Generating, Optimizing, and Scheduling a Compiler Level Representation of Stream Parallelism

Jeffrey M. Fifield
University of Colorado at Boulder, fifield@mtnhigh.net

Follow this and additional works at: https://scholar.colorado.edu/csci_gradetds

Part of the Computer Sciences Commons

Recommended Citation
https://scholar.colorado.edu/csci_gradetds/33

This Dissertation is brought to you for free and open access by Computer Science at CU Scholar. It has been accepted for inclusion in Computer Science Graduate Theses & Dissertations by an authorized administrator of CU Scholar. For more information, please contact cuscholaradmin@colorado.edu.
Generating, Optimizing, and Scheduling a Compiler Level Representation of Stream Parallelism

by

Jeffrey M. Fifield

B.S., University of Michigan, 2000

M.S., University of Colorado, 2006

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2011
This thesis entitled:
Generating, Optimizing, and Scheduling a Compiler Level Representation of Stream Parallelism
written by Jeffrey M. Fifield
has been approved for the Department of Computer Science

Dirk Grunwald

Jeremy Siek

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.
Stream parallelism is often cited as a powerful programming model for expressing parallel computation for multi-core and heterogeneous computers. It allows programmers to concisely describe the concurrency and communication requirements found in a program and it allows compilers and runtime systems to easily generate efficient code targeting parallel hardware. This type of stream parallelism is often restricted to use the Synchronous Dataflow (SDF) model and implemented using static compilation and scheduling techniques. While powerful, SDF and the associated static methods have real limitations when applied to general purpose programming on general purpose hardware.

To increase generality, we can define stream parallelism as a graph of processes communicating with one another over unidirectional data channels. Although dynamic scheduling techniques have been developed for this more general model, the powerful compiler transformations that are available under the SDF model no longer apply. This is made worse by the fact that general purpose models are typically implemented as software frameworks on top of high-level general purpose languages.

The Stream and Kernel Intermediate Representation (SKIR) is a compiler level representation of stream parallelism for general purpose languages. A SKIR compiler is able to recognize and take advantage of SDF style parallelism while allowing more general programs. This thesis presents the SKIR program representation and describes how it can be used as compiler target for several different high level languages. We show a dynamic scheduling mechanism for SKIR programs based on the concepts of coroutines and task stealing. We also propose code optimizations to reduce runtime overhead associated with dynamic scheduling. Such techniques are not possible in a high-level software framework and provide performance that meets or exceeds the performance
of existing systems while providing greater generality and portability than static methods.
Contents

Chapter

1 Introduction

1.1 Streaming Models .................................................. 3
1.2 Limitations of Existing Programming Methods .................. 5
  1.2.1 Dynamic Applications ....................................... 6
  1.2.2 Portability ..................................................... 10
1.3 Contributions ...................................................... 11

2 Language Constructs for Stream Parallel Computation ......... 15

2.1 Kernels ............................................................. 16
2.2 Streams ........................................................... 21
2.3 Stream Graph Construction ....................................... 22
2.4 Scheduling ........................................................ 24

3 The Stream and Kernel Intermediate Representation (SKIR) .... 25

3.1 Introduction ......................................................... 25
3.2 SKIR Kernels ....................................................... 26
  3.2.1 Hierarchical Kernels ....................................... 32
3.3 SKIR Streams ..................................................... 33
3.4 SKIR Memory Model ............................................. 34
3.5 SKIR Program Construction ..................................... 36
3.5.1 Example 1: Static Stream Graph .................................................. 36
3.5.2 Example 2: Dynamic Stream Graph ........................................... 36
3.6 Implementation .............................................................................. 39
4 Compiling High-Level Languages to SKIR ...................................... 41
  4.1 Compiler Intrinsics: C Language .................................................. 41
    4.1.1 Example ............................................................................. 42
    4.1.2 Implementation ................................................................. 42
    4.1.3 Discussion ........................................................................ 44
  4.2 User Library: C++ ...................................................................... 44
    4.2.1 Example ............................................................................. 45
    4.2.2 Implementation ................................................................. 47
  4.3 Stream Language Front-end: StreamIt ........................................ 50
    4.3.1 Generating Filters .............................................................. 50
    4.3.2 Generating Pipelines and Split-Joins .................................... 54
    4.3.3 Other Language Features .................................................. 58
  4.4 Embedded DSL: JavaScript ......................................................... 60
    4.4.1 Sluice ................................................................................. 62
    4.4.2 Dynamic Recompilation for SKIR ....................................... 64
    4.4.3 Offloading Kernel Execution ............................................. 66
    4.4.4 Discussion ........................................................................ 68
  4.5 Conclusion .................................................................................. 69
5 SKIR Dynamic Scheduling ............................................................... 70
  5.1 Stream Implementation ............................................................... 71
  5.2 Single Threaded Execution .......................................................... 73
  5.3 Multithreaded Execution ............................................................. 80
  5.4 Discussion ................................................................................. 83
6 SKIR Compilation

6.1 Static vs. Dynamic SKIR Compilation ............................................. 87
6.2 Kernel Analysis ................................................................. 90
   6.2.1 Detecting Hierarchical Kernels ............................................. 90
   6.2.2 Detecting Data Parallelism ................................................ 91
   6.2.3 Determining Rates ......................................................... 91
6.3 Kernel Specialization ............................................................. 94
   6.3.1 Kernel Batching ............................................................. 95
   6.3.2 Generating Kernels as Coroutines ....................................... 96
   6.3.3 Coroutine Elimination ..................................................... 101
   6.3.4 Specialization Templates ................................................ 103
   6.3.5 Summary ................................................................. 106
6.4 Code Generation ................................................................. 108
6.5 Compiling for GPUs ............................................................ 108
6.6 Stream Graph Level Transformations .......................................... 112
   6.6.1 Fusion: Reducing Parallelism ............................................. 113
   6.6.2 Fission: Increasing Parallelism ......................................... 115

7 Performance Evaluation ............................................................ 118

7.1 Benchmarks ................................................................. 118
7.2 Results ................................................................. 121
7.3 Sluice ................................................................. 134
   7.3.1 Single Threaded Offload ................................................ 136
   7.3.2 Task Parallel Execution ................................................. 138
   7.3.3 Data Parallel Execution ................................................. 139

8 Related Work ................................................................. 141

8.1 Compiler Representations for Stream Parallelism ....................... 141
8.2 Dynamic Scheduling of Stream Parallelism ............................... 143
8.3 Stream Parallelism for Heterogeneous Hardware ............................ 145

9 Conclusion ............................................................................. 146

Bibliography ............................................................................. 148
# Tables

## Table

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Overview of the Stream and Kernel Intermediate Representation (SKIR) operations.</td>
</tr>
<tr>
<td>7.1</td>
<td>The number and size of kernels for each benchmark. We bin the kernels in each benchmark according to their execution time in cycles. The table shows the number of kernels in each bin.</td>
</tr>
</tbody>
</table>
## Figures

### Figure

1.1 Pipeline found in the *dedup* benchmark. ........................................... 7
1.2 Prime sieve implemented in the Go language. ..................................... 9

2.1 A data parallel addition written AMD’s Brook+ language ....................... 16
2.2 A four stage pipeline in the language proposed by Kahn and MacQueen [40]. . 17
2.3 A four stage pipeline written in the StreamIt language. .......................... 18
2.4 A kernel object in the GNU Radio stream processing system. ..................... 19
2.5 An FM receiver application created for the GNU Radio stream processing framework using a graphical user interface. ................................. 23
2.6 Constructing a stream graph for an equalizer in StreamIt. ........................ 24

3.1 SKIR kernel state transitions ............................................................... 30
3.2 An example of a static SKIR program graph. The `main` procedure defines and executes a four stage pipeline similar to the one in Figure 2.2. The program is written in SKIR pseudo-code................................................................. 37
3.3 An example of a dynamic SKIR program graph. This figure shows the prime sieve program written as SKIR pseudo-code. It is similar to the Go program shown in Figure 1.2. ................................................................. 38

4.1 SKIR C Intrinsics. Each C intrinsic maps to one SKIR intrinsic ................. 42
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>A simple SKIR program constructed using C compiler intrinsics. This program creates two streams of integers, adds them together, then prints the result.</td>
</tr>
<tr>
<td>4.3</td>
<td>An example of a SKIR C++ program using the C++ user library from Figure 4.4.</td>
</tr>
<tr>
<td>4.4</td>
<td>An example of a C++ library providing an object oriented interface for stream parallelism using the SKIR C intrinsics.</td>
</tr>
<tr>
<td>4.5</td>
<td>A simple three stage pipeline written in StreamIt</td>
</tr>
<tr>
<td>4.6</td>
<td>The result of compiling the IntSource filter from Figure 4.5 with the StreamIt to SKIR compiler.</td>
</tr>
<tr>
<td>4.7</td>
<td>The result of compiling the Adder filter from Figure 4.5 with the StreamIt to SKIR compiler.</td>
</tr>
<tr>
<td>4.8</td>
<td>The result of compiling the Main filter from Figure 4.5 with the StreamIt to SKIR compiler.</td>
</tr>
<tr>
<td>4.9</td>
<td>The pipeline filter from the StreamIt to SKIR compiler runtime library.</td>
</tr>
<tr>
<td>4.10</td>
<td>A generic duplicating splitter from the StreamIt to SKIR compiler runtime library.</td>
</tr>
<tr>
<td>4.11</td>
<td>An example of a StreamIt prework function.</td>
</tr>
<tr>
<td>4.12</td>
<td>An example of a StreamIt feedback loop.</td>
</tr>
<tr>
<td>4.13</td>
<td>Array based vs. stream based addition in JavaScript</td>
</tr>
<tr>
<td>4.14</td>
<td>Two simple Sluice kernels. The kernel on the left implements a simple counter. The kernel on the right adds the same value to each item popped the input stream and pushes the result to its output stream.</td>
</tr>
<tr>
<td>4.15</td>
<td>An overview of Sluice kernel offload.</td>
</tr>
<tr>
<td>5.1</td>
<td>Basic lock free queue operations</td>
</tr>
<tr>
<td>5.2</td>
<td>Simplified version of the SKIR stream buffer data structure</td>
</tr>
<tr>
<td>5.3</td>
<td>An example of producer-consumer coroutines</td>
</tr>
<tr>
<td>5.4</td>
<td>Yielding lock free queue operations</td>
</tr>
<tr>
<td>5.5</td>
<td>An example of transforming a SKIR kernel to use coroutine scheduling.</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>5.6</td>
<td>A single threaded scheduling algorithm for coroutine style kernel work functions.</td>
</tr>
<tr>
<td>5.7</td>
<td>An example of single threaded execution of a three stage pipeline. The large downward arrows represent work being done by a kernel, the dotted lines represent work stealing, and the horizontal arrows represent scheduling events. Time flows down.</td>
</tr>
<tr>
<td>5.8</td>
<td>The <code>KERNEL_TASK::EXECUTE</code> method runs the kernel work function and makes scheduling decisions in conjunction with a work stealing algorithm. The next task to run is returned.</td>
</tr>
<tr>
<td>5.9</td>
<td>An example of two processors executing a three stage pipeline. The large downward arrows represent work being done by a kernel, the dotted lines represent work stealing, and the horizontal arrows represent scheduling events. Time flows down.</td>
</tr>
<tr>
<td>6.1</td>
<td>An example of a SKIR C program with a statically analyzable stream graph.</td>
</tr>
<tr>
<td>6.2</td>
<td>Applying batching to <code>SKIR_CONSUMER</code> produces <code>BATCHED_CONSUMER</code>.</td>
</tr>
<tr>
<td>6.3</td>
<td>Kernel work function template implementing batching.</td>
</tr>
<tr>
<td>6.4</td>
<td>An implementation of the <code>skir.pop</code> operation.</td>
</tr>
<tr>
<td>6.5</td>
<td>SKIR coroutine helper function for x86-64.</td>
</tr>
<tr>
<td>6.6</td>
<td>Implementation of <code>skir.yield</code> for x86-64.</td>
</tr>
<tr>
<td>6.7</td>
<td>Implementation of <code>skir.return</code> for x86-64.</td>
</tr>
<tr>
<td>6.8</td>
<td>A coroutine elimination work function template.</td>
</tr>
<tr>
<td>6.9</td>
<td>An implementation of <code>skir.pop</code> for use with coroutine elimination.</td>
</tr>
<tr>
<td>6.10</td>
<td>Generating specialized <code>skir.pop</code> templates with meta-programming.</td>
</tr>
<tr>
<td>6.11</td>
<td>Execution order of SKIR kernel specialization passes.</td>
</tr>
<tr>
<td>6.12</td>
<td>An example of the one work function instance per iteration execution strategy used by the OpenCL backend. In this example the kernel has an input rate of 2 and an output rate of 3.</td>
</tr>
<tr>
<td>6.13</td>
<td>The structure of OpenCL offload within the SKIR runtime.</td>
</tr>
<tr>
<td>6.14</td>
<td>SKIR kernel fusion algorithm.</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>6.15</td>
<td>SKIR kernel fusion algorithm.</td>
</tr>
<tr>
<td>6.16</td>
<td>SKIR kernel fission.</td>
</tr>
<tr>
<td>7.1</td>
<td>A SKIR version of the CalculateForces kernel from the nbbody benchmark. The code is written in C++ using a class library interface to the SKIR intrinsics.</td>
</tr>
<tr>
<td>7.2</td>
<td>Throughput obtained by the SKIR compiler and scheduler for the StreamIt benchmarks using a varying number of threads. Error bars indicate one standard deviation.</td>
</tr>
<tr>
<td>7.3</td>
<td>The results of performing coroutine elimination on the StreamIt benchmarks using the SKIR just-in-time compiler.</td>
</tr>
<tr>
<td>7.4</td>
<td>The results of performing dynamic fusion on the StreamIt benchmarks using the SKIR just-in-time compiler.</td>
</tr>
<tr>
<td>7.5</td>
<td>Throughput obtained by the SKIR dynamic scheduler, the StreamIt static scheduler, and the GNU Radio dynamic scheduler for the StreamIt benchmarks using a varying number of threads. Error bars indicate one standard deviation.</td>
</tr>
<tr>
<td>7.6</td>
<td>Throughput obtained by the SKIR dynamic scheduler, the StreamIt compiler targeting 8 threads, and the StreamIt compiler targeting the actual number of threads. Error bars indicate one standard deviation.</td>
</tr>
<tr>
<td>7.7</td>
<td>A comparison of SKIR performance to the performance of coarse-grained pipeline parallel applications parallelized using Pthreads, TBB, and FastFlow. Performance is measured as execution time.</td>
</tr>
<tr>
<td>7.8</td>
<td>Performance of the nbbody benchmark running on SKIR using kernel fission on the CPU and OpenCL on the GPU.</td>
</tr>
<tr>
<td>7.9</td>
<td>Comparison of Pixastic code (left) with the same code ported to Sluice (right). The code that is shown inverts all the pixels in an image.</td>
</tr>
<tr>
<td>7.10</td>
<td>The CalculateForces kernel found in the nbbody benchmark. Ported from the SKIR C++ version of the benchmark shown in Figure 7.1.</td>
</tr>
</tbody>
</table>
7.11 Performance of Pixastic image processing routines coded as ordinary JavaScript compared to the performance of the same routines running as Sluice kernels using the SKIR runtime.

7.12 Performance of the $nbody$ benchmark implemented as procedural JavaScript, as Sluice code, and as Sluice code with CalculateForces kernel running on SKIR runtime.

7.13 Per image processing time for the Pixastic benchmarks implemented when a varying number of images are processed using Sluice task parallelism.

7.14 Speedup of the $nbody$ benchmark due to data parallelism when the CalculateForces kernel is run on a multi-threaded SKIR runtime compared to the same benchmark using a single-threaded SKIR runtime.

8.1 An example of SVM-C pseudo code.
Chapter 1

Introduction

It has long been clear that the era of rapidly improving single processor performance is over. A decade ago, technological and economic realities forced the semiconductor industry to stop pursuing increased clock speed as the primary means of improving performance. At the same time, computer architects had already extracted much of the performance to be gained from the exploitation of instruction level parallelism and prediction. Nevertheless, Moore’s Law has continued to hold and the number of transistors available to hardware architects has continued to increase. As a result, the semiconductor industry has been forced to provide performance improvements by adding more processing elements per chip with every new silicon generation. Chips with 10’s to 100’s of processing cores are already a reality and will soon become the norm. Even mainstream mobile phones contain multiple processors, vector instruction set extensions, and programmable graphics hardware.

Chip manufacturers are also attempting to differentiate their products and tailor their designs to different markets. The result is increased heterogeneity among the individual cores in single systems and among different versions of the same architecture. Processing cores in the same product family can vary greatly in functionality, performance and energy efficiency. Compute accelerators that were previously fixed function and single purpose are becoming more programable and more general purpose. Looking forward, there is a continued desire to investigate and invest in heterogeneous processor designs, many-core architectures, and processors using reconfigurable logic.

