation, the benchmarks are mapped by applying coarse-grained data parallelism followed by coarse-grained software pipelining (CGDTP + CGSP). The SMP path of our compiler is covered in more detail in Section 6.6 and [Tan09]. The changes from these descriptions mirror the changes outlined for the TILE64 in the previous section, namely additional buffering may be necessary to achieve the CGSP steady-state.

Figure 8-16 gives the 16-core speedup comparison for dupdec versus general fission with sharing reduction for our target SMP architecture. Across the entire suite of benchmarks the combined techniques of this dissertation achieve a 14.0x parallelization speedup for 16 cores. Combining CGSP to CGDTP enables Vocoder and Radar to achieve 9.2x and 14.9x respectively. Without CGSP Vocoder and Radar achieve 2.5x and 7.6x, respectively (see Section 6.6). For the SMP architecture, 14.0x parallelization speedup is a remarkable achievement that demonstrates our techniques are robust to varying benchmarks and portable to this type of architecture.

Concentrating on the benefit of general fission, the average speedup increase for general fission with sharing reduction over dupdec for peeking benchmarks (ChannelVocoder, FilterBank, FMRadio, and Vocoder) is 5.5x. FMRadio again sees the largest speedup increase in the comparison at 13.0x. The reasons for this large speedup are similar as given in the previous section. However, the SMP communication mechanism is not as efficient as the TILE64, thus general fission gives a greater speedup.

It in interesting that the dupdec 16-core speedup over single core for each of these three benchmarks is greater for the TILE64 versus the SMP. Since, each is normalized to it's own single core performance, this is mostly likely due to the RSP paradigm of the TILE64 being more efficient than the SMP's shared-memory communication for streaming applications. When general fission is applied, the communication requirement of the mapping is reduced, and in the SMP case, we do

not overwhelm the cache-coherence mechanism, and we achieve super-linear speedups. General fission allows the compiler to achieve over a 15x 16-core speedup for each of the three benchmarks targeting the SMP architecture. Without general fission, the SMP compiler would not achieve scalable parallel mappings for these benchmarks.

8.10 Related Work

Printz's "signal flow graphs" includes nodes that performed a sliding window [Pri91]. Printz presents an implementation strategy translating the sharing requirement of data parallelization of peeking filters into communication. However, his approach does not alter the steady-state of the graph to reduce sharing to neighboring fission products, and it does not further alter the steady-state to reduce sharing to under a given threshold. Instead, Printz's technique parallelizes the communication required by sharing via a pipelined sequence of transfers between neighboring cores on the iWarp machine. Our techniques trade latency for reduced inter-core communication, a tradeoff that Printz did not explore. Finally, Printz did not implement his strategy; instead his evaluation relied on an model of a parallel architecture.

In the Warp project, the AL language [Tse89, Tse90] had a window operation for use with arrays. The AL compiler targeted the Warp machine and translated loops into systolic computation. The AL compiler did alter the blocking of distributed loop iterations to reduce the sharing requirement for the sharing of the sliding window among iterations of the loop. However, this was in the context of an imperative language with parallel arrays and considered a different application class: matrix computation.

The Brook language requires a programmer to explicitly represent the communication of sliding windows by specifying that a filter reads overlapping portions of the input. Liao et al. map Brook to multicore processors by leveraging the affine partitioning model [LDWL06]. However, they do not alter the steady-state to reduce the number of items shared between the data parallel products of sliding window filters. Also, it is unclear if their techniques guarantee that an item is shared by at most 2 cores.

Other languages have included the notion of a sliding window. The ECOS graphs language allows actors to specify how many items are read but not consumed [HMWZ92]; the Signal language allows access to the window of values that a variable assumed in the past [GBBG86]; and the SA-C language contains a two-dimensional windowing operation [DBH$^+$01]. However, to the best of our knowledge, translation systems for these languages do not utilize sliding windows to improve parallelism and reduce inter-core communication.

8.11 Chapter Summary

In this chapter we cover methods that further reduce the inter-core communication requirement of fission of peeking filters. First, an algorithm for removing synchronization nodes in a general stream graph is covered. This algorithm enables us to formulate fission of a peeking filter on the general graph. The general graph formulation of fission leverages the more expressive output distribution of the representation (as compared to StreamIt graphs) to precisely share duplicated items between products of a fissed peeking filter. This precision reduces inter-core communication as compared to the traditional technique for fission of peeking filters. The next technique actively

reduces the amount of sharing that occurs between products of a fissed peeking filter, by altering the steady-state schedule of the graph. CGDTP is altered to use the new techniques. Across the StreamIt Core benchmark suite, CGDTP employing these new techniques combined with CGSP attain a 14.0x mean parallelization speedup for the 16 core SMP and a 51.7x mean parallelization speedup for the 64 core TILE64.

The important properties of stream programs that enable the techniques in this chapter include:

1. The peeking idiom exposes sliding-window computation to the compiler, allowing it to aggressively transform the peeking filters to exploit data parallel with little inter-core communication.

2. Exposing producer-consumer relationships between filters. This enables us to precisely manage communication.

3. Exposing the outer loop around the entire stream graph. This enables the techniques to alter the steady-state so that sharing can be reduced between fission products.

4. Static rates and data distributions of filters enables the compiler to schedule the steady-state of the application, and precisely route data items between producers and consumers.

Chapter 9

Conclusions

We have evaluated the thesis that incorporating streaming abstractions into the execution model of a high-level language enables the compiler to effectively manage parallelism. Once appropriate abstractions are incorporated, compilation techniques must be informed by the unique properties of the streaming application domain and the target architecture class. Furthermore, the techniques must keep inter-core communication to a minimum and be governed by the limitations of what can be calculated statically. In this dissertation we demonstrate that our combined techniques achieve robust and uniform parallelization scaling for streaming applications when targeting diverse multi-core architectures. Our key results are the mean parallelization speedups across the StreamIt Core benchmark suite for three architectures:

			Speedup over Single Core for StreamIt Core Benchmarks		
Architecture	**Memory Model**	**Cores**	**Mean**	**Min**	**Std. Dev.**
Raw	Distributed Memory	16	14.9x	12.6x	1.8
Xeon E7350	Shared Memory	16	14.0x	9.2x	2.8
Tilera TILE64	Distributed-Shared Memory	64	51.7x	21.2x	13.1

The dissertation covers multiple techniques for solving the scheduling problem for stream graphs. The scheduling problem is to maximize the throughput of a stream graph by effectively allocating filters to cores. Each technique occupies a position on the space/time multiplexing continuum. Hardware pipelining and task parallelism combines filters of the original stream graph into larger filters in a partitioning step. Each of the filters of the partitioned graph is assigned to its own core. The mapping exploits pipeline and task parallelism. This mapping sits at the space-multiplexing end of the space/time multiplexing continuum. Hardware pipelining stands alone and does not interact nor complement other techniques. We do not present the performance results of hardware pipelining for the TILE64 nor the Xeon system because little would be gleaned from them over the Raw results.

