STREAMROLLER : A UNIFIED COMPILATION AND SYNTHESIS SYSTEM FOR STREAMING APPLICATIONS

by

Manjunath V. Kudlur

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2008

Doctoral Committee:

Associate Professor Scott A. Mahlke, Chair
Associate Professor Marina A. Epelman
Associate Professor Igor L. Markov
Assistant Professor Valeria M. Bertacco
Robert S. Schreiber, Hewlett-Packard Laboratories

© Manjunath V. Kudlur 2008
All Rights Reserved

To my grandfather, Yogappa.

ACKNOWLEDGEMENTS

I express my sincere gratitude towards my adviser, Professor Scott Mahlke, whose infectious enthusiasm carried my research through good times and bad. His mentorship and support over the years have made this thesis possible. I thank my dissertation committee members, Dr. Robert Schreiber, Professor Igor Markov, Professor Marina Epelman, and Professor Valeria Bertacco for their insightful comments and suggestions. Each of them brought a different perspective, and that helped me create a more polished thesis.

The members of the Compilers Creating Custom Processors (CCCP) group, my research group, have given me technical and moral support over the years, and made going to work, fun. I thank Michael Chu, Rajiv Ravindran, Nathan Clark, Kevin Fan, Yuan Lin, Hongtao Zhong, Hyunchul Park, Jason Blome, Ganesh Dasika, Shuguang Feng, Shantanu Gupta, Amin Ansari, Jeff Hao, and Mojtaba Mehrara for their camaraderie.

Last but not the least, I thank my family, who bring meaning to all my efforts.

TABLE OF CONTENTS

DEDICATION . . . ii

ACKNOWLEDGEMENTS . . . iii

LIST OF FIGURES . . . vi

LIST OF TABLES . . . viii

ABSTRACT . . . ix

CHAPTERS

1 Introduction . . . 1
  1.1 Contributions . . . 4
  1.2 Organization . . . 5

2 Background . . . 6
  2.1 Stream Programming . . . 6
  2.2 Multicore Compilation . . . 7
  2.3 Automated Synthesis . . . 8

3 Stream Graph Modulo Scheduling . . . 11
  3.1 Introduction . . . 11
  3.2 Background and Motivation . . . 14
    3.2.1 StreamIt Language . . . 14
    3.2.2 Cell Broadband Architecture . . . 16
    3.2.3 Motivation . . . 17
  3.3 Naïve Unfolding . . . 20
    3.3.1 Properties of Unfolded Stream Graph . . . 21
    3.3.2 Code Generation for Cell . . . 28
  3.4 Stream Graph Modulo Scheduling . . . 31
    3.4.1 Integrated Fission and Processor Assignment . . . 31
    3.4.2 Stage Assignment . . . 37
    3.4.3 Code Generation for Cell . . . 39

3.5 Evaluation 48
3.5.1 Experiments 48
3.6 Related Work 56

4 Memory Management for Stream Graph Modulo Scheduling 59
4.1 Introduction 59
4.2 Background 61
4.2.1 Memory System of the Cell Broadband Architecture . 61
4.2.2 Memory Requirements for SGMS 63
4.3 Block Allocation 64
4.4 ILP Formulation 70
4.5 Results 73

5 Automatic Synthesis of Prescribed Throughput Accelerator Pipelines 79
5.1 Introduction 79
5.2 System Overview 80
5.2.1 Input Specification 82
5.2.2 Accelerator Pipeline Hardware Schema 85
5.3 Design Methodology 88
5.3.1 Cost Components 88
5.3.2 ILP Formulation 91
5.3.3 Implementation 95
5.4 Case Studies 96

6 Conclusions and Future Directions 102

APPENDIX 106

BIBLIOGRAPHY 121

LIST OF FIGURES

Figure

3.1 (a) Example StreamIt program and (b) corresponding stream graph. . 15
3.2 The Cell broadband architecture. 16
3.3 Theoretical speedup for unmodified programmer-conceived stream graph. 18
3.4 A stream graph and its unfolded version showing data dependencies introduced due to unfolding. 22
3.5 A mapping of an unfolded stream graph on to 3 processors. 23
3.6 Code generation schema for Cell. (a) StreamIt program, (b) Unfolding and Mapping on 2 SPEs. (c) Multithreaded C code for Cell, with relevant synchronization and DMA copies. 29
3.7 Multithreaded C code for Cell, with relevant synchronization and DMA copies. 30
3.8 Example illustrating ILP formulation. 34
3.9 Properties of stages. 39
3.10 Main loop implementing the modulo schedule. 42
3.11 Buffer allocation for the modulo schedule. 43
3.12 Example illustrating fission, processor assignment and stage assignment. 45
3.13 Example illustrating a modulo schedule running on Cell. 46
3.14 Stream-graph modulo scheduling speedup normalized to single SPE. . 50
3.15 Comparing naïve unfolding to SGMS. 52
3.16 Effect of exposed DMA latency. 53
3.17 Comparing ILP partitioning to greedy partitioning. 54
4.1 Observed latencies for LS to LS and global memory to LS data transfers on the CBE. 62
4.2 A StreamIt filter and memory map on an SPE's local store. 63
4.3 Block storage option requiring the most storage space, but allowing complete overlap of DMAs at both producer and consumer processors. 68
4.4 Block storage option allowing overlap of DMA at the producer, but not at the consumer processor. 69

4.5 Block storage option allowing overlap of DMA at the consumer, but not at the producer processor. 70
4.6 Block storage option requiring the least storage space, but exposes DMA transfers at both the producer and consumer processors. 71
4.7 Buffer space required on each processor under different edge-cut minimization strategies. 75
4.8 Comparison of greedy with ILP-based block allocation. 76
4.9 Speedups on 8 SPEs with memory constraints. 77
5.1 Overview of the Streamroller synthesis system. 81
5.2 Input specification 83
5.3 Example accelerator pipeline (High performance) 87
5.4 Example accelerator pipeline (Low performance) 88
5.5 II vs. Cost of a loop accelerator 89
5.6 ILP formulation for system level synthesis 92
5.7 Cost vs. Throughput for Simple 97
5.8 Pipeline configurations for Simple 98
5.9 Cost vs. Throughput for FMRadio 99
5.10 Cost vs. Throughput for Beamformer 100
A.1 Graph construction for the example problem instance. 112
A.2 Generator 1 from Saucy's output for 3x3 graph 112
A.3 Generator 2 from Saucy's output for 3x3 graph 113
A.4 Time taken by SCIP solver on ILP models for assigning tasks to 8 processors. 117
A.5 Shifted column inequality. 119
A.6 Time taken by CPLEX solver on ILP models for assigning tasks to 32 processors. 119

LIST OF TABLES

Table

3.1 Benchmark characteristics. 49
4.1 Block storage options summary. 72

ABSTRACT

STREAMROLLER : A UNIFIED COMPILATION AND SYNTHESIS SYSTEM FOR STREAMING APPLICATIONS

by

Manjunath V. Kudlur

Chair: Scott A. Mahlke

The growing complexity of applications has increased the need for higher processing power. In the embedded domain, the convergence of audio, video, and networking on a handheld device has prompted the need for low cost, low power, and high performance implementations of these applications in the form of custom hardware. In a more mainstream domain like gaming consoles, the move towards more realism in physics simulations and graphics has forced the industry towards multicore systems. Many of the applications in these domains are streaming in nature. The key challenge is to get efficient implementations of custom hardware from these applications and map these applications efficiently onto multicore architectures.

This dissertation presents a unified methodology, referred to as *Streamroller*, that can be applied for the problem of scheduling stream programs to multicore architectures and to the problem of automatic synthesis of custom hardware for stream applications. Firstly, a method called *stream-graph modulo scheduling* is presented, which maps stream programs effectively onto a multicore architecture. Many aspects of a real system, like limited memory and explicit DMAs are modeled in the scheduler. The scheduler is evaluated for a set of stream programs on IBM's Cell processor.

Secondly, an automated high-level synthesis system for creating custom hardware for stream applications is presented. The template for the custom hardware is a pipeline of accelerators. The synthesis involves designing loop accelerators for individual kernels, instantiating buffers to store data passed between kernels, and linking these building blocks to form a pipeline. A unique aspect of this system is the use of multifunction accelerators, which improves cost by efficiently sharing hardware between multiple kernels.

Finally, a method to improve the integer linear program formulations used in the schedulers that exploits symmetry in the solution space is presented. Symmetry-breaking constraints are added to the formulation, and the performance of the solver is evaluated.

CHAPTER 1

Introduction

Emerging multimedia and wireless applications such as life-like realistic video games and mobile internet demand high processing power. In the embedded domain, this demand has been met with custom hardware tailored for a particular application. In more mainstream domains like gaming consoles, the industry has turned to multicore systems to meet the ever increasing demand for processing power. In both these domains, the nature of applications has become more complex over time. Developing custom hardware for these applications is hard, given the stringent area and energy constraints. Also, mapping these applications to multicore systems places the scheduling and orchestration burden on the compiler.

Fortunately, there is an abundance of parallelism in multimedia and wireless applications. The nature of these applications makes it easy for the programmer to express them using stream languages [42] in which pipeline parallelism is made explicit. In stream languages, the computation is represented as a directed graph where each node represents a piece of computation and each arc represents some data flow. Each

node (referred to as an actor or a filter), has an independent instruction stream and separate address space. It is easy to see that this model is motivated by application style used in media and network processing, where data “streams” through filters.

A key challenge in the embedded domain is to automatically create high quality custom hardware for these streaming applications. Automatic synthesis is important, as it enables short time-to-market and decreases system verification effort. The template for the custom hardware explored in this work is a system of accelerators communicating through memory buffers. Embedded applications often have real-time constraints, and the main goal of a synthesis system should be to synthesize hardware with enough resources to satisfy those constraints. At the same time, the synthesis system should provide good quality in several dimensions such as area and energy.

In the general purpose domain, the key challenge is to find an efficient mapping onto the target architecture. The compiler should plan and orchestrate the parallel execution on multiple cores. Often the benefit obtained from parallel execution is overshadowed by overheads due to communication and synchronization. Resource limitations, such as limited memory available on the processing elements and limited bandwidth available to transfer data between processing elements, should be carefully modeled during the mapping process to avoid stalls.

This work proposes a unified methodology, referred to as *Streamroller*, that can be applied to the problem of scheduling stream programs to multicore architectures and to the problem of automatic synthesis of custom hardware for stream applications. The unification comes about from the fact that both problems involve mapping

computation onto a set of processing elements. In the case of scheduling to multicore architectures, the set of processing elements are the cores available on the system. In the case of synthesis, computation is mapped onto a set of virtual accelerators. In the scheduling case, the mapping should honor resource constraints of an existing system, whereas in the synthesis case, the mapping should try to minimize the cost of resources used.

The component of Streamroller for mapping stream programs onto multicore systems is referred to as Stream Graph Modulo Scheduling (SGMS). Modulo scheduling is traditionally a form of software pipelining applied at the instruction level [58]. The same technique is applied on a coarse-grain stream graph to pipeline the actors across multiple cores. The objective is to maximize concurrent execution of actors while hiding communication overhead to minimize stalls. SGMS is a phase-ordered approach consisting of two steps. First, an integrated actor fission and partitioning step is performed to assign actors to each processor ensuring maximum work balance. Fission refers to the selective replication and splitting of parallel data actors to increase the opportunities for even work distribution. This first step is formulated as an integer linear program. The second step is stage assignment wherein each actor is assigned to a pipeline stage for execution. Stages are assigned to ensure data dependences are satisfied and inter-processor communication latency is maximally overlapped with computation. SGMS is evaluated on the Cell processor [31], which is a decoupled heterogeneous multicore system, and is representative of an important class of future multicore systems. Extensions are presented which make SGMS suit-

able for multicores in the embedded domain, which have more constrained memory systems.

The second part of this work is an automated system for designing stylized accelerator pipelines from streaming applications. This work focuses on synthesizing a highly customized pipeline that minimizes hardware cost while meeting a user-prescribed performance level. The cost of all components of the system are modeled in the design, including the data path cost of individual accelerators, and the memory buffers between pipeline stages. A unique aspect of the system is the utilization of multifunction loop accelerators [25] to enable multiple pipeline stages to time multiplex the hardware for a single pipeline stage. This approach sacrifices performance as kernel execution is sequentialized, but greatly increases the ability to share hardware in the design and thus drive down the overall cost.

1.1 Contributions

This dissertation makes the following contributions.

- A unified methodology for dealing with stream applications that can be applied in two problem settings : mapping stream applications to multicore systems, and synthesizing custom hardware for stream applications.

- A coarse-grain scheduling approach that maximally exploits pipeline parallelism in stream programs. The scheduler models realistic resource constraints, including limited local storage and data transfer overhead.

- A systematic design methodology for creating accelerator pipelines for stream applications with minimum cost at a user-specified throughput.

1.2 Organization

The remainder of this dissertation report is organized as follows. Chapter 2 provides brief background on stream programming, mapping to multicore processors and automated synthesis of custom accelerators. Chapter 3 describes Stream Graph Modulo Scheduling (SGMS), a coarse-grain software pipelining method for mapping stream programs to multicore processors. Extensions to SGMS, which make it more suitable for platforms with constrained memory systems is presented in Chapter 4. Chapter 5 presents a high level synthesis method that automatically produces a pipeline of loop accelerators for stream programs. The integer linear programming formulations used in both the compilation and synthesis systems is further examined in Appendix A. Symmetries present in the formulation are systematically discovered and exploited to speed up the solver runtime. Finally, Chapter 6 provides conclusions and directions that this research can be taken in the future.

CHAPTER 2

Background

2.1 Stream Programming

In the scientific community, there is a long history of successful parallelization efforts [3, 10, 17, 30, 37]. These techniques target counted loops that manipulate array accesses with affine indices, where memory dependence analysis can be precisely performed. Loop-level and single-instruction multiple-data (SIMD) parallelism are extracted to execute multiple loop iterations or process multiple data items in parallel. Unfortunately, these techniques do not often translate well to other applications. More sophisticated memory dependence analysis has been proposed, such as points-to analysis [52]. However, scientific parallelization techniques have yet to be successful outside their domain due to limitations of compiler analyses.

The stream programming paradigm offers a promising approach for programming multicore systems. Stream languages are motivated by the application style used in image processing, graphics, networking, and other media processing domains. Exam-

ple stream languages are StreamIt [64], Brook [12], CUDA [51], SPUR [71], Cg [46], Baker [13], and Spidle [16]. Stream languages represent computation as a directed graph where each node referred to as an actor or a filter represents computation, and each arc represents the flow of data [41]. In one model of stream programming, called Synchronous Data Flow (SDF) [42], the number of data samples produced and consumed by each node is specified a priori. For this work, we focus on StreamIt, which implements the SDF programming model.In StreamIt, a program is represented as a set of autonomous filters that communicate through first-in first-out (FIFO) data channels [64]. During program execution, filters fire repeatedly in a periodic schedule [26]. Each filter has a separate instruction stream and an independent address space, thus all dependences between filters are made explicit through the communication channels. Compilers can leverage these characteristics to plan and orchestrate parallel execution.

2.2 Multicore Compilation

Multicore systems have become the industry standard from high-end servers, down through desktops and gaming platforms, and finally into handheld devices. Example systems include the Sun UltraSparc T1 that has 8 cores [36], the Sony/Toshiba/IBM Cell processor that consists of 9 cores [31], the Intel IXP 2800 network processor that contains 16 microengines [2], and the Cisco CRS-1 Metro router that utilizes 192 Tensilica processors [22]. Intel and AMD are producing quad-core systems today and larger systems are on their near term roadmaps. Putting more cores on a chip

increases peak performance, but it has shifted the burden onto both the programmer and compiler to identify large amounts of coarse-grain parallelism to effectively utilize the cores. Highly threaded server workloads naturally take advantage of more cores to increase throughput. However, the performance of single-thread applications has dramatically lagged behind. Traditional programming models, such as C, C++, and Fortran, are poorly matched to multicore environments because they assume a single instruction stream and a centralized memory structure.

Programmers are slowly warming up to parallel programming to completely exploit multicores. Compilers for explicitly parallel languages targeting multicores face problems beyond compilers for single core. These compilers have to model and utilize macro resources such as processing elements, distributed memories and interconnects. Intricacies such as heterogeneity of processing elements, disjoint address spaces and explicit data transport mechanisms throws up new challenges for multicore compilers. Even though multicore architectures are quite mature now, the state-of-the-art of compilers for them lags far behind.

2.3 Automated Synthesis

As communication bandwidths are scaled or more features are added to portable devices, such as high-definition video, embedded computing systems are required to perform increasingly demanding computation tasks. Programmable processors are unable to meet increasing performance requirements and decreasing cost and energy budgets. Application specific hardware in the form of loop accelerators is often used

to address these issues. A loop accelerator implements a critical loop from an application with far greater performance and efficiency than would be possible with a programmable implementation. However, a single accelerator designed and operated in isolation is insufficient. These tasks are commonly streaming applications that consist of multiple compute-intensive functions (e.g., filters) that operate in turn on streaming data. The natural realization of these tasks is a hardware pipeline of accelerators, each implementing one or more functions that process the data. These accelerators must be designed to meet overall throughput requirements while minimizing the hardware cost.

Designing a highly customized system of accelerators presents several difficult challenges. The design space is enormous because of the large number of variables, which include the number, type, and specific design of each accelerator, mapping of application loops to accelerators, arrangement of accelerators in the overall pipeline, and method of inter-accelerator communication. In the face of such challenges, an automated accelerator pipeline design system enables the systematic exploration of a much larger portion of the design space than is possible with manual designs, leading to higher-quality results. In addition, an automated system designs pipelines that are correct by construction by using a parameterized template, greatly reducing the verification portion of the product cycle. All of these factors enable automated design systems to deliver high-performance, low-cost solutions with shorter time-to-market, which is critical particularly in the embedded domain.

Automated synthesis of hardware from high-level specifications has a long history

in the design automation field. Most high level synthesis systems in the past build an operation-level data flow representation of the specification, and derive the datapath for the hardware from the schedule of the data flow graph. However, the granularity of synthesis has varied over the years. One of the earliest systems, Chippe [11], used a PASCAL-like language as the input specification. It synthesized hardware for acyclic basic blocks without memory accesses. The granularity of synthesis was a basic block for most of the earlier systems [56, 15, 14, 21]. MIMOLA [47] and Cathedral III [50] represent comprehensive approaches to high level synthesis. They perform memory and data path synthesis for a restricted domain of applications. The works in [68, 49, 65] are also some of the earlier systems that varied widely in the input specification language and granularity of synthesis. PICO [61] takes a programmer centric view of the high level synthesis problem. The language of specification is sequential C so that the designer can focus on just expressing the algorithm, rather than worrying about hardware details. This makes it easy for the designer to develop and debug applications quickly, and the system automatically derives the architecture at a desired performance level.

CHAPTER 3

Stream Graph Modulo Scheduling

3.1 Introduction

Stream programs are explicitly parallel. The central challenge is obtaining an efficient mapping onto the target architecture. Often the gains obtained through parallel execution can be overshadowed by the costs of communication and synchronization. Resource limitations of the system must also be carefully modeled during the mapping process to avoid stalls. Resource limitations include finite processing capability and memory associated with each processing element, interconnect bandwidth, and direct memory access (DMA) latency. Lastly, stream programs contain multiple forms of parallelism that have different tradeoffs. It is critical that the compiler leverage a synergistic combination of parallelism, while avoiding both structural and resource hazards.

In this chapter, we present a modulo scheduling algorithm for mapping streaming applications onto multicore systems, referred to as *stream-graph modulo scheduling*

or SGMS [39, 44]. Modulo scheduling is traditionally a form of software pipelining applied at the instruction level [58]. We apply the same technique on a coarse-grain stream graph to pipeline the filters across multiple cores. The objective is to maximize concurrent execution of filters while hiding communication overhead to minimize stalls. SGMS is a phase-ordered approach consisting of two steps. First, an integrated filter fission and partitioning step is performed to assign filters to each processor ensuring maximum work balance. Parallel data filters are selectively replicated and split to increase the opportunities for even work distribution. This first step is formulated as an integer linear program. The second step is stage assignment wherein each filter is assigned to a pipeline stage for execution. Stages are assigned to ensure data dependences are satisfied and inter-processor communication latency is maximally overlapped with computation.

Our target platform is the Cell architecture, which represents the first tangible platform that is a decoupled multicore where there is no shared cache so code-data colocation is necessary [31]. SGMS is part of a fully automatic compilation system, known as StreamRoller, that maps StreamIt applications onto a Cell system. The SGMS schedule is output in the form of a C template that executes an arbitrary software pipeline. This template, combined with C versions of the filters, are compiled with the host compiler to execute on the target system. For our experiments, we use an IBM QS20 Blade Server running Fedora Core 6.0. It is a Cell system equipped with 16 3.2GHz synergistic processing engines (SPEs) on 2 chips, and 1 GB RAM.

Our work has the most overlap with the coarse-grained scheduling used in the

StreamIt compiler [27, 26]. The StreamIt scheduler consists of two major phases. First, a set of transformations are applied on the stream graph to combine and split filters to ensure the computation granularity is balanced. Second, a coarse-grain software pipeline is constructed by iteratively applying a greedy partitioning heuristic that assigns filters to processors. Each filter is considered in order of decreasing work and assigned to the processor with the least amount of work so far. To minimize synchronization, the partitioning algorithm is wrapped with a selective fusion pass that repeatedly fuses the two adjacent filters that have the smallest combined work. This process reduces communication overhead by forcing the combined filters to reside on the same processor.

Our work differs along two primary dimensions. First, the StreamIt compiler targets the Raw processor that has a traditional cache on each processor [63]. In [26], intermediate buffers needed by the software pipeline of the stream graph are relegated to the off-chip DRAM banks, and a separate communication stage is introduced between steady states to shuffle data between banks. Our formulation of pipeline stage assignment explicitly models DMA overhead and proactively overlaps data transfers for future iterations with computation on the current iteration. Second, we formulate the partitioning and filter fission step as an integer linear program rather than employing iterative partitioning and fusing to generate a schedule. Our approach combines packing and fission of filters, data transfers, and resource constraints to generate more balanced and higher quality schedules for architectures such as Cell.

This work offers the following contributions:

- The design, implementation, and evaluation of stream-graph modulo scheduling for efficiently mapping streaming applications onto decoupled multicore systems.
- An integer linear program formulation for integrated filter fission and partitioning to assign filters to processing elements maximizing workload balance.
- A pipeline stage assignment algorithm that proactively overlaps DMA transfers with computation to minimize stalls.
- A fully automated compilation system for Cell capable of generating performance results on real hardware.

3.2 Background and Motivation

3.2.1 StreamIt Language

StreamIt [64] is an explicitly parallel programming language that implements the synchronous data flow (SDF) [41] programming model. Filters are specified by parametrized classes, which are similar to Java classes. They can have local variables corresponding to local filter state, and methods that accesses these variables. An filter can have both read-only and read-write state. A stateful filter that modifies local state during the work function cannot be parallelized as the next invocation depends on the previous invocation. However, the SDF semantics allow the parallel replication of stateless filters. A special method called `work` is reserved to specify the

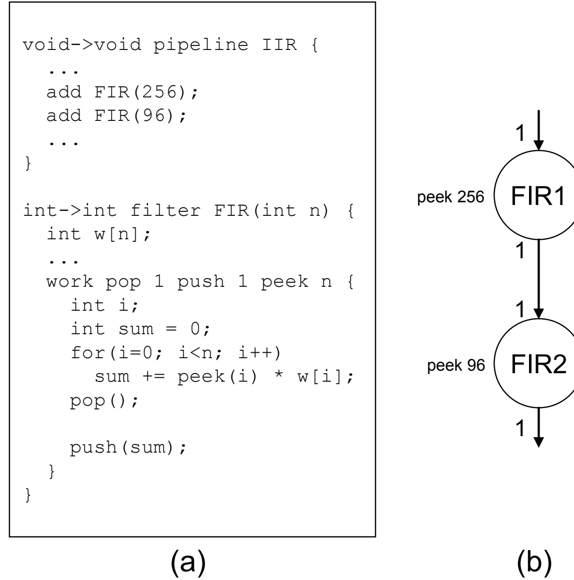


Figure 3.1: (a) Example StreamIt program and (b) corresponding stream graph.

work function that is executed when the filter is invoked in steady state. The stream rates (number of items *pushed* and *popped* on every invocation) of the work functions are specified statically in the program.

The stream graph is constructed by instantiating objects of the filter classes. StreamIt provides ways to construct specific structures like *pipeline*, *split-join*, and *feedback loop*. Using these primitives, the entire graph can be constructed hierarchically. Note that feedback loops provide a means to create cycles in the stream graph. Feedback loops are naïvely handled by fusing the entire loop into a single filter. More intelligent ways to handle nested loops is beyond the scope of this chapter. Further, the feedback loop pattern does not appear in any of the benchmarks that we evaluate. Hence, the rest of the chapter assumes an acyclic stream graph.