In the recent past, a programmer wanting to improve the performance of a piece of code
could simply wait for the next processor generation and see their program run much faster on the new chip. Faster hardware designs, not better programs, provided most of the performance gains. While this is still true to some extent, the trends in mainstream processor design listed above are causing a shift in who is responsible for the performance of computer software. The burden of taking advantage of new hardware capabilities is shifting from away from processor architects and onto the backs of programmers, languages designers, and compilers.

Unfortunately, writing algorithms and compilers that can take advantage of parallel processor designs is challenging. Creating parallel code that is efficient, correct, and portable is difficult even for the most skilled programmers. Mainstream languages are highly sequential and contain only primitive support for parallel constructs. Common parallel programming constructs such as threads, locks, and SIMD intrinsics are to parallel programing as assembly language is to sequential programming, and are notoriously difficult to use correctly. Decades of research on parallelizing sequential languages has failed to produce compiler technology that can automatically extract parallelism from much more than simple loop nests. Adding heterogeneity into the mix only makes the problem of parallelizing applications more difficult. To sustain the historical performance gains of the computer industry, new programming techniques must be developed for the new hardware architectures reaching the mainstream market.

Stream parallel programming is one technique that has been used for exploiting parallel hardware, such as multi-core processors and vector acceleration units, as well as non-traditional programmable hardware such as graphics processors (GPU) and field programmable gate arrays (FPGA). In the stream parallel model, a program is made up of a number of computational kernels that produce and consume streams of data. The kernels are linked together via their input and output streams to form a stream graph representing an algorithm. This abstraction is a natural fit for many application domains. Any data driven application that must process large flows of data with fairly regular computation is a good candidate for the stream parallel model. This includes many algorithms that exhibit data or pipeline parallelism such as those found in graphics, all kinds of signal processing (e.g. audio, video, radio), scientific codes, financial applications, network
processing, and most recently, distributed real-time processing of social network feeds [13].

The kernel and stream based construction of stream programs has two benefits. It makes parallelism and inter-task communication easy to express by programmers and it makes parallelism and inter-task communication easy to extract by compilers. There are three general types of parallelism that can be expressed by programmers and extracted by compilers in stream parallel programs. Because individual kernels are conceptually concurrent, with independent threads of control, kernels operating concurrently in different regions of a stream graph naturally exhibit task parallelism. Other kernels might be directly connected to one another in a producer consumer relationship. These kernels exhibit pipeline parallelism and can be scheduled to run on different processors with communication channels allocated between them. Finally, some kernels are stateless, and as a result do not carry dependencies from one execution to the next. These kernels exhibit data parallelism and many instances of the same kernel can be working on different parts of the input stream(s) at the same time.

1.1 Streaming Models

There are two formal models of computation that are generally usually used to describe stream parallel computation. They are Kahn Process Networks and Synchronous Dataflow Networks. A related model of computation that is also popular is Communicating Sequential Processes.

A Kahn Process Network (KPN) consists of a network of deterministic processes (kernels) communicating using unbounded first-in first-out data channels (streams) [39]. These channels are the only means with which processes can communication with one another. In the KPN model, processes can always enqueue data to their output stream, will always block on a read of an empty input stream, and are not allowed to test for the presence or absence of data in the channels. Because of these properties, Kahn process networks are deterministic. That is, the same input on a set of channels will always produce the same output, independent of processes execution order. A KPN is called bounded if it is possible to run the network to completion using a bounded amount of
buffer space between the processes. It is undecidable to statically determine the amount of buffer space required between processes. However, it is possible to dynamically adjust the amount of buffer space such that bounded networks require only bounded buffer space [48].

A restricted form of the KPN model is the Synchronous Dataflow Network (SDF) [43]. Under SDF, the processes in the network execute atomically, reading all their inputs and writing all their outputs in a single execution. If an execution of a kernel cannot run to completion because it lacks input data or lacks output buffer space, then it will not run at all. In a SDF computation, the number of inputs read and written to and from each data channel is fixed and known to the system at compile time. This means that for a given network, the compiler can prove the absence of deadlock, can compute a static execution schedule, and can determine static buffer sizes for that schedule. Because they can be statically analyzed and scheduled, SDF networks generally exhibit more potential for optimization than KPNs.

The Communicating Sequential Process (CSP) model of computation is a formal model closely related to Kahn process networks. A primary difference between KPN and CSP is that CSP communication channels involve synchronous unbuffered rendezvous between the sender and the receiver. That is, a sender cannot transmit a message until a receiver is ready to receive it. Another significant difference is that CSP processes are allowed to non-deterministically read a message from one of a set of input channels. As in KPNs, determining deadlock freedom is undecidable.

The motivation for this thesis is to enable flexible use of stream parallelism in general purpose languages and for a variety of application domains. Stream parallelism is therefore viewed as a slight generalization of the KPN model. We want to allow for programs with dynamic behaviors not supported under the SDF model (including dynamic input and output rates) or the KPN model (e.g. dynamic stream graphs, non-deterministic kernels). As with any programming construct, it is beneficial to write stream parallel programs in a certain way to promote compiler optimization. For stream parallel programs, this simply means writing code in a style close to the SDF model when compiler optimization is most important. An analogy is that C/C++ programmers can write loops following certain restrictions when auto-vectorization, loop unrolling, or other loop
transformations are important, and allow themselves more freedom when it is not.

1.2 Limitations of Existing Programming Methods

The authors of the StreamIt project correctly observed that languages prior to their own have either elegant and general ways for expressing streams, but are too general for aggressive optimization (e.g. functional languages), or provide good analysis and optimization of stream graphs, but are narrowly focused and exist largely outside of programming languages, applying mainly to specific domains (e.g. modeling systems for DSP) [52][32]. As a result, they propose a language, StreamIt, that is very close to the synchronous dataflow model, allowing for high performance implementation, but which also has a programmer friendly syntax [34]. Although they succeeded in their goals, the programming model is still very close to SDF and the language is not practical for general use.

In this dissertation we reject the notion that support for aggressive optimization and support for more general forms of stream parallelism are mutually exclusive. We argue for a middle way; that is, allow for a general form of process networks and apply aggressive SDF style optimization where possible. We can identify two major shortcomings with respect to existing methods of programming using the stream parallel model that prevent this from happening. These shortcomings are the result of the artificial division between support for optimizable stream graphs and support for generality.

The first shortcoming is the lack of stream parallelism in general purpose languages. As alluded to in the motivations for StreamIt cited above, there is little support for writing optimizable stream parallel algorithms in mainstream general purpose languages. If there is support available, it is in the form of a user library or framework which can provide stream parallel abstractions, but which cannot provide stream graph optimization. Recent examples of this approach for the C++ language are GNU Radio, Feedback-Directed Pipeline Parallelism, DoPE, and FastFlow [4][51][49][19]. While these frameworks employ efficient dynamic scheduling mechanisms, they cannot employ the optimization, parallel scheduling, or code generation techniques that make
a specialized stream parallel language attractive. A good example of this is seen in DoPE [49], where the best result required manual kernel fusion. This basic stream graph transformation is easily performed by stream parallel compilers but very difficult to achieve using only a C++ library.

The second shortcoming is the lack of generality in stream parallel languages. While a language like StreamIt is an excellent platform for research into static stream graph transformations, it does nothing to help programmers use stream parallelism in existing general purpose languages. It still requires that programmers use a specialized language of limited scope. In particular, the set of real applications that use only SDF or KPN style computation without requiring other abstractions or third party libraries is very small. Languages specific to stream parallelism feature powerful graph transformation, scheduling, and code generation techniques, but lack the ability to express or execute computation outside of stream parallel models.

Within the category of generality are two very useful features missing from high performance static compilers for stream parallelism. These are support for dynamic applications and support for portability and forward-scaling. It is useful to examine both of these in more detail.

1.2.1 Dynamic Applications

Our first missing feature with respect to generality in stream parallel languages – support for dynamic applications – can be viewed from multiple perspectives. We expand on three of them:

(1) Applications where individual kernels have dynamic performance characteristics.

(2) Applications where an actor outside of the stream graph wishes to reconfigure the graph.

(3) Applications where a kernel inside of the stream graph wishes to reconfigure itself.

As a simple example of the first case, we can consider the dedup program from the Parsec benchmark suite [21]. This application is a data compression/decompression benchmark. The compression algorithm is used by Parsec for actual benchmarking. It is written in a stream parallel style as a five stage dataflow pipeline, pictured in Figure 1.1. The developer of this program
implemented parallel execution with Pthreads and wrote a custom queuing mechanism to deal with stream communication. It would have been much easier to write the program in an environment with support for stream parallelism.

![Pipeline found in the dedup benchmark.](image)

The stages of the *dedup* compression algorithm are unchanging and can be viewed as kernels in a static program graph. Despite this, we cannot use SDF style scheduling or code transformation on the full graph. This is because two of the program stages, ChunkProcess and SendBlock, do not have fixed input and output rates. ChunkProcess conditionally sends its output to one of two destinations depending on whether or not the data being processed hits in a compression cache. If the data is not present in the cache, then the data must be sent to the Compression stage. Otherwise, it can be sent directly to the SendBlock stage. The result is that static compiler and scheduler cannot predict the output rates of ChunkProcess or the input rates of SendBlock. One could make the graph synchronous by tagging data as cached in the ChunkProcess stage, sending all data to the Compress stage, then using the identity operation in Compress for tagged data chunks. But doing this makes the runtime of Compress highly variable and dependent on the particular input data being compressed. While general purpose frameworks with dynamic scheduling can handle this gracefully, stream parallel languages using static scheduling can not.

The second case above – applications where an actor outside of a stream graph wishes to reconfigure the graph – is easily illustrated using an example from software defined radio. In many radio applications the signal processing and modulation schemes for a particular protocol can change over time depending on the radio environment and user requirements. As an example we describe at a high level the structure of a software receiver for orthogonal frequency division multiple access (OFDMA). One place this technology is used is in the WiMAX standard.
In a software implementation of a receiver for OFDMA, we begin with time domain samples converted from the analog domain to the digital domain by an analog-to-digital converter (A/D). The samples obtained from the A/D must be preprocessed before we can apply OFMDA demodulation and obtain the encoded digital information. This preprocessing step includes things like automatic gain control and filtering. Here we encounter the first place where the stream graph might be significantly reconfigured while running. Filter operations are used to smooth data or to eliminate interference from frequencies the receiver is not interested in. The effectiveness of any particular filter is largely determined by the number of taps that are used. Each tap serves as a coefficient in a dot product operation. This dot product operation is performed to produce one output sample for every input sample using a sliding window of data in the input stream. By increasing the number of taps, we generally increase the effectiveness of the filter but we also increase the computation requirements of the filter. One of the benefits of software defined radio is that an algorithm can make this trade-off dynamically. Unfortunately, for a SDF style stream parallel program this means the weighting of the filter kernel might change significantly in the static execution schedule or that buffer sizes related to the windowing operation (i.e. the input rate) must change. Either of these events could require recompilation of the entire stream graph.

After preprocessing, our software OFDMA receiver will perform a Fast Fourier Transform (FFT) operation to transform the signal from the time domain to the frequency domain. Depending on the OFDMA protocol being used, the width of the FFT may, like the number of filter taps, require dynamic modification. The implications of this change with regard to the stream graph are similar to the number taps used by filters. That is, the computation and communication requirements of the FFT can change significantly. After the FFT operation, our partially demodulated OFDMA signal consists of some number of small frequency bands called sub-carriers. Each of the sub-carriers is demodulated using a scheme dictated by the radio environment and user requirements. As with the filter and FFT, dynamic modification of the sub-carrier modulation schemes is desirable, and can have significant implications on the stream graph. After demodulating our individual sub-carriers, we finally have bits. It is common for these bits to be coded using an error
coding scheme such as Viterbi or Reed-Solomon encoding. One of the advantages of software-defined radio is that these too can be dynamically changed. Again, this can have a significant impact on the structure of the stream graph with respect to computation requirements and stream communication rates. It is desirable to have a system that easily accommodates these kinds of dynamic changes to a stream graph.

The third case listed above – a kernel inside of the stream graph that wishes to reconfigure itself – can be illustrated using an amusing example credited to Doug McIlroy but made popular by Rob Pike. The example is the implementation of the prime sieve of Eratosthenes written as a self-modifying pipeline of kernels. A version of the algorithm implemented in the CSP language Go is shown in Figure 1.2 [5].

```go
package main
import "fmt"

// Send the sequence 2, 3, 4, ... to channel 'ch'.
func generate(ch chan int) {
    for i := 2; ; i++ {
        ch <- i // Send 'i' to channel 'ch'.
    }
}

// Copy the values from channel 'in' to channel 'out', removing those divisible by 'prime'.
func filter(in, out chan int, prime int) {
    for {
        i := <-in // Receive value of new variable 'i' from 'in'.
        if i%prime != 0 {
            out <- i // Send 'i' to channel 'out'.
        }
    }
}

// The prime sieve: Daisy-chain filter processes together.
func main() {
    ch := make(chan int) // Create a new channel.
    go generate(ch) // Start generate() as a goroutine.
    for i := 0; i < 100; i++ { // Print the first hundred primes.
        prime := <-ch
        fmt.Println(prime)
        ch1 := make(chan int)
        go filter(ch, ch1, prime)
        ch = ch1
    }
}
```

Figure 1.2: Prime sieve implemented in the Go language.

The sieve program begins by connecting a goroutine computing a sequence of integers to a printer goroutine that prints its input. The sequence of integers starts with the prime number 2.
Each time the printer goroutine receives an input it constructs new a goroutine which filters out all multiples of that number. It then inserts the new goroutine into the pipeline between itself and and its source. By induction, all the numbers received by the printing filter are prime. The result is a pipeline which grows by one each time a new prime is discovered. Although this program is typically given as an example in a CSP language like Go, it is more of a stream parallel style dataflow algorithm than a CSP style rendezvous algorithm. Furthermore, each configuration of the pipeline can be expressed as a KPN computation. More complex examples of this type of recursively defined stream graphs can be found in McIlroy’s paper on computing power sets [46].

A more practical example of this kind of dynamic graph reconfiguration is found in programs where a kernel must perform some initialization work before the stream graph enters a steady state. In stream graphs with feedback loops it might be necessary to initialize the back edge stream in the graph with data before starting the computation. In DSP applications it might be necessary to write some amount of data to an output stream to implement a delay. Both cases are easily implemented by a kernel that pushes some data to an output stream, then replaces itself with another kernel which provides the steady state functionality.

1.2.2 Portability

Another desirable feature with respect to generality in stream parallel languages - support for portability and forward-scaling – is based on the observation that computing platforms and processor designs, even within a single product line of a single architecture, are increasingly diverse and heterogeneous. These trends require that parallel codes written for high performance, like stream parallel algorithms, be written with a certain amount of robustness to architectural variation. Unfortunately, static scheduling and compilation methods fail in this regard, and not just for stream parallel applications. For example, if a program using vector ISA extensions must be compiled to run on a wide variety of end user machines then the developer has two choices. Use the lowest common denominator, and compile for the most widely available (i.e. oldest) ISA extensions, giving up performance on newer processors. Or, somehow ship a “fat binary” which can select
at runtime the appropriate version of code to run. One way to get this for free is to compile to a common intermediate form, such as a virtual machine, and use runtime compilation.

*Forward Scaling* is a term used by Intel to describe the increasing need for dynamic compilation on Intel platforms to provide portability with respect to future (and past) products. They say it is required because “the performance of parallel applications is very sensitive to core count, vector ISA width (e.g., SSE), core-to-core latencies, memory hierarchy design, and synchronization costs,” and because all of these factors are changing rapidly in the x86 architecture [29]. They propose that one solution is to write code using a well defined programming model (in their case parallel arrays) in a way that allows an algorithm to be analyzed at runtime, then re-optimize the program dynamically as appropriate for the algorithm being executed and the hardware that it is executing on. This thesis advocates a similar approach, only for stream parallel computation.

Synchronous dataflow style stream parallelism has been shown to be a good programming model for targeting non-traditional processors such as GPUs [23][38][54], and FPGAs [24][25] [31][36]. However, the portability problems above are made worse when heterogeneous computation is added to the mix. As an example, consider a program where some piece of the code is well suited for execution on a GPU. It may be the case that doing so on one system hurts performance because the cost of communicating data between the CPU and a discrete GPU overwhelms the performance gains. On another system, the same program might have access to low latency shared memory between the CPU and an integrated GPU resulting in increased performance. Achieving the best performance for this kind of a program requires a sophisticated awareness of the execution environment. In the case of stream parallelism, the static optimization techniques of existing stream parallel languages should be coupled with dynamic compilation and execution mechanisms in order to gain the desired runtime platform awareness and portability.