Coarse-grained data and task parallelism (CGDTP) sits very near the time-multiplexing end of the space/time-multiplexing continuum. First, filters are fused to remove any inter-core communication caused by data distribution between producers and consumers. Next, data parallelism is leveraged by fissing filters at appropriate spans such that each task-parallel level of the graph is load-balanced and occupies all the cores of the processor. The levels of the graph are executed in a time-multiplexed fashion. Task parallelism is retained in a level when it can reduce the inter-core communication requirement of the data parallelization of peeking filters. Task parallelism is har-

nessed within a level via space-multiplexing. We describe the techniques as "time-multiplexing" because most of the parallelism that exists in the transformed graph is data parallelism, and task parallelism is included only to reduce the communication requirements, not as means of load-balancing.

Coarse-grained software pipelining (CGSP) complements CGDTP to parallelize filters that cannot be data-parallelized. This technique is akin to traditional software pipelining of instructions nested in a loop, though here it is applied to coarse-grained filters. CGSP allows a space-multiplexing strategy to schedule the steady-state without regard for the data dependences of the stream graph. CGSP, on its own, exploits pipeline and task parallelism in a space-multiplexed fashion. However, we combine the technique with CGDTP to incorporate all three forms of parallelism in the mapping. First the stream graph is transformed by CGDTP to leverage data parallelism, then stateful filters (if present) are parallelized by CGSP. The combination of techniques sits somewhere in the middle of the space/time continuum

This dissertation is an important step towards shifting the burden of scalable parallel performance for the streaming application domain from the programmer to the translation system. This shift will bring to parallel programming the important high-level programming properties of scalability, portability, malleability and composability. This dissertation makes the following contributions:

1. We develop an analytical model of an important class of stream programs targeting multicore architectures. The analytical model considers data parallel streaming programs, comparing the scalability prospects of two mapping strategies: time-multiplexed data parallelism and space-multiplexed data parallel. Analytical throughput equations are developed for each, extracting and comparing the key differences in the models in terms of load-balancing effectiveness, communication, and memory requirement. The model provides evidence that optimizing the communication of the data-parallelization of peeking filters is important to enabling scalable parallel mappings. Evidence is offered that shows pipeline parallelism will become increasingly important as multicore continue to scale by adding cores. We develop asymptotic communication bounds for the two strategies given a low-latency, near-neighbor mesh network, concluding that the strategies are asymptotically equal.

2. We present a detailed description and analysis of the StreamIt Core benchmark suite. We provide an analysis of the suite of 12 programs from the perspectives of composition, computation, communication, load-balancing, and parallelism.

3. We develop the first fully-automated infrastructure for exploiting data and task parallelism in streaming programs that is informed by the unique characteristics of the streaming domain and multicore architectures. The technique first automatically matches the granularity of the stream graph to the target without obscuring data-parallelism. The programmer is not required to think about parallelism nor provide any additional information to enable parallelization. When data-parallelism is exploited, the task parallelism of the graph is retained so that the span of the introduced data parallelism can be minimized. This serves to reduce communication and synchronization. These techniques provide a 12.2x mean speedup on a 16-core machine for the StreamIt Core benchmark suite.

4. We develop the first fully-automated framework for exploiting pipeline parallelism via coarse-grained software pipelining in stream programs. Data parallel techniques cannot explicitly parallelize filters with iteration-carried state. Leveraging the outer scheduling loop of the stream program, we apply software pipelining techniques at a coarse granularity to schedule stateful filters in parallel. Our techniques partition the graph to minimize the critical path workload of a core, while reducing inter-core communication. For the stateful benchmarks of the StreamIt Core benchmark suite targeting 16 cores, adding coarse-grained software pipelining to data and task parallel techniques improves throughput by 3.2x.

5. We develop a novel framework for reducing the inter-core communication requirement for data-parallelizing filters with sliding window operations. Filters that operate on a sliding window of their input must remember input items between iterations. Data-parallelization of these filters requires sharing of input across data-parallel units. We introduce new techniques that reduce the sharing requirement by altering the steady-state schedule and precisely routing communication. When combined with the mapping techniques above, this data-parallelization technique enables scalable parallel mappings for traditional shared-memory architectures, achieving a mean 14x speedup when targeting 16 cores for our Xeon system. The techniques also enable scalable parallel mappings to aggressively parallel, distributed shared-memory architectures, achieving a mean 52x speedup when targeting the 64 cores of the TILE64.

9.1 Overall Analysis and Lessons Learned

This section collects the important insights gleaned from the research presented in this dissertation.

9.1.1 Execution Model

The execution model developed and employed in this dissertation has its foundations in the synchronous dataflow model. We demonstrate that a high-level streaming language can be transformed into our execution model. The following proprieties of the model are important to enabling our parallelization and mapping techniques:

1. **Coarse-grained parallelism is easy to discover.** Stream programs offer three types of coarse-grained parallelism: task, data, and pipeline parallelism. Each type of parallelism does not require heroic compiler analysis nor programmer annotations to extract from the program. Task parallelism exists between pairs of filters that are on different parallel branches of the original stream graph, as written by the programmer. Pipeline parallelism exists between producers and consumers that are directly connected in the stream graph. Data parallelism exists at any filter that has no dependencies between one execution and the next. Each type of parallelism is utilized in our overall solution. Although parallelism is easy to discovery, care must be exercised when exploiting the parallelism; communication must be minimized and a load-balanced mapping must be produced.

2. **Communication and synchronization are explicit.** Communication between coarse-grained units is explicit and regular in our execution model. Communication between filters is represented via the channels (edges) in the graph. Communication patterns are described by fine-grained, statically-determinable schedules. Synchronization in the model exists only between

producers and consumers. These properties enable the compiler to have complete knowledge of the parallel behavior of a streaming application. The compiler can effectively manage communication and synchronization. Furthermore, the communication and synchronization model is easy for the programmer to understand and is a natural fit for representing streaming applications.

3. **Regular and repeating computation.** The computation of a filter is described by an atomic function that is repeated during the steady-state of the application. In the streaming domain, this function rarely contains data dependent computation. These properties enable techniques that exploit data parallelism to achieve effective load-balancing. Furthermore, these properties enable our static techniques to compare workload between filters when incorporating task and pipeline parallelism into the mapping.

4. **Continuous stream of inputs entering application.** Inputs are continuously processed by a stream graph. The graph describes the steady-state behavior of the application, however, this steady-state executes for many iterations. This property enables the compiler to aggressively transform the application, scheduling computation and communication from multiple steady-states in parallel. We take advantage of this property in all of our techniques.

5. **Two common sources of stateful computation are explicitly exposed.** Stateful computation prevents data parallelization, and thus other techniques must be employed to parallelize stateful filters. These techniques are difficult to load-balance and represent a limitation to scalability because parallelism is limited to the programmer-defined granularity of the graph. The execution model represents and exposes two common sources of state: sliding windows and initialization behavior. When sources of stateful computation are exposed to the compiler, the compiler can reason about them, and apply data parallelization techniques that distribute the state across the data-parallelized units.