Figure 3.1 shows an example StreamIt program and its corresponding stream

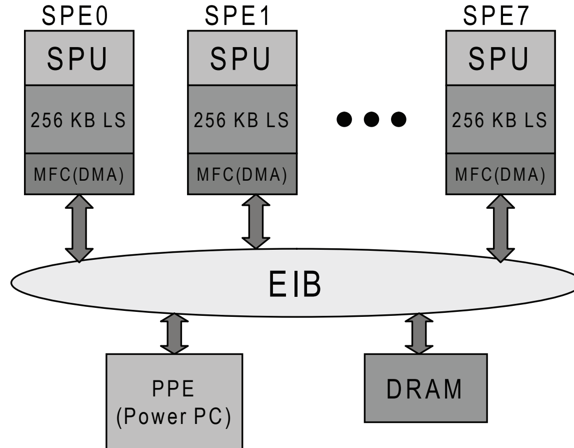


Figure 3.2: The Cell broadband architecture.

graph. StreamIt provides the `peek` primitive to the programmer, which can be used to non-destructively read values off the input channel. Note that this is only for convenience, and does not make StreamIt deviate from the pure SDF model. This is because a program with `peek` can always be reimplemented with just `pushes` and `pops`, and some local state that holds a subset of values seen so far.

3.2.2 Cell Broadband Architecture

Our compilation target is the Cell Broadband Engine (CBE), shown in Figure 3.2. The CBE is a heterogeneous multicore system, consisting of one 64bit PowerPC core called the Power Processing Element (PPE) and eight Synergistic Processing Elements (SPEs). Each SPE has a Single-Instruction Multiple-Data (SIMD) engine called the synergistic processing unit (SPU), 256 KB of local memory and a memory flow control (MFC) unit which can perform Direct Memory Access (DMA) operations to and from the local stores independent of the SPUs. The SPUs can only access the local store,

so any sharing of data has to be performed through explicit DMA operations. The SPEs and PPE are connected via a high bandwidth interconnect called the Element Interconnect Bus (EIB). The main memory and peripheral devices are also connected to the EIB. The feature of the CBE most relevant to this work is the ability of the MFCs to do non-blocking DMA operations independent of the SPUs. The SPUs can issue DMA requests that are added to hardware queues of the MFCs. The SPU can continue doing computation while the DMA operation is in progress. The SPU can query the MFC for DMA completion status and block only when the needed data has not yet arrived. The ability to perform asynchronous DMA operations allow overlap of computation and communication, and is leveraged for efficient software pipelining of stream graphs.

3.2.3 Motivation

Stream programs are replete with pipeline parallelism. A filter can start working on the next data item as soon as it is done with the current item, even when other filters in the downward stream of the graph are still working on the current item. In a multiprocessor environment, by running different filters on different processors and overlapping iterations, the outer loop can be greatly sped up. Trying to exploit pipeline parallelism requires (1) a good distribution of work among the available processors and (2) managing the communication overhead resulting because of producers and consumers running on different processors.

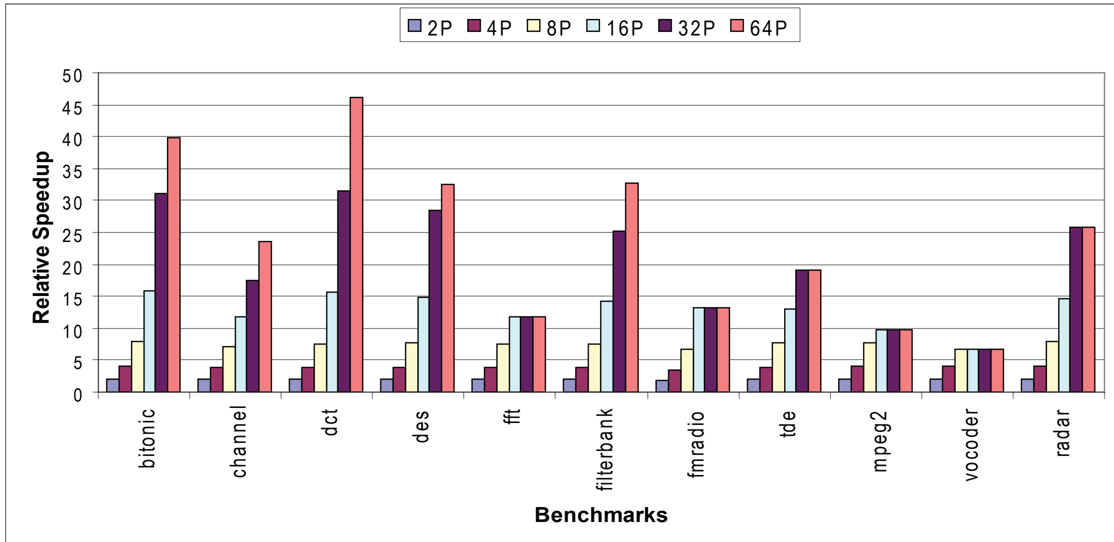


Figure 3.3: Theoretical speedup for unmodified programmer-conceived stream graph.

The partitioning problem. Figure 3.3 shows the theoretical speedup possible for a set of unmodified stream programs for 2 to 64 processors.[1] The filters present in the programmer-conceived stream graph are assigned to processors in an optimal fashion such that the maximal load on any processor is minimized. Speedup is calculated by dividing the single processor runtime by the load on the maximally loaded processor. The programmer-conceived stream graph has ample parallelism that can be exploited on up to 8 processors. Beyond 8 processors, the speedup begins to level off. Most benchmarks just do not have enough filters to span all processors. For example, `fft` has only 17 filters in its stream graph, therefore no speedup is possible beyond 17 processors. The other reason is that work is not evenly distributed across the filters. Even though the computation has been split into multiple filters, the programmer has no accurate idea of how long a filter's work function will take to

[1]More details of the applications are provided in Section 3.5.

execute on a processor when coding the function. This combined with the fact that work functions are indivisible units leads to less scaling on 16 or more processors. For example, in the `vocoder` benchmark, the longest running filter contributes to 12% of the work, thus limiting the theoretical speedup to $\frac{100}{12} = 8.3$.

Most of the benchmarks are completely stateless, i.e., all filters are data parallel [26]. In fact, only `mpeg2`, `vocoder`, and `radar` have filters that are stateful. Data parallel filters can be replicated (or fissed) any number of times without changing the meaning of the program. The longest running filter in `vocoder` benchmark is stateless, and can be fissed to reduce the amount of work done in a single filter. Fissing data parallel filters not only allows work to span more processors, it also allows work to be evenly distributed across processors by making the largest indivisible unit of work smaller.

Even though data parallel filters provide ample opportunity to divide up work evenly across processors, it is not obvious how many times a filter has to be fissed to achieve load balance. An actual partitioning has to be performed to decide if filters have been fissed enough number of times. On the other hand, a good partitioning is achieved only when filters have been fissed into suitably small units. This circular cause and consequence warrants an integrated solution that considers the problems of fission and partitioning in a holistic manner.

Communication overhead. When a filter that produces data and the filter(s) that consume that data are mapped to different processors, the data must be communicated to the consumers. In our implementation on the Cell system, filters are

mapped to the SPEs that have disjoint address spaces. Therefore, communicating data to consumers is through an explicit DMA. When such transfers are not avoided, or not carefully overlapped with useful work, the overhead could dominate the execution times.

Section 3.3 presents a simple scheme which naïvely unfolds the entire stream graph. This scheme serves as the baseline for comparison. The Section 3.4 addresses the problem of partitioning and communication overhead. First, an integrated fission and partitioning method is presented that fisses the filters just enough to span all processors, and also obtain an even work distribution. Next, the stage assignment step divides up the filters into pipeline stages in which all communication is overlapped with computations.

3.3 Naïve Unfolding

This technique is based on a simple observation: when all filters in a stream program are stateless, the graph can be unfolded P times (where P is the number of available processors) and each copy of the graph can be run on one of the processors without incurring any communication overhead, and thus achieving a speedup of P. Unfolding [53] refers to the process of making multiple copies of the stream program and is analogous to unrolling a loop in traditional compilation. Many realistic stream programs do have filters with persistent state. Unfolding such graphs introduces new data dependencies between different copies of a stateful filter. The following section describes the properties of an unfolded stream graph and establishes the limits of

speedup achievable by unfolding.

3.3.1 Properties of Unfolded Stream Graph

Consider an SDFG, $G = (V, E)$. The graph G is assumed to be *rate matched*, i.e., each node $v \in V$ actually represents executing the original filter one or more times as specified by the basic repetition vector. The set V of vertices can be partitioned into two disjoint subsets V_s and V_u. V_s is the set of all filters with state. In the context of StreamIt, this set includes all filters with explicit state and all peeking filters, whose peek amount is larger than the pop amount, thus requiring implicit state. V_u is the set of all stateless filters. Now consider unfolding the graph G, n times. Unfolding basically is making n copies of the graph. More formally, a new graph $G' = (V', E')$ is created with the following properties.

- For every node $v \in V$, the graph G' has n nodes, $\{v_1, v_2, ...v_n\} \in V'$. Thus $|V'| = n \times |V|$.
- For every edge $(v, u) \in E$, the graph G' has n edges, $\{(v_1, u_1), (v_2, u_2), ...(v_n, u_n)\} \in E'$.
- For every stateful node $v \in V_s$ in the original graph G, n new edges are introduced in G'. The new edges are $\{(v_1, v_2), (v_2, v_3), ..., (v_n, v_1)\}$.

The new edges, $\{(v_1, v_2), (v_2, v_3), ..., (v_n, v_1)\} \in E'$, are introduced so that data dependencies due to persistent state can be honored. Note that the new edges force the sequentialization of the execution of consecutive instances of stateful filters. We

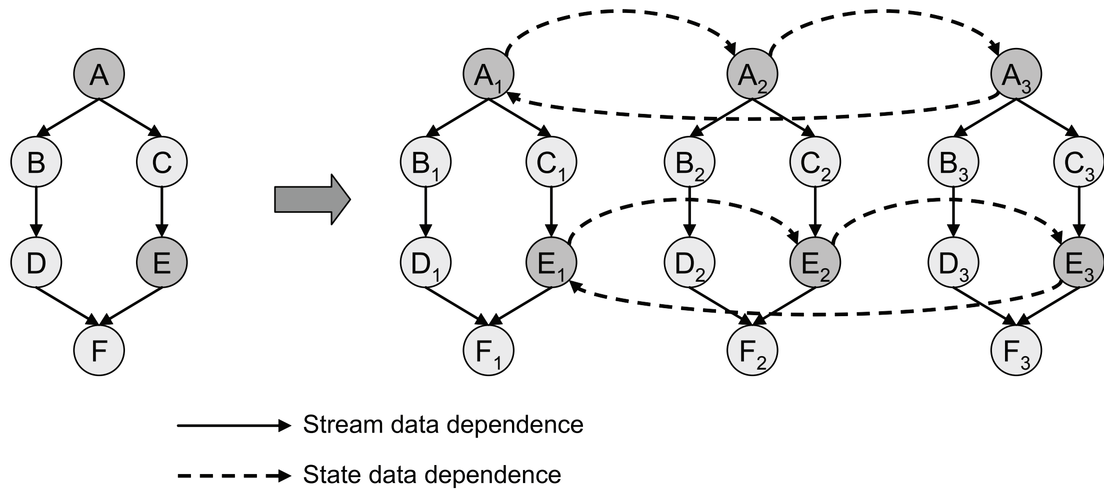


Figure 3.4: A stream graph and its unfolded version showing data dependencies introduced due to unfolding.

differentiate the data dependencies created due to unfolding from those present in original stream graph. The original stream graph has producer-consumer data dependencies, called *stream data dependencies*. The unfolded graph also has stream data dependencies as well as additional data dependencies introduced due to state data, called *state data dependence*. Note that state data dependencies are manifested in the original graph too, as a self loop on each stateful filter.

Figure 3.4 shows a stream graph, and its unfolded version, which has been unfolded 3 times. Nodes A and E shown as darker circles are stateful filters in the original graph. In the unfolded version, new edges $A_1 \rightarrow A_2, A_2 \rightarrow A_3$, and $A_3 \rightarrow A_1$ enforce dependencies due to persistent state in filter A.

Now, consider software pipelining the unfolded stream graph. The throughput of a software pipeline determines the speedup obtained, and the throughput depends on the critical cycles in the graph [40]. Let $t(v)$ be the execution time for the filter

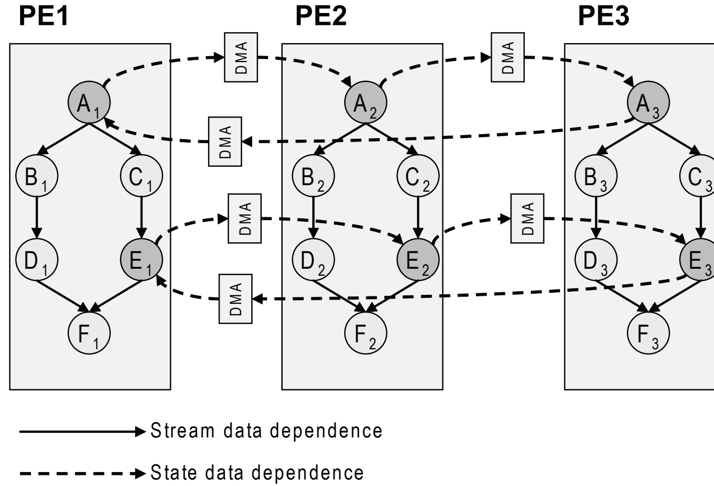


Figure 3.5: A mapping of an unfolded stream graph on to 3 processors.

$v \in V$ in the original graph. As stated in Section 3.2.1, we only consider stream programs without feedback loops, i.e., the original stream graph does not have any cycles. However, as shown in Figure 3.4, unfolding introduces cycles in the graph. In the unfolded graph, only stateful filters are involved in cycles. Therefore the critical cycle in the unfolded graph is determined by the longest running stateful filter. Thus, the length of the longest cycle can easily be calculated as

$$n \times \max_{v \in V_s} t(v)$$

where n is the number of times the graph has been unfolded. This value assumes zero communication overhead. However, a realistic software pipeline implemented on a multiprocessor system would incur communication overhead that should be filtered into critical cycle and throughput calculation. In this section, we consider a simple mapping of the unfolded stream graph on to multiple processors and establish the properties of that mapping.

Consider Figure 3.5, the original stream graph from Figure 3.4 is unfolded 3 times and mapped on to 3 processors. The mapping is such that all filters corresponding to one iteration of the original stream graph are mapped to the same processor. In this mapping, every stateful filter $v_i, i > 1$ should wait for the completion of v_{i-1} on a different processor to ensure state data dependencies are honored. Also, the stateful filter v_1 should wait for the completion of v_n. The state data must be copied from $(i-1)$-th processor before filter v_i can begin execution on processor i. This inter-processor copy has been shown as "DMA" on state data dependence edges in Figure 3.5.

Suppose $s(v)$ is the amount of time taken to transfer the state data associated with filter v. $s(v)$ depends on the size of the persistent state of filter v and the communication latency of the multiprocessor system. We assume that the size of persistent is constant and does not grow during runtime. StreamIt does not allow dynamic memory allocation, and thus this property holds for all benchmarks in our evaluation. The length of the critical path in the mapping we have adopted not only depends on the execution time $t(v)$ of a stateful filter, but also on $s(v)$, the time required to transfer the state from a different processor. For every stateful node $v \in V_s$, the mapping has a cycle of length

$$n \times s(v) + n \times t(v)$$

The longest cycle in the graph constrains the maximum throughput achievable for the graph. We adopt the terminology used in traditional instruction centric software pipelining, and refer to the critical path length as "recurrence constrained minimum

initiation interval", or RecMII. Thus, the RecMII in the unfolded graph using the mapping shown in Figure 3.5 is given by

$$RecMII = \max_{v \in V_s} (n \times s(v) + n \times t(v)) \tag{3.1}$$

The maximum throughput achievable for the graph is also constrained by the resources, in this case the limited number of processors available to execute the graph. The constraint on throughput due to resources is referred to as "resource constrained minimum initiation interval", or ResMII. In general, ResMII is calculated by performing a bin-packing of nodes on to processors, and looking at processor with maximum load. However, we already have a mapping of nodes to processors, which simplifies the calculation of ResMII. Ignoring the time to transfer the state data, the load on any processor is given by

$$\sum_{v \in V} t(v)$$

This is because each processor executes *all* filters corresponding to one iteration of the original graph. Stateful nodes incur extra overhead for data transfer in the form of DMAs. Consider a stateful filter $v \in V_s$, the i-th instance v_i of v in the unfolded graph must get the data from the $(i-1)$-th instance. We assume the filter i is responsible for this data transfer, and attribute the time needed for the DMA to processor i. We attribute the DMA required for transferring state from filter v_n to v_1, to processor 1. In steady state, each processor executes one iteration of the original graph. In addition, it also performs the data transfer for every stateful filter. Thus

every processor is equally loaded, and the load on any processor, is given by

$$ResMII = \sum_{v \in V_s} s(v) + \sum_{v \in V} t(v) \tag{3.2}$$

The best throughput for the graph using the above mapping described, referred to as the "minimum initiation interval", or MII, is simple the maximum of RecMII and ResMII given by Equations 3.1 and 3.2.

$$MII = MAX(RecMII, ResMII) \tag{3.3}$$

Suppose the number of filters in the stream program are much larger than the number of available processors, i.e., $|V| \gg P$. And suppose that work is mostly evenly distributed across filters, and there is no one filter that dominates the runtime. This is always true for realistic well written stream programs, as the programmer would have had load balance in mind while parallelizing an application into a stream program, thus not putting all the work into one large filter. With these two pre-conditions, RecMII is much smaller than ResMII because ResMII is the sum of work on all filters, whereas RecMII depends on the work on one filter. The rest of the section assumes ResMII > RecMII. Equation 3.1 is used as one of the filters that helps in choosing a scheduling method for a particular graph.

Given that MII = ResMII, in steady state, the above mapping on n processors completes n iterations in MII cycles. Thus the speedup achieved by this mapping over one processor is given by

$$Speedup \;=\; \frac{\sum_{v \in V_s} s(v) + \sum_{v \in V} t(v)}{n \times \sum_{v \in V} t(v)} \tag{3.4}$$

With small number of processors n, and a relatively larger number of stateful filters, the value $\sum_{v \in V_s} s(v)$ is going to dominate the numerator value in Equation 3.4. However, until now, the discussion has assumed that the number of times a filter is executed is specified by the *basic* repetition vector. The semantics of the SDF model allows any integer multiple of the basic repetition vector also as a valid repetition vector. We refer to this multiplier as the “coarsening filter”, as it coarsens the granularity of computation per filter.

Consider a coarsening filter k, i.e., the number of steady state executions of each filter is specified by the basic repetition vector multiplied by k. The execution time of a filter is given by $k \times t(v)$. Starting from this graph and applying the same unfolding and mapping process described above, the speedup achieved as a function of k is given by

$$Speedup(k) \;=\; \frac{\sum_{v \in V_s} s(v) + \sum_{v \in V} k \times t(v)}{n \times k \times \sum_{v \in V} t(v)} \tag{3.5}$$

Note that the value of $\sum_{v \in V_s} s(v)$ in the numerator does not have the multiplier k. This is because, using a coarsening filter of k is semantically equivalent to executing first k iterations of the stream graph in processor 1, $k+1$ to $2k$ iterations on processor 2 and so on. The first k iterations running on processor 1 do not incur any communication overhead as the copies are run sequentially on the same processor. At the end of k iterations, the amount of state that must be transferred is still a constant, and does not change with k. This is in stark contrast to stream data, which increases linearly with respect to the coarsening filter. Because the mapping described above

only cuts state data edges across processors, an arbitrarily large coarsening filter still incurs a constant communication overhead.

Using unfolding and the mapping strategy described above, the next section presents a code generation schema for the Cell processor, which faithfully implements the software pipeline obtained from the mapping.

3.3.2 Code Generation for Cell

The SPEs are independent processors with disjoint address spaces. The general code generation strategy is to spawn one thread per SPE to run code that makes calls to work functions corresponding to filters belonging to one iteration of the stream graph. The main program running on the Power processor just spawns the SPE threads and does not intervene thereafter. The rest of this section is devoted to describing the schema of per SPE code.

Figure 3.6(a) shows a StreamIt program with 4 filters connected in a pipeline. `Upsample` and `Downsample` are stateless filters, while `FIR1` and `FIR2` have implicit state due to peeking. Figure 3.6(b) shows unfolding and mapping of this program on 2 SPEs. One of the requirements for implementing this schedule is efficient synchronization. The execution of `FIR1` on SPE2 must wait for the completion of previous iteration of `FIR1` on SPE1. We chose the First-In First-out (FIFO) data structure as our synchronization primitive. SPEs synchronize by pushing and popping "tokens" from FIFOs. Tokens are just dummy integers that are used to denote the readiness of data. The FIFO implementation is blocking, i.e., a producer blocks when trying

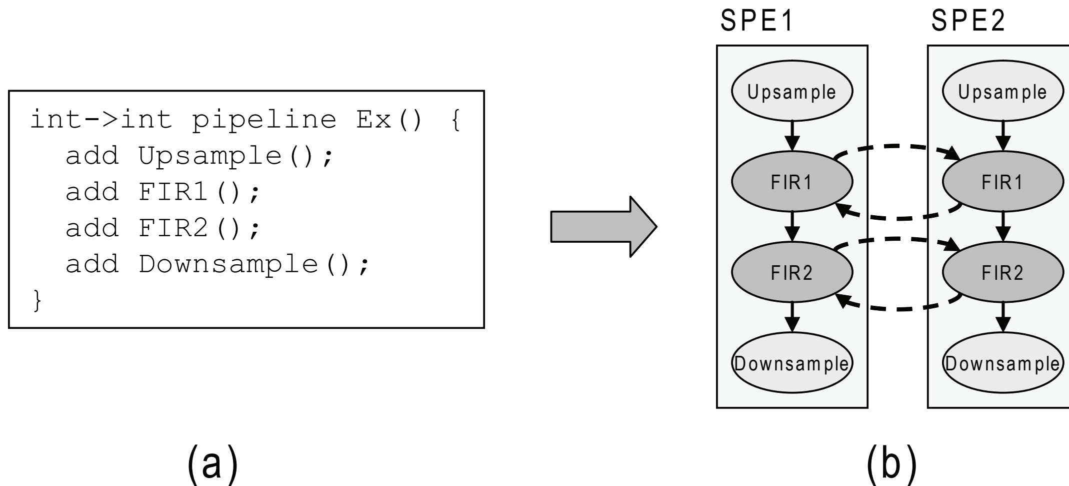


Figure 3.6: Code generation schema for Cell. (a) StreamIt program, (b) Unfolding and Mapping on 2 SPEs. (c) Multithreaded C code for Cell, with relevant synchronization and DMA copies.

to write to a full FIFO, and the consumer blocks when trying to read from an empty FIFO. The FIFO has been implemented using a simple lock-free technique [67], which ensures fast operation. By choosing a size of 1 for the FIFO, SPE2 blocks until SPE1 is done with one iteration of `FIR1`.

Figure 3.7 shows C code snippets for the threads running on two SPEs. When SPE1 runs the first iteration of the stream graph, there is no state data dependence for `FIR1` and `FIR2`. Therefore, the first iteration is peeled out, and calls are made to work functions of the filters in dataflow order. As soon as a stateful filter's work function is done, a token is pushed into the fifo to indicate that SPE2 can start copying state over, and start its own version of the filter's work function. `token_push_fir1()` in `spe1_code()` performs this operation. All instances of `FIR1` on SPE2 have to wait for previous instance of `FIR1` on SPE1. `token_pop_fir1()` on SPE2 waits for the

```
spe1_code() {
  Upsample();

  FIR1();
  token_push_fir1();

  FIR2();
  token_push_fir2();
  Downsample();

  for(i=1; i<max_iter; i++) {
    Upsample();

    token_pop_fir1();
    dma_get_state_fir1();
    FIR1();
    token_push_fir1();

    token_pop_fir2();
    dma_get_state_fir2();
    FIR2();
    token_push_fir2();

    Downsample();
  }
}
```

```
spe2_code() {

  for(i=0; i<max_iter; i++) {
    Upsample();

    token_pop_fir1();
    dma_get_state_fir1();
    FIR1();
    token_push_fir1();

    token_pop_fir2();
    dma_get_state_fir2();
    FIR2();
    token_push_fir2();

    Downsample();
  }
}
```

Figure 3.7: Multithreaded C code for Cell, with relevant synchronization and DMA copies.

completion of `FIR1` on SPE1, and `dma_get_state_fir1()` copies over the relevant state information from SPE1. After the work function `FIR1()` is called on SPE2, `token_push_fir1()` signals SPE1 so that the next instance of FIR1 can be started. Similar synchronization is performed before and after calls to FIR2. Thus, in steady state, each SPE waits for the previous SPE before calling the work function of a stateful filter. This waiting is accomplished by a blocking pop from relevant token fifo. Then, the state information is copied over using DMA, and the work function is called. After calling the work function, the SPE signals the next SPE by performing a push on the relevant token FIFO.

3.4 Stream Graph Modulo Scheduling

This section describes our method for scheduling a stream graph onto a multicore system. The objective is to obtain a maximum throughput software pipeline taking both the computation and communication overhead into account. The stream graph modulo scheduling (SGMS) algorithm is divided into two phases. The first phase is an integrated fission and processor assignment step based on an integer linear program formulation. It fisses data parallel filters as necessary to get maximal load balance across the given number of processors. The second phase assigns filters to pipeline stages in such a manner that all communication is overlapped with computation on the processors.