1.3 Contributions

In this dissertation we present a compiler level representation for stream parallel computation which addresses the limitations discussed in the previous section. We describe a general
purpose solution which can be implemented in a fully dynamic and high performance manner. It is general purpose because it is source language and target architecture independent. It is dynamic because it supports dynamic programs, dynamic compilation, and dynamic scheduling. Because it is a compiler level representation, execution can be implemented in a high performance manner.

The underlying thesis for this work can be summarized by the following statement:

**Thesis Statement:** By preserving high level stream graph information, a low level representation can provide portability and generality in the context of stream processing on parallel and heterogeneous hardware. Furthermore, such a representation can be parallelized and executed efficiently on such hardware using generic parallelization mechanisms, without limiting ease of programming.

The contributions of this dissertation can be summarized as follows:

1. **A low level representation for stream parallel computation, SKIR (Chapter 3).** The Stream and Kernel Intermediate Representation (SKIR) allows for the expression of stream parallel computation at the level of a compiler intermediate representation or virtual machine instruction set. It can be used by both stream specific and general purpose programming languages. It allows static and dynamic compilation and execution strategies as well as static and dynamic stream graphs.

2. **A set of case studies showing that SKIR can be used effectively as a compilation target for stream parallel computation written in high level languages (Chapter 4).** We present four very different high level language interfaces to SKIR: a set of C language compiler intrinsics, giving almost direct access to the SKIR primitives; an object oriented C++ user library, similar in style to other C++ frameworks for parallelism; a language front end for StreamIt, currently the standard language for experimentation with static stream program transformations; and a JavaScript to SKIR compiler, showing how even very
high level dynamic languages can benefit from the stream parallel model using a dynamic SKIR implementation.

(3) **A scheme for the dynamic scheduling of stream parallel computation combining cooperative multitasking with randomized task stealing (Chapter 5).** With the advent of parallel programming environments like Cilk and Thread Building Blocks, randomized task stealing has become a popular mechanism for expressing concurrency and obtaining parallel execution. The reasons for this are that task stealing naturally provides good load balancing and low overhead task switching. We show how to combine this technique with one of the oldest mechanisms for cooperative multitasking in stream parallel programs to obtain efficient parallel execution.

(4) **An optimizing just-in-time compiler for stream parallel programs (Chapter 6)** To enable dynamic compilation and optimization of stream parallel programs, we have built a just-in-time (JIT) compiler for SKIR programs. The JIT compiler transforms SKIR kernels into a form that can execute concurrently and provides the implementation of the data stream abstraction. The compiler can also apply static analysis to determine which parts of a stream graph conform to the synchronous data flow model and apply additional kernel optimization and stream graph transformations those kernels and sub-graphs.

(5) **A performance evaluation of the SKIR runtime, JIT compiler, and dynamic scheduler on a diverse set of benchmarks (Chapter 7).** The dynamic SKIR environment is evaluated on two sets of benchmarks. The first is a set of StreamIt benchmarks where SDF semantics are present, low overhead execution is important because kernels are fine grained, and high performance is the primary goal. The performance is compared to a fully dynamic system and a fully static system. The second set of benchmarks is taken from a variety of application domains and includes data parallel and pipeline parallel stream graph topologies. These benchmarks use a variety of implementation technologies and contain one or two compute intense kernels and one or more lightweight kernels. Per-
formance is again the primary goal and SKIR performance is compared with the original implementation of each benchmark.

Additionally, an overview of the stream parallel programming model is given in Chapter 2 and Chapter 8 reviews related work. We conclude in Chapter 9.
Chapter 2

Language Constructs for Stream Parallel Computation

The stream parallel model is widely used for data-centric computations that have high computational intensity, that can be decomposed into independent components, and that use their inputs very few times before throwing them away. This is traditionally true of application domains such as graphics and signal processing but increasingly applies to other application domains as well.

This chapter presents features common to different stream parallel programming languages, differentiating characteristics of different stream parallel programming languages, and defines terminology used in the rest of this dissertation. This is done in the context of four example stream processing systems. It is important that any low level abstraction for stream parallelism – like SKIR – is able to express common high level stream parallel programming constructs.

Stream parallel computation is based on two main concepts: streams and kernels. A stream is typically an unbounded collection of data elements. It is much like a common queue abstraction except that it is allowed to be conceptually infinite in size. It has push and pop operations like a first-in first-out queue where elements are written by pushing data to one end of the stream and read by popping from the other end of the stream.

A kernel is simply a procedure that is specifically designed to execute using streams as its primary inputs and outputs. It may contain state but does not share data with other parts of the program except using streams. To create a stream parallel computation, streams and kernels are connected together to form a stream graph. The stream graph is a directed graph with kernels as the nodes and streams as the edges. Stream data flows along the edges in the graph and is processed
by the kernels at the nodes in the graph. Cycles are allowed in the graph but the stream parallel model does not prevent programs containing deadlock from being written.

2.1 Kernels

There are two basic conceptual representations of kernels used in stream programming languages. They are either viewed as special kernel function or as special kernel objects.

Using the function representation, kernels are defined and executed similar to an ordinary procedure call. This is the case in the StreamC/KernelC [41] and Brook family of languages [22][23][44]. An example of a Brook program is shown in Figure 2.1. Using this approach,

```c
kernel void sum(float a<>, float b<>, out float c<>)
{
    c = a + b;
}

int main(int argc, char** argv)
{
    int i, j;
    float a<10, 10>;
    float b<10, 10>;
    float c<10, 10>;
    float input_a[10][10];
    float input_b[10][10];
    float input_c[10][10];

    for(i=0; i<10; i++) {
        for(j=0; j<10; j++) {
            input_a[i][j] = (float) i;
            input_b[i][j] = (float) j;
        }
    }
    streamRead(a, input_a);
    streamRead(b, input_b);
    sum(a, b, c);
    streamWrite(c, input_c);
}
```

Figure 2.1: A data parallel addition written AMD’s Brook+ language

the main distinction between kernel and non-kernel functions is that kernel functions can operate directly on streams (declared using angle brackets in Brook) while non-kernel functions must use an API to manipulate streams. The Brook language also requires that kernels be side-effect free, implying that they are data parallel. This is because this particular language specifically targets data parallel hardware such as GPUs. Also, unlike some of the other stream parallel languages
described in this dissertation, Brook operates on finite size streams. Other programming languages targeting graphics processors, such as CUDA and OpenCL, inherit many characteristics from the Brook language.

A similar kernel model is found in one of the earliest stream processing languages, proposed by Kahn and MacQueen [40], and shown in Figure 2.2. The code in the figure describes and executes a four stage pipeline. Like in Brook, kernels (Processes) in this language operate on streams (channels) and are invoked using an ordinary procedure call syntax. One difference is that this language allows writable state to be present in the kernels. This feature increases generality but makes it harder to detect data parallelism. The TRANSDUCER kernel exhibits data parallelism, but in order to take advantage of that parallelism, the compiler will have to determine that it exists. The example also shows pipeline parallelism between the kernels.

The other common language construct for representing kernels is as a special object. Using this model, the kernel is an object or similar structure containing a kernel work function. The kernel work function is equivalent to a kernel represented as a special function as described above. In addition to the kernel work function, the object might contain additional metadata describing the operation of the kernel, encapsulated program state, and helper methods such as constructors and destructors. One form of metadata sometimes associated with the kernel is the amount of data read from each input stream and written to each output stream each time the kernel work function

```
Process PRODUCER out Q0;
    vars N; 0 -> N;
    repeat INCREMENT N; PUT(N,Q0) forever
Endprocess;
Process TRANSDUCER A in QI out QO;
    repeat PUT(A + GET(QI), Q0) forever
Endprocess;
Process CONSUMER in QI;
    repeat 20 times PRINT(GET(QI)) close
Endprocess;
Process GO;
    doco channels Q1 Q2 Q3;
    PRODUCER(Q1); TRANSDUCER(1,Q1,Q2);
    TRANSDUCER(-1,Q2,Q3); CONSUMER(Q3); closeco
Endprocess;
Start doco GO() closeco;
```

Figure 2.2: A four stage pipeline in the language proposed by Kahn and MacQueen [40].
is executed. This information, which we call the *input rates* and *output rates* of the kernel, gives
the compiler insight into the behavior of the kernel and can be used to find efficient schedules for
executing a stream graph or to enable stream graph transformations. An example of a programming
language featuring object based kernels with rate metadata is the StreamIt programming language
[34]. An example StreamIt program is shown in Figure 2.3. This is essentially the same program

```c
void->int filter IntSource {
    int x;
    init { x = 0; }
    work push 1 {
        push(x);
        x = x + 1;
    }
}

int->int filter Adder(int A) {
    work push 1 pop 1 { push(A+pop()); }
}

int->void filter IntPrinter {
    work pop 1 { println(pop()); }
}

void->void pipeline Main {
    add IntSource;
    add Adder(1);
    add Adder(-1);
    add IntPrinter;
}
```

Figure 2.3: A four stage pipeline written in the StreamIt language.

as the one shown in Figure 2.2, a simple four stage pipeline. The three kernels in this program
are the objects *IntSource*, *Adder*, and *IntPrinter*. In StreamIt, kernel work functions are
named *work* and constructors are named *init*. Kernels can also contain other helper functions.
In the example, the *IntSource* kernel contains encapsulated state *x* and a constructor function.
The *Adder* kernel also has read-only state in the form of the kernel parameter *A*. The kernels in
the figure are also annotated with their input and output rates. These are the statements *push 1*
and *pop 1*.

When a kernel work function is annotated with its rate information or when the compiler can
determine this information through compiler analysis, we call the kernel a *synchronous kernel* with
*static rates*. This terminology comes from the the synchronous dataflow model in which all kernels
must have static rates. Although kernels in the StreamIt language are not technically required to be synchronous, in practice they are because this information is provided as programmer annotations and because the original StreamIt compiler depends on synchronous kernels for scheduling. For more general models such as KPN, static or dynamic rates are possible.

Another object oriented stream parallel programing environment is GNU Radio [4]. GNU Radio is a C++ programming framework and runtime system designed for digital signal processing. It is an example of a framework for expressing stream parallelism layered on top of an existing general purpose language. In it, hand tuned kernels written in C++ are connected together using a Python API. An example of a constructor and work function for a kernel object from the GNU Radio tutorial are shown in Figure 2.4. In GNU Radio, kernels are called Blocks. The block in

```c++
howto_square_ff::howto_square_ff () :
  gr_block ("square_ff",
    gr_make_io_signature (MIN_IN, MAX_IN, sizeof (float)),
    gr_make_io_signature (MIN_OUT, MAX_OUT, sizeof (float)))
{
}

int howto_square_ff::general_work (int noutput_items,
  gr_vector_int &ninput_items,
  gr_vector_const_void_star &input_items,
  gr_vector_void_star &output_items)
{
  const float *in = (const float *) input_items[0];
  float *out = (float *) output_items[0];
  for (int i = 0; i < noutput_items; i++){
    out[i] = in[i] * in[i];
  }

  // Tell runtime system how many input items we consumed on
  // each input stream.
  consume_each (noutput_items);

  // Tell runtime system how many output items we produced.
  return noutput_items;
}
```

Figure 2.4: A kernel object in the GNU Radio stream processing system.

the figure computes the square of its input stream and writes the result to its output stream. In the figure we can see that the block constructor declares the number of input and output streams the block can handle, as well as the size of data items in the input and output streams. The kernel work function is the method named general_work. This function extracts data from its input
buffers, performs the required computation and writes the result to its output buffers. It then tells the runtime how much data was read/written and returns control to the scheduler.

In both the function and object forms of kernels, we end up with a kernel work function describing the computation performed by the kernel. There are two ways that kernel work functions are written in stream parallel languages. For convenience, we call them *implicitly looped* and *explicitly looped*.

The kernels in the Brook (Figure 2.1) and StreamIt (Figure 2.3) examples above are implicitly looped. In systems with implicitly looped kernels, kernel work functions are written to consume the least amount of input necessary to produce output. The functions are assumed to be wrapped in an implicit loop which will repeatedly call the function until all input data is consumed or until the program terminates. Another way to think of this is that implicitly looped kernel work functions implement one iteration of some loop. Within implicitly looped kernel work functions, the programmer does not have to manage blocking or buffering of stream communication or otherwise be aware of stream communication implementation details. Similarly, the programmer does not have to be aware of runtime scheduling mechanisms. Instead, it is assumed that when an implicitly looped kernel work function is called the kernel can execute without blocking to wait for input data in an input stream or buffer space in an output stream.

In contrast, the kernels in the language of Kahn and MacQueen (Figure 2.2) and in the GNU Radio framework (Figure 2.4) are explicitly looped. In systems with explicitly looped kernels, kernel work functions do not have an implicit loop around them. Instead, the programmer is responsible for constructing any loops needed to consume input during the execution of the stream graph. Depending on the system, the programmer may need to be aware of stream communication implementation details and/or interact with the scheduler in order to block or yield for buffer space correctly. In Figure 2.2, the programmer constructs while loops inside of the kernel work functions to consume the appropriate amount of input. The PRODUCER and TRANSDUCER kernels will run as long as the stream graph is active, and the CONSUMER kernel runs for 20 iterations. When CONSUMER finishes, the others will as well. We see more extensive programmer involvement in
the GNU Radio example. Here the kernel work function is passed an input buffer and an output buffer. The programmer must manually extract the inputs and write the outputs in an explicit loop. After completing the available work, the kernel tells the scheduler how much data it has consumed and how much data it has produced. By returning, the kernel effectively blocks until more input is available.

The advantage of implicit looping is that the compiler has more freedom in how it schedules and executes the kernel. In particular, it allows easy decomposition of data parallel kernels and merging of fine grained kernels. In contrast, the explicit looping style often gives the programmer more control over the execution of the kernel. This means the programmer can potentially hand tune the kernel for performance, but the compiler has less freedom to perform code transformations or to alter scheduling mechanisms.

2.2 Streams

The way that streams work is fairly uniform across stream parallel languages, but there are a few variations. Streams are typically unbounded, as in the Kahn/MacQueen, StreamIt, and GNU Radio examples above, although in some systems they are declared with a finite size similar to an ordinary array, as in Brook. They have first-in first-out semantics with some form of push and pop operators to add and remove elements from the stream. In the Brook and GNU Radio examples these are expressed as ordinary memory reads and writes while in Kahn/MacQueen and StreamIt, these are named constructs in the language.

Some languages add a peek operator that can read a data item from a stream without removing it from the stream. This can be useful for kernels such as filters which read a sliding window of stream data for each output produced. Without the peek operator these types of kernels would need to maintain state between iterations. With a peek operator they can be state free and potentially data parallel. In the example languages, peek is explicit in StreamIt and available in GNU Radio as raw stream buffer access.

The push, pop and peek operations are almost always viewed as synchronous and blocking
operators. That is, it is never necessary for the programmer to test a input stream for data or an output stream for buffer space. They only return after completing their operation. Testing for presence or absence of data is not allowed under either the SDF or KPN programming models.

2.3 Stream Graph Construction

It is convenient and correct to think of kernels and streams as nodes and edges in a stream graph. There are several different ways that the structure of stream graphs can be expressed in stream parallel programming languages.

One method is to simply invoke kernels with the syntax of an ordinary procedure call. Examples of this approach are found in the GO process in the Kahn/MacQueen example in Figure 2.2 and in the main function shown in the Brook example in Figure 2.1. Using this approach, the compiler must either run the kernels one at a time, or it must extract the edges of the stream graph by examining the uses of streams by the kernel call statements.

A second approach is to programmatically construct the stream graph. This is the approach taken in the GNU Radio framework. In that system, kernels are compiled into a library of processing blocks. At runtime, a Python program allocates instance of blocks, connects them together into a stream graph, and triggers execution. For example, to connect two signal source kernels to a single audio sink kernel (i.e. a speaker) in GNU Radio, we can use the following Python code which produces a U.S. dialtone:

```python
src0 = gr.sig_source_f(sample_rate, gr.GR_SIN_WAVE, 350, ampl)
src1 = gr.sig_source_f(sample_rate, gr.GR_SIN_WAVE, 440, ampl)
dst = audio.sink(sample_rate, "")
self.connect(src0, (dst, 0))
self.connect(src1, (dst, 1))
...
graph.run()
```

A third approach is to specify the structure of the stream graph directly in a text based language or using a graphical interface. In GNU Radio, for example, it is possible to create stream graphs using a graphical interface called GRC, instead of using Python directly. Figure 2.5 shows
Figure 2.5: An FM receiver application created for the GNU Radio stream processing framework using a graphical user interface.

a graphical example of a signal processing application created using this software. It is easy to imagine how the GRC software can translate this graphical representation into the appropriate sequence of stream graph operations in Python code.