   Peeking is a useful construct that allows the programmer to represent a variety of sliding window computations. Without including the peeking idiom in StreamIt, sliding windows would have to be implemented in a stateful manner where the programmer explicitly introduces state to remember the dequeued item across work function firings. The implementation could be very difficult for the compiler to analyze and data parallelize. The prework function allows a filter to have different behavior on its first invocation. The most common use of prework is for implementing a delay; on the first execution, the filter pushes N placeholder items, while on subsequent executions it acts like an identity filter. Without prework, the delayed items would need to be buffered internally to the filter, introducing state into the computation.

High-level languages that can be directly translated into our execution model, such as StreamIt, are natural representations for applications in the streaming domain. These languages increase programmer productivity in expressing the application, while simultaneously preserving the rich static properties of the streaming domain. This dissertation develops techniques that effectively map programs written in these languages to multicore architectures.

9.1.2 Benchmarks

This dissertation includes a detailed analysis of the StreamIt benchmark Core suite. Our experience with this suite of 12 real-world application informed our techniques for mapping and paral-

lelization. The following highlights the properties of the StreamIt Core benchmark suite and the application domain, and how these properties informed our techniques:

1. **The programmer-conceived versions of the stream graphs have limited parallelism and thus limited scalability beyond 16 cores.** The original versions of the stream graphs include only task and pipeline parallelism. It is not sufficient to rely only on these two types of parallelism when computing a scalable mapping. Thus the compiler must aggressively transform the application by leveraging data parallelism and pipeline parallelism across steady-states.

2. **Data-parallel, stateless filters are the norm.** Across our benchmark suite, 95% of static filter implementations are stateless, while 85% of filter instantiations are stateless. Due to this, and due to the positive qualities of data parallelism (namely load-balancing), mapping techniques should prioritize data parallel techniques.

3. **Stateful filters, though not common, are required to represent some algorithms.** The stateful computation in two of our applications is inherent to the underlying algorithm at the represented granularity. For the sake of expressiveness, a way to represented stateful computations should be included in the execution model. Mapping and scheduling techniques, in order to be robust across applications, must parallelize stateful components. This consideration motivated our coarse-grained software pipelining technique.

4. **Most benchmarks include complex fine-grained data distribution between producers and consumers.** 10 of our 12 benchmarks include a splitjoin construct that describes scatter/gather communication and synchronization. When data-parallelizing, scatter/gather operations may require inter-core communication. Our techniques explicitly remove the inter-core communication requirement for splitjoin communication when peeking is not present. This is accomplished by coarsening the stream graph such that producers and consumers are mapped to the same core when data parallelizing.

5. **Peeking is useful to represent sliding-window computation, thus preventing stateful computation.** Peeking is present in 4 of the 12 benchmarks. The peeking filters often implement a sliding widow operation, e.g., FIR LowPass and HighPass filters in ChannelVocoder, FilterBank, and FMRadio. Furthermore, for these benchmarks, the peeking filters represent a significant percentage of the workload of the benchmark.

9.1.3 Parallelization Techniques

This dissertation develops multiple parallelization techniques that together calculate an effective mapping for a given streaming application. The techniques are informed by the properties of the execution model and the application domain detailed above. Managing parallelism requires both exposing parallelism and optimizing the performance of the parallel mapping. This later task can be broken down further into the following concerns: (i) matching granularity, (ii) load-balancing, (iii) synchronization, and (iv) communication. This section highlights insights and conclusions regarding these concerns.

1. **Data parallelism is load-balanced.** Data parallelism is the preferred form of parallelism in the streaming domain. We note above that data parallelism is readily available. For mapping, data

parallelism is favored because it produces load-balanced mappings to multicore architectures. When data-parallelizing a single filter, the product filters are executing the same code, but with different inputs. As filters rarely contain data-dependent computation, the product filters of data-parallelization are load-balanced.

2. **The communication requirement of pipeline parallelism is identical to data parallelism for simple pipelines.** Given a simple pipeline, there are two broad strategies for mapping the filters to cores: space-multiplexing and time-multiplexing. Time-multiplexing entails data parallelism, where each filter is spread across all cores. Space-multiplexing entails pipeline parallelism, where each filter is assigned to it's own core. It seems counter-intuitive, but the communication requirement of the strategies are equivalent when normalized by the number of output items. Although this analysis was conducted on a mesh network with duplication handled via the network, the finding should inform decisions across architectures. Given data parallelism's load-balancing qualities, this equivalence is additional evidence that data parallelism should be favored over pipeline parallelism.

3. **Static work estimations are often inaccurate.** Even though computation in our execution model is regular, it is difficult to accurately compare workload between filters at compile time. It is our experience that optimal strategies for partitioning and mapping rely heavily on work estimation and tend to over-fit. Optimal strategies fare poorly at runtime because of this over-reliance on accurate work estimation. We instead present heuristic techniques that only rely on workload comparisons in circumstances that tend to have reduced demands on accuracy or when necessary to parallelize state. For example, we incorporate task parallelism to reduce the sharing requirement of data parallelization peeking filters. Task parallelism tends to be load-balanced, as filters that are task parallel are often performing the same computation but with different parameters. Thus, in this case, there is less demand for accuracy for static estimation.

4. **Pipeline parallelism is worth considering.** Our techniques are rather unique in their leveraging of pipeline parallelism. Exploiting data parallelism is the norm when mapping streaming programs to parallel architectures. We have demonstrated that if there exists stateful computation, leveraging pipeline parallelism is required to achieve scalable parallel performance. Furthermore, if a load-balanced pipeline can be calculated, the partitioning of instructions and data of pipeline parallelism (as compared to the duplication of instructions and data of data parallelism) often leads to super-linear parallelization speedups. Reduced data and instruction footprints are important when targeting an embedded multicore and as larger applications are considered.

5. **Contiguous partitioning hampers the load-balancing of a hardware-pipelined mapping.** Leveraging pipeline parallelism is an effective strategy for parallelizing stateful computation. However, hardware pipelining, the traditional approach, requires that a contiguous set of filters be mapped to give core. We demonstrate that this approach hinders load-balancing. This insight led to the development of our coarse-grained software-pipelining technique for parallelizing stateful filters. This approach does not require contiguous partitioning of filters, and thus has more flexibility to arrive at a load-balancing partitioning of filters when compared to hardware-pipelining.

6. **For data parallelism, the two sources of inter-core communication are peeking and data distribution.** Our techniques explicitly reduce the inter-core communication requirement engendered by data parallelism by (i) first coarsening the filters of the stream graph via fusion to keep communicating filters on the same core, (ii) incorporating task parallelism to reduce the span of data parallelization of peeking filters, and (iii) directly reducing the sharing requirement of the fission of peeking filters by altering the steady-state to coarsen data-parallelized filters.

The techniques described in this thesis require static rate programs, i.e., programs in which the input and output rates of the filters are statically-determinable. We appreciate that some streaming algorithms cannot be represented given this restriction. Considering the complete picture, we envision that our transformations will be embedded in a more flexible and hybrid programming model and translation system. Employing an analogy to traditional optimizing compilers, we consider a static rate graph as akin to a function or method: a manageable unit of optimization that the compiler can reason about, and that can be stitched together to form a complete program. Furthering the analogy, we consider the filter as the basic block of parallel computation: aggressive transformations are performed at this level, and it is straightforward to summarize the effects of a filter.