3.4.1 Integrated Fission and Processor Assignment

Consider a dataflow graph $G = (V, E)$ corresponding to a stream program. Let $|V| = N$ be the number of filters. Let the basic repetition vector be r, where r_i specifies the number of times v_i is executed in a static schedule. Let $t(v_i)$ be the time taken to execute r_i copies of v_i. The rest of the section assumes r_i executions of v_i as the basic schedulable unit. Given P processors, a software pipeline needs some assignment of the filters to the processors. The throughput of the software pipeline is determined by the load on the maximally loaded processor. As shown in Section 3.2, even an optimal assignment on the unmodified programmer conceived stream graph does not provide linear speedups beyond 8 processors. Some data parallel filters need to be fissed into two or more copies so that there is more freedom in distributing

work evenly across the processors. For each filter in the stream graph, the following ILP formulation comes up with the number of times the filter has to be fissed, and an assignment of each copy of the filter to a processor. The objective function is the maximal load on any processor, which is minimized.

A set of 0-1 integer variables $a_{i,j,k,l}$ is introduced for every filter v_i. The meaning of the four suffixes is explained below:

- i identifies the filter.
- j identifies the *version* of the filter that would appear in the final graph. For every filter v_i, the formulation considers multiple versions of the filter. Version 0 of the filter is fissed 0 times (no copies made), version 1 of the filter is fissed once so that two copies of the filter are considered for scheduling, and so on.
- k identifies the copy of the jth version of the filter v_i. Version 0 has only one copy. Version 1 has 2 copies of the filter and a splitter and joiner. The splitter and joiner have to run on some processor, therefore, they are considered as independent schedulable units. Thus there are $(j+3)$ schedulable filters in the jth version. We have either $0 \leq k < j+3$ when $j \geq 1$, or $k = 0$ when $j = 0$.
- l identifies the processor to which the kth copy is assigned.

Let Q be the maximum number of versions considered for a filter. Filters with carried state cannot be fissed at all and $Q = 1$ for such filters. On the other hand, stateless filters can be fissed any number of times. The choice of Q affects the load balance obtained from the processor assignment. Choosing a low value for Q would

inhibit the freedom of distributing copies of a filter to many processors. We observed that the maximum number of copies of a filter that appear in the best partitions is always less than P for all benchmarks. Therefore, in the experiments Q was set to P, the number of processors under consideration. The following equation ensures that a copy of an filter is either assigned to one processor or not assigned to any processor at all, implying that a different version was chosen.

$$\sum_{l=1}^{P} a_{i,j,k,l} \leq 1 \quad \forall i, 0 \leq j < Q, 0 \leq k < j+3 \tag{3.6}$$

When a copy of a filter is indeed assigned to a processor, all other copies in the same version have to be assigned to processors, and all other versions should not be assigned to processors. To ensure this, a set of Q indicator variables, $b_{i,q}, 0 \leq q < Q$, are introduced for every filter v_i. These indicator variables are 0-1 variables which serve two purposes. First, they indicate which version of the filter was chosen. Second, by virtue of being either 0 or 1 only, ensure that either all copies of a version are assigned to processors, or no copy is assigned to any processor. The following set of equations show the relation between the indicator variables $b_{i,q}$ and the assignment variables $a_{i,j,k,l}$.

$$\sum_{l=1}^{P} a_{i,0,0,l} - b_{i,0} = 0 \quad \forall i \tag{3.7}$$

$$\sum_{l=1}^{P} \sum_{k=0}^{j+2} a_{i,j,k,l} - b_{i,j} \leq M \times b_{i,j} \quad \forall i, 1 \leq j < Q \tag{3.8}$$

$$\sum_{l=1}^{P} \sum_{k=0}^{j+2} a_{i,j,k,l} - (j+2) - b_{i,j} \geq -M + M \times b_{i,j} \quad \forall i, 1 \leq j < Q \tag{3.9}$$

M in Equations 3.8 and 3.9 is a constant that is larger than the upper bound of $\sum_{l=1}^{P} \sum_{k=0}^{j+2} a_{i,j,k,l}$. Note that Equations 3.8 and 3.9 are standard ILP tricks to ensure that

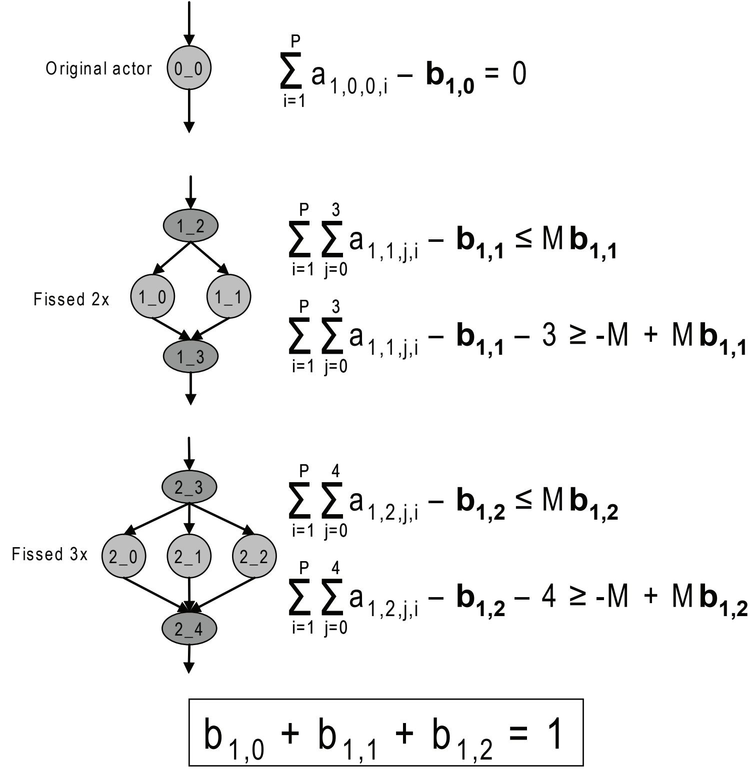


Figure 3.8: Example illustrating ILP formulation.

a linear sum either equals a constant or is zero. In this case, the sum $\sum_{l=1}^{P}\sum_{k=0}^{j+2} a_{i,j,k,l}$ either has to be $(j+3)$, denoting that all copies of a version were assigned to some processor, or has to be 0, denoting that none of the copies were assigned to any processor. $b_{i,j}$ conveniently takes on 1 or 0, respectively. The following equation ensures that one and only one version of a filter is chosen in the final assignment.

$$\sum_{j=0}^{Q} b_{i,j} = 1 \quad \forall i \tag{3.10}$$

Figure 3.8 illustrates the above set of equations for an example filter. Q is chosen to be 3 in the example. Three versions of the filter are shown in the figure. The labels on the nodes indicate the version number and copy number. The last equation

$b_{1,0} + b_{1,1} + b_{1,2} = 1$ ensures that only one version is chosen, and the rest of the equations ensure that all copies of the chosen version are assigned to processors.

To determine the quality of an assignment, the amount of work assigned to each processor has to be calculated. The following equation computes the work (in terms of time) done by a copy of a filter.

$$W_{i,j,k,l} = \begin{cases} t(v_i) & \text{if j = 0} \\ \frac{t(v_i)}{j+1} + \epsilon & \text{if j > 1 and k < j+1} \\ splitter_work(v_i) & \text{if j > 1 and k = j+1} \\ joiner_work(v_i) & \text{if j > 1 and k = j+2} \end{cases} \tag{3.11}$$

Version 0 of the filter is same as the original filter. Therefore, the work done by version 0 is the original work $t(v_i)$. In version 1, there are 2 copies of the filter that do half the work as the original filter. Note that there is a small overhead of ϵ when fissing filters which peek more elements than they pop. This is due to the introduction of a decimation stage on each copy which just pops and ignores part of the data to maintain correct semantics. In addition, there is additional work done by the splitter and joiner in version 1. The last three cases in Equation 3.11 compute the work done by copies of the filter, splitter, and joiner. Note that the work done in splitter and joiner depends on the implementation. However, they both are constants given the number of items popped by the corresponding filter. For some assignment of filters to processors, the following equation computes the total work TW_p that gets assigned

to a processor p.

$$TW_p = \sum_{i=1}^{N} \sum_{j=0}^{Q} \sum_{valid\ k} a_{i,j,k,l} \times W_{i,j,k,p} \tag{3.12}$$

The processor p with maximum work TW_p assigned to it constitutes the bottleneck processor, and thus TW_p denotes the inverse of the throughput of the overall pipeline. We borrow the terminology from operation-centric modulo scheduling used in compiler backends, and use the term Initiation Interval (II) to denote the inverse of the throughput. The following set of equations compute II from the TW_p's.

$$TW_p \leq II \quad 1 \leq p \leq P \tag{3.13}$$

The ILP program that minimizes II subject to constraints given by Equations 3.6 to 3.13 provides the following information.

- The value of j for which $b_{i,j} = 1$ identifies the version of the filter chosen. Note that Equation 3.10 ensures that only one of the $b_{i,j}$'s have the value 1.
- Given a copy k of the chosen version j, the set of values $a_{i,j,k,l}$ that are 1 identify the processors to which the copy is assigned. For example, if $a_{i,j,k,4} = 1$, then the kth copy the filter is assigned to processor 4.

The above formulation does not account for any communication overhead. The data produced by a filter has to be communicated to a consuming filter if that filter was assigned to a different processor. The following section shows how all such communication can be hidden, thus achieving the exact throughput obtained from the processor assignment step.

3.4.2 Stage Assignment

The processor assignment obtained by the method described in the previous section provides only partial information for a pipeline schedule. Namely, it specifies how filter executions are overlapped across processors. It does not specify how they are overlapped in time. To realize the throughput, which is the load on the maximally loaded processor obtained from processor assignment, all filters assigned to a processor including the necessary DMAs have to be completed within a window of II time units. The only goal of processor assignment step is load balance, therefore it assigns filters to different processors without taking any data precedence constraints into consideration. An filter assigned to a processor could have its producer assigned to a different processor, and have its consumer assigned to yet another processor. To honor data dependence constraints and still realize the throughput obtained from processor assignment, the filter executions corresponding to a single iteration of the entire stream graph are grouped into *stages*. Note that the concept of stage is adapted from traditional VLIW modulo scheduling [58]. Across all processors, stages of a single iteration execute sequentially, thus honoring data dependences. Within a single processor, no stages are *active* at the beginning of execution. During the initial few iterations, stages are activated sequentially, thus filling up the pipeline and enabling executions of data dependent filters belonging to earlier iterations concurrently with filters from later iterations. In steady state, all stages are active on a processor, thus realizing the throughput obtained from processor assignment. The pipeline is drained by deactivating stages during the final few iterations.

The overarching goal of the stage assignment step is to overlap all data communication (DMAs) between filters. To achieve this, the stage assignment step considers the DMAs as schedulable units. To honor data dependences and ensure DMAs can be overlapped with filter executions, certain properties are enforced on the stage numbers of filters. Consider a stream graph $G = (V, E)$. The stage to which a filter v_i is assigned is denoted S_i. In addition, the processor to which v_i is assigned to is denoted by p_i. The following rules enforce data dependences and ensure DMA overlap.

- $(v_i, v_j) \in E \Rightarrow S_j \geq S_i$, i.e., the stage number of a consuming filter should come after the producing filter. This is to preserve data dependence.

- If $(v_i, v_j) \in E$ and $p_i \neq p_j$, then a DMA operation must be performed to get the data from p_i to p_j. The DMA operation is given a separate stage number S_{DMA}. As shown in Figure 3.9, the inequality $S_i < S_{DMA} < S_j$ is enforced between the stages of the different filters and the DMA operation. The DMA operation is separated from the producer by at least one stage, and similarly, the consumer is separated from the DMA operation by one stage. This ensures decoupling, and allows the overlap of the producer and the DMA, as well as the DMA and the consumer.

- Within the set of filters assigned to some processor p, the inequality $\sum_{S_j=s} t(v_j) \leq II, \forall s$ is enforced. In other words, the sum of execution times of filters (S_j) assigned to a stage (s) should be less than the desired II. This is the basic modulo scheduling constraint, which ensures that the stages are not overloaded, and that a new iteration can be initiated every II time units.

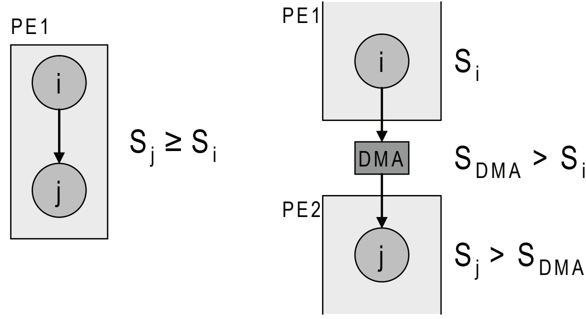


Figure 3.9: Properties of stages.

A simple data flow traversal of the stream graph is used to assign stages to filters as shown in Algorithm 1. For each filter in dataflow order, the **FindStage** procedure assigns a stage to the filter. The `for` loop beginning on the line marked 1 computes the maximum stage of the producers of the filter under consideration. If any of the producers are assigned to a different processor, the earliest stage considered for filter is $maxstage + 2$, which leaves room for DMAs in $maxstage + 1$. Otherwise, the filter could be placed on $maxstage$. The `while` loop beginning on the line marked 4 finds a stage number later than $stage$ on which the load is less than the II obtained from processor assignment.

3.4.3 Code Generation for Cell

This section describes a code generation strategy to implement the modulo schedule obtained for a stream program on a Cell system. The target of our code generation are the multiple SPEs, as opposed to the PPE. This section describes the general code generation schema, the buffer allocation strategy, and provides a complete example.

```
FindStage (filter) :
maxstage ← 0 ;
flag ← false ;
1 foreach producer p of filter do
    if stage(p) > maxstage then
        maxstage ← stage(p) ;
    end
2   if Proc(p) ≠ Proc(filter) then
        flag ← true ;
    end
end
3 if flag then
    stage ← maxstage + 2 ;
else
    stage ← maxstage ;
end
4 while Load(Proc(p), stage) + t(filter) > II do
    stage ← stage + 1
end
Load(Proc(p), stage) += t(filter) ;
return stage
```

Algorithm 1: Stage assignment procedure

Code generation schema. The SPEs are independent processors with disjoint address spaces. The general code generation strategy is to spawn one thread per SPE. Each thread makes calls to work functions corresponding to filters that are assigned to the respective SPEs, and perform DMAs to get data from other SPEs. The main program, running on the PPE, just spawns the SPE threads and does not intervene thereafter.

Figure 3.10 shows pseudo C code that runs on each SPE thread. It mimics the kernel-only [59] code of modulo scheduling for a VLIW processor. The array `stage` functions similar to the *staging predicate*, and its size (`N`) is the maximum number of stages. The main loop starts off with only the first stage active. The `if` conditions that test different elements of `stage` ensure only filters assigned to a particular stage are executed. The last part of the loop *shifts* the elements of the array `stage` to the left, which has the effect of filling up the software pipeline. Finally, when all iterations are done, draining the software pipeline is accomplished by shifting a `0` into the last element of `stage`.

The code corresponding to each active stage are calls to the work functions of the filters assigned to this SPE and the corresponding stage, and the necessary DMAs to fetch data from other SPEs. The Cell processor provides non-blocking DMA functionality [32], which is leveraged for overlapping DMAs and computation. A DMA operation assigned to a particular stage is implemented using the `mfc_get` primitive, which enters the DMA command into a queue and returns immediately. The MFC engine in each SPE processes the queue asynchronously and independent of the

```
void spe_work()
{
  char stage[N] = {0};

  stage[0] = 1;

  for (i=0; i<max_iter+N-1; i++) {
    if (stage[N-1]) {
      // Begin DMA operations
      // Call to filter work functions
    }
    if (stage[N-2]) {
    }
    ...
    if (stage[0]) {
    }
    wait_for_dma_completion();
    // start epilogue
    if (i == max_iter-1)
      stage[0] = 0;
    // Shift-left staging predicate
    for(j=N-1; j>=1; j--)
      stage[j] = stage[j-1];
    barrier();
}
```

Figure 3.10: Main loop implementing the modulo schedule.

processor. After enqueuing the DMA request, the code proceeds to execute work functions for filters. Note that even though the actual DMA operations are asynchronous, the SPE should queue up the DMA requests synchronously using the `mfc_get` primitive. No more DMAs can be queued once work functions begin execution. Therefore, all DMA operations belonging to a stage are queued up before any work functions are called to ensure maximal overlap of actual DMAs and computation. Finally, the `wait_for_dma_completion` uses the `mfc_read_tag_status_all` primitive to ensure all

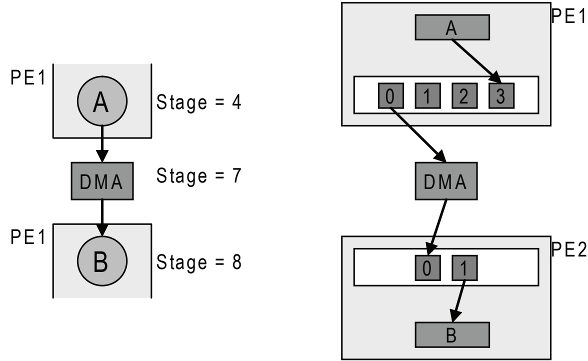


Figure 3.11: Buffer allocation for the modulo schedule.

DMAs issued in the current iteration are completed, and a barrier synchronization is executed to ensure the current iteration is completed on all SPEs. `barrier()` is implemented using the `signal` mechanism available on the SPEs, and with the current implementation, 2×10^6 barriers can be performed in 1 second.

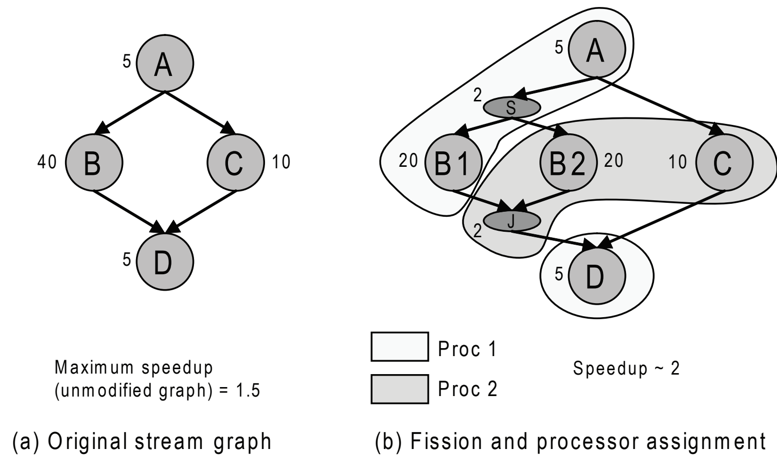
Buffer allocation. In the code generation schema described above, several iterations of the original stream graph are in flight concurrently. A producer filter could be executed multiple times before one of its consumers is ever executed. To ensure correct operation, multiple buffers are used to store the outputs of producer filters. The buffers are used in a fashion similar to rotating registers in a traditional modulo schedule. The number of buffers needed for the output of a producer filter assigned to stage S_p feeding a consumer filter on stage S_c can easily be calculated as $S_c - S_p + 1$.

Figure 3.11 shows the buffer allocation for a producer filter A and consumer filter B. They are assigned to different processors with an intervening DMA. Since the stage separation between A and the DMA is 3, 4 buffers are allocated on the local memory of PE1, and A uses them in a round-robin fashion. The arrows on the picture on the

right shows the current buffers being used. Note that the DMA operation and filter A are executing concurrently by using different buffers. Similarly, B is using a buffer different from the DMA. In the current implementation, all buffers are allocated on the local memories of the SPEs. The buffers between a producer filter and a DMA operation are stored on the SPE on which the producer is running. Symmetrically, the buffers between the DMA operation and the consuming filter are stored on the consumer SPE. 256KB of local store is sufficient to hold all the buffers needed by the benchmarks evaluated. This is corroborated by the authors of [26], who report that the buffers needed by the benchmarks would fit on the 512KB cache of the Cell processor.

3.4.3.1 Example

Figure 3.12(a) shows an example stream graph. Assume that all filters in the graph are data parallel, i.e., they can be fissed any number of times. The numbers beside the nodes represent the amount of work done by the filters. Note that B does the most work of 40 units and the sum of work done by all filters is 60 units. When trying to schedule the unmodified graph on to 2 processors, the maximum achievable speedup is $\frac{60}{40} = 1.5$. Figure 3.12(b) shows the result of the integrated fission on processor assignment step. Node B has been fissed once, resulting in two new nodes $B1$ and $B2$, and the corresponding splitter S and joiner J, whose work are assumed to be 2 units. The processor assignment obtained has an II of 32, thus resulting in a speedup of $\frac{60}{32} \sim 2$. Finally, Figure 3.12(c) shows the stage assignment in which



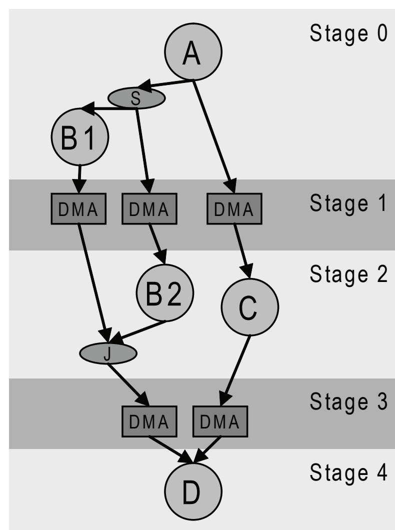


Figure 3.12: Example illustrating fission, processor assignment and stage assignment.

DMAs are separated from consumers by one stage, thus ensuring complete overlap of computation and communication.

Figure 3.13 shows the execution timeline of the code running on two SPEs. The main feature to note is the steady state execution, which starts from the 5th iteration in Figure 3.13. In the steady state, all filters and all DMAs are active. The 4 iterations shown before the steady state correspond to the prologue of the modulo schedule, in

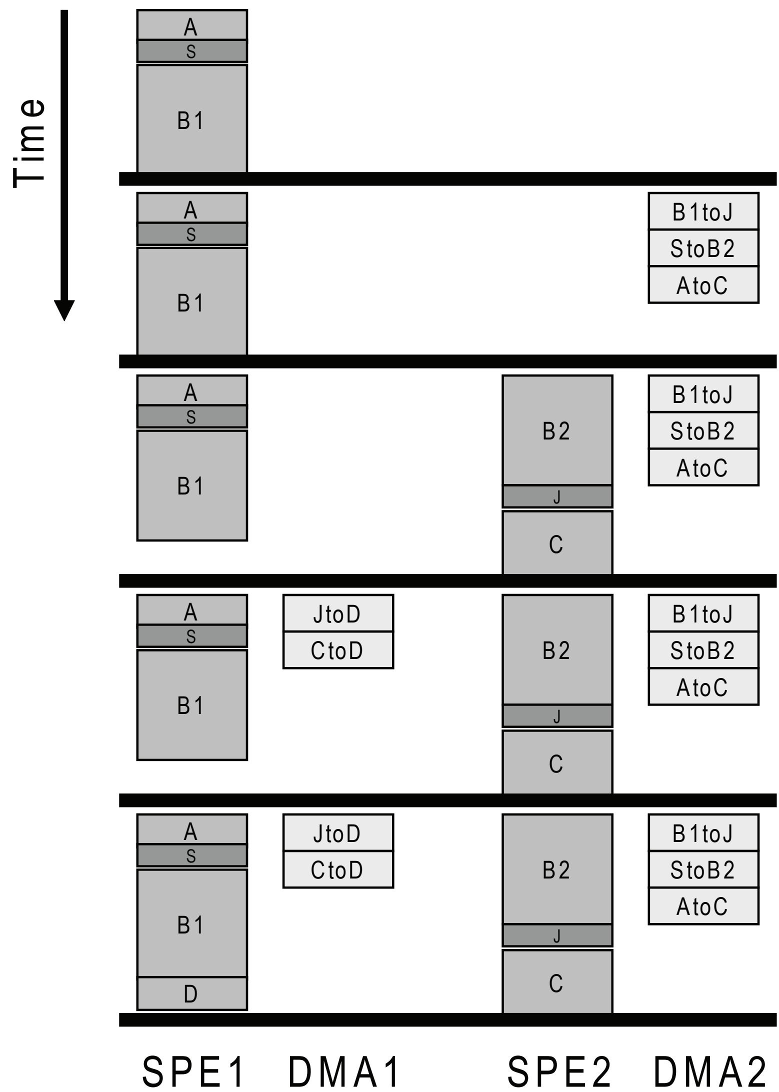


Figure 3.13: Example illustrating a modulo schedule running on Cell.

which some filter executions and DMAs do not happen as they are predicated by the `stage` array. The DMA operations are started before filter executions on the SPEs, thus ensuring overlap with computation. Due to the overlap, the purported speedup of 2 is achieved by the schedule.

The main differences between naïve unfolding and SGMS can be summarized as below.

- All DMA transfers of stream data can be overlapped with computation in SGMS where as DMA transfers of state data cannot be overlapped with any computation as it is present in the critical path.