A hierarchical stream graph construction method is built into the StreamIt language. Kernel work functions can perform work on their input streams or they can declare a set of children kernels to do the work for them. These children kernels are connected to one another using several patterns built into the language. They can be connected as pipelines or in various split-join patterns. A split-join is a pattern where a parent kernel’s input stream is split or duplicated among the inputs of its children kernels, then the outputs of those children are recombined into a single stream which serves as the output of the parent kernel. The pipeline pattern can be seen in the Main kernel in Figure 2.3. Another StreamIt example is shown in Figure 2.6. This example is from the StreamIt Cookbook [16]. In the program fragment shown in the figure, the pipeline pattern is used in the BandPassFilter kernel and the split-join pattern is used in the BPFCore kernel. The
float->float pipeline BandPassFilter(float rate, float low, float high, int taps)
{
    add BPFCore(rate, low, high, taps);
    add Subtracter();
}

float->float splitjoin BPFCore (float rate, float low, float high, int taps)
{
    split duplicate;
    add LowPassFilter(rate, low, taps, 0);
    add LowPassFilter(rate, high, taps, 0);
    join roundrobin;
}

float->float filter Subtracter
{
    work pop 2 push 1 {
        push(peek(1) - peek(0));
        pop(); pop();
    }
}

Figure 2.6: Constructing a stream graph for an equalizer in StreamIt.

hierarchically defined structure of the computation is shown graphically alongside the program text.

2.4 Scheduling

The individual kernels in stream programs can be statically scheduled prior to execution or dynamically scheduled at runtime. This is not really a feature of the language as much as a feature of the implementation of the compiler or runtime. Nevertheless, the design of the language can impact the ability to use one or the other form of scheduling. For example, in a language like StreamIt, where the structure of the stream graph is explicit, it is much easier to come up with a static schedule than in other languages where the structure of the stream graph is not obvious. In a system like GNU Radio, the opposite is true. Because the stream graph is constructed at runtime using pre-compiled components, effective static scheduling is difficult.
Chapter 3

The Stream and Kernel Intermediate Representation (SKIR)

3.1 Introduction

The Stream and Kernel Intermediate Representation (SKIR) provides the two basic constructs, streams and kernels, required by stream parallel programs. It also includes operations to create streams, to identify and execute kernels, and to construct and modify stream graphs. It can be used as an extension to low level program representations targeted by front-end compilers – such as JVM, LLVM, or CIL – or as a set low level primitives exposed as compiler intrinsics to domain specific libraries and frameworks. The stream and kernel abstractions are provided by SKIR as a set of low-level instruction-like operations. Applications create and execute streaming computation by sequentially executing procedural SKIR operations at runtime. A complete list of SKIR operations is shown in Table 3.1.

The design goals for SKIR are to address the limitations detailed in Chapter 1. That is,

(1) Support a variety of stream parallel and general purpose source languages.

(2) Support a variety of stream parallel programing models.

(3) Support static and/or dynamic compilation and optimization.

(4) Support dynamic stream graphs.

In addition, SKIR aims to,

(5) Provide an abstraction layer to hide the implementation details of parallel execution and stream communication.

(6) Allow for performance greater than or equal to existing systems.
### Table 3.1: Overview of the Stream and Kernel Intermediate Representation (SKIR) operations.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( k = \text{skir.kernel} \text{ work, arg} )</td>
<td>Create a new runtime kernel object with the work function \text{ work} and kernel state \text{ arg}. Store a handle to the resulting kernel object in ( k ).</td>
</tr>
<tr>
<td>skir.call ( k, \text{ins}, \text{outs} )</td>
<td>Execute kernel ( k ) with the input streams \text{ins} and the output streams \text{outs}. \text{ins} and \text{outs} are arrays of stream objects.</td>
</tr>
<tr>
<td>skir.uncall ( k )</td>
<td>Stop execution of ( k ) and remove it from the stream graph.</td>
</tr>
<tr>
<td>skir.wait ( k )</td>
<td>Block until kernel ( k ) finishes execution.</td>
</tr>
<tr>
<td>skir.become ( k )</td>
<td>Replace the currently executing kernel with ( k ). Must be called from within a kernel work function.</td>
</tr>
<tr>
<td>( s = \text{skir.stream} \text{ size} )</td>
<td>Create a new runtime stream object and store a handle to the resulting object in ( s ). \text{size} is the size in bytes of the elements in the stream.</td>
</tr>
<tr>
<td>skir.push ( \text{idx}, \text{data} )</td>
<td>Push \text{data} onto output stream ( \text{idx} ).</td>
</tr>
<tr>
<td>skir.pop ( \text{idx}, \text{data} )</td>
<td>Pop an element from input stream ( \text{idx} ) and store the result into \text{data}.</td>
</tr>
<tr>
<td>skir.peek ( \text{idx}, \text{data}, \text{off} )</td>
<td>Read the stream element from input stream ( \text{idx} ) at offset \text{off} and store the result into \text{data}.</td>
</tr>
</tbody>
</table>

#### 3.2 SKIR Kernels

SKIR kernels are kernel objects managed by the SKIR compiler and runtime which encapsulate a single kernel work function and any associated kernel state. They are created with the \text{skir.kernel} operation,

\[
\text{kernel} = \text{skir.kernel}(\text{work, arg})
\]

This operation takes two parameters, \text{work} and \text{arg}. The \text{work} parameter is a pointer to a kernel work function. This is the function to be called when a kernel is scheduled for execution. The required form of this function is discussed below. The \text{arg} parameter is a pointer that will be passed to the kernel work function when it is called by the SKIR runtime. This parameter is useful for passing kernel state or arguments to the work function. The stream parallel programming model used by SKIR assumes that program state, including state accessed through the \text{arg} parameter, is
for the most part not shared between kernels. This is discussed in more detail in Section 3.4. However, a SKIR implementation cannot assume that a kernel is side-effect free. That is, a SKIR implementation must determine using static analysis or other methods that a kernel is data parallel. As a practical matter, we note that if the arg parameter is a typed pointer (e.g. not void* in C), then an implementation can likely perform better analysis with respect to data parallel execution and execution on non-CPU devices. However, the use of a typed pointer is not a requirement of SKIR.

The result of the skir.kernel operation is an opaque reference to an implementation defined SKIR kernel object. These objects are used as an arguments to the skir.call, skir.wait, skir.become, and skir.uncall operations. The behavior of any other operation performed on kernel objects is undefined. A kernel object can be in one of four states: allocated, active, finished, or deallocated. These states are used by SKIR implementations and are not made visible to the program. The skir.kernel operation returns a kernel in the allocated state. A SKIR implementation may start analyzing the kernel work function associated with the kernel object any time after the skir.kernel operation but the implementation cannot start executing the kernel.

Kernels are spawned for execution using the skir.call operation,

```plaintext
stream ins[] = {..., null}
stream outs[] = (..., null)
skir.call(kernel, ins, outs)
```

The parameters to skir.call are the kernel to execute followed by null-terminated arrays of input and output streams. The input and output streams in the arrays must be stream objects returned by skir.stream operations. Stream objects can be present in at most one input stream array and at most one output stream array. The effect of a skir.call operation is to connect the given input and output streams to the given kernel object then transition an allocated or finished kernel object to the active state. Any stream already connected to the kernel different from the streams given to skir.call is disconnected. A SKIR implementation may execute a kernel if and only if the kernel object is in the active state. The behavior of skir.call for active,
 deallocated, or invalid kernel objects is undefined. The SKIR implementation is not required to check that the number of input and output streams passed to a skir.call operation or the size of data the streams contain is compatible with the given kernel’s work function. If it is not compatible, the runtime behavior is undefined.

The skir.wait operation provides synchronization between kernels and the threads that spawned them,

\[
\text{skir.wait}(\text{kernel})
\]

The skir.wait operation takes a kernel object as its single parameter. When executed it will block if \text{kernel} is in the allocated or active states. The skir.wait operation returns immediately when the kernel is in, or transitions to, the finished or deallocated states. The skir.wait operation also returns immediately if \text{kernel} is an invalid kernel object.

To forcibly stop a kernel from executing and/or deallocate a kernel object, a program can execute a skir.uncall operation,

\[
\text{skir.uncall}(\text{kernel})
\]

The skir.uncall operation stops the execution of \text{kernel}, disconnects any input or output streams connected to the kernel object, transitions the kernel object to the deallocated state, then returns once the SKIR implementation can guarantee that the kernel will no longer execute. An implementation must also guarantee that execution stops normally, with the kernel work function retuning a boolean value. That is, it will not preemptively suspend a kernel. Upon returning from skir.uncall, the value of \text{kernel} is undefined with respect to the SKIR program. Internally, a SKIR implementation may free any resources associated with any kernel object in the deallocated state.

Kernel work functions in SKIR are ordinary procedures with a certain form. They are implicitly looped and always have three pointers as their only parameters. The first parameter receives the arg pointer that was given as an argument to the skir.kernel operation. The second and third parameters receive the arrays of input and output streams that were passed as arguments to
the skir.call operation. These arrays are primarily for building hierarchical kernels (described below) and for internal use by a SKIR implementation. We could have made the second and third parameters optional, but we favored a uniform kernel work function prototype instead. The result of directly accessing or manipulating the arrays of input and output streams or the stream objects they contain is undefined for non-hierarchical work functions. Stream data is accessed from within kernel work functions using the skir.push, skir.pop, and skir.peek operations. The behavior of these operations is undefined outside of the work functions of active kernels. Kernel work functions always return a boolean value. A return value of true indicates that the kernel has finished executing and should be transitioned to the finished state by the SKIR implementation. A return value of false indicates that the kernel can process more data and should remain in the active state.

Non-hierarchical kernels can replace their functionality in an active program by executing a skir.become operation,

    skir.become(kernel)

The skir.become operation takes a single parameter, the new kernel to be used in place of the currently executing kernel. The kernel parameter must be the result of a skir.call operation invoked from within the kernel work function executing the skir.become operation. The effect of the skir.become operation is to immediately and atomically replace the kernel work function and the kernel state of the running kernel object with that of kernel. The kernel remains in the active state and the streams connected to the kernel object remain the same. This implies that the new kernel work function must be compatible with the number of connected input and output streams and the size of data they contain. If it is not compatible, the behavior of the kernel object is now undefined. When the kernel object transitions to the finished or deallocated state, the SKIR implementation must reset the kernel object’s kernel work function and kernel state to the values passed as arguments to the original skir.kernel operation for the kernel object. The implementation can then deallocate any resources associated with the internal kernel object which was the argument to skir.become.
The `skir.become` operation is useful in cases where a kernel has to perform some initialization work before the stream graph enters a steady state. For example in stream graphs with feedback loops it might be necessary to initialize the back edge stream in the graph with some data. Another simple example is a kernel implementing a delay in a DSP application. In this case, the kernel might push some number of zeros to an output stream, then replace itself with an identity function. By using the `skir.become` operation, the kernel can avoid internal branches and have fixed output rate(s).

```
allocated
active
finished
deallocated

return true,
return false,
blocked on finished kernel
blocked on active kernel

skir.call
skir.uncall
skir.uncall
skir.uncall
skir.call
skir.call
```

Figure 3.1: SKIR kernel state transitions

The state transitions of kernel objects are summarized in Figure 3.1. Kernel objects are created in the `allocated` state. Once a kernel has been spawned for execution using `skir.call`, it is in the `active` state and the kernel work function will be executed repeatedly by the SKIR runtime until it leaves that state. An `active` kernel transitions to `finished` if it returns `true` or if it becomes blocked on a stream operation where the kernel connected to the other end of the stream is in the `finished` state. For example, if a kernel `A` tries to execute a `skir.pop` instruction on
an empty stream, and the kernel $B$ that was writing to the stream returned `true` the last time it ran, then kernel $A$ is also marked as `finished` as if it had returned `true`. A kernel is transitioned to `deallocated` if a `skir.uncall` operation is called on it. Kernels in the `allocated` or `finished` states are ready to be spawned for execution with `skir.call`.

Using the SKIR operations, the pieces of a stream graph are constructed sequentially but potentially execute in parallel. This implies that there are times when the graph contains `active` kernels but is also being constructed, modified, or deconstructed. During these transition periods, the stream graph may contain `active` kernels whose input and output streams are not yet connected. Whether or not these kernels will run while their neighbors are unconnected is implementation defined. In the implementation of SKIR described in this dissertation, for example, a kernel can always safely execute and write data to an output stream until the data buffer associated with that stream is full. After blocking on the full buffer, the kernel will only run again if a kernel is connected to the other end of the stream and executes long enough to start draining the buffer. Another SKIR implementation might show different behavior because of a different implementation of stream communication, but the output of the program will be the same. The same applies when streams are disconnected as the result of an `skir.uncall` operation. Whether or not the SKIR implementation executes `active` kernels attached to the other end of the uncalled kernel’s former stream objects is implementation dependent. The only thing SKIR guarantees is that the stream objects are left in a state that allows them to be connected to a new kernel with a `skir.call` operation.

Once stream objects have been connected to kernel objects, they can be deallocated by a SKIR implementation if and only if both ends of the stream have been disconnected from kernels transitioning to the `deallocated` state without any `skir.call` operations referencing the stream object between these two events. This means that both kernel objects and stream objects allocated using `skir.kernel` or `skir.stream` will only be deallocated as the result of deliberate program actions and not as the result of simple program execution or modification. For stream objects, this also means that a program must be careful to always leave one end of a stream connected to an
active, allocated or finished kernel if the stream object is to be reused. These properties also imply that a steam object can never be deallocated if one or both of its ends are never connected to kernel objects. However, based on the discussion in the previous paragraph, the execution behavior of such a program may not be well defined.

### 3.2.1 Hierarchical Kernels

There is a special type of kernel object defined by SKIR called a *hierarchical kernel*. Hierarchical kernels are defined as kernels whose work functions do not contain any skir.push, skir.pop, skir.peek, or skir.become operations. Instead, the kernel work function may contain skir.kernel, skir.stream, and skir.call operations necessary to instantiate, connect, and spawn child kernels. These child kernels are allocated and executed using skir.kernel and skir.call operations with the usual semantics. It is legal and desirable for the kernel work function of hierarchical kernels to access the input and output stream arrays of the kernel work function and pass the streams found therein to skir.call operations. Such streams are then considered to be connected to the corresponding child kernel and associated with that kernel’s arrays of input and/or output streams. Hierarchical kernels allow the recursive definition of stream graphs using ordinary skir.kernel and skir.call operations, increasing code modularity. For example, a program could use a kernel from a separate code library without knowing that the kernel is a hierarchical kernel expanding into many smaller kernels implementing the desired functionality. Hierarchical kernels also simplify the implementation of stream parallel programming languages such as StreamIt where programs are composed in this manner.

Unlike ordinary kernel objects which potentially execute their kernel work function many times, the kernel work function of hierarchical kernels must be executed by the SKIR implementation exactly once. A hierarchical kernel remains in the active state until all children allocated and called from within the hierarchical kernel’s work function reach the finished or deallocated state. If all child kernels reach the finished state, then the hierarchical parent kernel is transitioned to finished. If all child kernels reach the deallocated state, then the hierarchical parent kernel is
transitioned to *deallocated*. When a \texttt{skir.uncall} operation is invoked on an *active* or *finished* hierarchical kernel, the SKIR implementation must invoke \texttt{skir.uncall} on each child kernel. When a \texttt{skir.uncall} operation is invoked on an *allocated* hierarchical kernel, no children will be present and the hierarchical kernel is immediately transitioned to the *deallocated* state. References to child kernels are considered part of the state of the kernel. Therefore it can be assumed by a SKIR implementation that references to these kernel objects are not accessible outside of the hierarchical kernel’s work function. This implies that the only way to execute \texttt{skir.uncall} on a child kernel is to execute \texttt{skir.uncall} on the hierarchical parent kernel. This also implies that children of a hierarchical kernel cannot be in a mixture of *finished* and *deallocated* states except during the execution of a \texttt{skir.uncall} operation on the hierarchical parent kernel. All of these properties ensure correct operation of \texttt{skir.wait} for hierarchical kernels.

### 3.3 SKIR Streams

SKIR streams use the usual stream abstraction of an unbounded first-in first-out queue of data to communicate between a single source kernel and a single destination kernel. Data written to a stream by the source kernel can be later read by the destination kernel in the order it was written. Streams are allocated using the \texttt{skir.stream} operation which takes as a single parameter the desired size, in bytes, of data elements stored in the stream. The result is an opaque reference to a stream object that can be used as an argument to the \texttt{skir.call} operation.