The StreamIt Core benchmark suite includes 12 static applications, however many more applications have been developed in StreamIt. Out of 29 real-world applications in the full StreamIt benchmark suite, 24 are completely static-rate, and out of those with any dynamic rates, only 3% of the user-defined filters have a dynamic rate [Thi09]. Thus, the compiler can apply our techniques on static-rate subgraphs of an application that includes dynamic rates. A runtime system can be employed to orchestrate execution at the dynamic-rate boundaries between the static-rate subgraphs.

The goal of our research is not to promote StreamIt as a general-purpose language for multicore architectures. StreamIt will probably not displace imperative languages in the future. However, we do seek to influence language design and library support such that a stream programming solution does become standard. When developing the StreamIt language, we sought to design a language that increased both programmer productivity and compiler analyzability. With this dissertation, we now understand how our decisions affect the compiler's ability to perform automatic parallelization. Given the success of the techniques of this dissertation, many features of the StreamIt language are ready to be included in a mainstream language. As the streaming application domain increases in importance (e.g., audio, video, communications, image, compression, etc.) and multicores continue to scale, we anticipate that stream programming will gain acceptance. Stream programming will become another essential tool in the toolbox of the programmer, increasing her productivity when developing a streaming application and liberating her from having to deal with the nasty issues of parallelism.

Bibliography

[ABK] R. Anderson, E. Biham, and L. Knudsen, *Serpent: A proposal for the advanced encryption standard*, `http://www.cl.cam.ac.uk/~rja14/Papers/ventura.pdf`.

[Agh85] G. Agha, *Actors: A model of concurrent computation in distributed systems*, Ph.D. Thesis, Massachusetts Insitute of Technology, 1985.

[AGK+05] S. Amarasinghe, M. I. Gordon, M. Karczmarek, J. Lin, D. Maze, R. M. Rabbah, and W. Thies, *Language and compiler design for streaming applications*, International Journal of Parallel Programming, 2005.

[Agr04] S. Agrawal, *Linear state-space analysis and optimization of StreamIt programs*, M.Eng. Thesis, Massachusetts Institute of Technology, 2004.

[AHH89] A. Agarwal, J. Hennessy, and M. Horowitz, *An analytical cache model*, ACM Trans. Comput. Syst. **7** (1989), no. 2, 184–215.

[ALP97] M. Adé, R. Lauwereins, and J. A. Peperstraete, *Data memory minimisation for synchronous data flow graphs emulated on DSP-FPGA targets*, Design Automation Conference (DAC), 1997.

[And99] G. R. Andrews, *Foundations of Parallel and Distributed Programming*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[And07] J. Andersson, *Modelling and evaluating the StreamBits language*, Master's thesis, Halmstad University, 2007.

[Arm07] J. Armstrong, *A history of Erlang*, Conference on History of Programming Languages (HOPL), ACM, 2007.

[ATA05] S. Agrawal, W. Thies, and S. Amarasinghe, *Optimizing stream programs using linear state space analysis*, International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), 2005.

[AVW93] J. Armstrong, R. Virding, and M. Williams, *Concurrent programming in Erlang*, Prentice Hall, 1993.

[AW77] E. A. Ashcroft and W. W. Wadge, *Lucid, a nonprocedural language with iteration*, Communications of the ACM **20** (1977), no. 7, 519–526.

[BBB+91] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, *The nas parallel benchmarks*.

[BCC+88] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J. Webb, *iWarp: An integrated solution to high-speed parallel computing*, Supercomputing, 1988.

[BCG+03] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski, *Streaming XPath processing with orward and backward axes*, International Conference on Data Engineering, 2003.

[BELP95] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, *Cyclo-static data flow*, International Conference on Acoustics, Speech, and Signal Processing (ICASSP), 1995.

[BFH+04] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, *Brook for GPUs: Stream Computing on Graphics Hardware*, SIGGRAPH, 2004.

[BG92] G. Berry and G. Gonthier, *The ESTEREL synchronous programming language: Design, semantics, implementation*, Science of Computer Programming **19** (1992), no. 2, 87–152.

[BG99] S. Bakshi and D. D. Gajski, *Partitioning and pipelining for performance-constrained hardware/software systems*, IEEE Trans. Very Large Scale Integr. Syst. **7** (1999), no. 4, 419–432.

[BHLM91] J. Buck, S. Ha, E. Lee, and D. Messerschmitt, *Multirate signal processing in Ptolemy*, International Conference on Acoustics, Speech, and Signal Processing (ICASSP), 1991.

[bit] *Bitonic sort*, `http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm`.

[BML95] S. Bhattacharyya, P. Murthy, and E. Lee, *Optimal parenthesization of lexical orderings for DSP block diagrams*, International Workshop on VLSI Signal Processing, 1995.

[BML96] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software synthesis from dataflow graphs*, Kluwer Academic Publishers, 1996.

[BSL96] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, *Self-timed resynchronization: A post-optimization for static multiprocessor schedules*, International Parallel Processing Symposium, 1996, pp. 199–205.

[CCD+08] Y.-K. Chen, J. Chhugani, P. Dubey, C. Hughes, D. Kim, S. Kumar, V. Lee, A. Nguyen, and M. Smelyanskiy, *Convergence of recognition, mining, and synthesis workloads and its implications*, Proceedings of the IEEE **96** (2008), no. 5, 790–807.

[CCH+00] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon, *Stream Computations Organized for Reconfigurable Execution (SCORE): Extended Abstract*, Proceedings of the Conference on Field Programmable Logic and Applications, 2000.

[CFKA90] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal, *Directory-based cache coherence in large-scale multiprocessors*, Computer **23** (1990), no. 6, 49–58.

[CG02] P. Cremonesi and C. Gennaro, *Integrated performance models for spmd applications and mimd architectures*, IEEE Trans. Parallel Distrib. Syst. **13** (2002), no. 12, 1320–1332.

[CGKS05] D. Chandra, F. Guo, S. Kim, and Y. Solihin, *Predicting inter-thread cache contention on a chip multi-processor architecture*, High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on (12-16 Feb. 2005), 340–351.

[CGT+05] J. Chen, M. I. Gordon, W. Thies, M. Zwicker, K. Pulli, and F. Durand, *A reconfigurable architecture for load-balanced rendering*, SIGGRAPH / Eurographics Workshop on Graphics Hardware, 2005.

[Cha01] R. Chandra, *Parallel programming in OpenMP*, Morgan Kaufmann, 2001.

[CHR+03] C. Consel, H. Hamdi, L. RÃl'veillÃÍre, L. Singaravelu, H. Yu, and C. Pu, *Spidle: A DSL Approach to Specifying Streaming Applications*, 2nd Int. Conf. on Generative Prog. and Component Engineering, 2003.

[CL94] M. E. Crovella and T. J. LeBlanc, *Parallel performance prediction using lost cycles analysis*, Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing (New York, NY, USA), ACM, 1994, pp. 600–609.

[CLC+09] Y. Choi, Y. Lin, N. Chong, S. Mahlke, and T. Mudge, *Stream compilation for real-time embedded multicore systems*, CGO '09: Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization (Washington, DC, USA), IEEE Computer Society, 2009, pp. 210–220.