- In the naïve unfolding method, each SPE runs all filters in the original stream graph, whereas in SGMS, an SPE runs only a subset of the filters. Therefore, the memory footprint of code for naïve unfolding is much larger than for SGMS.

- The latency for one iteration of the original stream graph is equal to the uniprocessor execution time of an iteration in the naïve unfolding method. This is because all filters belonging to one iteration is executed sequentially by an SPE. In contrast, task level parallelism is exploited within an iteration in SGMS, and therefore, the latency for an iteration could be much smaller.

Despite the shortcomings compared to SGMS, naïve unfolding is a simple method which requires no sophisticated compiler analyses, and is straightforward to implement for the Cell processor. We compare SGMS with naïve unfolding in the following section.

3.5 Evaluation

This section evaluations of various aspects of SGMS, including a comparison to naïve unfolding.

3.5.1 Experiments

This section presents the results of the experimental evaluation of SGMS, and comparison to the naïve unfolding method. A uniprocessor schedule was first generated for one SPE, with instrumentations added for measuring running time of each filter. The SPU "decrementer", a low overhead timing measurement mechanism, is used for profiling. The timing profile for each filter is used by the SGMS scheduler that generates schedules for 2-16 processors. The scheduler uses the CPLEX mixed integer program solver during the integrated fission and processor assignment phase. The code generation phase outputs plain C code that is divided into code that runs on the Power processor and code that runs on individual SPEs. The main thread running on Power processor spawns one thread per SPE. Each SPE thread executes a code pattern that was described in Section 3.4.3. IBM's Cell SDK 2.1 was used to implement the DMA copies, and the barrier synchronization. The GNU C compiler gcc 4.1.1 targeting the SPE was used to compile the programs. Note that only vectorization that was automatically discovered by gcc were performed on the filters' codes. The hardware used for our evaluation is an IBM QS20 Blade server. It is equipped with 2 Cell BE processors and 1 GB XDRAM.

Benchmark	Filters	Stateful	Peeking	State size (bytes)
bitonic	28	2	0	4
channel	54	2	34	252
dct	36	2	0	4
des	33	2	0	4
fft	17	2	0	4
filterbank	68	2	32	508
fmradio	29	2	14	508
tde	28	2	0	4
mpeg2	26	3	0	4
vocoder	96	11	17	112
radar	54	44	0	1032

Table 3.1: Benchmark characteristics.

Benchmark suite. The set of benchmarks available with StreamIt software version 2.1.1 was used to evaluate the scheduling methods. Most benchmarks are from the signal processing domain. `bitonic` implements the parallel bitonic sorting algorithm. `des` is a pipelined version of DES encryption cipher. [26] provides descriptions of the benchmarks. Table 3.1 shows the details relevant to our evaluation. Number of stateful filters with explicit state and peeking filters with implicit state are important to understand the speedups from naïve unfolding. Typical sizes of states in these benchmarks are also shown.

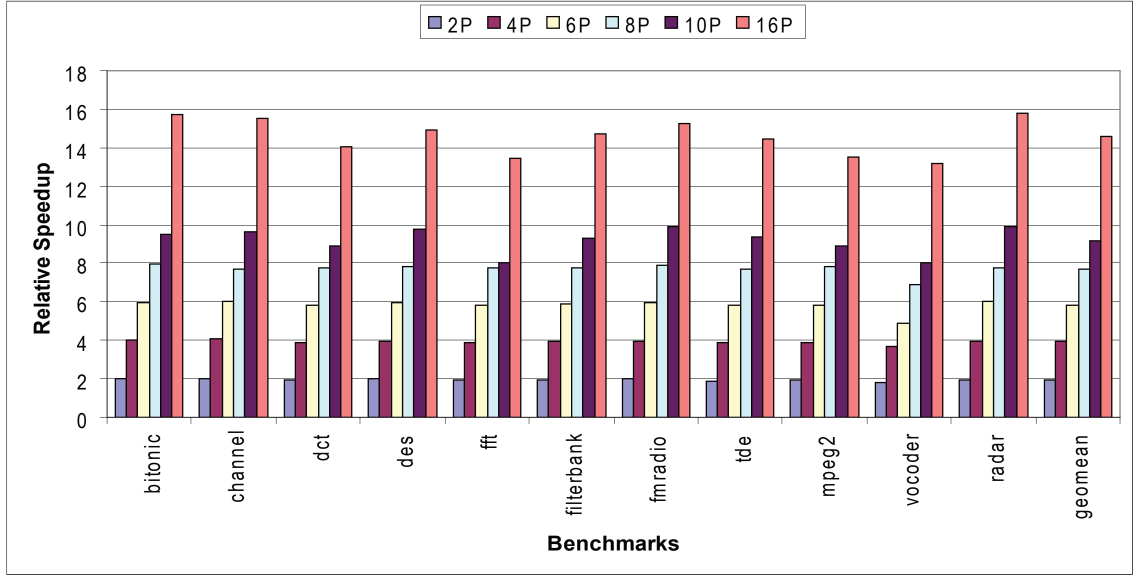


Figure 3.14: Stream-graph modulo scheduling speedup normalized to single SPE.

SGMS performance. Figure 3.14 shows the speedups obtained by SGMS over single processor execution on 2 to 16 processors for the benchmark suite. SGMS obtains near linear speedup for all benchmarks, resulting in the geometric mean speedup of 14.7x on 16 processors. The main reasons for near linear speedups are listed below.

- The integrated fission and partitioning step fisses enough data parallel filters and the resulting number of filters is enough to span all available processors.
- The partitioning assigns filters to processors with maximal load balance.
- Stage assignment separates data transfers and filters that use the data into different stages. This ensures that all data transfers are overlapped with computation.

Note that with perfect load balance and complete overlap of all communication with computation should always result in a speedup of N on N processors. However, the

observed geometric mean speedup is only 14.7x on 16 processors. One of the main overheads in our implementation arises from the barrier synchronization. As shown in Figure 3.10, all SPEs do a barrier synchronization at the end of every iteration of the loop implementing the modulo schedule. Our implementation of the `barrier` on the SPEs adds an overhead of 1 second for every 2×10^6 calls. Depending on the number iterations the stream graph is executed, barrier synchronization adds an overhead of up to 3 seconds in some benchmarks. A notable benchmark is `vocoder` for which the 16 processor speedup is only 13x. `vocoder` has 96 filters in the stream graph. On 16 processors, the partitioning results in over 30 DMA operations being in flight at the same time, which adds some overhead to the steady state. SGMS relies on static work estimates during the partitioning phase. Any deviation from the static estimate during runtime would change the balance of work across processors and cause a reduction in speedup. However, this effect is difficult to quantify.

Comparing naïve unfolding to SGMS. Figure 3.15 compares the speedup obtained by SGMS and naïve unfolding on 16 processors. There are 3 bars per benchmark. The first bar is the speedup obtained by naïve unfolding for the original stream graph. The second bar is the speedup obtained by naïve unfolding on the same set of benchmarks, but with the size of state variables artificially increased by 16x compared to the original implementation. The last bar the speedup obtained by SGMS for the original stream graph. Figure 3.15 has to be correlated with Table 3.1 for better understanding. For benchmarks that are almost completely stateless, such as `dct`, `des` and `mpeg2`, naïve unfolding achieves over 15.5x speedup on 16 processors.

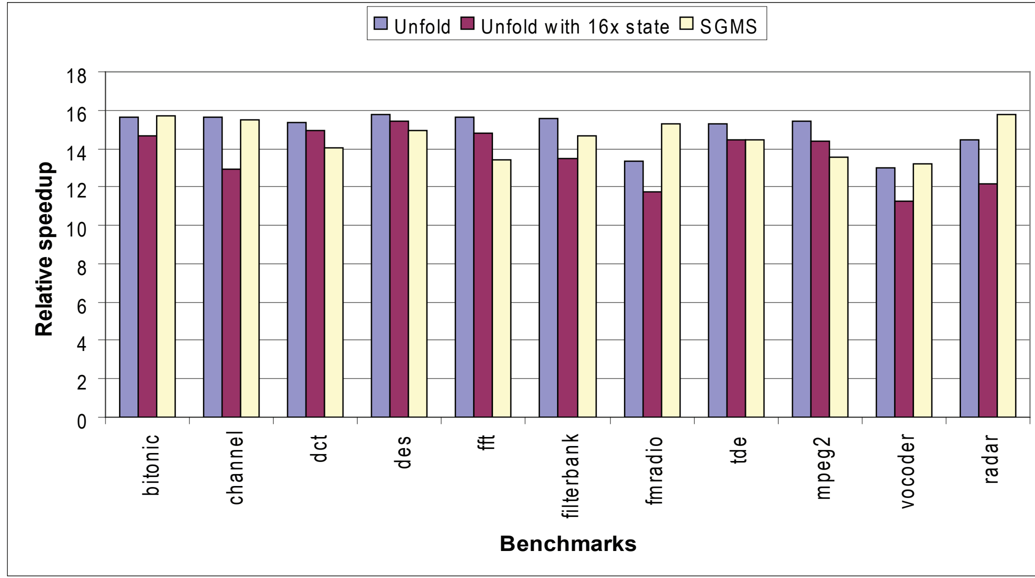


Figure 3.15: Comparing naïve unfolding to SGMS.

This is not surprising as independent iterations run on different processors without any communication. Note that each benchmark nominally has 2 stateful filters, which are the input and output filters. These are used for preserving program order. The small amount of communication needed for these two stateful filters adds very little overhead, and thus completely stateless stream programs achieve close to 16x speedup on 16 processors. The SGMS method for these programs does not unfold the stream graph completely, but only fisses enough filters to get an even work distribution. The selective fissing adds extra splitters and joiners that add non-zero overhead to the steady state. Also, SGMS uses a barrier synchronization at the end of each iteration, whereas in naïve unfolding, the stateful filters perform a point to point synchronization. Because of these two facts, naïve unfolding performs 5-10% better than SGMS for completely stateless stream programs.

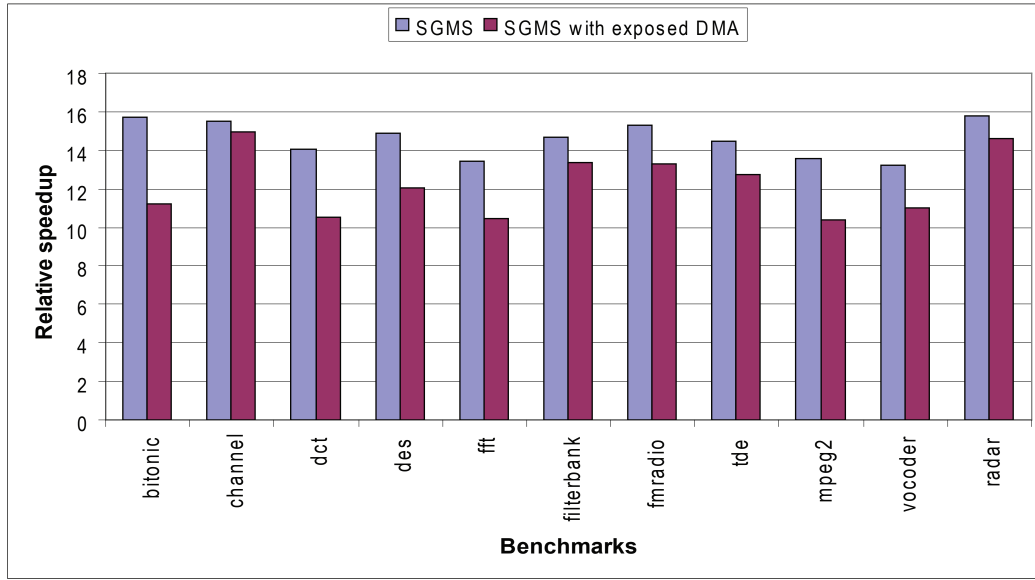


Figure 3.16: Effect of exposed DMA latency.

For stream programs with many stateful and peeking filters, such as `vocoder`, `radar`, and `fmradio`, SGMS outperforms naïve unfolding by up to 20%. The DMA transfer of state data in naïve unfolding is completely exposed as it is in the critical path. However, all DMA transfers of stream data are overlapped with computation in SGMS. The exposed DMA overhead for naïve unfolding is more pronounced when the state size is artificially increased to 16x the original state size. In this case, SGMS, whose performance is unaffected by the state size increase, outperforms naïve unfolding by up to 35%.

Effect of exposed DMA latency. Figure 3.16 illustrates the effectiveness of computation/communication overlap. For each benchmark, a version of the C code for SPEs was generated in which the data transfer overhead was completely exposed. For this case, the stage assignment did not separate the DMA operation and the

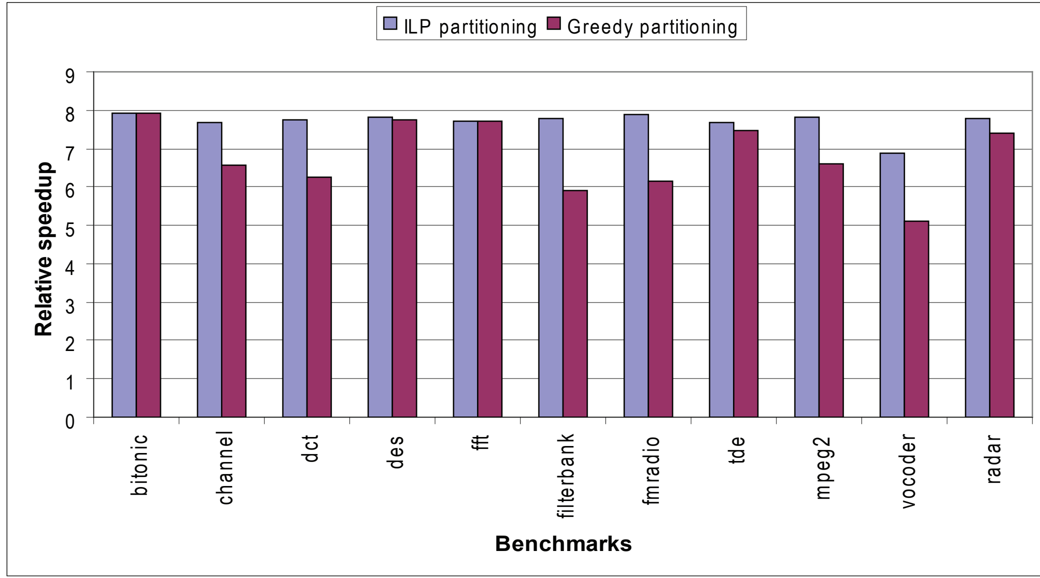


Figure 3.17: Comparing ILP partitioning to greedy partitioning.

consumer filter into different stages. Rather, they were put in the same stage and the consumer SPE stalls until the DMA operation is completed. The effect of exposed DMA latency is detrimental for all benchmarks. For `channel`, `filterbank`, and `radar`, which have high computation to communication ratios, the effect is not very pronounced and they retain most of their speedups even with exposed DMA latency. `bitonic` and `des` have low computation to communication ratios, and they suffer up to 25% perfomance loss when the DMA latencies are exposed.

Comparing ILP partitioning to greedy partitioning. The integrated fission and processor assignment phase is in part an optimal formulation for bin packing. In addition to deciding how many times each filter has to be fissed, this phase also does the assignment with maximal load balancing. Figure 3.17 compares the optimal formulation with a greedy heuristic. We only compare the 8 processor speedup. This

is because the programmer conceived stream graph already has enough parallelism to span 8 processors as shown in Figure 3.3 and the fission part of the formulation does not fiss any filters. Thus, Figure 3.17 effectively compares an optimal bin packing formulation to a greedy strategy. We use the Metis [34] graph partitioner as our greedy strategy. The original stream graph is partitioned into N parts using Metis, where N is the number of processors. The same work estimates are used as weights on the nodes of the graph. Note that this greedy partitioning is similar to the one used in [26]. In [26], a separate communication stage is introduced between steady states to shuffle data between banks. However, to make the comparison fair, the same algorithm for stage assignment is used in both cases which overlaps all DMA transfers with computation. Figure 3.17 shows that the quality of graph partition using a greedy method depends greatly on the structure of the graph. For example, `fft` and `tde` are just linear graphs with no splitters or joiners. For these cases, the greedy graph partitioner is able to achieve the same load balance as the optimal partitioner. For highly parallel graphs like `filterbank` and `vocoder`, heuristics perform up to 35% worse than an optimal formulation. Overall, the optimal partitioner achieves a geometric mean speedup of 7.6x, whereas the greedy partitioner achieves 6.7x on 8 processors.

Scaling of ILP formulation. The `vocoder` benchmark is used to study how the CPLEX solver runtimes scales when trying to partition the graph for 2 to 128 processors. `vocoder` is the largest benchmark in the suite, and the solver runtimes are smaller for all other benchmarks. The solver run times were under 30 seconds for

up to 16 processors. The time taken for partitioning on 32, 64 and 128 processors were 2, 6, and 16 minutes, respectively on a Intel Pentium D running at 3.2GHz. Appendix A further explores the scalability of the ILP formulation. Symmetries in the formulation are exploited by adding symmetry breaking constraints. As much as 25% improvement in the solver run time was observed by adding a reduced set of symmetry breaking constraints to the problem.

3.6 Related Work

There is a large body of literature on synchronous dataflow graphs, on languages to express stream graphs, and methods to exploit the parallelism expressed in stream graphs. Even though SDF is a powerful explicitly parallel programming model, its niche has been in DSP domain for a long time. Early works from the Ptolemy group [43, 42, 41] has focused on expressing DSP algorithms as stream graphs. Some of their scheduling techniques [54, 29] have focused on scheduling stream graphs to multiprocessor systems. However, they focus on acyclic scheduling and do not evaluate scheduling to a real architecture.

There has been other programming systems based on the stream programming paradigm, and each of those systems have compilers which target multiprocessors. [28] maps StreamC to a multithreaded processor. This was more of a feasibility study, and the scheduling was done manually. In [69], the authors map the Brook language to a multicore processor. They make use of affine partitioning techniques which are more suitable for parameterized loop based programs. With StreamIt, the stream graph

is completely resolved at compile time, and a direct scheduling technique like ours is more effective. Note that any stream programming system in which the computation can be expressed as an stream graph could utilize our scheduling method.

There has been a recent spur of research in the domain of compiling to the Cell processor. CellSs [7] is a stylized C model for programming the cell. The computation is expressed as functions which make all their inputs and outputs explicit in terms of parameters. Functions can be stringed together to form a data flow graph. A run time scheduler treats this graph in the same way a superscalar processor treats operations, and schedules these functions on to the cell SPEs as soon as their inputs are ready. Our work is distinctly different from theirs in that, we use a static compile time schedule which does not have run time scheduling overheads. [35] talks about compiling the Sequoia language to the Cell processor. This paper's focus is more on representing machines with multiple levels of memories, possibly with disjoint address spaces, in a reusable way, and a compiler to automatically target such representations. Our work focuses more on the actual scheduler, and assumes a fixed machine. [9] talks about parallelizing a specific application at multi levels of granularity on the Cell processor. This is more of an experiences paper, and the parallelization was done manually.

The problem scheduling coarse-grain filters to processors on a multicore with distributed memory is conceptually similar to scheduling operations to the function units in a multicluster VLIW processor [58, 60]. However, stream graph exposes more optimization opportunities such as the ability to fiss filters. Also, the constraints of limited register space is not an issue on multicores as there is ample memory avail-

able to hold the intermediate buffers.

CHAPTER 4

Memory Management for Stream Graph Modulo Scheduling

4.1 Introduction

Stream Graph Modulo Scheduling presented in Chapter 3 targeted the Cell processor. The data engines (SPEs) on the Cell processor have 256KB of memory each, which was sufficient to hold the data needed by the SGMS schedule. Embedded processor tend to have much less memory on-chip due to area and power concerns. This Chapter extends SGMS by proposing optimizations for mapping stream programs to systems with constrained memory. Concurrent execution of filters invariably results in live data that need to be stored and communicated. On a platform like Cell with multi-level hierarchical memory, placing the data close to the processing elements ensures low latency transfers. The buffer allocation strategy presented in this chapter chooses a memory hierarchy level for each block of live data. Live blocks are placed

on local stores of the SPEs as much as possible to leverage low latency communication between SPEs. Blocks are copied over to main memory when local store overflows. The strategy presented in this chapter decouples the placement of blocks at producer and consumer sites so that local memories on both sites can be fully utilized. Local to local memory, global to local memory and global to global memory transfers are considered simultaneously to minimize the amount of exposed transfer latency. First, the schedule obtained through stream graph modulo scheduled in transformed to reduce memory requirements, without sacrificing throughput. This involves minimizing the number of cuts to data flow edges. This phase evens out the memory required on each SPE. Then, an efficient integer linear programming formulation is used to allocate the live blocks on either the local or global memory. The ILP formulation minimizes the exposed transfer latency across processors while ensuring that the local memory capacities are not exceeded. Experiments on the Cell processor show that the memory allocation strategy is able to increase performance by up to 120% when memory constraints are removed from a system with nominal memory to run the software pipeline. On the other hand, the method is also able to maintain most of the performance as more and more memory constraints are added to the system.

4.2 Background

4.2.1 Memory System of the Cell Broadband Architecture

The Cell Broadband Engine (CBE) is a heterogeneous multicore system, consisting of one 64bit PowerPC core called the Power Processing Element (PPE) and eight Synergistic Processing Elements (SPEs). Each SPE has a SIMD engine called the synergistic processing unit (SPU), 256 KB of local memory and a memory flow control (MFC) unit which can perform DMA operations to and from the local stores independent of the SPUs. The property of CBE most pertinent to this chapter is the fact that the SPUs can only access the local store, so any sharing of data has to be performed through explicit DMA operations. The SPEs and PPE are connected via a high bandwidth interconnect called the Element Interconnect Bus (EIB). The main memory and peripheral devices are also connected to the EIB. The MFCs can perform asynchronous non-blocking DMA transfers, independent of the SPUs. The SPUs can issue DMA requests which are added to hardware queues of the MFCs. The SPU can continue doing computation while the DMA operation is in progress. The SPU can query the MFC for DMA completion status and block only when the needed data has not yet arrived. The ability to perform asynchronous DMAs operations allow overlap of computation and communication.

The EIB has a peak bandwidth of 204.8 GB/s and can support multiple outstanding requests from the SPEs. However, the observed transfer latency could vary widely depending on the traffic pattern and the number of SPEs participating in the

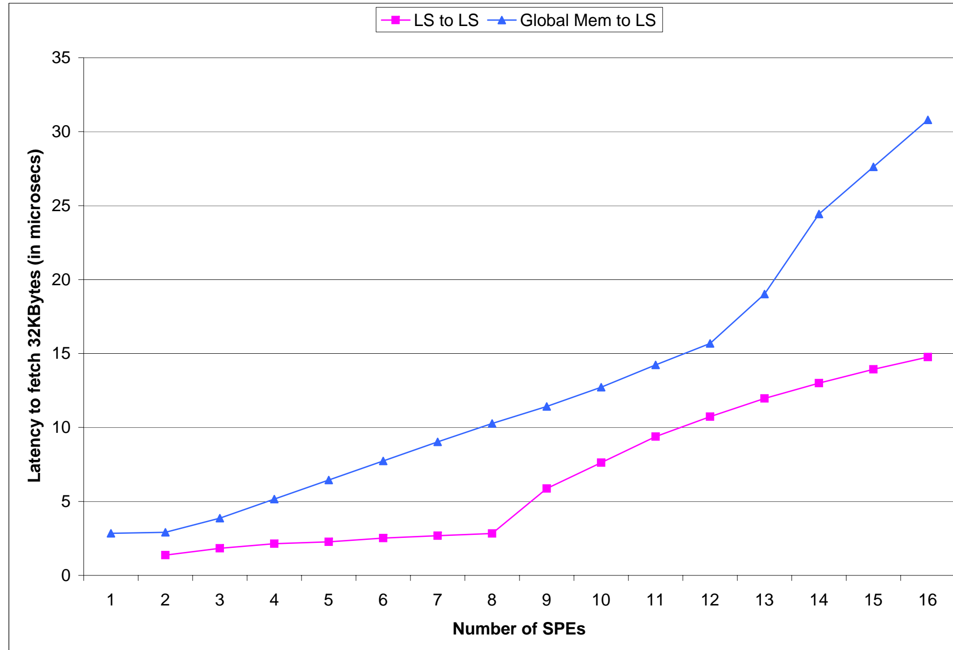


Figure 4.1: Observed latencies for LS to LS and global memory to LS data transfers on the CBE.

communication. Figure 4.1 shows the result of a latency measurement experiment. Threads running on the SPEs fetch data from other SPEs' local stores or from the global memory using the `mfc_get` API call, and the observed latencies are measured. In the LS-to-LS case, each SPE thread chooses one of the other SPEs at random and fetches 32 KB of data from its local store. As the number of participating SPEs increase, the observed latency increases. The observed latencies are much higher when the SPEs are fetching a random 32KB block from the global memory as illustrated in the Global Mem-to-LS case. Global memory to local store goes through the L2 cache in the system, and the limited ports on L2 causes the latencies to be up to 2 times higher than for LS-to-LS transfers.