The operations defined on stream objects are the three stream operations: \texttt{skir.push}, \texttt{skir.pop}, and \texttt{skir.peek},

\begin{verbatim}
skir.push(idx, ptr)
skir.pop(idx, ptr)
skir.peek(idx, ptr, offset)
\end{verbatim}

The parameters to \texttt{skir.push}, \texttt{skir.pop}, and \texttt{skir.peek} are an index indicating the stream to operate on along with a pointer to the source data (for \texttt{skir.push}) or destination location (for \texttt{skir.pop} and \texttt{skir.peek}). Streams are indexed starting with zero for both input and output
streams. For example, a \texttt{skir.push} operation with an index argument of one indicates that data should be written to the second output stream that was passed to the \texttt{skir.call} operation. As one would expect, \texttt{skir.push} writes data to an output stream and \texttt{skir.pop} reads (and removes) data from an input stream. The \texttt{skir.peek} operation reads data from an input stream but does not remove it and takes an additional unsigned offset parameter. When a non-zero offset is specified, \texttt{skir.peek} reads ahead in the stream by the given amount.

Streams are allocated using the \texttt{skir.stream} operation which takes as its sole parameter the desired size, in bytes, of data elements stored in the stream,

\begin{verbatim}
stream = skir.stream(nbytes)
\end{verbatim}

The resulting stream can be used as an argument to the \texttt{skir.call} operation by including it in one of the null terminated arrays passed to that operation. SKIR kernels are the only code in the program that can operate on streams, and they must do so using the stream operations.

### 3.4 SKIR Memory Model

The stream parallel model makes an important assumption about memory accesses which is necessary for parallel execution. As stated in section 3.2, the stream parallel programming model assumes that program state is not shared between kernels. The result of this assumption is that execution of kernels can be freely interleaved by a compiler or scheduler without introducing data races or other synchronization errors. This is also an important property of both Kahn Process Networks and Synchronous Dataflow Networks.

It is helpful to examine this assumption with respect to SKIR. There are two types of program state that are purposely passed to SKIR kernels. These are the private \textit{kernel state} passed to a program kernel as the \texttt{arg} parameter of a \texttt{skir.call} operation and \textit{stream data} passed between kernels over streams.

Kernel state is defined to be any memory location read or written by the kernel work function whose address was obtained through arithmetic involving \texttt{arg}. A SKIR implementation can
assume that memory operations on kernel state do not conflict with memory operations performed by any other kernel work function. Two memory operations conflict if they access the same memory location and at least one of them is a store operation. Furthermore, it is assumed that memory operations on kernel state do not conflict with other parts of the program (outside of kernel work functions) while the kernel is active. A SKIR implementation must only ensure that memory operations on kernel state are sequentially consistent between invocations of the kernel work function. For example, if a SKIR implementation runs a kernel \( K \) on processor \( A \), then on processor \( B \), then on processor \( C \), then any modification to \( K \)’s kernel state by \( A \) and \( B \) must appear to \( C \) as if \( K \) ran only on \( C \).

Stream data is defined as any data read directly from an input stream as well as any memory location read or written by a kernel work function whose addresses was obtained through arithmetic involving memory addresses read directly from an input stream. For the latter, a SKIR implementation must provide sequential consistency between such memory operations and \texttt{skir.push}, \texttt{skir.peek}, or \texttt{skir.pop} operations. In practice this means that for kernels operating on streams of pointers, a memory fence might be inserted for every \texttt{skir.push}, \texttt{skir.peek}, or \texttt{skir.pop} operation unless the compiler can prove it is not needed based on behaviors such as the target hardware’s consistency model, the control flow of the kernel work function, or the behavior of the underlying kernel scheduler (e.g. single threaded or multi-threaded). A SKIR implementation can assume that a kernel obtains exclusive ownership of stream data when it is read from an input stream. Ownership is released by writing the stream data (or the pointer used to reference the stream data) to an output stream or by returning control to the scheduler. While a kernel has ownership of stream data, a SKIR implementation can assume that memory operations on stream data do not conflict with memory operations in other parts of the program. The behavior of a program that violates these assumptions is undefined.

For all other types of memory operations not mentioned above, such as loads and stores to global variables or procedure calls with side effects, a SKIR compiler is not required to provide any consistency or synchronization guarantees beyond those provided to these operations by the
underlying implementation IR. In the presence of such operations, the SKIR implementation may still assume that the execution of kernels can be freely interleaved without introducing data races or other synchronization errors.

3.5 **SKIR Program Construction**

The previous sections described the individual SKIR operations and execution model in detail. This section shows two examples of how these operations can be used to build and execute stream parallel computation. The first example is a static stream graph that could be analyzed and compiled statically using a SDF compiler. The second example is a stream graph which uses hierarchical kernels and reconfigures itself using the `skir.become` operation.

3.5.1 **Example 1: Static Stream Graph**

An example of how the SKIR operations can be combined to create and execute a stream parallel computation is shown in Figure 3.2 using SKIR pseudo-code. The C like pseudo-code syntax used in the figure contains SKIR operations similar to the SKIR compiler intrinsics that have been implemented for the C language. The program in the figure has the same structure as the program by Kahn and McQueen in Figure 2.2. The `main` procedure creates four kernels, allocates three streams to connect them, then calls each kernel to begin execution. A `skir.wait` operation on the last kernel prevents `main` from returning until the computation has finished.

3.5.2 **Example 2: Dynamic Stream Graph**

Section 1.2.1 stated that two desirable features for stream parallel computation are to allow dynamically constructed stream graphs and to allow dynamically modifiable stream graphs. The first feature, dynamically constructed graphs, is trivially provided by the SKIR operations `skir.stream`, `skir.kernel`, and `skir.call`; graphs are constructed dynamically by default. The second feature, dynamically modifiable graphs, is provided by the two SKIR operations
```c
bool SKIR_PRODUCER (int *state, void *ins[], void *outs[]) {
    *state = *state + 1
    skir.push(0, state)
}

bool SKIR_TRANSDUCER (int *state, void *ins[], void *outs[]) {
    int data
    skir.pop(0, &data)
    data += *state
    skir.push(0, &data)
    return false
}

bool SKIR_CONSUMER (int *state, void *ins[], void *outs[]) {
    int data
    if (*state == 0) return true
    skir.pop(0, &data)
    print(data)
    *state = *state - 1
    return false
}

main()
{
    int counter = 0
    int limit = 20
    int one = 1
    int neg = -1

    stream Q1[2], Q2[2], Q3[2]
    kernel K1, K2, K3, K4

    Q1[0] = skir.stream(sizeof(int)); Q1[1] = 0
    Q2[0] = skir.stream(sizeof(int)); Q2[1] = 0
    Q3[0] = skir.stream(sizeof(int)); Q3[1] = 0

    K1 = skir.kernel(SKIR_PRODUCER, &counter)
    K2 = skir.kernel(SKIR_TRANSDUCER, &one)
    K3 = skir.kernel(SKIR_TRANSDUCER, &neg)
    K4 = skir.kernel(SKIR_CONSUMER, &limit)

    skir.call(K1, NULL, Q1)
    skir.call(K1, Q1, Q2)
    skir.call(K2, Q2, Q3)
    skir.call(K3, Q3, NULL)
    skir.wait(K3)
}
```

Figure 3.2: An example of a static SKIR program graph. The main procedure defines and executes a four stage pipeline similar to the one in Figure 2.2. The program is written in SKIR pseudo-code.
struct printer_state_t {
    int limit; int cnt; int last_prime
}

bool generate (int *cnt, void *ins[], void *outs[]) {
    *cnt = *cnt + 1
    skir.push(0, cnt)
    return false
}

bool filter (int *prime, void *ins[], void *outs[]) {
    int e
    skir.pop(0, &e)
    if (e % *prime)
        skir.push(0, &e)
    return false
}

bool printer (printer_state_t *ps, void *ins[], void *outs[]) {
    skir.pop(0, &ps->last_prime)
    print("prime %u: %u\n", ++ps->cnt, ps->last_prime)
    if (ps->cnt >= ps->limit) return true
    kernel = skir.kernel(make_printer_filter, ps)
    skir.become(k)
    return false // unreachable
}

bool make_printer_filter (printer_state_t *ps, void *ins[], void *outs[]) {
    int *p = malloc(sizeof(int))
    *p = ps->last_prime
    kernel k0 = skir.kernel(filter, p)
    kernel k1 = skir.kernel(printer, ps)
    stream strm[2] = { skir.stream(sizeof(int)), 0 }
    skir.call(k0, ins, strm)
    skir.call(k1, strm, outs)
    return false
}

void run_sieve (int nprime) {
    stream strm[2] = { skir.stream(sizeof(int)), 0 }
    stream nul[1] = {0}
    int gen_state = 1
    kernel src = skir.kernel(generate, &gen_state)
    skir.call(src, nul, strm)

    printer_state_t printer_state
    printer_state.limit = nprime
    printer_state.cnt = 0
    kernel sink = skir.kernel(printer, &printer_state)
    skir.call(sink, strm, nul)
    skir.wait(sink)
    print("done\n")
}

Figure 3.3: An example of a dynamic SKIR program graph. This figure shows the prime sieve program written as SKIR pseudo-code. It is similar to the Go program shown in Figure 1.2.
skir.uncall and skir.become. The use of skir.uncall for this task is fairly obvious. It allows a kernel to be removed from an active stream graph and potentially replaced by one or more new kernels using the other SKIR operations. The skir.become operator allows the same thing to occur from within a non-hierarchical kernel work function.

Figure 3.3 shows the prime sieve program from Figure 1.2 written as SKIR pseudo-code. The program begins by constructing and executing a two stage pipeline consisting of a generate kernel and a printer kernel. Each time the printer kernel receives a new prime from its input stream it allocates a new hierarchical kernel, make.printer_filter, then replaces itself with that kernel using skir.become. The hierarchical kernel is itself a two stage pipeline containing a filter kernel, to filter out multiples of a single prime, and a new printer kernel.

3.6 Implementation

In this dissertation, SKIR is implemented as an extension to the Low Level Virtual Machine (LLVM) code representation version 2.9. The LLVM representation is a “Static Single Assignment (SSA) based representation that provides type safety, low-level operations, flexibility, and the capability of representing ’all’ high-level languages cleanly ... designed to be used in three different forms: as an in-memory compiler IR, as an on-disk bitcode representation ..., and as a human readable assembly language” [7]. The SKIR operations are implemented using the LLVM intrinsic facility. It is generally much easier to extend LLVM using intrinsics rather than adding a new instruction. This is because an intrinsic looks like an ordinary function call to compiler passes that do not have any knowledge of an intrinsic’s semantics while appearing as a LLVM instruction to compiler passes that have been modified to recognize the intrinsic. Intrinsics also support custom code generation in the compiler backend like an ordinary LLVM instruction. Adding an instruction rather than an intrinsic also changes the LLVM bitcode format and therefore breaks compatibility with other LLVM versions. Other features of LLVM implemented as intrinsics include vararg support, basic support for garbage collection, atomic instructions, exception handling, and some standard library operations such as memcopy and trigonometric functions. As in SKIR,
these intrinsics allow for easier code transformation and optimization by the LLVM compiler.
Chapter 4

Compiling High-Level Languages to SKIR

One of the design goals of SKIR is for it to be suitable as a compilation target for multiple languages. This chapter presents four case studies involving the compilation of high-level languages to SKIR. In Section 4.1, C language compiler intrinsics providing direct access to SKIR operations are described. Section 4.2 shows how these intrinsics can be used to build a user library in the C++ language. The library provides a high level object oriented abstraction for stream processing. In Section 4.3, the StreamIt compiler frontend for SKIR is described. Finally, Section 4.4 shows how we can use the SKIR compiler and runtime system to provide an out of process acceleration layer for an embedded domain specific language in JavaScript.

4.1 Compiler Intrinsics: C Language

The SKIR operations can be exposed to programmers of low level languages like C as compiler intrinsics. Using compiler intrinsics in this way is a common method of exposing low level functionality to a programmer. Compiler intrinsics take the form of special functions that are handled differently than other functions in the compiler. They are most often used to provide compiler optimized implementations of certain function (e.g. math libraries), to provide low level synchronization methods (e.g atomic operations), or to expose features of a specific processor (e.g. vector instructions). The SKIR intrinsic declarations for the C language are shown in Figure 4.1. Each of the SKIR C intrinsics corresponds to one of the SKIR operations shown in Table 3.1.
typedef void* skir_stream_ptr_t;
typedef unsigned int skir_stream_idx_t;
typedef void* skir_stream_element_t;
typedef void* skir_kernel_ptr_t;

extern skir_kernel_ptr_t* __SKIR_kernel(void *workfn, void *args);
extern void __SKIR_call(skir_kernel_ptr_t k, skir_stream_ptr_t ins[], skir_stream_ptr_t outs[]);
extern void __SKIR_wait(skir_kernel_ptr_t k);
extern void __SKIRBecome(skir_kernel_ptr_t k);
extern void __SKIRUncall(skir_kernel_ptr_t k);
extern skir_stream_ptr_t __SKIR_stream(unsigned int size);
extern void __SKIR_push(skir_stream_idx_t s, skir_stream_element_t e);
extern void __SKIR_pop(skir_stream_idx_t s, skir_stream_element_t e);
extern void __SKIR_peek(skir_stream_idx_t s, skir_stream_element_t e, unsigned int offset);

Figure 4.1: SKIR C Intrinsics. Each C intrinsic maps to one SKIR intrinsic

4.1 Example

An example program written using the SKIR C intrinsics is shown in Figure 4.2. This program reads an integer from the command line and uses it to initialize the state (cnt0 and cnt1) of two counter kernels. It then allocates 3 integer streams, in0, in1, and out. The streams in0 and in1 are used as the outputs of two counter kernels src0 and src1, and as the two inputs of a single adder kernel, add. The adder kernel computes the sum of these inputs then pushes the result to the out stream. The stream out is the sole input of the kernel sink. This kernel simply prints its inputs.

The work function for the counter kernels src0 and src1, named counter_work, and the work function for the printer kernel sink, named int_printer_work, are not shown. In this example, they are found in the header file simple_kernels.h.

4.1.2 Implementation

Instead of implementing the C intrinsics as an extension to each front-end compiler we might want to use (e.g. llvm-gcc, llvm-g++, and clang), they are implemented as a LLVM compiler pass that can run from the command line. The pass recognizes the SKIR C intrinsics in LLVM bitcode and translates them to the appropriate SKIR operations. Thus to compile a SKIR C
#include "skir_intrinsics.h"
#include "simple_kernels.h"

int add_work(void **, skir_stream_ptr_t ins[], skir_stream_ptr_t outs[])
{
    int a, b, c;
    _SKIR_pop(0, &a);
    _SKIR_pop(1, &b);
    c = a + b;
    _SKIR_push(0, &c);
    return 0;
}

int main(int argc, char *argv[])
{
    if (argc != 2) {
        printf("usage: %s <num samples>\n", argv[0]);
        exit(0);
    }

    int cnt = atoi(argv[1]);
    int cnt0 = cnt;
    int cnt1 = cnt;

    skir_stream_ptr_t in0, in1, out, ins[3], outs[2];
    in0 = _SKIR_stream(sizeof(int));
    in1 = _SKIR_stream(sizeof(int));
    out = _SKIR_stream(sizeof(int));
    ins[0] = 0;
    outs[1] = 0;

    skir_kernel_ptr_t src0 = _SKIR_kernel((void*)counter_work, &cnt0);
    outs[0] = in0;
    _SKIR_call(src0, ins, outs);

    skir_kernel_ptr_t src1 = _SKIR_kernel((void*)counter_work, &cnt1);
    outs[0] = in1;
    _SKIR_call(src1, ins, outs);

    skir_kernel_ptr_t add = _SKIR_kernel((void*)add_work, 0);
    ins[0] = in0;
    ins[1] = in1;
    ins[2] = 0;
    outs[0] = out;
    _SKIR_call(add, ins, outs);

    skir_kernel_ptr_t sink = _SKIR_kernel((void*)int_printer_work, 0);
    ins[0] = out;
    ins[1] = 0;
    outs[0] = 0;
    _SKIR_call(sink, ins, outs);

    _SKIR_wait(sink);
    return 0;
}

Figure 4.2: A simple SKIR program constructed using C compiler intrinsics. This program creates two streams of integers, adds them together, then prints the result.

program to SKIR bitcode we can use the following sequence of commands:
The program `llvm-gcc` is the LLVM version of the GCC C front-end which knows how to emit LLVM bitcode. The program `opt` is the LLVM command line code optimizer. The command line option `-skir-c-to-intr` tells `opt` to run the C intrinsic to SKIR intrinsic transformation pass. This sequence of commands is easily wrapped in a shell script, resulting in SKIR compilers called `skircc` and `skir++` for compiling SKIR C and SKIR C++ programs, respectively.

4.1.3 Discussion

Despite the inconvenience of using compiler intrinsics and the verbosity of the resulting code, the SKIR C intrinsics have proven to be quite useful for producing high performance stream parallel programs. This is largely because of the abstractions provided by the SKIR operations that the SKIR C intrinsics represent. For comparison, imagine the amount of code that would need to be written to implement the program in Figure 4.2 using traditional concurrency mechanisms in C. The programmer would first have to come up with a scheduling scheme to execute the kernels using a single thread or multiple threads. If multithreaded execution was desired, the programmer would then have to choose an implementation technology for threading, implement the communication mechanism, then hook it all together and test it. The stream parallel abstractions in SKIR provide all of these things for us.