[Cli81] W. D. Clinger, *Foundations of actor semantics*, Ph.D. Thesis, Massachusetts Insitute of Technology, 1981.

[Cop94] D. Coppersmith, *The data encryption standard (DES) and its strength against attacks*, IBM Journal of Research and Development **38** (1994), no. 3.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, *LUSTRE: A declarative language for real-time programming*, International Symposium on Principles of Programming Languages (POPL), 1987.

[CQ93] M. J. Clement and M. J. Quinn, *Analytical performance prediction on multicomputers*, Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing (New York, NY, USA), ACM, 1993, pp. 886–894.

[CR92] Y. Chung and S. Ranka, *Application and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed-memory multiprocessors*, Proceedings of Supercomputing '92 (1992).

[CRA09] P. M. Carpenter, A. Ramirez, and E. Ayguade, *Mapping stream programs onto heterogeneous multiprocessor systems*, CASES '09: Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems (New York, NY, USA), ACM, 2009, pp. 57–66.

[CRA10] P. M. Carpenter, A. Ramírez, and E. Ayguadé, *Buffer sizing for self-timed stream programs on heterogeneous distributed memory multiprocessors*, HiPEAC (Y. N. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell, eds.), Lecture Notes in Computer Science, vol. 5952, Springer, 2010, pp. 96–110.

[CS97] L.-F. Chao and E. H.-M. Sha, *Scheduling Data-Flow Graphs via Retiming and Unfolding*, IEEE Trans. on Parallel and Distributed Systems **08** (1997), no. 12.

[CSM93] H. Chen, B. Shirazi, and J. Marquis, *Performance evaluation of a novel scheduling method: Linear clustering with task duplication*, Proceedings of International Conference on Parallel and Distributed Systems (1993), 270–275.

[CV02] K. S. Chatha and R. Vemuri, *Hardware-Software partitioning and pipelined scheduling of transformative applications*, IEEE Trans. Very Large Scale Integr. Syst. **10** (2002), no. 3.

[DBH+01] B. A. Draper, A. P. W. Böhm, J. Hammes, W. A. Najjar, J. R. Beveridge, C. Ross, M. Chawathe, M. Desai, and J. Bins, *Compiling SA-C programs to FPGAs: Performance results*, International Workshop on Computer Vision Systems, 2001.

[DDM06] A. Das, W. J. Dally, and P. Mattson, *Compiling for stream processing*, PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques (New York, NY, USA), ACM, 2006, pp. 33–42.

[DFA05] W. Du, R. Ferreira, and G. Agrawal, *Compiler Support for Exploiting Coarse-Grained Pipelined Parallelism*, Supercomputing, 2005.

[Dix93] K. M. Dixit, *The SPEC benchmarks*, 149–163.

[Duc04] N. Duca, *Applications and execution of stream graphs*, Senior Undergraduate Thesis, Johns Hopkins University, 2004.

[EJL+03] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, *Taming heterogeneity – the Ptolemy approach*, Proceedings of the IEEE **91** (2003), no. 1, 127–144.

[Ell85] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, Ph.D. thesis, Yale University, 1985.

[EM87] E. and D. Messerschmitt, *Pipeline interleaved programmable DSP's: Synchronous data flow programming*, IEEE Trans. on Signal Processing **35** (1987), no. 9.

[ERAL95] H. El-Rewini, H. Ali, and T. Lewis, *Task scheduling in multiprocessing systems*, IEEE Computer (1995).

[GBBG86] P. L. Guernic, A. Benveniste, P. Bournai, and T. Gautier, *Signal – A data flow-oriented language for signal processing*, IEEE Transactions on Acoustics, Speech and Signal Processing **34** (1986), no. 2, 362–374.

[GBS05] M. Geilen, T. Basten, and S. Stuijk, *Minimising buffer requirements of synchronous dataflow graphs with model checking*, Design Automation Conference (DAC) (Anaheim, California, USA), ACM, 2005, pp. 819–824.

[GGD02] R. Govindarajan, G. R. Gao, and P. Desai, *Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks*, Journal of VLSI Signal Processing **31** (2002), no. 3, 207–229.

[GJ79] M. R. Garey and D. S. Johnson, *Computers and intractibility : A guide to the theory of np-completeness*, San Francisco : W. H. Freeman and Company, 1979.

[GO98] T. Gross and D. R. O'Halloron, *iWarp, Anatomy of a Parallel Computing System*, MIT Press, 1998.

[Gor02] M. Gordon, *A stream-aware compiler for communication-exposed architectures*, S.M. Thesis, Massachusetts Institute of Technology, 2002.

[GPGLW01] V. Gay-Para, T. Graf, A.-G. Lemonnier, and E. Wais, *Kopi Reference manual*, `http://www.dms.at/kopi/docs/kopi.html`, 2001.

[GR05] J. Gummaraju and M. Rosenblum, *Stream Programming on General-Purpose Processors*, MICRO, 2005.

[Gre] J. Greenberg, *Biomedical signal and image processing (MIT 6.555)*, `http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm`.

[Gre75] I. Greif, *Semantics of communicating parallel processes*, Ph.D. Thesis, Massachusetts Insitute of Technology, 1975.

[GRE+01] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, *Mibench: A free, commercially representative embedded benchmark suite*, WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop (Washington, DC, USA), IEEE Computer Society, 2001, pp. 3–14.

[GTA06] M. I. Gordon, W. Thies, and S. Amarasinghe, *Exploiting coarse-grained task, data, pipeline parallelism in stream programs*, International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2006.

[GTK+02] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe, *A Stream Compiler for Communication-Exposed Architectures*, ASPLOS, 2002.

[HBS73] C. Hewitt, P. Bishop, and R. Steiger, *A universal modular ACTOR formalism for artificial intelligence*, International Joint Conferences on Artificial Intelligence (IJCAI), 1973.

[HCK+09] A. H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke, *Flextream: Adaptive compilation of streaming applications for heterogeneous architectures*, PACT '09: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques (Washington, DC, USA), IEEE Computer Society, 2009, pp. 214–223.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, *The synchronous data flow language LUSTRE*, Proc. of the IEEE **79** (1991), no. 1.

[HCW+10] A. H. Hormati, Y. Choi, M. Woh, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke, *Macross: macro-simdization of streaming applications*, ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems (New York, NY, USA), ACM, 2010, pp. 285–296.

[HKM+08] A. Hormati, M. Kudlur, S. Mahlke, D. Bacon, and R. Rabbah, *Optimus: Efficient realization of streaming applications on FPGAs*, International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES) (Atlanta, GA, USA), ACM, 2008, pp. 41–50.

[HMH01] R. Ho, K. Mai, and M. Horowitz, *The Future of Wires*, Proc. of the IEEE, 2001.

[HMWZ92] J. C. Huang, J. Muñoz, H. Watt, and G. Zvara, *ECOS graphs: A dataflow programming language*, Symposium on Applied Computing, 1992.

[Hoa78] C. A. R. Hoare, *Communicating sequential processes*, Communications of the ACM **21** (1978), no. 8, 666–677.