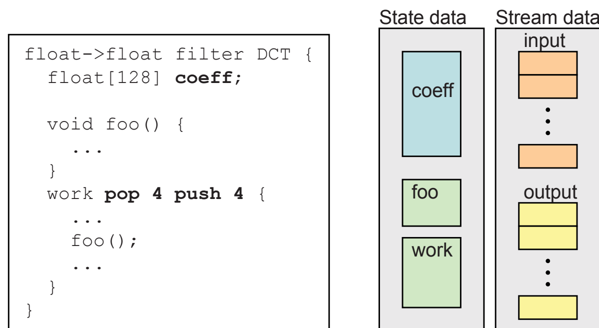


Figure 4.2: A StreamIt filter and memory map on an SPE's local store.

4.2.2 Memory Requirements for SGMS

The software pipeline obtained from SGMS translates to running the filters' work functions on the SPEs of the Cell processor. As mentioned in Section 4.2.1, each SPE can only access its local store. Thus, all data relevant for running an filter's work function need to be present on the local store.

Figure 4.2 shows the `DCT` filter class. It has a local array `coeff`, the work function and a helper function `foo`. The code for the `work` and `foo` functions and member variables are categorized as state data. The rest of the chapter assumes that the local store has at least enough space to hold the state data of all filters mapped to an SPE. Note that implicit state needed by peeking filters is also counted under state data. Every schedulable instance of a filter also needs space for storing the stream data that is pushed and popped in the `work` function. The amount of data pushed or popped during one schedulable instance (some number of repetitions of the work function) of a filter is henceforth referred to as a *block*. The rest of the chapter assumes that there is enough space for at least one input and one output block. As shown in Figure 4.2, a

filter could have produced multiple output blocks before the consuming filter executes. With multiple filters mapped to an SPE, many blocks could be live during the steady state execution before they are consumed during the successive iterations. Figure 4.1 clearly shows that the placement of these live blocks has a significant impact on the latency to transfer the blocks to the consumer filters. It is evident that storing the blocks in the local stores of SPEs is better. However, the number of blocks live during an iteration of the steady state could exceed the capacity of the local store. Live blocks that do not fit in the local store need to be spilled to the global memory. The higher latency of global memory to LS transfers necessitates careful placement and orchestration of transfers of the live blocks. Section 4.3 describes our solution for efficient management of local store for SGMS.

4.3 Block Allocation

The software pipeline for a stream graph produced by SGMS has both producers and consumers running simultaneously in the steady state. As described in Section 3, this is accomplished by running the producer filter enough times to fill the pipeline. The execution schema necessitates storage for two kinds of blocks produced during the steady state : (a) intra blocks, that are produced and consumed on the same processor within the same iteration of the stream graph, and (b) inter blocks, that are produced by an filter in the current iteration and used by an filter mapped to a different processor during a future iteration. As mentioned in Section 4.2.2, we assume that there is enough space for storing the intra blocks, and code for the filters

and any local filter state. This section describes how the inter blocks are allocated and communicated.

Edge cut minimization. The first step in the block allocation process is to minimize the overall space required for the inter blocks. The space required for the inter blocks is a function of the number of instances in which producer and consumer filters are assigned to different processors. In other words, it is a function of the number of data flow edges "cut" by the schedule. SGMS described in Chapter 3 pays no special attention to the number of edges cut. The processor assignment phase only minimizes the initiation interval (II). It is evident that there could be many assignments with the same II. The assignment with fewer edge cuts is preferrable to one with more edge cuts. The goal of this phase is to transform the assignment such that the number of edge cuts is minimized, without changing the II.

An initial assignment of filters to processors is obtained by partitioning the data flow graph. This partitioning is heuristic based, and has no control over the resulting throughput. The goal of graph partitioning is to minimize the number of edges cut. The edges are weighted by the amount of data produced by the producer of the edge in steady state. Then, a second step changes this assignment with a small number of edge cuts to an assignment with the original II. This is done by "moving" filters between processors. The goal is to obtain an assignment with the original II, but using fewest moves. This tends to make the number of edges cut close to the one obtained from graph partitioning, while achieving the original II. The second step is formulated as an integer linear program. Let N be the number of filters and P, the number of

processors. Let p_i denote the processor to which filter i was assigned by the graph partitioner. This formulation considers moving filter i to every other processor (and also, not moving it at all). 0/1 variables $x_{i,j}$ are introduced, $j \in \{1, \ldots, P\} - \{p_i\}$, to denote the fact that filter i is being moved to one of the processors other than p_i. The following equation ensures that filter i is either moved to one and only one of the other processors, or not moved at all.

$$\sum_{j \in \{1,\ldots,P\}-\{p_i\}} x_{i,j} \leq 1 \qquad i \in \{1, \ldots, N\} \tag{4.1}$$

Let t_i be the time taken to run filter i in the steady state. The following equation calculates the load L_j on processor j.

$$L_j = \sum_{p_i=j} (t_i - t_i \times (\sum_{k \in \{1,\ldots,P\}-\{p_i\}} x_{i,k})) + \sum_{p_i \neq j} t_i \times x_{i,j} \tag{4.2}$$

The first summation term includes the times of filters that were originally assigned to processor j, if they were not moved to any other processor. The second summation term includes times of all filters that were moved from some other processor to processor j. Let II be the initiation interval obtained from the original processor assignment phase. The following equation ensures that the load on each processor is lower than II.

$$L_j \leq II \qquad j \in \{1, \ldots, P\} \tag{4.3}$$

Using the variables $x_{i,j}$, it is easy to measure the number of "moves" made, which is a proxy for the number of edges cut by the new assignment.

$$MOVES = \sum_{i \in \{1,\ldots,N\}} \sum_{j \in \{1,\ldots,P\}-\{p_i\}} x_{i,j} \tag{4.4}$$

When a filter has all its producers and consumers assigned to the same processor as itself, moving it to a different processor would cause many more edge-cuts. However, Equation 4.4 treats all moves equally. Using different weights to cases like the one described above could lead to a better solution. However, it was empirically found that using different weights did not affect solution quality significantly. Therefore, the rest of the discussion assumes all moves are weighed equally.

Equation 4.4 measures the number of edges cut across all processors. This may result in uneven distribution of edge cuts across processors. To ensure that the memory requirement at each processor is balanced, it may be more important to get an even distribution of edge-cuts across all processors. The following equation bounds the maximum number of edges cuts at any processor by the variable EC.

$$\sum_{p_i=j} \sum_{k \in \{1,\ldots,P\}-\{p_i\}} x_{i,k} \leq EC \qquad j \in \{1,\ldots,P\} \tag{4.5}$$

Solving equations 4.1 through 4.3 while minimizing $MOVES$ or EC results in an assignment with minimal overall edge-cuts or evenly distributed edge cuts, respectively.

Block allocation. This step takes a processor assignment, with possibly reduced memory requirements, and assigns a location for each block of live data. Consider an filter A mapped to processor $SPE1$ with an output edge to an filter B mapped to $SPE2$. This mapping of filters to processors implies that both filters execute concurrently in the steady state. Therefore, when A is running iteration i, B is running iteration $i - \delta$, consuming the block produced by $(i - \delta)$th iteration of A. Figures 4.3 through 4.6 enumerates some of the options available for storing the inter

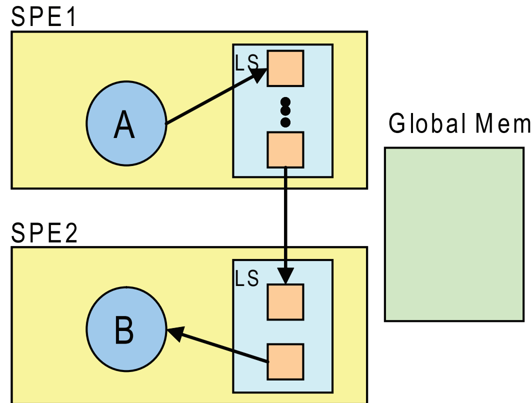


Figure 4.3: Block storage option requiring the most storage space, but allowing complete overlap of DMAs at both producer and consumer processors.

blocks produced by A and consumed by B. Each of these options have different δ values and varying levels of overlap of computation and communication.

Option 1. Figure 4.3 illustrates this option. δ live blocks are stored in the local store of SPE1. Iteration i of A writes to one of those blocks, while a concurrent DMA operation transfers the block produced by iteration $i - \delta - 1$ of A to SPE2. This DMA operation writes to one of the two live blocks stored in SPE2, while B reads the other live block, corresponding to iteration $i - \delta$. Thus, all communication is completely overlapped with communication. Note that all transfers are LS-to-LS as no live blocks are stored in the global memory.

Option 2. As shown in figure 4.4, two blocks are stored in the local memory. Iteration i of A writes to a live block in the local store of SPE1, and initiates a DMA copy to one of the blocks in the global memory. Note that this DMA copy is overlapped with execution of other filters executing on SPE1, and that is the reason why this block cannot be reclaimed and reused. On the consumer side, no live blocks are allocated on

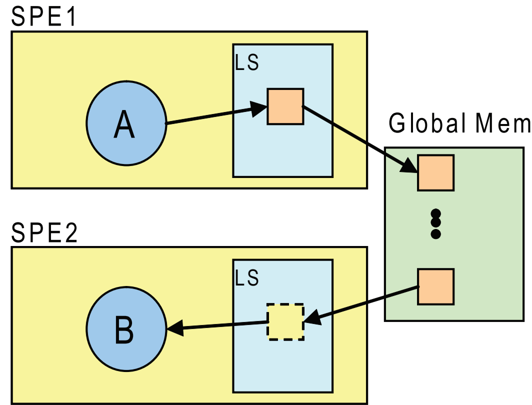


Figure 4.4: Block storage option allowing overlap of DMA at the producer, but not at the consumer processor.

the local store of SPE2. Iteration $i-1$ of B fetches the other live block from global memory, and blocks until the DMA is completed. After B's execution, the input block is no longer needed, and can be reused by other filters on SPE2. In Option 2, δ blocks are stored in the global memory, and the global memory to LS DMA transfer is completely exposed on the consumer side.

Option 3. Using this option exposes the DMA transfer at the producer side. Iteration i of A writes to a block on the local store of SPE1, and transfers that block to global memory. This DMA transfer is blocking, and therefore its latency is completely exposed. This block, however, can be reused by other filters on SPE1. On the consumer side, two live blocks are allocated. Iteration $i-2$ of B uses one of the blocks, while a concurrent DMA operation transfers the block needed for iteration $i-1$. δ blocks are stored in the global memory in this option.

Option 4. This option uses the least amount of storage among the four. However, this results in the DMA transfer being exposed on both the producer and consumer side.

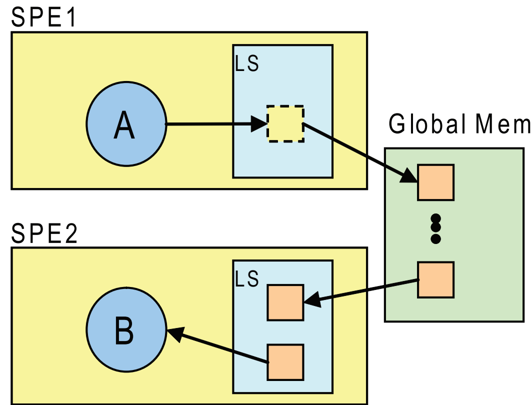


Figure 4.5: Block storage option allowing overlap of DMA at the consumer, but not at the producer processor.

On SPE1, iteration i of A is executed and the resulting live block is immediately transferred to the global memory, thus enabling reuse of the block. Similarly, the block need by iteration $i-1$ of B is fetched onto the local store of SPE2 with a blocking DMA operation. The 4 options are summarized in Table 4.1.

4.4 ILP Formulation

Given a partition of filters to the available processors, one of the options described above has to be chosen for each edge in the stream graph whose producers and consumers are on different processors. The amount of storage needed on the local store of each SPE, and the decrease of throughput caused by exposed DMAs can be calculated only when options have been chosen for *all* the edges crossing processors in the partition. The following integer linear programming (ILP) formulation chooses an option for each edge. The sizes of local stores allowed for live blocks are taken as input, and the formulation constrains the option assignment such that all the live

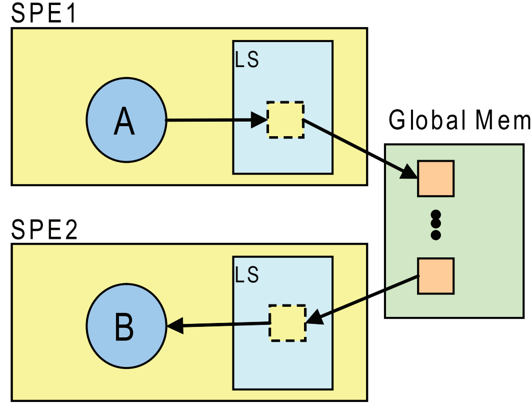


Figure 4.6: Block storage option requiring the least storage space, but exposes DMA transfers at both the producer and consumer processors.

blocks fit within the allowed size. The throughput decrease due to the exposed DMAs is minimized by the formulation.

Suppose $E = \{e_1, e_2, ..., e_n\}$ are the edges whose producers and consumers are on different processors. Let $p(e_i)$ and $c(e_i)$ denote the processors to which the producing filter and consuming filter of edge e_i are assigned and $d(e_i)$ denote the block size associated with e_i. Let P be the total number of processors. Four binary variables $x_{i,1}, x_{i,2}, x_{i,3}$, and $x_{i,4}$ are introduced for each edge $e_i \in E$. $x_i = 1$ implies option 1 described above is chosen for e_i. The following equation ensures that one (and only) option is chosen for each edge.

$$x_{i,1} + x_{i,2} + x_{i,3} + x_{i,4} = 1 \quad i \in \{1, n\} \tag{4.6}$$

On each processor j, the size of local store taken up by the live blocks can be calculated

	# live blocks		DMA Hidden	
	@Producer	@Consumer	@Producer	@Consumer
Option 1	δ	2	✓	✓
Option 2	1	0	✓	×
Option 3	0	2	×	✓
Option 4	0	0	×	×

Table 4.1: Block storage options summary.

from the above binary variables as follows.

$$size(j) = \sum_{p(e_i)=j} \delta \times d_i \times x_{i,1} + d_i \times x_{i,2} + \sum_{c(e_i)=j} 2d_i \times x_{i,1} + 2d_i \times x_{i,3} \quad (4.7)$$

Suppose the maximum local store space allowed on processor j is S_j. The following equation is used to enforce this constraint.

$$size(j) \leq S_j \quad j \in \{1, P\} \quad (4.8)$$

Choosing options 2, 3, or 4 for an edge also incurs DMAs on the processors, whose latencies are exposed in the schedule. Suppose l_i is the DMA latency for transferring the block associated with e_i. The following equation calculates the total latency exposed at each processor j.

$$lat(j) = \sum_{p(e_i)=j} l_i \times x_{i,3} + l_i \times x_{i,4} + \sum_{c(e_i)=j} l_i \times x_{i,2} + l_i \times x_{i,4} \quad (4.9)$$

The following equation introduces a variable *maxlat* which measures the maximum exposed latency across all processors.

$$lat(j) \leq maxlat \quad j \in \{1, P\} \tag{4.10}$$

The variable *maxlat* is used as the objective function, and is minimized subject the constraints given by Equations 4.6 through 4.10.

4.5 Results

This section presents the results of the experimental evaluation of block allocation for SGMS. The partitioning method presented in Section 3.4 is used to assign stream actors to SPEs. This partition assumes that all live blocks can be allocated in the local store of the SPEs, and all DMA transfers can be overlapped. In other words, option 1 shown in Figure 4.3 is used to allocate live blocks for all the inter SPE edges. The throughput achievable by this partition, referred to as Initiation Interval (II), represents the maximum speedup possible, with no memory constraints. Buffer allocation described in Section 4.3 is performed with a given memory constraint which chooses one of the 4 options for each inter SPE edge. The edge cut minimization is performed on the original partition using the Metis [34] graph partitioner followed by the ILP phase. The ILP program for the buffer allocation problem is solved using CPLEX solver. This allocation reduces the memory required on each SPE. It also exposes some DMA transfer latencies for some edges, which has the effect of increasing the II. After an option is chosen for each edge, the code generation schema presented in

Section 3.4.3 is used to implement the software pipeline on each of the SPEs. The live blocks on SPEs are declared as arrays on the corresponding SPEs' C files. The main thread running on Power processor spawns one thread per SPE. Each SPE thread executes a code pattern that was described in Section 3.4.3. IBM's Cell SDK 3.0 was used to implement the DMA copies, and the barrier synchronization. The GNU C compiler gcc 4.1.1 targeting the SPE was used to compile the programs. Note that only vectorization that was automatically discovered by gcc were performed on the actors' codes. The hardware used for our evaluation is an IBM QS20 Blade server. It is equipped with 2 Cell BE processors and 1 GB XDRAM. The set of benchmarks available with StreamIt software version 2.1.1 was used to evaluate the scheduling methods. Most benchmarks are from the signal processing domain. `bitonic` implements the parallel bitonic sorting algorithm. `des` is a pipelined version of DES encryption cipher. Gordon et al. describe the benchmarks in [26].

Each SPE on the Cell processor has 256KB local store. The memory requirements of the original partition, assuming option 1 for all inter SPE edges, were less than 256KB for all benchmarks that were evaluated. The actual memory requirements for the benchmarks ranged from 16KB for `bitonic` to 98KB for `mpeg2`. To evaluate the buffer allocation method, the available memory space was artificially constrained for each of the benchmarks.

Figure 4.7 illustrates the effectiveness of edge cut minimization. The bar chart shows the buffer space required at each of the processors (1 through 4) as a fraction of total space required on all the processors for `vocoder` and `bitonic` benchmarks. The

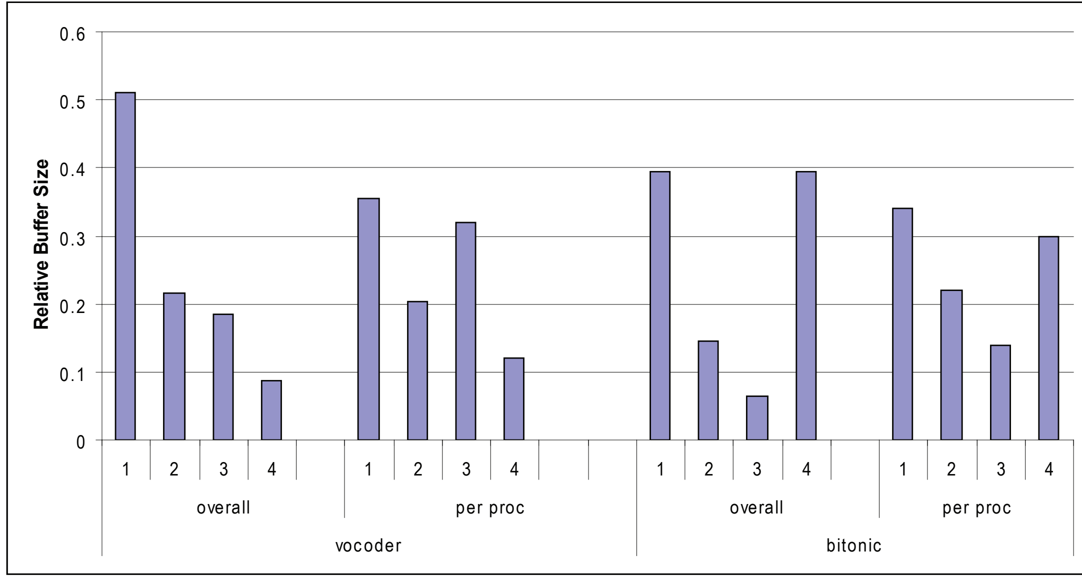


Figure 4.7: Buffer space required on each processor under different edge-cut minimization strategies.

set of bars marked "overall" shows the space required on each processor when the edge cut minimization tries to minimize the overall number of edges cut, i.e., it minimizes Equation 4.4, subject to Equations 4.1 through 4.3. The set of bars marked "per proc" shows the space required when the edge cut minimization tries to get an even distribution of edge cuts across processors, i.e., it minimizes Equation 4.5. Figure 4.7 shows that the buffer requirement is more evenly distributed across processors when Equation 4.5 is used as the objective function during edge cut minimization. For example, in the case of `vocoder`, the standard deviation of buffer requirement across the processors is 15% in the "overall" case, whereas it is only 8% in the "per proc" case.

Figure 4.8 compares the ILP-based block allocation with a greedy heuristic based allocation. The greedy block allocation is performed locally on every processor. Ini-

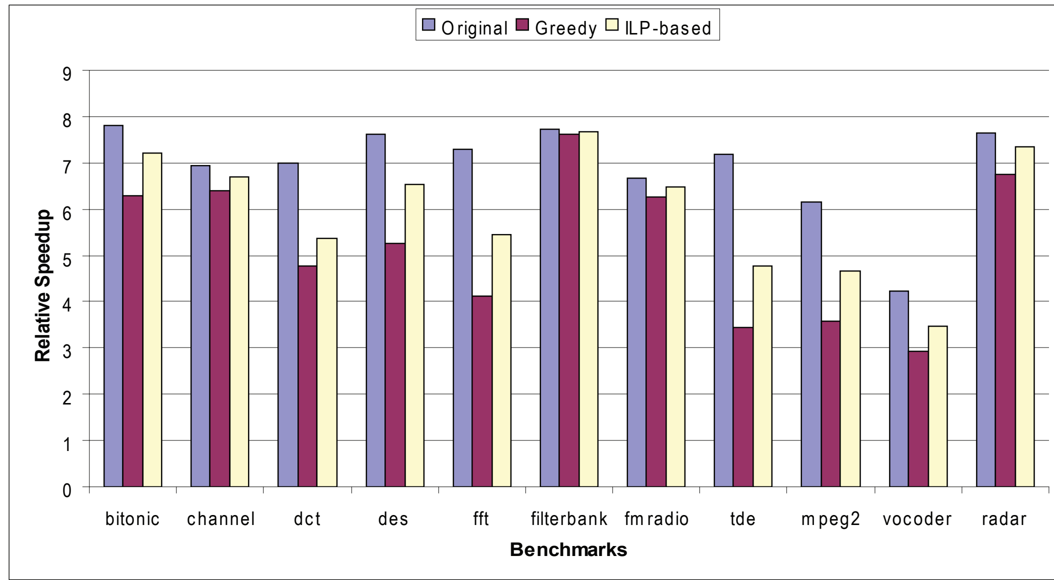


Figure 4.8: Comparison of greedy with ILP-based block allocation.

tially, all live blocks are assumed to be allocated on the local memory. Then, the live blocks are ordered according to their sizes, from the smallest to the largest. Live blocks are considered in this order, and are spilled to the global memory. This process is continued until the remaining blocks fit in the local memory. Figure 4.8 compares the 8 processor speedups of benchmarks. The leftmost bars marked "Original" show the speedups with unconstrained memory. The other two bars show the speedups when the memory was constrained to 50% of the space required by all live blocks. The ILP-based allocation out-performs the greedy heuristic by as much as 30%. Since the ILP-based allocation has a global view across all processor, it is able to reduce the exposed DMAs more than the greedy heuristic.

Figure 4.9 shows the speedup obtained on 8 processors over single processor execution. Different bars represent varying memory constraints. The leftmost bar

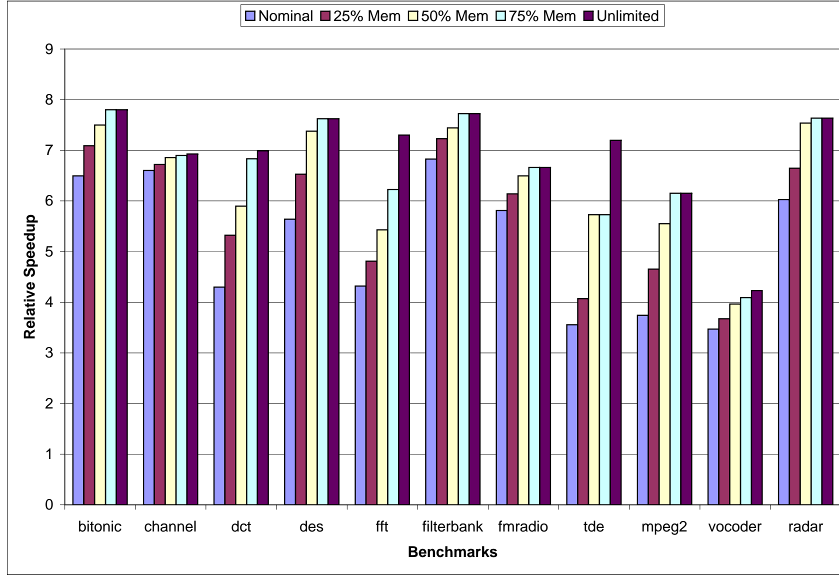


Figure 4.9: Speedups on 8 SPEs with memory constraints.

corresponds to the most memory constrained experiment. For that experiment, no space was available for live blocks on the local stores of SPEs. The local store only contained the code for actors, and their local state. The right most bar represents the unconstrained system, which has enough space in the local store of the SPEs to hold all the live blocks needed by the schedule. The 3 intermediate bars represent systems with 25%, 50%, and 75% respectively, of the total space required for all live blocks for each of the benchmarks. Speedup losses of up to 120% are observed on the system with nominal memory. The actual speedup loss is dependent on the original partition. Note that live blocks are only needed for edges whose producing and consuming actors are on different SPEs. The partitioner tries to minimize II, which is the maximum load on any SPE. The number of edge cuts produced by a partitioner working with such an objective depends on the distribution of work among the ac-

tors in the stream graph. The partitioner produced partitions with few edge cuts for benchmarks like `channel`, `fmradio`, and `vocoder`. The difference between nominal and unlimited cases is only 20% in these cases. For `dct`, `fft`, and `tde` the partitioned had many edge cuts, thus increasing the number of live blocks. Constraining the memory for such benchmarks results in speedup loss of as much as 120%.