4.2 User Library: C++

It is not uncommon to for high level user libraries and frameworks to provide functionality similar to the abstractions provided by SKIR. Such systems often provide runtime code scheduling abstractions as well communication abstractions. Examples are the FastFlow, FDP, DoPE, and GNU Radio systems, all written in C++ [19][49][51][4]. The main difference between SKIR and these systems is that SKIR is a compiler level representation which allows for code transformation and optimization specific to the stream parallel model. The other systems provide only runtime
scheduling and communication abstractions and do not allow for stream parallel code transformation. Nevertheless, if it is not possible to build such frameworks in high level languages on top of SKIR, then its usefulness is reduced. In this section, we show how to build such a framework in the C++ language using the SKIR abstractions. The example framework is a simple object oriented user library built on top of the SKIR C intrinsics.

### 4.2.1 Example

An example program using the C++ user library is shown in Figure 4.3. This program constructs and executes a three stage pipeline. There are three classes defined in the program, `Adder`, `IntSource`, and `IntSink`. Each class represents one type of kernel and contains one kernel work function, a static member function named `work`. In addition, the kernels encapsulate their own state as private data fields. Using the C++ library, the pieces of a stream graph can be constructed procedurally in the typical SKIR fashion:

```cpp
StreamVector s0;
StreamVector s1;
StreamVector nul;

s0.push_back(new Stream<int>);
s1.push_back(new Stream<int>);
IntSource *src = new IntSource(10);
Adder *inc = new Adder(1);
IntPrinter *snk = new IntPrinter;
```

In this code, two new `int` streams are allocated and added to vectors called `StreamVectors`. These vectors are analogous to the null terminated arrays used in the `skir.call` operation. After adding the streams to the vectors, an instance of each of the kernel types is created. The only thing that remains is to execute each of the kernels:

```cpp
(*src)(nul, s0);
(*inc)(s0, s1);
(*snk)(s1, nul);
```
```cpp
#include <iostream>
#include "SKIR.hpp"

using namespace skir;

class Adder : public Kernel<Adder> {
    private:
        int a;
    public:
        Adder(int _a) : a(_a) {}
        static int work(Adder *me, InputStreams ins, OutputStreams outs) {
            Stream<int> in(ins, 0);
            Stream<int> out(outs, 0);
            int i = in.pop();
            out.push(i + me->a);
            return 0;
        }
};

class IntSource : public Kernel<IntSource> {
    int count;
    public:
        IntSource(int i) : count(i) {}
        static int work(IntSource *me, InputStreams ins, OutputStreams outs) {
            Stream<int> out(outs, 0);
            bool ret = (me->count <= 0);
            if (!ret) {
                out.push(me->count);
                me->count--;
            }
            return ret;
        }
};

class IntSink : public Kernel<IntSink> {
    int sum;
    public:
        static int work(IntPrinter *me, InputStreams ins, OutputStreams outs) {
            Stream<int> in(ins, 0);
            sum += in.pop();
            return 0;
        }
};

int main(int argc, char **argv) {
    StreamVector s0;
    StreamVector s1;
    StreamVector nul;
    s0.push_back(new Stream<int>);
    s1.push_back(new Stream<int>);
    IntSource *src = new IntSource(10);
    Adder *inc = new Adder(1);
    IntPrinter *snk = new IntPrinter;
    (*src)(nul, s0);
    (*inc)(s0, s1);
    (*snk)(s1, nul);
    snk->wait();
}
```

Figure 4.3: An example of a SKIR C++ program using the C++ user library from Figure 4.4.
Here the kernel objects are spawned for execution using a functor interface. The StreamVectors describe the edges in the stream graph. The program then blocks the main thread until snk is finished, to avoid returning from `main` before the stream graph has finished:

```cpp
snk->wait();
```

Each of the work functions defined in the three kernel objects have a typical kernel work function structure. Inside of the kernel work function, streams are accessed by first naming each of the input and output streams that will be used:

```cpp
static int work(Adder *me,
       InputStreams ins, OutputStreams outs)
{
  Stream<int> in(ins,0);
  Stream<int> out(outs,0);
  int i = in.pop();
  out.push(i + me->a);
  return 0;
}
```

Here the code assigns the name `in` to the first input stream (i.e. index 0) and the name `out` to the first output stream. It also declares the types of the streams. The `Stream` class provides `push`, `pop`, and `peek` member functions, allowing for the usual SKIR operations on streams:

```cpp
int i = in.pop();
out.push(i + me->a);
return 0;
```

The other two kernel in the program, IntSource and IntSink, are defined in a similar manner.

### 4.2.2 Implementation

The complete implementation of the SKIR C++ library is shown in Figure 4.4. As mentioned above, it implements stream parallel constructs using the SKIR intrinsics. We briefly describe its operation in this section.

User kernels are defined by deriving a class from the Kernel base class. This class provides the functionality of three SKIR kernel operations, `skir.kernel`, `skir.call`, and `skir.wait`. The SKIR `skir.kernel` operation is executed in Kernel’s constructor function. Because the
extern "C" { 
#include "skir_intrinsics.h"
}

typedef skir_stream_element_t StreamElement;
typedef skir_stream_ptr_t StreamPtr;
typedef skir_kernel_ptr_t KernelPtr;
typedef StreamPtr* InputStreams;
typedef StreamPtr* OutputStreams;

class StreamBase {
private:
  StreamPtr stream;
  int offset;
public:
  StreamBase(int size) {
    stream = ..SKIR_stream(size);
  }
  StreamBase(StreamPtr *, int i) : offset(i) {} 
  inline int getOffset() { return offset; }
  inline StreamPtr getStream() { return stream; }
};
typedef std::vector<StreamBase*> StreamVector;
template<class T> class Stream : public StreamBase {
public:
  Stream() : StreamBase(sizeof(T)) {}
  Stream(StreamPtr *p, int i) : StreamBase(p, i) {}
  inline void push(T t) {
    void *p = &t;
    ..SKIR_push(getOffset(), p);
  }
  inline T pop() {
    T t;
    void *p = &t;
    ..SKIR_pop(getOffset(), p);
    return t;
  }
  inline T peek(int o) {
    T t;
    void *p = &t;
    ..SKIR Peek(getOffset(), p, o);
    return t;
  }
};
template<class D> class Kernel {
private:
  KernelPtr kernel;
public:
  // call a kernel
  void operator() (const StreamVector& ins, const StreamVector& outs) {
    unsigned int i;
    StreamPtr* _ins = new StreamPtr[ins.size()+1];
    for (i=0; i<ins.size(); i++) {
      _ins[i] = ins[i]->getStream();
    }
    _ins[i] = 0;
    StreamPtr* _outs = new StreamPtr[outs.size()+1];
    for (i=0; i<outs.size(); i++) {
      _outs[i] = outs[i]->getStream();
    }
    _outs[i] = 0;
    ..SKIR_call(kernel, _ins, _outs);
    delete[] _ins;
    delete[] _outs;
  }
  static int work(void *s, StreamPtr ins[]), StreamPtr outs[]) {
    assert(0 && "error: called Kernel::work!");
    return 0;
  }
  void wait() {
    ..SKIR_wait(kernel);
  }
};

Figure 4.4: An example of a C++ library providing an object oriented interface for stream parallelism using the SKIR C intrinsics.
skir.kernel operation requires the address of the derived class’s work function, the library makes use of static polymorphism. This is the reason that the derived kernel object must provide itself as a template argument to the Kernel base class. This is also the reason why the work function in the derived class is made static. The result of the skir.kernel operation is encapsulated by the Kernel class. This is used in the wait method, which wraps skir.wait, and in the operator() method, which wraps skir.call. The operator() method also receives two StreamVectors as arguments. The method unwraps the SKIR stream objects stored in these vectors, and passes them to the skir.call operation.

StreamVectors are defined as std::vector<StreamBase*>. StreamBase is used to call the skir.stream operation (in its constructor) and to store the stream index information used by Kernel::operator(). Derived from StreamBase is the Stream class that is used directly by user programs. This class extends StreamBase with type information and with the implementation of the pop, peek, and push operations.

The Stream class constructor is overloaded so that the class can be used in two different places by programs. The first place Stream objects get used is during stream graph construction. In this use case, the constructor takes no parameters. The object itself is parameterized using the type of data stored in the stream as an argument to the Stream class template. The sizeof this type is taken to pass as an argument to the skir.stream operation. The other use of Stream objects in user programs is to get a handle on input and output streams inside of the kernel work function. In this case the object is constructed taking two parameters. The first parameter is the StreamVector of input or output streams the desired stream is contained in. The second parameter is the index of the desired stream in the given vector. This ensures that writing,

```cpp
Stream<int> in(ins, 0);
int i = in.pop();
```

using the C++ library compiles into SKIR code equivalent to,

```cpp
int i
skir.pop(0, &i)
```
4.3 Stream Language Front-end: StreamIt

StreamIt has in recent years become the most popular stream processing language for research of stream parallelism [33][36][37][38][54]. It is obvious that a compiler level representation like SKIR should be able to represent StreamIt style programs. StreamIt is supported in the SKIR environment with a StreamIt to SKIR front-end compiler. This compiler takes StreamIt programs as input and produces SKIR bitcode as output.

An example StreamIt program is shown in Figure 4.5. The program in this figure is a simple three stage pipeline. The IntSource filter produces integers starting with zero, the Adder kernel adds three to its input, and the IntPrinter kernel prints its input. The three kernels are connected together into a pipeline in Main. This section describes how this program is compiled to SKIR using the StreamIt to SKIR front-end compiler.

Lexing and parsing of the StreamIt language is accomplished by directly porting the ANTLR parser from the StreamIt project from Java to C++. ANTLR includes support for both languages so this is a straightforward porting process. In the StreamIt compiler this parser generates an intermediate representation known as the FEIR, or Front-End IR, which is simply an object oriented representation of the StreamIt abstract syntax tree. In the original StreamIt compiler this is a Java based representation. The StreamIt to SKIR compiler implements the same FEIR in C++.

After generating the FEIR for a StreamIt program, the main task of the StreamIt to SKIR compiler is to translate the FEIR into LLVM and SKIR bitcode implementing the program. This is done as a direct translation to SKIR without performing optimization. Instead of performing optimizing code transformations in the language front-end, we rely exclusively on the SKIR compiler to perform any optimization and scheduling necessary to efficiently execute the program.

4.3.1 Generating Filters

Computation in StreamIt takes place in filters. Filters in StreamIt contain an optional initialization function, called init, a kernel work function, called work, and optionally contain filter
void->int filter IntSource { int x; init { x = -1; } work push 1 { x = x + 1; push(x); } }

int->int filter Adder(int A) { work pop 1 push 1 { push(A+pop()); } }

int->void filter IntPrinter { work pop 1 { println(pop()); } }

void->void pipeline Main { add IntSource; add Adder(3); add IntPrinter; }

Figure 4.5: A simple three stage pipeline written in StreamIt

state. StreamIt kernel work functions are limited to have a single input stream and a single output stream. All the basic features of StreamIt filters are present in the IntSource filter shown in Figure 4.5. The init function initializes the filter’s state, an integer named x. The work function increments x then pushes the result to its output stream.

The output of the StreamIt to SKIR compiler for IntSource is shown in Figure 4.6. In order to implement a StreamIt filter in SKIR, the compiler creates two functions: an initialization function and a kernel work function. The initialization function allocates the filter’s state and contains the contents of the filter’s init function. In our example, the initialization function allocates four bytes, for the integer variable x, then stores the value -1 into this memory location, as in the original init function. A pointer to the allocated state is returned by the generated init function. This value will be passed to the kernel work function as kernel state. The kernel work
function generated for IntSource is also quite simple. It loads $x$ from the kernel state (which was passed in as argument zero), adds one to it, then pushes the result to its output stream using `skir.push`.

```
define noalias i8* @__streamit_IntSource_init(i8* nocapture) nounwind {
  entry:
    %malloccall = tail call i8* @malloc(i64 4) ; <i8*> [#uses=2]
    %1 = bitcast i8* %malloccall to i32* ; <i32*> [#uses=1]
    store i32 -1, i32* %1
    ret i8* %malloccall
}

define i32 @__streamit_IntSource_work(i8* nocapture,
                                      i8** nocapture,
                                      i8** nocapture) nounwind {
  entry:
    %3 = alloca i32, align 4 ; <i32 *> [#uses=2]
    %4 = bitcast i8* %0 to i32* ; <i32*> [#uses=2]
    %x = load i32* %4 ; <i32> [#uses=1]
    %5 = add i32 %x, 1 ; <i32> [#uses=2]
    store i32 %5, i32* %4
    store i32 %5, i32* %3
    %6 = bitcast i32* %3 to i8* ; <i8*> [#uses=1]
    call void @llvm.skir.push(i32 0, i8* %6)
    ret i32 0
}
```

Figure 4.6: The result of compiling the IntSource filter from Figure 4.5 with the StreamIt to SKIR compiler.

At runtime, the SKIR version of the IntSource filter is constructed using the following sequence of generated instructions:

```
%1 = call i8*, @__streamit_IntSource_init(i8* null)
%2 = i8* bitcast (i32 (i8*, i8***, i8***)* @__streamit_IntSource_work to i8*)
%3 = call i8* @llvm.skir.kernel(i8* %2, i8* %1)
```

StreamIt filters can also take parameters that are visible to the entire filter. An example of this is seen in the Adder filter in Figure 4.5. This filter takes a parameter $A$, then uses $A$ inside of its work function. The output of the StreamIt to SKIR compiler for Adder is shown in Figure 4.7. The SKIR implementation of the initialization function for Adder is very similar to the one
for IntSource. This is because filter arguments are captured and stored as part of the filter’s state. The only difference is that instead of being initialized explicitly in the init function, the value that was passed as an argument to the kernel is used as the initializer. The initialization function generated by the StreamIt to SKIR compiler always takes a single parameter. At runtime, any arguments to the filter are packed into a single structure and passed as a single pointer to the initialization function.

```c
define noalias i8* __streamit_Adder_init(i8* nocapture) nounwind {
  entry:
  %malloccall = tail call i8* @malloc(i64 4) ; <i8*> [#uses=2]
  %1 = bitcast i8* %0 to i32* ; <i32*> [#uses=1]
  %2 = load i32* %1 ; <i32> [#uses=1]
  %3 = bitcast i8* %malloccall to i32* ; <i32*> [#uses=1]
  store i32 %2, i32* %3
  ret i8* %malloccall
}

define i32 __streamit_Adder_work(i8* nocapture,
                               i8** nocapture,
                               i8** nocapture) nounwind {
  entry:
  %3 = alloca i32, align 4 ; <i32> [#uses=2]
  %4 = alloca i32, align 4 ; <i32> [#uses=2]
  %5 = bitcast i8* %0 to i32* ; <i32*> [#uses=1]
  %A = load i32* %5 ; <i32> [#uses=1]
  %6 = bitcast i32* %3 to i8* ; <i8*> [#uses=1]
  call void @llvm.skir.pop(i32 0, i8* %6)
  %7 = load i32* %3 ; <i32> [#uses=1]
  %8 = add i32 %7, %A ; <i32> [#uses=1]
  store i32 %8, i32* %4
  %9 = bitcast i32* %4 to i8* ; <i8*> [#uses=1]
  call void @llvm.skir.push(i32 0, i8* %9)
  ret i32 0
}
```

Figure 4.7: The result of compiling the Adder filter from Figure 4.5 with the StreamIt to SKIR compiler.

The construction of the Adder filter with the argument \(3\), as in the Main pipeline of Figure 4.5, is generated in SKIR as the following sequence of instructions:

```c
%1 = alloca %Adder_init_type, align 8
```
4.3.2 Generating Pipelines and Split-Joins

StreamIt programs are composed using composite pipeline and split-join filters. These filters do not perform work themselves. Instead, they describe the interconnection of some number of child filters to do work for them. An example of a pipeline declaration is the Main filter in Figure 4.5. This pipeline has three children: an IntSource filter, an Adder filter, and an IntSink filter. The output of the IntSource filter is connected to the input of the Adder filter, and the output of the Adder filter is connected to the IntSink filter. In SKIR terminology, Main describes a hierarchical kernel.