[Hof05] H. P. Hofstee, *Power Efficient Processor Architecture and The Cell Processor*, HPCA **00** (2005), 258–262.

[HWA10] H. Hoffman, D. Wentzlaff, and A. Agarwal, *Remote store programming: A memory model for embedded multicore*, HiPEAC, January 2010.

[HWBR09] A. Hagiescu, W.-F. Wong, D. F. Bacon, and R. Rabbah, *A computing origami: folding streams in FPGAs*, DAC '09: Proceedings of the 46th Annual Design Automation Conference (New York, NY, USA), ACM, 2009, pp. 282–287.

[Inm88] Inmos Corporation, *Occam 2 Reference Manual*, Prentice Hall, 1988.

[int] *Intel® Microarchitecture, Codenamed Nehalem*, `http://www.intel.com/technology/architecture-silicon/next-gen/`.

[JSuA05] O. Johnsson, M. Stenemo, and Z. ul Abdin, *Programming and implementation of streaming applications*, Tech. Report IDE0405, Halmstad University, 2005.

[KA99a] Y.-K. Kwok and I. Ahmad, *FASTEST: A Practical Low-Complexity Algorithm for Compile-Time Assignment of Parallel Programs to Multiprocessors*, IEEE Trans. on Parallel and Distributed Systems **10** (1999), no. 2.

[KA99b] Y.-K. Kwok and I. Ahmad, *Static scheduling algorithms for allocating directed task graphs to multiprocessors*, ACM Comput. Surv. **31** (1999), no. 4, 406–471.

[Kah74] G. Kahn, *The semantics of a simple language for parallel programming*, Information Processing (1974), 471–475.

[Kar02] M. A. Karczmarek, *Constrained and Phased Scheduling of Synchronous Data Flow Graphs for the StreamIt Language*, Master's thesis, MIT, 2002.

[KCGV83] S. Kirkpatrick, J. C.D. Gelatt, and M. Vecchi, *Optimization by Simulated Annealing*, Science **220** (1983), no. 4598.

[KM66] R. M. Karp and R. E. Miller, *Properties of a model for parallel computations: Determinacy, termination, queueing*, SIAM Journal on Applied Mathematics **14** (1966), no. 6, 1390–1411.

[KM08] M. Kudlur and S. Mahlke, *Orchestrating the execution of stream programs on multicore platforms*, Conference on Programming Language Design and Implementation (PLDI), 2008.

[KMD+01] U. J. Kapasi, P. Mattson, W. J. Dally, J. D. Owens, and B. Towles, *Stream scheduling*, Proc. of the 3rd Workshop on Media and Streaming Processors, 2001.

[Knu98] D. E. Knuth, *Art of computer programming*, 2 ed., Addison-Wesley, 1998.

[KP08] W. Ko and C. D. Polychronopoulos, *Automatic granularity selection and OpenMP directive generation via extended machine descriptors in the promis parallelizing compiler*, OpenMP Shared Memory Parallel Programming, vol. 4315, Springer Berlin, 2008, pp. 207–216.

[KRD+03] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens, *Programmable stream processors*, IEEE Computer (2003).

[KTA03] M. Karczmarek, W. Thies, and S. Amarasinghe, *Phased scheduling of stream programs*, Conference on Languages, Compilers, Tools for Embedded Systems (LCTES), 2003.

[Lam88] M. Lam, *Software pipelining: An effective scheduling technique for vliw machines*, PLDI (New York, NY, USA), ACM Press, 1988.

[Lam03] A. A. Lamb, *Linear analysis and optimization of stream programs*, M.Eng. Thesis, Massachusetts Institute of Technology, 2003.

[LaP96] A. S. LaPaugh, *Layout Algorithms for VLSI Design*, ACM Computing Surveys **28** (1996), no. 1.

[Lar06] S. Larsen, *Compilation techniques for short-vector instructions*, Ph.D. thesis, Massachusetts Institute of Technology, April 2006.

[LBF+98] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. P. Amarasinghe, *Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine*, ASPLOS, 1998.

[LDWL06] S.-W. Liao, Z. Du, G. Wu, and G.-Y. Lueh, *Data and Computation Transformations for Brook Streaming Applications on Multiprocessors*, CGO, 2006.

[Leb01] J. Lebak, *Polymorphous Computing Architecture (PCA) Example Applications and Description*, External Report, Lincoln Laboratory, Mass. Inst. of Technology, 2001.

[Lee01] E. A. Lee, *Overview of the Ptolemy Project*, UCB/ERL Technical Memorandum UCB/ERL M01/11, Dept. EECS, University of California, Berkeley, CA, March 2001.

[LFK+93] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg, *The Multiflow Trace Scheduling Compiler*, Journal of Supercomputing **7** (1993), no. 1-2, 51–142.

[LHG+89] E. Lee, W.-H. Ho, E. Goei, J. Bier, and S. Bhattacharyya, *Gabriel: A design environment for DSP*, IEEE Transactions on Acoustics, Speech and Signal Processing **37** (1989), no. 11, 1751–1762.

[LM87a] E. A. Lee and D. G. Messerschmitt, *Synchronous data flow*, Proceedings of the IEEE **75** (1987), no. 9, 1235–1245.

[LM87b] E. A. Lee and D. G. Messerschmitt, *Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing*, IEEE Trans. Comput. **36** (1987), no. 1, 24–35.

[LPMS97] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, *MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems*, MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture (Washington, DC, USA), IEEE Computer Society, 1997, pp. 330–335.

[LTA03] A. A. Lamb, W. Thies, and S. Amarasinghe, *Linear analysis and optimization of stream programs*, Conference on Programming Language Design and Implementation (PLDI), 2003.

[Mac] B. Machrone, *Mythmash: Frontside bus - bottleneck or room to grow?*, `http://www.intelcapabilitiesforum.net`.

[MB01] P. Murthy and S. Bhattacharyya, *Shared buffer implementations of signal processing systems using lifetime analysis techniques*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **20** (2001), no. 2, 177–198.

[MB04] P. K. Murthy and S. S. Bhattacharyya, *Buffer merging – a powerful technique for reducing memory requirements of synchronous dataflow specifications*, ACM Transactions on Design Automation for Electronic Systems **9** (2004), no. 2, 212–237.

[MDH+06] Matthew, Drake, H. Hoffman, R. Rabbah, and S. Amarasinghe, *MPEG-2 decoding in a stream programming language*, International Parallel and Distributed Processing Symposium (IPDPS), 2006.

[MG94] N. Mehdiratta and K. Ghose, *A bottom-up approach to task scheduling on distributed memory multiprocessor*, Proceedings of International Conference on Parallel Processing **2** (1994).

[MGAK03] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, *Cg: A System for Programming Graphics Hardware in a C-like Language*, SIGGRAPH, 2003.

[Mic97] Michael Sipser, *Introduction to the theory of computation*, PWS Publishing Company, 1997.

[ML90] V. W. Mak and S. F. Lundstrom, *Predicting performance of parallel computations*, IEEE Trans. Parallel Distrib. Syst. **1** (1990), no. 3, 257–270.

[ML02] P. Murthy and E. Lee, *Multidimensional synchronous dataflow*, IEEE Transactions on Signal Processing **50** (2002), no. 8, 2064–2079.