CHAPTER 5

Automatic Synthesis of Prescribed Throughput Accelerator Pipelines

5.1 Introduction

In this chapter, we present an automated system for designing stylized accelerator pipelines from streaming applications. The system synthesizes a highly customized pipeline that minimizes hardware cost while meeting a user-prescribed performance level. The input to the system is a behavioral description of the application specified in C that is comprised of a system specification and a set of kernels. The system specification describes the organization and communication in the pipeline, while the kernels describe the functionality of each stage on a single packet of data. The system designs the complete accelerator pipeline by determining the throughput of each stage as well as the inter-stage buffer organization. A unique aspect of the system is the utilization of multifunction loop accelerators to enable multiple pipeline stages to

time multiplex the hardware for a single pipeline stage. This approach sacrifices performance as kernel execution is sequentialized, but greatly increases the ability to share hardware in the design and thus drive down the overall cost.

The contributions of this work are threefold:

- A systematic design methodology for creating rate-matched accelerator pipelines with minumum cost at a user-specified throughput.
- A system that can exploit high degrees of hardware reuse by mapping multiple loops to multifunction accelerators.
- A stylized accelerator pipeline template optimized for streaming applications.

5.2 System Overview

Figure 5.1 shows a broad overview of our accelerator pipeline synthesis system. The system takes as input the application written in C, expressed as a set of communicating kernels. Performance and design constraints, such as overall throughput of the pipeline, clock period and memory bandwidth, are also specified. The frontend of the system performs data dependency analysis on the application to derive the loop graph, which is a representation of the communication structure between the kernels. The system synthesizes an accelerator pipeline with minimum cost to meet the performance constraints. The pipeline consists of a number of loop accelerators (LAs) to execute the kernels in the application and ping-pong memory buffers for commu-

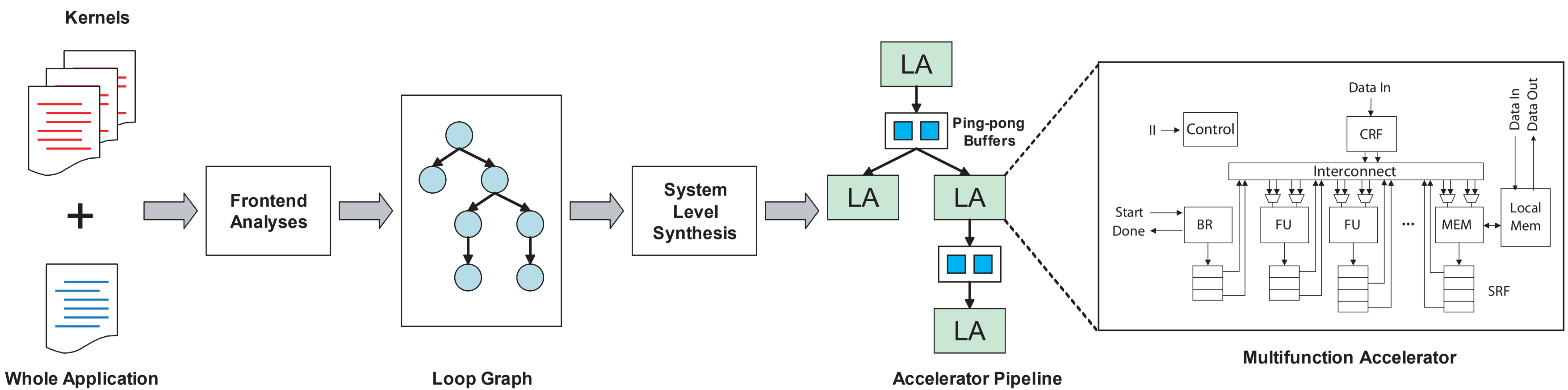


Figure 5.1: Overview of the Streamroller synthesis system.

nicating values. The following subsections describe the input specification and the accelerator pipeline schema in more detail.

5.2.1 Input Specification

The input to the synthesis system is the whole application written in C. Simple stylizations are imposed on the structure of the C program. The stylizations make the analysis of the program simpler, but still enable a wide variety of media and network applications to be expressed. Sequential C semantics make it easy for applications to be developed and debugged quickly, even with the stylization restrictions. The input program consists of two logical parts, viz., a set of *kernel specifications* and the *system specification*.

Kernel specification. Conceptually, kernels form one stage of processing in the application. For example, in wireless applications, a low pass filter can be a single kernel. In our system, a kernel is expressed as a single C function. All inputs and outputs to the kernel have to be provided as arguments to the function. Arguments can be C arrays or scalars. The body of a kernel function has to be a perfectly nested `for` loop. Separating kernels into independent functions enables reuse and modularity. For example, many image procecessing applications perform the same transform on an input image in multiple stages. The same kernel function can be called with appropriate arguments to accomplish this.

System specification. The system specification describes one "packet's" forward flow through the pipeline. The system specification is expressed as a C function

```
void FirFilter (short inp[N1],
                short coeffs[N2],
                short out[N1-N2])
{
  int i, j;

  for(i=0; i<N1-N2; i++) {
    for(j=0; j<N2; j++) {
      out[i] += coeffs[j] * inp[i+j];
    }
  }
}

void distribute (short inp[N1] ,
                short out1[N1],
                short out2[N1] )
{
   ....
   ....
}

void demod (short inp[N1-N2],
            short out[M])
{
   ....
   ....
}
```

(a) Kernel Specification

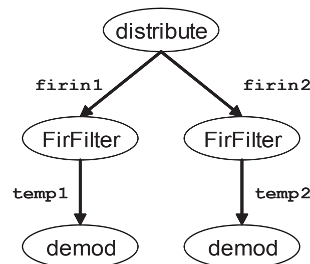
```
void fmradio (short inp[N1],
              short out[M])
{
  short coeffs[N2];
  short firin1[N1], firin2[N1];
  short temp1[N1-N2], temp2[N1-N2];
  short out1[M], out2[M];

  distribute (inp, firin1, firin2);

  FirFilter (firin1, coeffs, temp1);
  FirFilter (firin2, coeffs, temp2);

  demod (temp1, out1);
  demod (temp2, out2);
  ....
  ....
}
```

(b) System Specification



(c) Loop Graph

Figure 5.2: Input specification

whose body contains a sequence of calls to the kernel functions. The system function will be invoked continuously on consecutive packets of data. What constitutes a packet depends on the application. In image processing applications, a packet can be a sub-block of a bigger image. In wireless applications, a packet can be a chunk of data received over the wireless channel. Typically, the applications in these domains process continuous streams of such packets. However, the processing that happens on a single packet is sequential in nature. Thus, our input specification fits well for expressing applications in these domains.

Figure 5.2 shows an example input specification. The system specification function is `fmradio`, which is shown in Figure 5.2(b). It takes the array `inp` as the input and

outputs `out`. The body is made of calls to different kernels shown in Figure 5.2(a). `fmradio` uses local arrays to pass data between the kernels. A simple dataflow analysis of the system specification yields the *loop graph* shown in Figure 5.2(c). The nodes in the loop graph correspond to kernels whereas the edges indicates dataflow through an array between kernels. Note that arbitrarily complex loop graphs can be expressed by using just straight line C code.

Performance specification. The applications targeted by our synthesis system have real-time requirements usually expressed at the highest level in terms of, say, frames/second or Kbps. Since the input specification corresponds to end-to-end processing of one packet, the real-time constraint can be easily translated to the number of times the system specification function has to be called per second. Given the clock period, the performance specification reduces to the number of cycles between consecutive invocations of the system specification function. For example, consider the application in Figure 5.2. The `fmradio` function completely processes one input packet `inp`. Suppose `N1` is 512, `fmradio` processes $512 \times 16 = 8192$ bits per call. To achieve a real-time requirement of 128 Mbps, `fmradio` has to be called $\frac{(128 \times 1024 \times 1024)}{8192} = 16384$ times every second. If the clock frequency under consideration is 200 MHz, the `fmradio` function has to be invoked approximately once every 12208 cycles to meet the real-time requirement.

5.2.2 Accelerator Pipeline Hardware Schema

Each kernel is mapped to a *loop accelerator* (LA). Depending on the performance requirements, multiple kernels can be mapped to the same LA, which performs the functions of both the kernels. These multifunction LAs form the building blocks for the accelerator pipeline. The intermediate arrays used in the input specification are mapped to SRAMs with ports connected to producer and consumer LAs. This section presents the hardware schema of the individual LAs and the accelerator pipeline.

Multifunction Loop Accelerator. The hardware schema used in this chapter is shown in the inset on the right side of Figure 5.1. The innermost loops of the kernel specification function are modulo scheduled, and the architecture for the LAs is derived directly from the schedule. Modulo schedules [58] are characterized by initiation interval (II), which is the number of cycles between invocations of successive iterations of a loop. Therefore, high performance schedules of a loop have lower IIs. The accelerator is designed to exploit the high degree of parallelism available in modulo scheduled loops with a large number of function units (FUs). Each FU writes to a dedicated shift register file (SRF); in each cycle, the register contents shift downwards by one register. Wires from the registers back to the FU inputs allow data transfer from producers to consumers.

Multifunction LAs are designed to execute more than one loop nest. The general hardware schema for multifunction LAs is similar to a single function LA. The set of FUs in the multifunction LA is the union of FUs required by the individual loops. The widths and depths of SRFs are set such that they can support values of operations

from all loops assigned to the corresponding FUs. A detailed description of how an optimal datapath for a multifunction LA is derived is beyond the scope of this chapter; readers are referred to [25] for more information. Instead, this chapter [38] focuses on how multifunction LAs can be used as building blocks to build an accelerator for the whole application.

Accelerator Pipeline Schema. The accelerator pipeline is designed such that all processing on a single packet of data (henceforth referred to as a *task*) is done sequentially to respect the program order in the system specification function. However, multiple tasks can be in progress in the pipeline at the same time. The sequential execution is achieved by using the two control signals `START` and `DONE`. The LA begins execution of a loop when the `START` signal goes high, and raises the `DONE` signal at the end. By connecting the `DONE` signal of a producer LA to the `START` signal of a consumer LA, sequential execution of a task is ensured. Arrays used for communication are mapped to SRAMs in the accelerator. Since multiple tasks can be in flight in the pipeline, more than one SRAM buffer could be allocated to a program array.

Figure 5.3(b) shows the accelerator pipeline corresponding to the loop graph in Figure 5.3(a). There are three loops `K1`, `K2`, and `K3` in the application, each with a trip count (**TC**) of 100. The accelerator shown in Figure 5.3(b) is capable of executing the application with an overall throughput of 100 cycles. Each loop is modulo scheduled with II=1. Therefore, the approximate latency of each loop (one stage in the accelerator pipeline) is 100 cycles. Figure 5.3(c) shows the execution

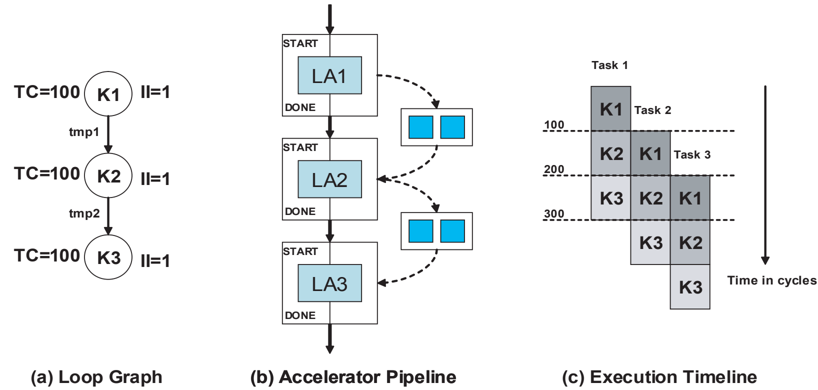


Figure 5.3: Example accelerator pipeline (High performance)

timeline for 3 tasks executing in the pipeline. Note that execution of `K1` in task 2 is overlapped with execution of `K2` in task 1. This means that `K1` will be producing new values for the array `tmp1` while `K2` is still using the old values. To avoid this and still provide overlapped execution of tasks, two SRAM buffers are allocated to `tmp1`. Alternate tasks ping-pong between these 2 buffers. Similarly, two buffers are allocated for `tmp2`.

Figure 5.4(b) shows a different accelerator pipeline for the same application. This pipeline has a lower throughput of 200 cycles, as opposed to 100 cycles capable by Figure 5.3(b). In this pipeline, `K1` is modulo scheduled with II=2, which is a lower performance implementation. Also, LA2 is a multifunction accelerator capable of executing `K2` and `K3`, each with an II of 1. Figure 5.4(c) shows the execution timeline of 2 tasks through the pipeline. Note that execution of `K2` and `K3` never overlap across tasks. Therefore, only one buffer is allocated for `tmp2`.

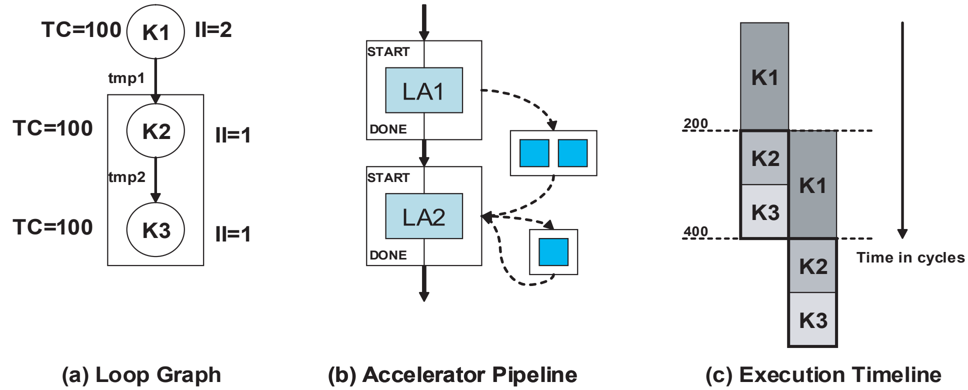


Figure 5.4: Example accelerator pipeline (Low performance)

5.3 Design Methodology

This section presents our methodology for designing an accelerator pipeline for an application. Section 5.3.1 describes the cost tradeoffs for various components of the accelerator pipeline. Section 5.3.2 describes an integer linear programming (ILP) formulation for finding an optimal cost accelerator pipeline at a prescribed throughput. Section 5.3.3 presents a practical end-to-end system that generates Verilog RTL from the sequential C application.

5.3.1 Cost Components

The cost of individual loop accelerators forms a major component of the cost of the accelerator pipeline. As described in Section 5.2.2, the datapath for an LA is derived from the modulo schedule of the innermost loop of a kernel function. An LA consists of a number of FUs connected to SRFs. Since all operations in the loop body are packed within II cycles in a modulo schedule, the number of FUs is determined

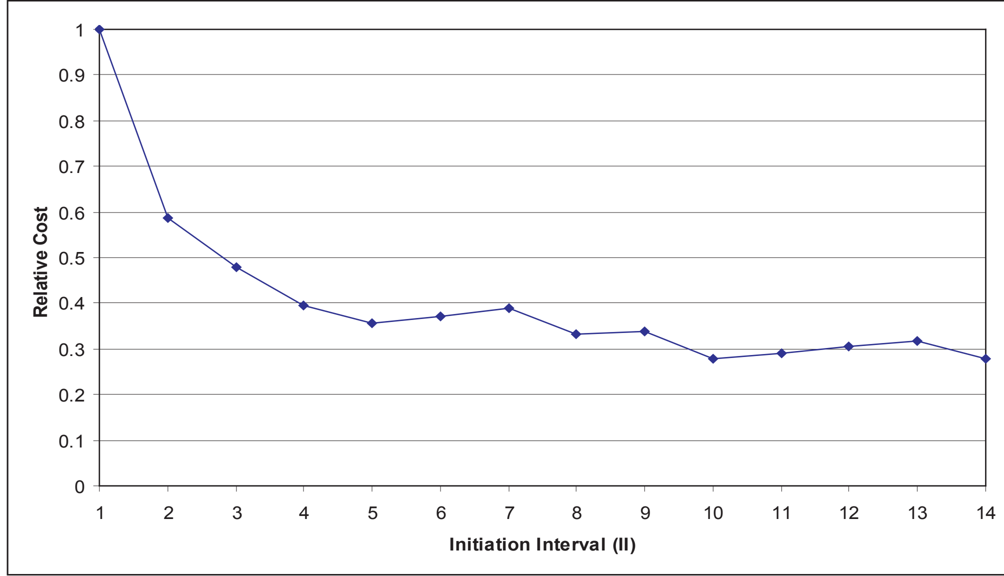


Figure 5.5: II vs. Cost of a loop accelerator

solely by the II. The cost of FUs is also determined by the bitwidths of operations assigned to it. Note that the bitwidth of an FU has to be as large as the widest operation assigned to it. The bitwidth of a SRF is same as the FU connected to it. The depth, however, depends on the schedule. The SRF has to hold all live values produced by an FU. If the consumer of an operation is scheduled far away from its producer, the SRF connected to the producer FU will be deep. Also, when the loops have recurrences, the variables that carry the dependences across the recurrence must be live for at least II cycles. Thus, the cost of SRFs tends to increase for very large IIs.

Figure 5.5 shows a plot of the cost of a single LA with increasing II. The loop for which this LA was built is a part of the *Beamformer* [55] application. The costs are obtained by scheduling the loop at different IIs using a cost sensitive modulo sched-

uler [24] and synthesizing the resultant Verilog using the Synopsys design compiler. The highest cost corresponds to the lowest II of 1, which is the highest performance implementation of the loop. As II is increased, the number of FUs in the datapath and the corresponding SRFs decrease. Therefore, we see the general trend of decreasing overall cost with increasing IIs. However, beyond II of 10, there is no decrease in cost. At high IIs, the depth of SRFs corresponding to the recurrence variables grow disproportionately compared to other SRFs. Thus, there is no further cost improvement beyond II of 10.

Apart from II, the other dimension that affects multifunction accelerator cost is the different loops that the LA implements. A multifunction LA saves cost over two single function LAs. The amount of cost saved depends on the mix of operations in the two loop bodies. The more similar they are, the more cost saved. However, a multifunction LA has an adverse effect on performance. Since there is only one physical hardware to execute two loops, instances of these two loops in different tasks cannot be overlapped. To achieve the same overall throughput of the pipeline, the multifunction LA might have to implement a higher performance version of the individual loops. Thus, the tradeoff of independent LAs with low performance versus one multifunction LA implementing high performance versions of the loops, has to be considered.

The other major component of cost of the accelerator pipeline is that of the memory buffers. Note that there has to be at least one buffer for every array in the application. To allow for task overlap, more than one buffer might have to allocated

to an array. The size and bitwidth of an array is application dependent and the modulo schedule for a loop has no control over the cost of a memory buffer. However, depending on the amount of task overlap required, which is dictated by the overall throughput requirement, the number of buffers for an array vary.

5.3.2 ILP Formulation

The accelerator pipeline design system has to judiciously choose the IIs for each kernel in the loop, and the number of buffers allocated for each program array such that the overall throughput is achieved, and at the same time, cost is minimized. The system also has to consider combining many loops into one multifunction LA whenever cost savings are possible. In this section, we develop an integer linear programming (ILP) formulation that optimizes the overall cost of the accelerator pipeline by choosing IIs and the number of buffers for the arrays. Figure 5.6 shows the integer linear program for the optimization problem.

Consider the *loop graph* constructed from the system specification function. For every kernel instance i in the system specification function, the loop graph has a vertex v_i. If a kernel i writes to an array a, and kernel j reads from the array, and edge $e_{a,i,j}$ is added between vertices v_i and v_j. Note that the array name a is also a part of the edge label. This is because more than one array could be used for communication between a pair of loops. An integer variable II_i is introduced for every kernel i, and equation 1 bounds it between $IIMIN_i$ and $IIMAX_i$. Section 5.3.3 describes how $IIMIN_i$ and $IIMAX_i$ are determined. If the trip count of loop i is TC_i, then the

Minimize: $\sum_{i=1}^{p} C_i + BUFCOST$

Subject to:

$$IIMIN_i \leq II_i \leq IIMAX_i \qquad \forall i \qquad \mathbf{(1)}$$

$$L_i = TC_i \times II_i \qquad \mathbf{(2)}$$

$$L_i \leq \tau \qquad \forall i \qquad \mathbf{(3)}$$

$$\sum_{k \in path\ i \to j} L_k \leq d_{a,i,j} \times \tau \qquad \forall paths\ i \to j \qquad \mathbf{(4)}$$

$$\sum_{j=1}^{p} b_{i,j} = 1 \qquad \forall i \qquad \mathbf{(5)}$$

$$\sum_{i=1}^{p} b_{i,j} \times L_i \leq \tau \qquad 1 \leq j \leq p \qquad \mathbf{(6)}$$

$$\left.\begin{array}{lcl} TL_{i,j} & \geq & 0 \\ TL_{i,j} & \leq & P \times b_{i,j} \\ TL_{i,j} & \leq & L_i \\ TL_{i,j} & \geq & L_i - (1 - b_{i,j}) \times P \\ \sum_{i=1}^{p} TL_{i,j} & \leq & \tau \end{array}\right\} \qquad \mathbf{(7)}$$

$$II_i = \sum_{k=IIMIN_i}^{IIMAX_i} k \times ii_{i,k} \qquad ii_{i,k} \in \{0,1\} \qquad \mathbf{(8)}$$

$$\sum_{k=IIMIN_i}^{IIMAX_i} ii_{i,k} = 1 \qquad \mathbf{(9)}$$

$$CL_i = \sum_{k=IIMIN_i}^{IIMAX_i} Cost(i,k) \times ii_{i,k} \qquad \mathbf{(10)}$$

$$SUMC_j = \sum_{i=1}^{p} b_{i,j} \times CL_i \qquad \mathbf{(11)}$$

$$MAXC_j \geq b_{i,j} \times CL_i \qquad 1 \leq i \leq p \qquad \mathbf{(12)}$$

$$C_j = MAXC_j + 0.5 \times (SUMC_j - MAXC_j) \qquad \mathbf{(13)}$$

$$BUFCOST = \sum_a d_{a,i,j} \times Memcost(a) \qquad \mathbf{(14)}$$

Figure 5.6: ILP formulation for system level synthesis

latency L_i of the LA implementing loop i can be approximated by equation 2.

For every edge $e_{a,i,j}$, an integer variable $d_{a,i,j}$ is introduced to denote the number of buffers that are synthesized for the array a. Let the overall throughput for the entire pipeline be denoted by τ. The latency of any loop i should be no more than the throughput τ, as shown in equation 3. Note that, if there is only one buffer for an array a, then the producer i cannot begin execution in the next task before the consumer j in the previous task is done executing. Effectively, the buffer is occupied for $L_i + L_j$ cycles. However, i and j can be overlapped across consecutive tasks if more than one buffer is allocated for array a. Given that $d_{a,i,j}$ denotes the number of buffers allocated for array a, equation 4 formalizes the above constraint. Equation 4 simplifies to $L_i + L_j \leq d_{a,i,j} \times \tau$ when there is a direct edge from i to j. In this case, $d_{a,i,j}$ can have a maximum value of 2, i.e., two buffers for the array a is sufficient to support maximal task overlap. However, i and j may not be directly connected, and there could be many paths (possibly of length longer than 2) from i to j in the loop graph. In the general case, more than 2 buffers may be required for array a, and consecutive tasks use the buffers in a round-robin fashion.

Multiple loops can be combined into one multifunction LA. The assignment of IIs to the loops is independent of whether or not they becomes part of a multifunction LA. Suppose the number of kernel instances in the system specification function is p. There can be a maximum of p accelerators in the pipeline. This extreme case corresponds to the design where there are no multifunction LAs in the system. Binary variables $b_{i,j}$ are introduced to denote the assignment of loop i to the accelerator j.

Equation 5 ensures that every loop is assigned to exactly one LA. The latency of a multifunction accelerator now becomes the sum of the latencies of individual loops assigned to it, which should be no more than the overall throughput τ, as shown in equation 6. Equation 6 involves the product of a binary variable and an integer variable, and is non-linear. However, it can be linearized using auxiliary variables $TL_{i,j}$ as shown in equation 7. P is a suitable large constant in equation 7.