The StreamIt to SKIR compiler generates code that instantiates the children of a composite filter in that filter’s initialization function. The generated work function is then responsible for allocating streams to connect the children kernels using skir.stream and executing the children using skir.call. The initialization and kernel work functions for Main are shown in Figure 4.8. The figure shows that the children filters of Main are allocated in its initialization function. These filters will be passed as state to the generated work function when it executes. But the work function does not construct a sub-graph from its children itself. Instead it is another hierarchical kernel which allocates and calls a single child kernel named pipeline. This helper kernel is part of a compiler library written in SKIR C and compiled into a static bitcode library. The compiler can import bitcode functions from this library and emit them into the SKIR code module being generated. From an implementation perspective, it is much easier and far less error prone to implement runtime functionality in this manner than it is to directly generate LLVM/SKIR bitcode correctly. The implementation of pipeline is shown in Figure 4.9.

The __streamit_pipeline_work function receives as its state a list of children filters
Figure 4.8: The result of compiling the Main filter from Figure 4.5 with the StreamIt to SKIR compiler.
Figure 4.9: The pipeline filter from the StreamIt to SKIR compiler runtime library.

to call. For the Main filter in our example, this list was constructed during the initialization function. If there are zero children in the list then the pipeline is empty, and the pipeline work function does nothing. If there is one child in the list, the pipeline work function only has to execute skir.call for this kernel and is done. If there are two or more children, the pipeline work function has to execute skir.call for each child and also connect them together serially using streams allocated with the skir.stream operation.
template< int NOUT >
static int split_dup_work_N_ (void *state,
    skir_stream_ptr_t ins[],
    skir_stream_ptr_t outs[])
{
    split_dup_t *s = (split_dup_t*)state;

    int e;
    __SKIR_pop(0, &e);
    for (int i=0; i<NOUT; i++)
        __SKIR_push(i,&e);
    return 0;
}

#define SPLIT_DUP(DNUM)  
    static int (* split_dup_work_##DNUM)(void*,skir_stream_ptr_t*,skir_stream_ptr_t*) = \  
    &split_dup_work_N_< DNUM >;

SPLIT_DUP(4)
...

Figure 4.10: A generic duplicating splitter from the StreamIt to SKIR compiler runtime library.

Composite split-join filters are constructed in the same way as pipeline filters. That is, the initialization function of the filter is generated to allocate and initialize each child filter. Then, the appropriate function from the compiler bytecode library is used to connect and execute the children filters in the work function. In addition, the compiler must allocate the actual split and join kernels. These are also implemented in the compiler library. There are two type of splitting kernels, split_{rr} kernels, which implement round-robin splitting of an input stream, and split_{dup} kernels, which duplicate the input stream among the children kernels. Figure 4.10 shows a generic duplicating split kernel defined in the compiler library. In practice, it is often possible to emit split and join kernels specialized for a specific number children. This allows for easier kernel analysis (e.g. rate detection) and optimization (e.g. loop unrolling) by the SKIR and LLVM optimization routines in the back-end compiler. The runtime library predefines a number of common versions of these kernels and also includes a fallback version which takes the number of children kernels (i.e. NOUT) as a runtime parameter. It is also possible to generate specialized versions on the fly, but this is unimplemented.
4.3.3 Other Language Features

Several StreamIt language features remain unimplemented in the StreamIt to SKIR at the time of writing this dissertation. This is simply because they do not show up in any of the benchmarks we used. They are feedback loops, pre-work functions, and teleport messaging.

Prework functions are StreamIt filter work functions that execute instead of the actual work function the first time the filter executes. An example of a prework function is shown in Figure 4.11.

```c
float->float filter Delay(int N) {
    prework push N {
        for (int i=0; i<N; i++) {
            push(0.0);
        }
    }

    work push 1 pop 1{
        push(pop());
    }
}
```

Figure 4.11: An example of a StreamIt prework function.

Prework functions could easily be added to SKIR using the `skir.become` operation. With this scheme, the compiler would simply generate the filter as usual, but use the prework function as the kernel work function. Then, instead of generating a return statement at the end of the prework kernel work function, a `skir.become` would be generated instead.

An example of a StreamIt feedbackloop construct is shown in Figure 4.12. A feedback loop contains a `join`, a `split`, and exactly two children. The inputs to the `join` are the input

```c
float->float feedbackloop AddFb(float scaling) {
    join roundrobin;
    body FloatAdder();
    loop FloatScaler(scaling);
    split duplicate;
    enqueue 0.0;
}
```

Figure 4.12: An example of a StreamIt feedback loop.
stream of the entire feedback loop and a back edge stream created by the split. The output of the join goes to a body filter. The output of the body filter is the input of the split. One output of the split is the back edge of the loop. This stream is feed through the loop filter before ending at the join. The other output of the split is the output of the entire feedback loop.

StreamIt feedback loops could be supported in SKIR the same way pipelines and split-joins are supported; a hierarchical kernel work function in the compiler library implements the details of constructing the composite structure and the compiler itself generates the code to build the feedback loop at runtime using that library function. The enqueue operation shown in the figure adds data to the backward edge of the feedback loop before the program starts running. This operation can be thought of as syntactic sugar on top of the prework construct, and could be supported in SKIR in the same way prework functions could be implemented using skir.become.

Teleport messaging is a unique StreamIt feature which allows inter-kernel synchronization outside of the SDF model [53]. Filters can receive messages from other filters by defining special handler methods to be called when a message is delivered. Filters can send messages with a syntax similar to calling these handler methods directly. For example,

\[
\text{GainFilter.changeGain}(1.2) [\min: \max];
\]

causes the value 1.2 to be delivered to the changeGain handler of the filter GainFilter. The notation \([\min: \max]\) indicates the minimum and maximum latency for handler invocation at the receiver. Latency is defined in terms of iterations of the sending filter \(S\). A latency of \(k\) means that the message should be delivered \(k\) iterations of \(S\) after the message was sent. To compute the iteration of the receiving filter \(R\) corresponding to latency \(k\), a function called the stream dependence function, \(SDEP_{R \leftarrow S}(n)\), is used. This function determines, for all valid execution schedules of the stream graph, the minimum number of iterations of \(R\) required for \(S\) to execute \(n\) times. So, if \(S\) sends a message with latency \([k_{\min}: k_{\max}]\) during its \(n\)th iteration, then the message must be delivered during \(R\)'s \(m\)th iteration where \(n + k_{\min} \leq SDEP_{R \leftarrow S}(m) \leq n + k_{\max}\) if \(R\) is a downstream filter and \(SDEP_{R \leftarrow S}(n + k_{\min}) \leq m \leq SDEP_{R \leftarrow S}(n + k_{\max})\) if \(R\) is an upstream filter.
Teleport messaging could be supported in SKIR by counting kernel iterations and constraining the execution schedule appropriately. When a teleport message is sent by $S$, its iteration count could be used to compute a valid iteration of $R$ before which the message handler should execute. It is not difficult to add iteration counting to SKIR kernels. In fact, this is already done for profiling. The challenge is in constraining the scheduler so that $R$ does not run too far ahead of $S$ and vice versa. One way to implement such constraints in a dynamically scheduled stream graph is to size the stream buffers between $S$ and $R$ appropriately. However, to support this we would need to provide an annotation mechanism for language frontends to communicate scheduling constraints to the SKIR scheduler, which we currently do not provide. Such a mechanism would be easy to provide in our LLVM based implementation, but the correct form for such annotations is the subject of future research.

4.4 Embedded DSL: JavaScript

A key benefit of the SKIR approach to stream parallelism is the potential for dynamic compilation and scheduling. It is hard to find programming languages more dynamic than weakly typed dynamic languages like JavaScript. Since its introduction by Netscape in the 1990’s, JavaScript – formally known as ECMAScript – has long been the dominant language for client-side web development. The size and complexity of client-side JavaScript programs has grown considerably since that time, and now includes applications such as games, office suites, and image editing tools traditionally developed using high performance statically compiled languages. More recently, developers have been expanding the use of JavaScript with standards and implementations for server-side JavaScript such as CommonJS [3] and node.js [8]. JavaScript based query processing also powers several popular “NoSQL” style databases such as Riak [2] and CouchDB [1]. This is largely due to the wide use of JavaScript in these communities and the ease with which JavaScript execution engines can be embedded into larger projects. These trends are all driving a need for high performance JavaScript implementations.

The stream parallel model can help with two of JavaScript’s main shortcomings moving
function array_adder(A, B, n) {
    var C = [];
    while (n--) {
    }
    return C;
}

function stream_adder(s0, s1, s2) {
    var a = s0.pop();
    var b = s1.pop();
    var c = a + b;
    s2.push(c);
    return false;
}

Figure 4.13: Array based vs. stream based addition in JavaScript

The JavaScript language itself is difficult to implement in a high performance fashion. This is largely due to the language’s types, or more correctly, the lack of types. The fundamental problem is that the type of a variable can change between executions of the same expression. This means that unless the runtime compiler can prove that something is always of a certain type, it has to insert dynamic checks into the generated code. The use of the stream parallel programing model can make it much easier to perform this kind of type inference. To see how this might work, we can look at the examples in Figure 4.13.

The array_adder function in the figure takes two arrays as parameters, adds them together, then returns the result. The other function, stream_adder, does the same thing using a stream-like API. In this stream parallel version, the computation is defined as a kernel work function. Instead of passing the data into the function as arrays, it is read into the kernel using stream operations. If we assume that streams are typed, then in this version of the computation there is little doubt as to the type of the addition expression or the type of the output data c at the time the
call to `stream_adder` is made. This is because streams are defined to be a homogeneous data structure. In contrast, the implementation of the array based method has to check the type of $A[n]$ and $B[n]$ for each iteration of the loop, since arrays can be heterogeneous.

### 4.4.1 Sluice

We have implemented an API similar to the one shown in Figure 4.13 as the *Sluice* library for the node.js JavaScript environment [27]. Node.js [8] is built on the high performance V8 JavaScript execution engine [15] and is intended to be used as a platform for creating network services, in particular web servers. We choose node.js over other JavaScript implementations as the target for Sluice because it runs on the server and not in a browser (i.e. no DOM and accessible from a command line), because V8 is fairly easy to extend using C++, and because the stream parallel model has already been shown to be a useful programming model for real time web services [13].

Stream program kernels in Sluice are simply JavaScript objects containing a method called `work`. This work function is called to do work on behalf of the kernel when it is scheduled to run. Once a kernel is executing as part of a streaming computation, its work function is called repeatedly by the Sluice scheduling algorithm until the kernel finishes executing. A kernel finishes executing when it becomes blocked on an empty input stream whose source has finished executing or when its work function returns `true`. This is the only requirement of a kernel work function – that it return a value of `true` if it has finished executing or return a value of `false` if it can keep processing input data if more is available. This is essentially the same execution model used by SKIR kernels. Two example Sluice kernels are shown in Figure 4.14.

Sluice uses StreamIt style stream graph construction. The following code shows how we can create and run a three stage pipeline using the two kernels from Figure 4.14 and an additional kernel called `Printer` which simply prints everything in its input stream:

```javascript
var src = new Counter(10);
var add = new Adder(1);
var snk = new Printer();
var p = Sluice.Pipeline(src, add, snk);
p.run();
```
function Counter(cnt) {
    this.i = 0;
    this.cnt = cnt;
    this.work = function() {
        if (this.i < this.cnt) {
            this.push(this.i++);
            return false;
        }
        return true;
    }
    return true;
}

function Adder(arg) {
    this.a = arg;
    this.work = function() {
        var e = this.pop();
        e = e + this.a;
        this.push(e);
        return false;
    }
    return true;
}

Figure 4.14: Two simple Sluice kernels. The kernel on the left implements a simple counter. The kernel on the right adds the same value to each item popped the input stream and pushes the result to its output stream.

The result of running this code is to print the numbers 1 through 10. We can obtain the same result in a more complicated way using a split-join:

```javascript
var add0 = new Adder(1);
var add1 = new Adder(1);
var add2 = new Adder(1);
var sj = Sluice.SplitRR(1, add0, add1, add2).JoinRR(1);
var p = Sluice.Pipeline(new Counter(10), sj, new Printer());
p.run();
```

In this example, the split-join distributes one element at a time, in a round-robin fashion, to each of the three Adder kernels. The results of the Adder kernels are then combined in the same round-robin manner. The entire split-join is then used as a stage in a three stage pipeline.

The `run` method is used to execute the top level pipeline or split-join in a Sluice computation. This is an asynchronous method. That is, it does not wait for the Sluice computation to complete before returning. Instead, it optionally takes as an argument a callback function that will be called when the computation finishes. This style of asynchronous programming is very common in JavaScript.

As long as a Sluice kernel contains a `work` method that returns `true` or `false`, the rest of the kernel and kernel work function can contain arbitrary JavaScript code. However, as we will show in the next section, it is advantageous for us follow a few simple coding conventions when creating Sluice kernels. First, we prefer that program state encapsulated in kernel objects be
stored in the this scope of that kernel object. Second, we prefer that such state be initialized by
the kernel object constructor. These conventions allow the implementation described in the next
section to more easily obtain the program information it needs to perform optimization. A more
mature implementation or one more tightly integrated into the JavaScript engine could relax these
constraints.

4.4.2 Dynamic Recompilation for SKIR

Using Sluice, program kernels are written in an object oriented style, encapsulating program
state within program kernels. Because of this, we can more easily inspect and manipulate the
state associated with a given program kernel at runtime. In addition, communication and syn-
chronization within the streaming computation is limited to push and pop operations on streams.
Together, these characteristics provide a great deal of flexibility in deciding where and when code
should execute. We can take advantage of this flexibility to dynamically recompile Sluice kernels
from JavaScript to SKIR then offload their execution to a SKIR runtime executing in a separate
process. Stream communication and communication of kernel state between SKIR and Sluice is
maintained using shared memory between the two processes. Because we restrict the features of
JavaScript that we can use with Sluice and because we sometimes recompile Sluice code to execute
outside of the JavaScript execution engine, we prefer to call Sluice an embedded domain specific
language.

JavaScript applications are distributed as source code and the source code for a particular
function is available to the program itself. Because of this, we can implement runtime code trans-
lation as part of the Sluice user library. For the purposes of this thesis, kernels to be accelerated
are identified to the library by the programmer. This is equivalent to using program annotations
or pragmas in a statically compiled language. Instances of kernels are translated after they are
allocated (using the new operation), but before they are added to a stream graph using the pipeline
or split-join constructs. The following code example shows how a Sluice kernel can be translated
to SKIR using our current implementation:
In this code, the kernel `k` is translated to SKIR using the Sluice library’s `toSkir` method. The result is passed to the anonymous callback then added to a pipeline which is immediately executed by calling `run`. The `toSkir` method transparently compiles the Sluice kernel to SKIR, sends it off to the SKIR runtime, and sets up the required shared memory buffers.

Once a kernel has been identified for optimization with SKIR, its source code is parsed and an abstract syntax tree (AST) for its work function is constructed. This AST is then used to generate SKIR code corresponding to the original Sluice kernel. The main problem we face in this process is the same one faced by any code generation system for JavaScript - mapping dynamic types to static types. The issue is that the type of a particular variable cannot, in general, be statically determined by a compiler whereas the machine code generated by a JavaScript compiler or the SKIR code generated by the Sluice compiler requires static types.

All high performance JavaScript implementations obtain that performance in part by making assumptions about the types contained in a piece of code. In trace-based compilers, type information is recorded during trace collection. For types that remain static during trace collection, machine code specialized to that type can be emitted [28]. In compilers employing direct translation from JavaScript to machine code, such as the V8 compiler [15], code is emitted for the first type seen, then patched if the type changes. Both approaches are based on the observation that although JavaScript supports dynamic types, in practice types remain fairly stable at runtime. Both approaches must also insert checks into the generated machine code to make sure assumptions about types are not violated.

For Sluice kernels written using the coding conventions discussed in Section 4.4.1, most types will be known before kernel execution. This is the direct result of the use of encapsulated program state and the use of the stream communication abstraction. The use of encapsulated program state means that any program state used by the kernel is already initialized when a kernel
object is passed to Sluice for translation to SKIR. This in turn means that the Sluice compiler can simply access the data to determine its type. The type must be rechecked each time the kernel is executed. However, because the kernel work function is typically called many times during kernel execution, the cost of these type checks is very small when compared to the inline checks generated by a typical JavaScript compiler.

The stream operations `pop` and `push` are trivially translated to SKIR as they are supported directly by the representation. Stream communication is the only source of data from outside of the kernel during its execution. By definition we know that the streams contain only a single type. Thus we know that interaction with stream objects will not result in dynamic type behavior.

A remaining source of dynamic types is from variables created during execution of the kernel work function. For example, different types could be assigned to the same variable depending on the path taken through an if-then-else statement. The Sluice to SKIR compiler does not currently handle this behavior, however this is a limitation of our current implementation, not of our approach. Existing mechanisms such as those used in the V8 compiler could be used.

Once static type information is computed for a Sluice kernel, the system translates the AST form of the kernel work function to SKIR code. Instead of generating low level SKIR code (i.e. LLVM code) directly, Sluice generates C code which includes the SKIR C intrinsics. Because JavaScript is syntactically similar to C and because good C to LLVM compilers already exist, this design decision simplifies our implementation enormously. The C code is then compiled down to SKIR code. A side benefit of this approach is that much of the compilation process automatically runs in parallel with the Sluice program.