[ML03] P. Mattson and R. Lethin, *"Streaming" as a pattern*, August 2003.

[MMT95] B. M. Maggs, L. R. Matheson, and R. E. Tarjan, *Models of parallel computation: a survey and synthesis*, HICSS '95: Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS'95) (Washington, DC, USA), IEEE Computer Society, 1995, p. 61.

[MQP02] M. D. McCool, Z. Qin, and T. S. Popa, *Shader metaprogramming*, SIGGRAPH, 2002.

[MSA+85] J. McGraw, S. Skedzielewski, S. Allan, R. Oldhoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas, *SISAL: Streams and iteration in a single assignment language*, Language reference manual, version 1.2, Lawrence Livermore National Laboratory, 1985.

[MSK87] D. May, R. Shepherd, and C. Keane, *Communicating Process Architecture: Transputers and Occam*, Future Parallel Computers: An Advanced Course, Pisa, Lecture Notes in Computer Science **272** (1987).

[MT93] A. Mitschele-Thiel, *Automatic Configuration and Optimization of Parallel Transputer Applications*, Transputer Applications and Systems '93 (1993).

[MTP+04] M. McCool, S. D. Toit, T. Popa, B. Chan, and K. Moule, *Shader algebra*, SIGGRAPH, 2004.

[Mur89] T. Murata, *Petri nets: Properties, analysis and applications*, Proceedings of the IEEE **77** (1989), no. 4, 541–580.

[NIO+03] H. Nakano, K. Ishizaka, M. Obata1, K. Kimura, and H. Kasahara, *Static coarse grain task scheduling with cache optimization using OpenMP*, International Journal of Parallel Programming, vol. 31, Springer, 2003, pp. 211–223.

[NY03] M. Narayanan and K. A. Yelick, *Generating permutation instructions from a high-level description*, Tech. Report UCB/CSD-03-1287, EECS Department, University of California, Berkeley, 2003.

[NY04] M. Narayanan and K. Yelick, *Generating permutation instructions from a high-level description*, Workshop on Media and Streaming Processors (MSP), 2004.

[O'H91] D. R. O'Hallaron, *The ASSIGN Parallel Program Generator*, Carnegie Mellon Technical Report CMU-CS-91-141, 1991.

[ORSA05] G. Ottoni, R. Rangan, A. Stoler, and D. I. August, *Automatic Thread Extraction with Decoupled Software Pipelining*, MICRO, 2005.

[PBL95] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee, *A Hierarchical Multiprocessor Scheduling Framework for Synchronous Dataflow Graphs*, Technical Report UCB/ERL M95/36, May 1995.

[Pet62] C. Petri, *Communication with automata*, Ph.D. Thesis, Darmstadt Institue of Technology, 1962.

[PL95] J. L. Pino and E. A. Lee, *Hierarchical Static Scheduling of Dataflow Graphs onto Multiple Processors*, Proc. of the IEEE Conference on Acoustics, Speech, and Signal Processing (1995).

[PLW96] M. A. Palis, J.-C. Liou, and D. S. Wei, *Task clustering and scheduling for distributed memory parallel architectures*, IEEE Transactions on Parallel and Distributed Systems **7** (1996), 46–55.

[PM91] K. Parhi and D. Messerschmitt, *Static Rate-Optimal Scheduling of Iterative Data-Flow Programs Via Optimum Unfolding*, IEEE Transactions on Computers **40** (1991), no. 2.

[PPL95] T. M. Parks, J. L. Pino, and E. A. Lee, *A comparison of synchronous and cycle-static dataflow*, Asilomar Conference on Signals, Systems, and Computers, 1995.

[Pri91] H. Printz, *Automatic mapping of large signal processing systems to a parallel machine*, Ph.D. Thesis, Carnegie Mellon University, 1991.

[PW96] T. A. Proebsting and S. A. Watterson, *Filter Fusion*, POPL, 1996.

[QCS02] Y. Qian, S. Carr, and P. Sweany, *Loop fusion for clustered VLIW architectures*, LCTES/SCOPES (New York, NY, USA), ACM Press, 2002, pp. 112–119.

[RDK+98] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens, *A bandwidth-efficient architecture for media processing*, MICRO-31, 1998.

[Reu04] A. Reuther, *Preliminary design review: Gmti narrowband for the basic pca integrated radar-tracker application*, Tech. Report PCA-IRT-3, Lincoln Laboratory: Massachusetts Institute of Technology, 2004.

[RG81] B. R. Rau and C. D. Glaeser, *Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing*, MICRO-14 (Piscataway, NJ, USA), IEEE Press, 1981.

[Sen80] S. Seneff, *Speech transformation system (spectrum and/or excitation) without pitch extraction*, Master's thesis, Massachussetts Institute of Technology, 1980.

[Ser05] J. Sermulins, *Cache Optimizations for Stream Programs*, M.Eng. Thesis, Massachusetts Institute of Technology, 2005.

[SGB06] S. Stuijk, M. Geilen, and T. Basten, *Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs*, Design Automation Conference (DAC), 2006.

[SLRBE05] A. Solar-Lezama, R. Rabbah, R. Bodik, and K. Ebcioglu, *Programming by sketching for bit-streaming programs*, Conference on Programming Language Design and Implementation (PLDI), 2005.

[So07] W. So, *Software thread integration for instruction level parallelism*, Ph.D. Thesis, North Carolina State University, 2007.

[SRG94] J. P. Singh, E. Rothberg, and A. Gupta, *Modeling communication in parallel algorithms: a fruitful interaction between theory and systems?*, SPAA '94: Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures (New York, NY, USA), ACM, 1994, pp. 189–199.

[ST09] D. Seo and M. Thottethodi, *Disjoint-path routing: Efficient communication for streaming applications*, IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing (Washington, DC, USA), IEEE Computer Society, 2009, pp. 1–12.

[Ste97] R. Stephens, *A Survey of Stream Processing*, Acta Informatica **34** (1997), no. 7.

[stra] *StreamIt homepage*, `http://compiler.lcs.mit.edu/streamit`.

[strb] *StreamIt Language Specification*, `http://cag.lcs.mit.edu/streamit/papers/streamit-lang-spec.pdf`.

[Strc] *StreamIt cookbook*, `http://cag.csail.mit.edu/streamit/papers/streamit-cookbook.pdf`.

[STRA05] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe, *Cache aware optimization of stream programs*, Conference on Languages, Compilers, Tools for Embedded Systems (LCTES), 2005.

[T+02] M. B. Taylor et al., *The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs*, IEEE Micro vol 22, Issue 2, 2002.

[Tai93] E. Taillard, *Benchmarks for basic scheduling problems*, European Journal of Operational Research **64** (1993), no. 2, 278–285.

[Tan09] C. Tan, *Dynamic load-balancing of streamit cluster computations*, M.eng thesis, Massachusetts Institute of Technology, Cambridge, MA, August 2009.

[TCA07] W. Thies, V. Chandrasekhar, and S. Amarasinghe, *A practical approach to exploiting coarse-grained pipeline parallelism in C programs*, International Symposium on Microarchitecture (MICRO), 2007.