Equations 1–7 can provide a valid selection of IIs for the loops and number of buffers for the arrays, and a valid combination of loops into multifunction LAs. An objective function is designed such that the cost of the overall pipeline is minimized. First, variables CL_i are introduced to denote the cost of a single function LA that just implements loop i. Note that this cost depends solely on II_i. However, CL_i is not a linear function of II_i as shown in Figure 5.5. To overcome this, a one-hot encoding of II_i is used to express CL_i in terms of II_i as shown in equations 8–10. Note that $Cost(i, k)$ denotes the cost of a single function LA implementing loop i with II=k, and is a constant. The cost C_j of a multifunction LA j is a function of the loops assigned to it. It cannot be expressed as a simple linear function, and can be obtained only by actually synthesizing the multifunction LA. To approximate C_j, we introduce two variables, $SUMC_j$ and $MAXC_j$ in equations 11 and 12 which represent the sum of costs of single function LAs that implement loops assigned to j, and the maximum of costs of those LAs, respectively. Equation 11 is not linear. However it can be linearized using the same technique shown in equation 7. The cost C_j of a multifunction LA j is bounded by $MAXC_j$ and $SUMC_j$. As an approximation, we

use equation 13 to represent C_j. The actual cost of a multifunction might vary widely between $MAXC_j$ and $SUMC_j$. If the loops that are combined are exactly identical, the cost will be same as $MAXC_j$. If they have less overlap in terms of kinds of operations in the loop body, then the cost of combined LA will be close to $SUMC_j$. Empirically, we found setting C_j at the mid-point between $MAXC_j$ and $SUMC_j$ worked satisfactorily for a wide range of applications. Getting a better estimate for C_j without actually synthesizing the multifunction LAs will strengthen the objective function, and is a subject of future research.

The other component of cost, the array buffers, does not depend on assignment of loops to a multifunction LA, and can be easily calculated from $d_{a,i,j}$'s. Since $d_{a,i,j}$ denotes the number of buffers allocated for the array a, the overall cost of memory buffers in the system is given by equation 14. $Memcost(a)$ is a constant depending on the size and bitwidth of the array a. The objective of the ILP solver is to minimize the sum, $sum_{i=1}^{p} C_j + BUFCOST$, subject to the constraints given by equations 1–14.

5.3.3 Implementation

We use the SUIF compiler infrastructure [70] to process the input specification and build the loop graph. High level information like trip counts of the kernels, the sizes and bitwidths of arrays used for communication, and the communication structure between kernels is gathered in a SUIF pass. The application is converted to assembly format, which is the input to the Trimaran [66] compiler tool chain. Operation level data flow analysis is performed to determine $IIMIN_i$'s for each kernel. Note that

the minimum achievable II is a function of recurrence cycles in the loop body. A cost sensitive modulo scheduler is used to schedule the loop bodies at increasing IIs, beginning with $IIMIN_i$. The datapath for the LA is derived from the schedule and synthesized using Synopsys design compiler. Thus, $Cost(i,k)$'s, the gate count estimates for the LA implementing loop i at II=k, are obtained. As II is increased, $Cost(i,k)$ decreases up to a point. As described in Section 5.3.1, the cost of LAs begins to increase at higher IIs. The value of $IIMAX_i$ is set to the point where this happens. $Memcost(a)$'s are computed using the Artisan memory compiler which synthesizes SRAMs for the communication arrays. Using the constants obtained as above and the throughput specification, the ILP program is formed and solved using the CPLEX solver. Thus, the II_i's for all loops and $d_{a,i,j}$'s, the number of buffers allocated for the arrays, are obtained.

5.4 Case Studies

This section presents accelerator pipelines for different applications designed using our system. `Simple` is a synthetic application included to illustrate how different components of cost are optimized. `Beamformer` and `FMRadio` are streaming applications from the VersaBench [55] suite. For each application, accelerator pipelines were designed with varying throughputs, and the system area results are presented.

Simple. This applications consists of a sequence of eight loops, each with a trip count of 256 and containing a mix of add and multiply operations. The loop iterations are completely parallel, allowing modulo schedules with IIs of 1. The highest perfor-

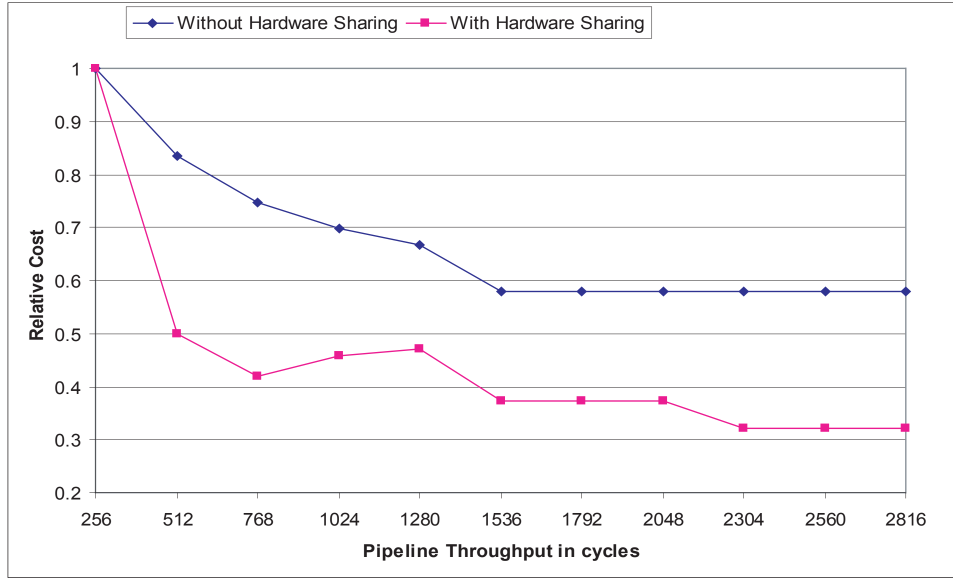


Figure 5.7: Cost vs. Throughput for Simple

mance pipeline for this application can have a throughput of 256 cycles. Figure 5.7 shows the cost of accelerator pipelines designed for `Simple` for throughputs varying from 256 cycles to 2816 cycles.

The costs shown are gate counts, and are relative to the cost of the pipeline with a throughput of 256 cycles. The set of points labeled "Without Hardware Sharing" correspond to the designs with no multifunction LAs. The ILP formulation was modified to get the lowest cost design without combining any loops. We see that multifunction LAs are able to achieve significant cost savings through sharing hardware across multiple loops. On an average, 40% cost savings are achieved by hardware sharing.

Figure 5.8 illustrates the use of multifunction LAs in the accelerator pipeline. The loop graph is shown with the IIs next to the nodes. Boxes indicate which loops were combined into multifunction LAs. When the required throughput is 512 cycles

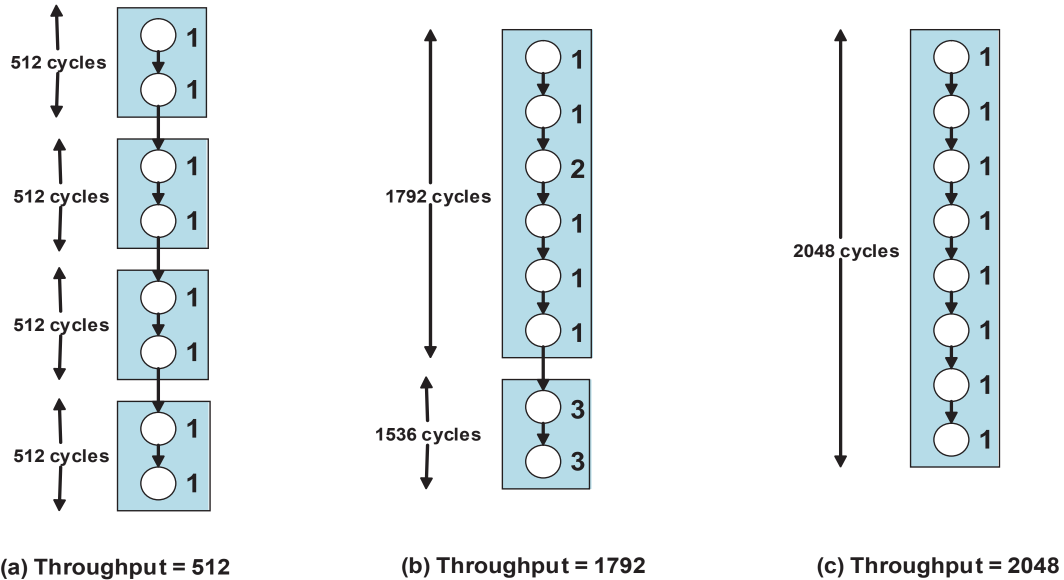


Figure 5.8: Pipeline configurations for Simple

(Figure 5.8(a)), adjacent loops are combined into multifunction LAs, resulting in only 4 stages in the pipeline. Each multifunction LA now has a latency of 512 cycles, as it implements two loops at II of 1. Figure 5.8(b) illustrates the complexity of designing a pipeline for a large application. When the required throughput is 1792, the number of possibilities are too many for a designer to manually explore. Our automated method systematically derives the pipeline configuration with minimum cost as shown.

FMRadio. FMRadio is a software implementation of an FM Radio receiver. Figure 5.9 shows the costs of the accelerator pipeline for FMRadio capable of receiving varying maximum frequencies. The frequencies were derived assuming a clock rate of 200 MHz. Our implementation has 27 loops in the loop graph. On an average, multifunction LAs cause 20% cost savings by sharing resources across multiple loops. Even though the general trend is lower cost for lower throughput configurations,

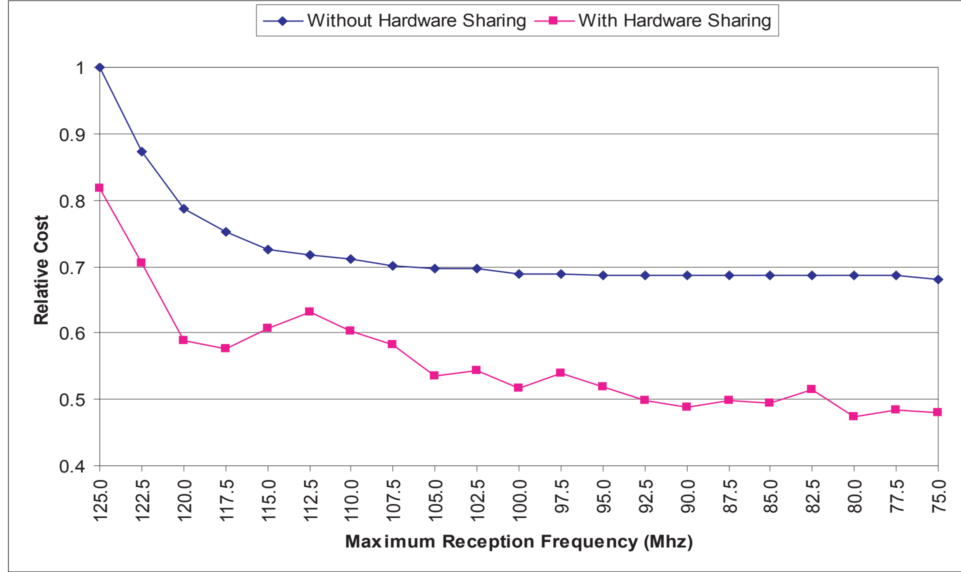


Figure 5.9: Cost vs. Throughput for FMRadio

the "With hardware sharing" curve is not smooth. For example, the cost for 112.5 MHz configuration is about 10% higher than the 117.5 MHz configuration. The approximation adopted using Equation 13 causes this sub-optimality in some cases. However, the cost of the pipeline with multifunction LAs is still much lower than without hardware sharing. The memory component of the cost was 45% of the overall cost for low II designs, indicating that more FUs are required in the data path in high performance implementations. The memory component of the cost was up to 70% of the overall cost for high II, low performance designs.

Beamformer. Beamformer is a spatial filter operating on data from an array of sensors. Again, the data rates shown in Figure 5.10 are derived assuming a 200 MHz clock. Our implementation has 10 loops in the loop graph. 15% cost savings are achieved due to hardware sharing on an average. The memory component of the overall cost ranged from 60% for low II designs to 70% for high II designs.

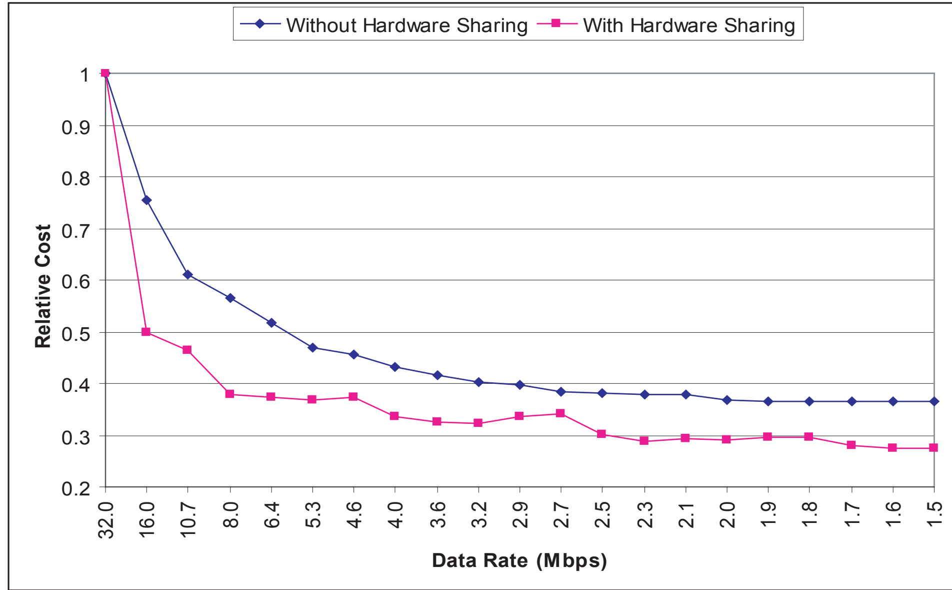


Figure 5.10: Cost vs. Throughput for Beamformer

Discussion. Some of the salient points about the designs generated in our case studies are as follows:

- Multifunction LAs reduce cost over having two independent low performance accelerators. As Figure 5.5 shows, the reduction in cost of a single accelerator is not linear with respect to decrease in performance. However, combining two similar loops can halve the cost. Thus, the system combines as many loops as possible to save cost.

- Pipelines with a low prescribed throughput often contain overdesigned LAs. Increasing the IIs beyond a certain point only increases the cost. Therefore, the system just picks a higher performance LA to save cost.

- Memory buffers are a significant portion of overall cost. In some cases, reducing the number of buffers to 1 while retaining the producer and consumer loops at

high performance saved more cost than synthesizing low performance LAs for the loops with two buffers for the array.

CHAPTER 6

Conclusions and Future Directions

The widespread use of multicore processors is pushing explicitly parallel high-level programming models to the forefront. Stream programming is a promising approach as it naturally expresses parallelism in applications from a wide variety of domains. This thesis addresses the basic problems underlying both compilation and hardware synthesis for stream programs. A unified set of mapping and scheduling algorithms have been developed for automatically mapping stream programs to multicore platforms and synthesizing custom hardware for stream programs. The main results of compilation and synthesis are summarized below.

Summary of compilation results. This part of the dissertation presents a compilation method for orchestrating the execution of stream programs on multicore platforms. One of the main issues in compilation is getting an even distribution of computation across processors. This is handled in an integrated fission and partitioning step that breaks up computation units just enough to span the available processors. The issue of communication overhead is overcome by an intelligent stage

assignment, which overlaps all communication with computation. A detailed evaluation of our method on real hardware shows consistent speedup for a wide range of benchmarks. Stream graph modulo scheduling provides a geometric mean speedup of 14.7x over single processor execution across the StreamIt benchmark suite. SGMS method is compared to naïve unfolding that unfolds all filters as many times as the number of processors. Even though naïve unfolding gets speedups similar to SGMS for completely stateless programs, SGMS demonstrates wider applicability by offering consistent speedups on both stateless and stateful programs. SGMS has been extended for embedded multicore platforms with constrained memory systems. As many blocks of data that are live during the schedule are allocated to the memories local to the processors on which they are produced and consumed. The exposed communication due to the spilling of live blocks to global memory is minimized as much as possible, thus resulting in a high-quality schedule even for a constrained memory system.

Summary of synthesis results. This part of the dissertation presents an automated system for designing accelerator pipelines for compute-intensive stream programs at a user-prescribed performance level. Synthesis consists of designing a set of communicating loop accelerators and buffers for storing intermediate results, and orchestrating the pipeline execution. Multifunction accelerators are used to reduce cost through hardware sharing between pipeline stages. Three case studies are presented to highlight the capabilities and effectiveness of the design system. The studies reveal three important findings about accelerator pipelines: multifunction LAs are found to

be more cost efficient than having multiple independent, lower performance accelerators; reducing the performance of a single LA below a certain point only serves to increase cost; and the memory buffers are significant and the configuration must be optimized to minimize overall system cost.

The advent of multicore processors has brought parallel programming to the forefront. Tools that help programmers to write parallel programs efficiently and map those programs to a variety of parallel hardware systems are of great importance. This thesis has developed a compilation and synthesis system for one flavor of parallel programming, namely stream programming. The evaluation of this technology has shown that stream programming is amenable to efficient execution of multicore platforms with distributed memory system. The scheduling methods developed here can be easily extended to other types of multicore platforms. Tools such as the ones developed in this thesis should make stream programming the top choice for developing a wide variety of applications.

This research can be extended in many ways. Static compilation methods developed in this thesis have wide applicability for embedded systems. Although embedded systems have been built with multicore processors (called MPSoC in this domain), these systems tend to be much different from mainstream processors. Constrained memory and irregular interconnects are some of the characteristics that differentiate these systems. Streamroller handles memory constraints in a separate phase. Integrating resource constraints into the main scheduling phase of Streamroller will make it more suitable for embedded systems. Also, applications developed for embedded

systems tend to have strict real-time specifications. Since the scheduler in Streamroller has been formalized as an integer linear program, it should be easy to integrate the real-time constraints into the current set of inequalities.

Mainstream multicore processors tend to have less resource constraints compared to embedded processors. However, the kind of applications developed for mainstream domain tend to be much more dynamic. Some of the dynamism arises from the data-dependent nature of the filter functions and presence of control dependent operations. Such applications are better scheduled by a dynamic scheduler that runs as a part of the runtime support system. The static scheduling techniques developed in this thesis can still be used on the static parts of the applications, thus avoiding the overheads of the runtime system.

APPENDIX

APPENDIX A

Symmetry Breaking

A.1 Introduction

While being useful in many contexts, symmetry introduces severe inefficiency in the solution on many integer linear programming (ILP) models. In ILP solvers that use branch-and-bound search methods, symmetry causes the search tree to become large. Equivalent solutions are unnecessarily explored in different parts of the tree. This chapter first describes the symmetries present in the main integer programming model used in the thesis, namely, the model for optimal assignment of M tasks to N processors. Then, a set of linear inqualities are presented that effectively removes the symmetries from the model. The performance of ILP solvers are evaluated when these inequalities are added to the original model.

The crux of the ILP model presented in Chapters 3 and 5 is the assignment of M tasks to N processors, while minimizing the load on the maximally loaded processor.

The objective function is

$$\min \quad \mathcal{C} \tag{A.1}$$

subject to

$$\sum_{j=1}^{N} x_{ij} = 1 \qquad i \in \{1, \dots, M\} \tag{A.2}$$

$$\sum_{i=1}^{M} c_i \times x_{ij} \leq \mathcal{C} \qquad j \in \{1, \dots, N\} \tag{A.3}$$

where c_i is the time taken to execute task i on a processor. Note that Chapter 3 uses additional variables and constraints to determine fission factors for the tasks, while Chapter 5 uses them to determine cost of resulting hardware. The above reduced model is used in this chapter to simplify the exposition. The above model uses a $M \times N$ matrix of 0/1 variables x_{ij}. Equation A.2 ensures that each task gets assigned to one and only one processor. Equation A.3 "measures" the load on a processor. The main source of symmetry in this model comes about from the fact that the processors are indistinguishable from one another. In Section A.2, the tool Saucy [20] is used to get a compact functional description of the symmetries. Section A.3 presents the construction of a set of linear inequalities that effectively removes the symmetry from the model. Section A.4 evaluates the performance of an ILP solver on the model with symmetries removed.

A.2 Discovering Symmetry

For the problem of assigning M tasks to N identical processors, it can be intuitively seen that there can be many equivalent assignments. For example, with

4 tasks and 2 processors, assigning tasks $\{1,2\}$ to processor 1 and tasks $\{3,4\}$ to processor 2 is the same as assigning tasks $\{3,4\}$ to processor 1 and tasks $\{1,2\}$ to processor 2. To verify the presence of this symmetry and obtain a canonical representation that can be utilized in an ILP model, we use the framework presented in [5] and [6]. It was shown by Crawford [18] that detecting symmetries is equivalent to the problem of testing graph automorphism. [5] presented a graph construction methodology that transforms a boolean satisfiability (SAT) problem instance into a colored graph. Testing graph automorphism on the graph revealed the symmetries in the original problem. The work in [6] extended the graph construction method for pseudo-Boolean formulas. The ILP model presented in Equations A.1 through A.3 is not strictly pseudo-Boolean, as it involves a free real variable $\mathcal{C}$, and the coefficients are real instead of integers. To utilize the graph construction method in [6], a modified formulation equivalent to the original model is presented.

A modified formulation. The graph construction method presented in [6] works for a mix of SAT clauses and pseudo-Boolean formulas. Pseudo-Boolean formulas involve adding additional nodes to the graph to represent coefficients, which typically tend to increase the graph size. First, the Equation A.2 is converted to the following set of SAT clauses. This avoids adding the additional nodes, resulting in a more compact graph representation.

$$\bigvee_{j=1}^{N} x_{ij} == TRUE \qquad i \in \{1, \dots, M\} \tag{A.4}$$

$$x_{ij} \wedge x_{ik} == FALSE \qquad j,k \in \{1, \dots, N\}, j \neq k, i \in \{1, \dots, M\} \tag{A.5}$$

Note that the optimization problem presented in Equations A.1 through A.3 can

be solved by solving a sequence of just constraint problems (without any optimization criteria), in which the free real variable $\mathcal{C}$ is replaced by a real constant $\mathcal{C}_{const}$. In each of these steps, $\mathcal{C}_{const}$ is reduced successively until the following constraints are unsatisfiable.

$$\sum_{i=1}^{M} c_i \times x_{ij} \leq \mathcal{C}_{const} \qquad j \in \{1, \ldots, N\} \tag{A.6}$$

The optimal value for $\mathcal{C}$ is the highest $\mathcal{C}_{const}$ for which Equation A.6 becomes unsatisfiable. Equations A.4 and A.5 are SAT clauses and Equation A.6 is a pseudo-Boolean formula. Because the graph construction method in [6] does not depend on the actual value of $\mathcal{C}_{const}$, it can be used directly on Equations A.4 through A.6 to detect the symmetries in the original ILP model.

Graph construction. The graph construction method presented in [6] is reproduced here for the reader's convenience.

- Each variable is represented by two vertices: positive and negative literals (literal vertices).

- Each SAT clause if represented by a single vertex (clause vertices).

- Edges are added between opposite literals (boolean consistency edges).

- Edges are added connecting a clause vertex to its corresponding literal vertices (incidence edges). An optimization here is to connect the literals participating in a *binary* clause directly by an edge, skipping the clause vertex.

- Clause vertices are colored with color 1 and all literal vertices (both positive and negative) with color 2.

- Literals in a pseudo-Boolean constraint P_i are organized as follows:
  - The literals in P_i are sorted by coefficient value; literals with the same coefficient are grouped together. Thus, if there are P different coefficients in P_i, we have R disjoint groups of literals, $L_1, \ldots, L_R$. Note that the only coefficients in the above model are c_i's, the execution times of tasks.
  - For each group of literals L_j with the same coefficient, a single vertex $X_{i,j}$ (coefficient vertex) is created to represent the value. Edges are then added to connect this vertex to each literal vertex in the group.
  - A different color is used for each distinct coefficient value encountered in the formula. This means that coefficient vertices that represent the same coefficient value in different constraints are colored the same.
- Each pseudo-Boolean constraint P_i is itself represented by a single vertex Y_i (PB constraint vertex). Edges are added to connect Y_i to each of the coefficient vertices $X_{i,1}, \ldots, X_{i_R}$.
- The vertices Y_i's are colored according to the constraint's right-hand side value b. Every unique value b implies a new color and vertices representing different constraints with the same RHS value are colored the same. Note that in the above model, there in only one unique RHS value, namely $\mathcal{C}_{const}$. Therefore, all Y_i's are given the same color in this case.