[THA07] W. Thies, S. Hall, and S. Amarasinghe, *Mapping stream programs into the compressed domain*, Tech. Report MIT-CSAIL-TR-2007-055, Massachusetts Institute of Technology, 2007, `http://hdl.handle.net/1721.1/39651`.

[Thi09] W. Thies, *Language and compiler support for stream programs*, Ph.d. thesis, Massachusetts Institute of Technology, Cambridge, MA, Feb 2009.

[til] *TILE-Gx Processors Family*, http://www.tilera.com/products/TILE-Gx.php.

[TKA02] W. Thies, M. Karczmarek, and S. Amarasinghe, *StreamIt: A language for streaming applications*, International Conference on Compiler Construction (CC), 2002.

[TKG+02] W. Thies, M. Karczmarek, M. Gordon, D. Maze, J. Wong, H. Hoffmann, M. Brown, and S. Amarasinghe, *A common machine language for grid-based architectures*, ACM SIGARCH Computer Architecture News, 2002.

[TKS+05] W. Thies, M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe, *Teleport messaging for distributed stream programs*, Symposium on Principles and Practice of Parallel Programming (PPoPP), 2005.

[TLAA03] M. B. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal, *Scalar operand networks: On-chip interconnect for ilp in partitioned architectures*, HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture (Washington, DC, USA), IEEE Computer Society, 2003, p. 341.

[TLM+04] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, et al., *Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams*, ISCA (Munich, Germany), June 2004.

[tra88] *The Transputer Databook*, Inmos Corporation, 1988.

[Tse89] P.-S. Tseng, *A parallelizing compiler for distributed memory parallel computers*, Ph.D. thesis, Carnegie Mellon University, 1989.

[Tse90] P.-S. Tseng, *Compiling programs for a linear systolic array*, PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation (New York, NY, USA), ACM, 1990, pp. 311–321.

[TTRT00] R. Thomas, R. Thomas, S. Reader, and T. J. Thomas, *An architectural performance study of the fast fourier transform on vector IRAM*, Tech. Report UCB/CSD-00-1106, University of California, Berkeley, 2000.

[UGT09a] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil, *Software pipelined execution of stream programs on GPUs*, CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization (Washington, DC, USA), IEEE Computer Society, 2009, pp. 200–209.

[UGT09b] ———, *Synergistic execution of stream programs on multicores with accelerators*, LCTES '09: Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems (New York, NY, USA), ACM, 2009, pp. 99–108.

[VNL97] S. P. VanderWiel, D. Nathanson, and D. J. Lilja, *Complexity and performance in parallel programming languages*, HIPS '97: Proceedings of the 1997 Workshop on High-Level Programming Models and Supportive Environments (HIPS '97) (Washington, DC, USA), IEEE Computer Society, 1997, p. 3.

[Wat06] D. Watkins, *Mash Hits*, The Guardian (2006).

[WGH+07] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. M. J. F. B. III, and A. Agarwal, *On-chip interconnection architecture of the tile processor*, IEEE Micro **27** (Sept.-Oct. 2007), no. 5, 15–31.

[Won04] J. Wong, *Modeling the scalability of acyclic stream programs*, M.Eng. Thesis, Massachusetts Institute of Technology, 2004.

[WTS+97] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, et al., *Baring It All to Software: Raw Machines*, IEEE Computer **30** (1997), no. 9.

[YFH97] H. W. Yau, G. Fox, and K. A. Hawick, *Evaluation of high performance fortran through application kernels*, HPCN Europe '97: Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking (London, UK), Springer-Verlag, 1997, pp. 772–780.

[Zha07] X. D. Zhang, *A streaming computation framework for the Cell processor*, M.Eng. Thesis, Massachusetts Institute of Technology, Cambridge, MA, 2007.

[ZLRA08] D. Zhang, Q. J. Li, R. Rabbah, and S. Amarasinghe, *A lightweight streaming layer for multi-core execution*, Workshop on Design, Architecture, and Simulation of Chip Multi-Processors (dasCMP), 2008.

[ZLSL05] D. Zhang, Z.-Z. Li, H. Song, and L. Liu, *A Programming Model for an Embedded Media Processing Architecture*, SAMOS, 2005.

[ZTB00] E. Zitzler, J. Teich, and S. S. Bhattacharyya, *Multidimensional exploration of software implementations for DSP algorithms*, Journal of VLSI Signal Processing **24** (2000), no. 1, 83–98.

Appendix A

Guide to Notation

Notation	Explanation
$G = (V, E)$	The stream graph, where V is the set of filters and E is set of FIFO channels between filters.
$(f, g) \in E$	Denotes communication from f to g.
$M(\Sigma, F)$	The multiplicity of filter F in schedule Σ, where $\Sigma \in \{I, S\}$. I represents the multiplicities of filters in the initialization schedule. S represents the multiplicities of filters in the steady-state schedule.
W_F	Work function of F.
W_F^P	Prework function of F.
$o(\mathcal{F}, F)$	The number of items dequeued from F's input buffer per firing of $\mathcal{F}$. $\mathcal{F} \in \{W^p, W\}$. This quantity is termed the *pop* rate.
$e(\mathcal{F}, F)$	$1 +$ the greatest index that is read (but not necessarily dequeued) from F's input buffer per firing of $\mathcal{F}$. $\mathcal{F} \in \{W^p, W\}$. This quantity is termed the *peek* rate.
$u(\mathcal{F}, F)$	The number of items enqueued to F's output buffer per firing of $\mathcal{F}$. $\mathcal{F} \in \{W_F^p, W_F\}$. This quantity is termed the *push* rate.
$C(F)$	The number of items remaining on F's input channel(s) after execution of the initialization schedule.
$\text{OUT}(F)$	The set of output edges of F.
$\text{IN}(F)$	The set of input edges of F.
$ID(\Sigma, F)$	The input distribution pattern for F. $ID(\Sigma, F) \in (\mathbb{N} \times E)^n = ((w_1, e_1), (w_2, e_2), ..., (w_n, e_n))$ n is the width of the input distribution pattern. Describes the round robin joining pattern for organizing the input data into the filter's single internal FIFO buffer, where w_i items are received from edge e_i before proceeded to the next edge, e_{i+i}.
$OD(\Sigma, F)$	The output distribution pattern for F. $OD(\Sigma, F) \in (\mathbb{N} \times (P(E) - \emptyset))^m = ((w_1, d_1), (w_2, d_2), ..., (w_n, d_n))$ Each d_i is called the *dupset* of weight i. The dupset d_i specifies that w_i items be duplicated to the edges of the dupset. Each tuple denotes that w_i output items of F are duplicated to the edges of d_i before moving on to the next tuple.
$RO(F_1, F_2, \Sigma)$	The ratio of output items F_1 splits along the edge (F_1, F_2) to the total number of items that F_1 produces in the schedule Σ.
$RI(F_1, F_2, \Sigma)$	The percentage of total input items that F_2 receives from F_1 for Σ.
$s(\mathcal{F}, F)$	The execution time (in cycles, with respect to a given real or conceptual machine) of function $\mathcal{F}$ of filter F.

Table A-1: Guide to notation.