Figure A.1 shows the graph constructed for the model that maps 3 tasks to 3 processors. The figure also shows the matrix of 0/1 variables used by the model and

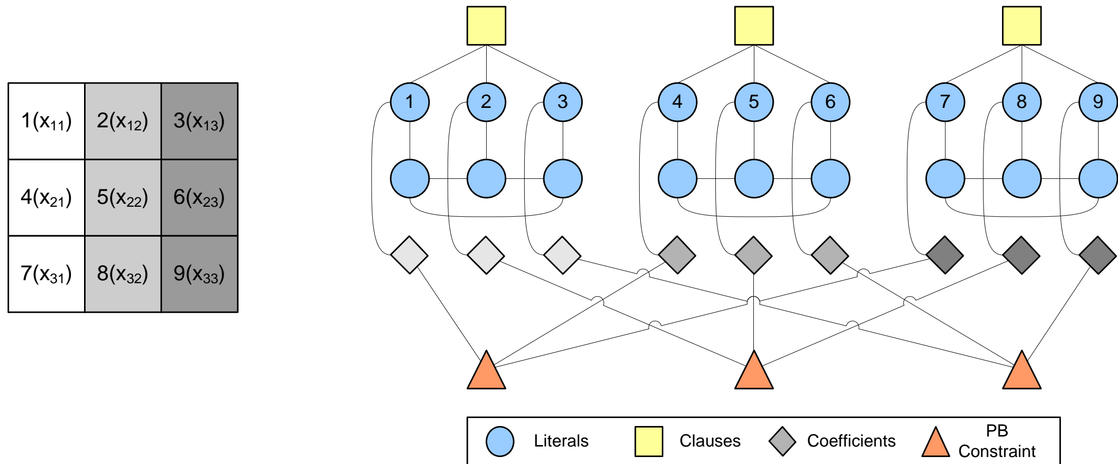


Figure A.1: Graph construction for the example problem instance.

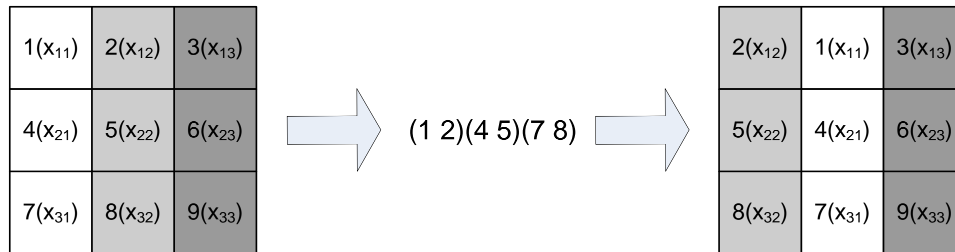


Figure A.2: Generator 1 from Saucy's output for 3x3 graph

their corresponding vertex numbers.

Symmetry discovery. The tool Saucy [20] is used for symmetry discovery. Given a graph, a symmetry (also called an automorphism) is a permutation of its vertices that maps edges to edges. The set of symmetries of a graph is closed under composition and is known as the *automorphism group* of the graph. Saucy outputs the *generators* of the automorphism group of the given graph. Generators are the irreducible set of group members, in this case permutations, from which all other members can be "generated" by repeated composition.

Figures A.2 and A.3 show the 2 generators output by Saucy for the $3x3$ problem

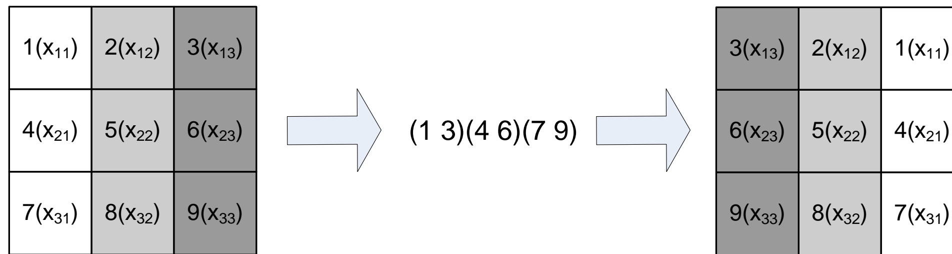


Figure A.3: Generator 2 from Saucy's output for 3x3 graph

shown in Figure A.1. The generators are output in the *cyclic form*. Note that the generator in Figure A.2 swaps column 1 with column 2 of the 0/1 variable matrix, while the generator in Figure A.3 swaps columns 1 and 3. The output of Saucy for all other configurations is similar, i.e., the list of generators output by Saucy swaps column 1 with every other column. It is easy to see that all permutations of the columns can be generated from these permutations. This is because the two generators provide the capability to swap any two columns (by using column 1 as the temporary column). Therefore, for M tasks with distinct execution times, the one and only symmetry present in the ILP model presented in Equations A.1 through A.3 is the permutation of columns.

A.3 Exploiting Symmetry

As noted in [19], symmetries induce an equivalence relation on the set of solutions. The solver can be sped up by considering only a few representatives (maybe only one) from each equivalence class. A classical approach to exploit symmetries is to add constraints that cut off equivalent copies of solutions. [19, 57, 5, 4] show how

this can be done in the context of SAT solving. They present symmetry breaking predicates (SBPs), that are clauses conjoined to the original CNF formula. SBPs restrict solutions to lexicographically smallest assignments in each equivalence class (lex-leaders). The work in [62, 48] describes linear symmetry breaking constraints that can be used in the context of integer linear programs. [45] uses a different approach to exploit symmetry. Instead of adding symmetry breaking constraints and using an off-the-shelf solver, symmetry is used during the solving process. Symmetries are used to devise cutting planes for the branch-and-cut method.

Linear constraints are used to enforce lexicographically ordered columns to exploit symmetry in the task assignment ILP model. Note that the model has a $M \times N$ matrix of 0/1 variables. Let us denote the column j of the matrix, $\{x_{ij}\}, i \in \{1, \ldots, M\}$, by X_j. A column is simply treated as a string of 0/1 characters, and the lexicographic relation follows the usual convention. More formally [6], $X_j \leq_{lex} X_k$ if the following boolean formula is satisfied.

$$\bigwedge_{i=1}^{M} \left[\bigwedge_{l=1}^{i-1} (x_{lj} = x_{lk}) \right] \rightarrow (x_{ij} \leq x_{ik}) \tag{A.7}$$

Equation A.7 has to be converted to a set of linear constraints, so that it can be added to the ILP model. For the sake of convenience, let us denote the bit strings by $A = a_1, a_2, \ldots, a_M$ and $B = b_1, b_2, \ldots, b_M$. A set of M variables ze_i and the following constraints are introduced.

$$ze_i + a_i + b_i \geq 1 \qquad i \in \{1, \ldots, M\} \tag{A.8}$$

$$ze_i + 1 \geq a_i + b_i \qquad i \in \{1, \ldots, M\} \tag{A.9}$$

$$a_i + 1 \geq ze_i + b_i \qquad i \in \{1, \ldots, M\} \tag{A.10}$$

$$b_i + 1 \geq ze_i + a_i \qquad i \in \{1, \ldots, M\} \tag{A.11}$$

ze_i is an *indicator* variable with the following property : $ze_i = 1 \Leftrightarrow a_i = b_i$. It is easy to see that Equation A.7 can be represented directly using ze_i as follows:

$$a_1 \leq b_1 \tag{A.12}$$

$$\left[\sum_{j=1}^{i-1}(1 - ze_j)\right] + (1 - a_i) + b_i \geq 1 \qquad i \in \{2, \ldots, M\} \tag{A.13}$$

Equation A.13 is basically asserting the boolean formula $(ze_1 \wedge ze_2 \wedge \ldots \wedge ze_{i-1}) \rightarrow (a_i \leq b_i)$, which is equivalent to $\overline{ze_1} \vee \overline{ze_2} \vee \ldots \vee \overline{ze_{i-1}} \vee \overline{a_i} \vee b_i$.

This linearization is used to lexicographically order all the columns, i.e.,

$$X_1 \leq_{lex} X_2 \leq_{lex} \ldots \leq_{lex} X_N \tag{A.14}$$

Note that Equations A.8 through A.13 introduce $M \times (N - 1)$ new variables and $6 \times M \times (N - 1)$ new constraints to the ILP model of assigning M tasks to N processors. The problem of lexicographically ordering columns of a $M \times N$ matrix has been studied extensively in [33] and [23]. Structures called orbitopes are introduced, that are convex hulls of 0/1-matrices that are lexicographically maximal subject to a group acting on the columns. The following compact set of constraints are developed

to completely describe the oribitopes.

$$
\begin{aligned}
z_{i,j} &= \sum_{k=i+1}^{N} x_{i,k} && i \in \{1, \ldots, M\}, j \in \{1, \ldots, N\} \\
w_{i+1,j+1} - w_{i,j+1} &\geq 0 && i \in \{0, \ldots, M-1\}, j \in \{0, \ldots, min(N, i+1)\} \\
w_{i,j} - w_{i+1,j+1} &\geq 0 && i \in \{1, \ldots, M\}, j \in \{1, \ldots, N\}, i < M \\
w_{M,1} &\leq 1 && \\
w_{i,j} - w_{i-1,j} - z_{i,j} + z_{i,j+1} &\leq 0 && i \in \{1, \ldots, M\}, j \in \{1, \ldots, N\} \\
z_{i,j} - w_{i,j} &\leq 0 && i \in \{1, \ldots, M\}, j \in \{1, \ldots, N\} \\
w_{i,min(i,N)} &\geq 0 && \qquad \text{(A.15)}
\end{aligned}
$$

$w_{i,j}$ and $z_{i,j}$ are new auxillary variables introduced into the model. Equations A.15 introduce $2 \times M \times N$ new variables and $4 \times M \times N$ constraints. The proof of the fact that these set of inequalities completely describe orbitopes is quite involved and extends over 10 pages. In contrast, the inequalities given by Equations A.8 through A.13 are straightforward, as they are direct linearizations of the boolean formula defining lexicographic properties.

A.4 Experiments

The effectivess of symmetry breaking constraints on the task assignment ILP model was evaluated using the mixed integer program solvers SCIP [8] and CPLEX [1]. CPLEX (version 10.2.0) is a commercial industry strength solver. CPLEX solver's runtime on small problem instances (assigning tasks to say, 8 processors) is small, of the order of few seconds. Observing speedup due to symmetry breaking constraints

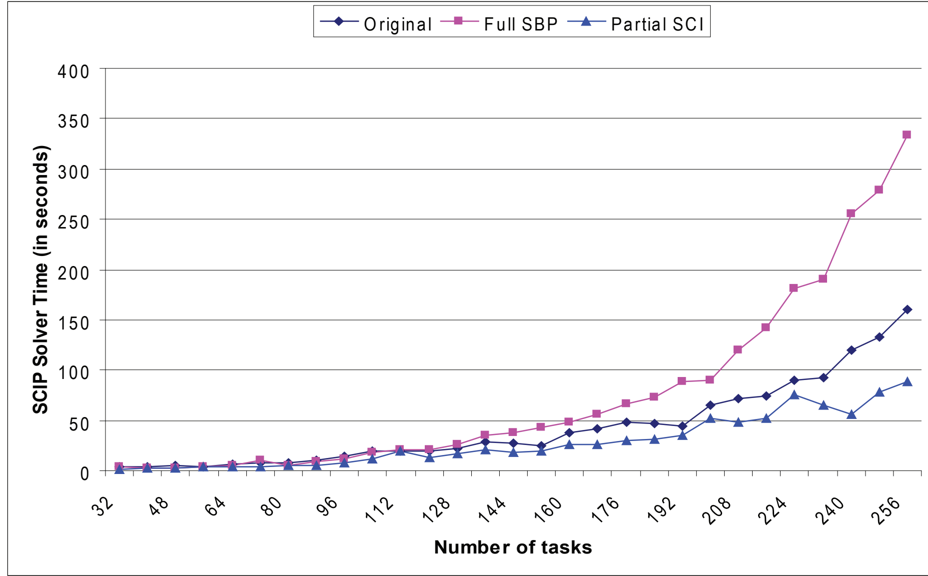


Figure A.4: Time taken by SCIP solver on ILP models for assigning tasks to 8 processors.

on small problem instances is quite hard with CPLEX. Therefore, the open source solver SCIP (version 1.00) was used for the experiments involving small instances.

Figure A.4 shows the runtime of SCIP on problem instances assigning tasks to 8 processors. The x-axis represents increasing number of tasks, and the y-axis represents the time taken, in seconds, by the SCIP solver. Three versions of the models were evaluated. The version marked "Original" in the graph consists of Equation A.1 through A.3. The verion marked "Full-SBP" containts the full set of symmetry breaking constraints, consisting of Equations A.8 through A.13, in addition to the original equations. Note that the time taken by the solver on the model with full symmetry breaking constraints is higher, up to 2x in the worst case. Even though the constraints restrict the solution space by a large amount, the $(6 \times M \times (N-1))$ extra constraints cause the LP relaxations in the branch-and-bound search to become much harder.

It is clear that using the full set of symmetry breaking constraints is detrimental to the performance of the solver. A reduced set of constraints was developed, which do not fully describe 0/1 matrices with lexicographically sorted columns, and therefore do not restrict the solution space as much as the full set of constraints. However, the smaller number of constraints is attractive because they are not likely to make the LP relaxations harder. Since a task can be assigned to one and only one processor, each row in the variable matrix can contain only a single 1. To have lexicographically decreasing column, this "single 1 per row" constraint forces the elements above the leading diagonal to become zeroes. The following equation enforces that constraint.

$$\sum_{k=j+1}^{N} x_{i,k} = 0 \qquad i \in \{1, \ldots, N-1\} \tag{A.16}$$

[33] proposes *Shifted Column Inequalities* (SCIs) which completely describe orbitopes, the 0/1 matrices with lexicographically sorted columns. However, the number of SCIs are exponential in M and N. The following equation captures a subset of size $M \times N$, of the SCIs.

$$\sum_{k=j}^{N} x_{i,k} - \sum_{l=j-1}^{i-1} x_{l,j-1} \leq 0 \qquad i \in \{2, \ldots, M\}, j \in \{2, \ldots, N\} \tag{A.17}$$

The reasoning behind Equation A.17 can be seen pictorially in Figure A.5.

The third version of the ILP model marked "Partial-SCI" in Figure A.4 consists of Equations A.16 and A.17 in addition to the original set of equations. Note that the solver performs much better with these reduced set of symmetry breaking constraints. The solver performance improved by up to 30% because of the symmetry breaking constraints. Figure A.6 shows the runtime of the CPLEX solver on problem

$$\sum_{k=j}^{N} x_{i,k} - \sum_{l=j-1}^{i-1} x_{l,j-1} \leq 0$$

Figure A.5: Shifted column inequality.

Figure A.6: Time taken by CPLEX solver on ILP models for assigning tasks to 32 processors.

instances assigning tasks to 32 processors. The version marked “Original” consists of the original set of equations. CPLEX has in-built symmetry-breaking optimizations. The “Original” version has this symmetry breaking optimization turned on. An experiment was performed to measure the runtime of the solver with this optimization turned off (`set pre symm 0` in the interactive optimizer). The solver’s performance was worse by 2-3x without the internal symmetry breaking optimizations, and it is not shown in the graph. Also, the performance of the solver with the full set of symmetry breaking constraints represented by Equations A.8 through A.13 was worse by 2x, and is not shown in the graph. This can again be attributed to the fact that the large set of constraints and extra variables make the LP relaxation instances much harder to solve. The version marked “Partial-SCI” in Figure A.6 contains the reduced set of symmetry breaking constraints. Note that the internal symmetry breaking was turned off for this version. Speedups of up to 25% was observed with the reduced set of symmetry breaking constraints. This shows that problem specific symmetry breaking is superior to the CPLEX’s internal generic optimizations.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] CPLEX. http://www.ilog.com/products/cplex/.

[2] M. Adiletta, M. Rosenbluth, and D. Bernstein. The next generation of Intel IXP network processors. *Intel Technology Journal*, 6(3):6–18, Aug. 2002.

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

[4] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Generic ILP versus specialized 0-1 ILP: an update. In *Proc. of the 2002 International Conference on Computer Aided Design*, pages 450–457, New York, NY, USA, 2002. ACM.

[5] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Solving difficult sat instances in the presence of symmetry. In *Proc. of the 39th Design Automation Conference*, pages 731–736, New York, NY, USA, 2002. ACM.

[6] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Symmetry breaking for pseudo-Boolean formulas. *Journal of Experimental Algorithmics*, 12(1):1.3, 2008.

[7] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: a programming model for the cell be architecture. *Proc. of Supercomputing '06*, 00(1):5, 2006.

[8] T. Berthold et al. Solving constraint integer programs. http://scip.zib.de.

[9] F. Blagojevic, D. S. Nikolopoulos, A. Stamatakis, and C. D. Antonopoulos. Dynamic multigrain parallelization on the cell broadband engine. In *Proc. of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 90–100, New York, NY, USA, 2007. ACM Press.

[10] W. Blume et al. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, Dec. 1996.

[11] F. Brewer and D. Gajski. Chippe: A system for constraint driven behavioral synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(7):681–695, July 1990.

[12] I. Buck et al. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, Aug. 2004.

[13] M. Chen, X. Li, R. Lian, J. Lin, L. Liu, T. Liu, and R. Ju. Shangri-la: Achieving high performance from compiled network applications while enabling ease of programming. In *Proc. of the SIGPLAN '05 Conference on Programming Language Design and Implementation*, pages 224–236, June 2005.

[14] C. M. Chu et al. Hyper: An interactive synthesis environment for high performance real time applications. In *Proc. of the 1989 International Conference on Computer Design*, pages 432–435, Oct. 1989.

[15] R. Composano. Design process model in the yorktown silicon compiler. In *Proc. of the 25st Design Automation Conference*, pages 489–494, Dec. 1988.

[16] C. Consel et al. Spidle: A DSL approach to specifying streaming applications. In *Proc. of the* 2nd*Intl. Conference on Generative Programming and Component Engineering*, pages 1–17, 2003.

[17] K. Cooper et al. The ParaScope parallel programming environment. *Proc. of the IEEE*, 81(2):244–263, Feb. 1993.

[18] J. Crawford. A theoretical analysis of reasoning by symmetry in first-order logic, 1992.

[19] J. Crawford, E. Luks, M. Ginsberg, and A. Roy. Symmetry-breaking predicates for search problems. In *Proc. of the 1996Principles of Knowledge Representation and Reasoning*, pages 148–159, Aug. 1996.

[20] P. T. Darga, K. A. Sakallah, and I. L.Markov. Faster symmetry discovery using sparsity of symmetries. In *Proc. of the 45th Design Automation Conference*, June 2008.

[21] S. Devadas and R. Newton. Data path synthesis from behavioral descriptions: An algorithmic approach. In *Proc. of the 1987 International Symposium on on Circuits and Systems*, pages 398–401, May 1987.

[22] W. Eatherton. The push of network processing to the top of the pyramid, 2005. Keynote address: Symposium on Architectures for Networking and Communications Systems.

[23] Y. Faenza and V. Kaibel. Extended formulations for packing and partitioning orbitopes, 2008. arXiv:0806.0233v1.

[24] K. Fan, M. Kudlur, H. Park, and S. Mahlke. Cost sensitive modulo scheduling in a loop accelerator synthesis system. In *Proc. of the 38th Annual International Symposium on Microarchitecture*, pages 219–230, Nov. 2005.

[25] K. Fan, M. Kudlur, H. Park, and S. Mahlke. Increasing hardware efficiency with multifunction loop accelerators. In *Proc. of the 4th International Conference on Hardware/Software Codesign and System Synthesis*, pages 276–281, Oct. 2006.

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

[27] M. I. 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. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, Oct. 2002.

[28] J. Gummaraju and M. Rosenblum. Stream programming on general-purpose processors. In *Proc. of the 38th Annual International Symposium on Microarchitecture*, pages 343–354, Washington, DC, USA, 2005. IEEE Computer Society.

[29] S. Ha and E. A. Lee. Compile-time scheduling and assignment of data-flow program graphs with data-dependent iteration. *IEEE Transactions on Computers*, 40(11):1225–1238, 1991.

[30] M. Hall et al. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, Dec. 1996.

[31] H. P. Hofstee. Power efficient processor design and the Cell processor. In *Proc. of the 11th International Symposium on High-Performance Computer Architecture*, pages 258–262, Feb. 2005.

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

[33] V. Kaibel and M. E. Pfetsch. Packing and partitioning orbitopes, 2006. arXiv:math/0603678.

[34] G. Karypis and V. Kumar. *Metis: A Software Package for Paritioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Orderings of Sparce Matrices*. University of Minnesota, Sept. 1998.

[35] T. J. Knight, J. Y. Park, M. Ren, M. Houston, M. Erez, K. Fatahalian, A. Aiken, W. J. Dally, and P. Hanrahan. Compilation for explicitly managed memory hierarchies. In *Proc. of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 226–236, New York, NY, USA, 2007. ACM Press.

[36] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro*, 25(2):21–29, Feb. 2005.

[37] D. J. Kuck. *The Structure of Computers and Computations*. John Wiley and Sons, New York, NY, 1978.

[38] M. Kudlur, K. Fan, and S. Mahlke. Streamroller: Automatic synthesis of prescribed throughput accelerator pipelines. In *Proc. of the 4th International Conference on Hardware/Software Codesign and System Synthesis*, pages 270–275, Oct. 2006.

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

[40] M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *Proc. of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–327, 1988.

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

[42] E. A. Lee and D. Messerschmitt. Pipeline interleaved programmable dsp's: Synchronous data flow programming. *IEEE Transactions on Signal Processing*, 35(9):1334–1345, 1987.

[43] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1):24–35, 1987.

[44] Y. Lin, M. Kudlur, S. Mahlke, and T. Mudge. Hierarchical coarse-grained stream compilation for software defined radio. In *Proc. of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 115–124, Oct. 2007.

[45] F. Margot. Pruning by isomorphism in branch-and-cut. *Lecture Notes In Computer Science*, 2081(1):304+, 2001.

[46] W. Mark, R. Glanville, K. Akeley, and J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. In *Proc. of the* 30th *International Conference on Computer Graphics and Interactive Techniques*, pages 893–907, July 2003.

[47] P. Marwedel. The MIMOLA system: Detailed description of the system software. In *Proc. of the 30th Design Automation Conference*, pages 59–63, 1993.

[48] I. Méndez-Díaz and P. Zabala. A polyhedral approach for graph coloring. *Electronic Notes in Discrete Mathematics*, 7(1):1–4, Apr. 2001.

[49] G. D. Micheli and D. C. Ku. HERCULES: System for high-level synthesis. In *Proc. of the 25st Design Automation Conference*, pages 483–488, May 1988.

[50] S. Note, W. Geurts, F. Catthoor, and H. D. Man. Cathedral-III: Architecture-driven high-level synthesis for high throughput DSP applications. In *Proc. of the 28th Design Automation Conference*, pages 597–602, June 1991.

[51] Nvidia. *CUDA Programming Guide*, June 2007. http://developer.download.nvidia.com/compute/cuda.

[52] E. Nystrom, H.-S. Kim, and W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Proc. of the 11th Static Analysis Symposium*, pages 165–180, Aug. 2004.

[53] K. Parhi and D. Messerschmitt. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. *IEEE Transactions on Computers*, 40(2):178–195, 1991.

[54] 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, University of California, Berkeley, May 1995.

[55] R. Rabbah, I. Bratt, K. Asanovic, and A. Agarwal. Versatility and versabench: A new metric and a benchmark suite for flexible architectures. Technical Report MIT-LCS-TM-646, Massachusetts Institute of Technology, June 2004.

[56] L. Ramachandran, V. Chaiyakul, and D. D. Gajski. Vhdl synthesis system (vss) user's manual version 5.0. Technical Report ICS-TR-92-52, University of California at Irvine, June 1992.

[57] A. Ramani, F. A. Aloul, I. L. Markov, and K. A. Sakallah. Breaking instance-independent symmetries in exact graph coloring. In *Proc. of the 2004 Design, Automation and Test in Europe*, page 10324, Washington, DC, USA, 2004. IEEE Computer Society.

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

[59] B. R. Rau, M. S. Schlansker, and P. P. Tirumalai. Code generation for modulo scheduled loops. In *Proc. of the 25th Annual International Symposium on Microarchitecture*, pages 158–169, Nov. 1992.

[60] J. Sánchez and A. González. Modulo scheduling for a fully-distributed clustered VLIW architecture. In *Proc. of the 33rd Annual International Symposium on Microarchitecture*, pages 124–133, Dec. 2000.

[61] R. Schreiber et al. PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. *Journal of VLSI Signal Processing*, 31(2):127–142, 2002.

[62] H. D. Sherali and J. C. Smith. Improving discrete model representations via symmetry considerations. *Management Science*, 47(10):1396–1407, 2001.

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

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

[65] D. E. Thomas, E. M. Dirkes, R. A. Walker, J. V. Rajan, J. A. Nestor, and R. L. Blackburn. The system architect's workbench. In *Proc. of the 25st Design Automation Conference*, pages 337–343, Dec. 1988.

[66] Trimaran. An infrastructure for research in ILP, 2000. http://www.trimaran.org/.

[67] J. D. Valois. *Lock-free data structures*. PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, USA, 1996.

[68] H. L. Verge, C. Mauras, and P. Quinton. The ALPHA language and its use for the design of systolic arrays. *Journal of VLSI Signal Processing*, 3(3):173–182, 1991.

[69] S. wei Liao, Z. Du, G. Wu, and G.-Y. Lueh. Data and computation transformations for brook streaming applications on multiprocessors. *Proc. of the 2006 International Symposium on Code Generation and Optimization*, 0(1):196–207, 2006.

[70] R. P. Wilson et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, 1994.

[71] D. Zhang, Z. Li, H. Song, and L. Liu. A programming model for an embedded media processing architecture. In *Proc. of the* 5th*International Symposium on Systems, Architectures, Modeling, and Simulation*, volume 3553 of *Lecture Notes in Computer Science*, pages 251–261, July 2005.