4.4.3 Offloading Kernel Execution

Because the SKIR scheduler executes as a separate process from Sluice, our main challenge is in obtaining high performance communication between offloaded Sluice kernels and the rest of the JavaScript program. Two things must be communicated efficiently between the separate parts of the program; these are any program state used by offloaded kernels and any streaming
communication between Sluice kernels and SKIR kernels. An overview of kernel offload is shown in Figure 4.15.

Stream communication between processes is made efficient by mapping the streams onto lock free queues in shared memory. For SKIR, this means that nothing further is needed because we are using the ordinary SKIR mechanism which supports shared memory. For Sluice, this functionality is provided using stream objects implemented as a C++ extension to V8 and made accessible through a JavaScript module. In both cases, the runtime remapping of the stream implementation is made possible because the program is written using the stream abstractions provided by Sluice and SKIR which do not assume any implementation details.

It is more difficult to efficiently communicate non-stream data used by a Sluice kernel from JavaScript to SKIR. Program state lacks the features that make implementing streaming communication fairly easy. It is not made structured or abstract by the programming model itself. Instead,
it’s structure depends on the algorithms used by the programmer and can vary greatly from program to program.

Because it is intractable to figure out what parts of a kernel’s state are read or written during an execution of its work function, all state must be copied from Sluice to SKIR when a kernel is called. As long as the kernel is executing, this data remains in the SKIR runtime. When the kernel finishes, the state is returned to Sluice.

Copying data between processes can incur high overheads compared to the granularity of computation the streaming model exposes. To reduce this overhead as much as possible, we again use shared memory between the processes. The first time the program state for a particular kernel is copied, the Sluice runtime requests a piece of shared memory from the SKIR runtime. The size of this memory is the same as the C language struct formed by combining all the state into a single buffer.

Each time a kernel is called to run in the SKIR layer, the kernel’s state is copied from JavaScript into the shared memory buffer. The kernel executing in the SKIR runtime interprets the data as a C struct, and can directly address the data without an additional copy. When the kernel finishes execution, the shared memory can be translated back into its JavaScript version. Thus the data is copied twice – once as it is translated from JavaScript to the C struct at the start of execution, and once as it is translated back to JavaScript data at the end of execution.

Other than the shared memory for streams and state, all interaction between SKIR and other processes is done through remote procedure calls (RPC) implemented using protocol buffers [11] over sockets. There are RPC interfaces corresponding to \texttt{skir.stream}, \texttt{skir.kernel}, and \texttt{skir.call} used to create and execute stream graphs. There are also commands to pass SKIR bitcode to the SKIR runtime and to create, read, and write shared memory buffers for kernel state.

### 4.4.4 Discussion

We evaluate the performance of the Sluice to SKIR accelerator in Chapter 7. The approach taken by Sluice is similar to several other projects. The RiverTrail project [12] introduces typed
parallel arrays into JavaScript and provides five data parallel methods for operating on them: map, combine, scan, filter, and scatter. By restricting the programming model around this specific data type, RiverTrail is able to perform type inference at runtime and generate OpenCL code for computation using the provided operators. This is analogous to Sluice restricting the programming model around stream parallel kernels and emitting and executing SKIR code. A Python embedded domain specific language similar to Sluice and RiverTrail is the Copperhead project [26]. Copperhead provides acceleration of computation involving nested parallel arrays by translating Python to CUDA code at runtime and running the result on graphics processors. Like Sluice and RiverTrail, it does this by restricting the host language until type inference can be successfully performed.

4.5 Conclusion

This chapter has shown that SKIR can support a variety of stream parallel programming environments. Exposed as low level C language compiler intrinsics, SKIR serves as a compilation target for the Sluice source to source compiler, serves as a implementation language for compiler libraries as part of our StreamIt to SKIR compiler, and provides stream parallel abstractions for our C++ library. A high level compiler can also emit SKIR code directly, as demonstrated by our StreamIt to SKIR compiler.
Chapter 5

SKIR Dynamic Scheduling

For streaming computations containing only synchronous kernels, a static schedule can be computed at compile time before the program executes. A valid static schedule guarantees bounded communication queues between program kernels and will not introduce deadlock. If the execution characteristics of kernels are known, static scheduling for this type of stream graph can also provide static load balancing.

For general purpose use, there are serious limitations with the synchronous dataflow model and associated static scheduling techniques. One problem is that static scheduling algorithms require accurate knowledge of both application and hardware execution characteristics as well as inter-kernel communication costs in order to make good scheduling and load balancing decisions [50]. Many factors found in real systems can cause static schedules to perform poorly. Some examples of these factors are kernels with variable runtimes, interference from other workloads, and hardware with dynamic performance characteristics. The problem is exacerbated when compiled software is widely distributed for use on many different end user machines. Another problem is that the synchronous model is quite restrictive for general purpose programming. One cannot, for example, write a program kernel that conditionally writes to an output stream. To do so would make it impossible to determine the kernel’s output rate.

Even if it were desirable, static scheduling is not possible for all SKIR programs. The main reason for this is that program kernels are not required to be synchronous. This means that a compiler or scheduler cannot, in general, compute bounded buffer sizes for communication in a fixed
schedule. In addition, the runtime construction of the stream graph means that a SKIR compiler cannot in general determine the stream graph structure until the program is running. In fact, entire portions of a program graph may conditionally execute (or not execute) based on program input and nothing prevents short running portions of the program graph from finishing before other portions of the graph are instantiated. Finally, for programs containing dynamic stream graphs, static scheduling may not be practical.

Instead of static scheduling, we use dynamic scheduling to coordinate the execution of SKIR kernels. In a dynamic scheduling algorithm, decisions about when to run a particular kernel are made dynamically at runtime. The goal of the SKIR dynamic scheduling algorithm is to provide the same things for a SKIR program that a static scheduling algorithm provides for a synchronous dataflow graph. In particular, such an algorithm should provide high performance, work with bounded communication queues, provide load balancing, and not cause deadlock.

This chapter describes the approach to dynamic scheduling implemented in the SKIR system. Section 5.2 describes how cooperative multitasking can be used to schedule SKIR kernels within a single thread. Section 5.3 shows how to extend this method using a randomized work stealing scheduler to provide multithreaded parallel execution.

5.1 Stream Implementation

Before discussing the SKIR scheduling algorithms, it is helpful to describe the baseline implementation of streams used by SKIR. We say baseline implementation because the actual stream implementation used changes slightly depending on the SKIR program optimizations employed, as discussed in Chapter 6. There are also implementations of SKIR streams that stray far from the baseline. For example there is an implementation that works over TCP/IP sockets and an implementation that enables communication with discrete GPU devices. This flexibility is a direct benefit of the stream abstraction provided by the SKIR representation. Baseline stream communication is implemented by the SKIR compiler using concurrent lock-free queue operations on a circular buffer. This data structure has been shown to efficiently support high speed communication
between processors in pipeline parallel applications as well as in streaming applications on shared memory multiprocessors [19][30]. The `skir.push`, `skir.pop`, and `skir.peek` operations for this data structure are shown in Figure 5.1. The `available` function referenced by the PEEK operation simply represents the calculation necessary to determine if there is data available at the given `offset` in the stream. During compilation, the low level code corresponding to the figure is emitted for each of the operations. A simplified version of the circular buffer data structure used by SKIR is shown in Figure 5.2. It contains the rates (if they are known), pointers to the source and destination kernels, the head and tail indexes, and the actual data buffer. Details such as alignment restrictions and padding to ensure proper cache line separation are omitted.
5.2 Single Threaded Execution

The challenge in providing efficient implementations of the operations of Figure 5.1 is figuring out how to implement lines 3, 10, and 18 in the figure. Because of the blocking semantics of the skir.push, skir.pop, and skir.peek operations, the easiest way to generate code for SKIR kernels is to place each kernel in its own operating system thread. Using this method, one can perform a simple translation of the stream operations into ones that sleep or busy-wait while input data or buffer space becomes available. The disadvantage of this approach is that for programs where the number of kernels is much higher than the number of processors, the overhead of thread creation, operating system context switching, and frequent signaling or busy-waiting is high.

The only alternative to using a thread per kernel is to map multiple kernels into a single thread. Multiple SKIR kernels are scheduled onto a single thread using a form of cooperative multitasking. Cooperative multitasking is a well known scheme for sharing a resource such as a CPU between multiple tasks or processes. Under this scheme a task uses a resource until it is finished then relinquishes control back to the system so that the resource can be assigned to another task. This stands in contrast with preemptive multitasking, where the underlying system forcefully suspends the execution of a task so that another task may use the resource.
A problem with cooperative multitasking is that a single poorly written task can tie up the resources of a system and starve other tasks. This is not a problem for SKIR programs for several reasons. First of all, SKIR kernel work functions are implicitly looped and typically written to do a single unit of work. That is, they pop some inputs, perform some computation, push some output, then return. Second, the resource sharing is between tasks in the same program working toward a common goal. So as long as a kernel is doing useful work (i.e. the kernel is not buggy or malicious), forward progress is being made by the program and starvation is not a concern. Third, stream communication buffers are implemented as fixed sized queues (despite being conceptually unbounded). This implies that a kernel can only execute until it runs out of input data or until it fills its output stream. Finally, the implementation of cooperative multitasking is controlled by the SKIR compiler, not by the programmer.

Cooperative multitasking is obtained in SKIR by transforming program kernels into coroutines. A coroutine is a special type of subroutine that can transfer control to another coroutine by executing a `yield` operation. When another coroutine yields execution back to the original coroutine, execution picks up where it left off. To the coroutine executing the `yield`, the operation looks just like an ordinary procedure call.

Coroutines naturally support producer-consumer relationships and this pattern is a common example of how coroutines can be used. Figure 5.3 shows an example of two coroutines in a producer-consumer relationship. In the figure, two coroutines communicate using a bounded queue. The program is executed by calling the `sched` routine, which initially calls `produce`. The coroutine `produce` runs until it has filled the queue. It then yields execution back to the `sched` routine, returning the address of the next routine to run (`consume`). Thus the `consume` coroutine will run next, drain the queue, yield control back to `sched`, and the whole process starts over again. This form of coroutines is often used to implement generators. The `yield` statement returns a value just like a `return` statement but does not unwind the call stack. Instead, it suspends execution of the current coroutine then jumps to the calling function. When the coroutine is called again later, execution resumes immediately following the original `yield`. 
produce(queue q) 
{
    while (true) {
        while (q.not_full()) {
            data = ...
            q.push_back(data)
        }
        yield(consume)
    }
}

consume(queue q) 
{
    while (true) {
        while (q.not_empty()) {
            data = queue.pop_front();
            ...
        }
        yield(produce)
    }
}

sched() 
{
    fn = produce
    while (true) fn = fn()
}

Figure 5.3: An example of producer-consumer coroutines

We can extend the ideas of Figure 5.3 to the scheduling of stream parallelism. Figure 5.4 shows the SKIR stream operations from Figure 5.1 using coroutine semantics to implement blocking. If we then inline these operations into the SKIR_TRANSUDER kernel from Figure 3.2, we get something like the kernel CORO_TRANSUDER shown in Figure 5.5. The use of coroutines allows the SKIR scheduler to suspend the execution of a kernel work function without resorting to unwieldy code transformations such as stack ripping [17].

The translation of the kernel from SKIR_TRANSUDER to CORO_TRANSUDER is essentially the code transformation that must be performed by the SKIR compiler during code generation in order to use coroutine scheduling. This transformation is described in more detail in Chapter 6. In Figure 5.5, the skir.pop and skir.push operations have been expanded into two parts. The first part of the expanded skir.pop (skir.push) checks to see if there is data (space)
available in the stream buffer. If there is not, then the kernel yields to the kernel attached the other end of the stream. If there is, then the second part of the expanded operation can proceed. For skir.pop, this means reading data out of the stream buffer. For skir.push, this means writing data to the stream buffer.

In Figure 5.3, the producer and consumer coroutines are scheduled using a very simple scheduler – it simply runs the coroutine yielded by the previous coroutine. In the SKIR form of coroutine this does not work because we have kernel work function coroutines that sometimes return another kernel coroutine (when they are blocked) and sometimes return a boolean (when they are not blocked). Instead, a scheduling routine like the one in Figure 5.6 can be used. Using this scheduling algorithm, a single thread is assigned some number of coroutines (i.e. kernel work functions) to execute. The coroutines are added to a single work queue, named run_q in the fig-
bool SKIR_TRANSDUCER (int *state, void *ins[], void *outs[]) {
  int d
  skir.pop(0, &d)
  data += *state
  skir.push(0, &d)
  return false
}

void * CORO_TRANSDUCER(int *state, void *ins[], void *outs[]) {
  skir_stream_t in = ins[0]
  skir_stream_t out = outs[0]
  // skir.pop
  while (in->head == in->tail)
    yield (in->src)
  int d = in->buf[in->tail] + *state
  // skir.push
  while (NEXT(out->head) == out->tail)
    yield (out->dst)
  out->buf[out->head] = d
  out->head = NEXT(out->head)
  return 0
}

Figure 5.5: An example of transforming a SKIR kernel to use coroutine scheduling.

kernel_coro *next = 0;

// while there are kernels to run
while (run_q.size()) {
  kernel_coro *k;

  // only run kernels assigned to this thread
  if (run_q.contains(next))
    k = run_q.find_and_remove(next);
  else
    k = run_q.pop_front();

  next = 0;
  do {
    next = k();
  } while (next == 0)

  // k returned true, it is done
  if (next & 1) continue;

  // k yielded
  run_q.push_back(k);
}

Figure 5.6: A single threaded scheduling algorithm for coroutine style kernel work functions.
ure, during the execution of the `skir.call` operation. The main outer loop of the scheduling algorithm runs as long as there are kernel coroutines present in the work queue. Each iteration of this loop first checks the value of the `next` pointer against the contents of `run.q`. If `next` is found in the queue, it is removed from the queue and executed. Otherwise, the kernel at the front of the queue is removed and executed. As mentioned above, the return value of a kernel work function coroutine can be a boolean or it can be the address of another coroutine. It is easy to represent boolean values within pointers using the old trick of pointer tagging. A pointer with its low order bit set represents true, while a null pointer represents false. Because pointers are always aligned to at least byte boundaries, the low bit of a valid address is never set. After executing a kernel work function coroutine, the scheduling algorithm checks this low order bit. If the bit is set, then the kernel returned `true`, indicating that it has finished. If the bit is not set, the kernel is either blocked and yielded a pointer to a coroutine, or it returned `false` indicating that it returned normally and is ready to run again. The inner loop of the scheduling algorithm runs the kernel until it becomes blocked or returns `true`. If the inner loop was exited because the kernel work function coroutine yielded, it is re-added to the end of the work queue.

The left side of Figure 5.7 shows the operation of the single threaded coroutine scheduling algorithm running the balanced three stage pipeline shown on the right side of the figure. The large block arrows represent the execution of program kernels and the horizontal arrows represent scheduling events. We can assume the three kernels have been added the work queue in the order K0, K1, K2 and that all input and output rates are one. The portions of the figure that correspond to overhead introduced by the scheduler and the portions of the figure that correspond to work present in the original program are labeled. Clearly, the ratio of work to overhead is not to scale in this diagram.

The scheduling algorithm begins by executing kernel K0. According to the algorithm, K0 will run until there is no buffer space available for it to write data to its output stream. It will then yield the blocking kernel, K1, to the scheduling algorithm. K1 then runs until it has consumed all the data in its input stream or it has filled its output stream. In this example, these conditions
Figure 5.7: An example of single threaded execution of a three stage pipeline. The large downward arrows represent work being done by a kernel, the dotted lines represent work stealing, and the horizontal arrows represent scheduling events. Time flows down.

happen at the same time but K1 will encounter the empty input stream first. Thus K0 executes next, and continues executing until it fills the buffer for its output stream. At this point K0 will yield to K1, which will immediately yield to K2. Kernel K2 runs until it consumes all its input, then yields to K1, which does the same. The state of the program is now the same as it was during a previous iteration of the scheduling loop (i.e. K1 has a full output buffer, but all other stream buffers are empty). For this example, we can observe the steady state execution schedule, K1 → K0 → K2. This is depicted by the dotted line arrow in the figure.
5.3 Multithreaded Execution

Coroutines alone do not provide parallel execution. To obtain parallel execution, we introduce the concept of task stealing into our scheduler. In task stealing systems, the total work to be done by a program is divided up into smaller tasks which can be executed in parallel. The tasks execute the work of the program and may spawn more tasks to further divide up their workload. Some number of worker threads (usually the number of processors) execute the tasks. When a task is spawned, it is placed in a work queue local to the currently executing worker thread. After a worker thread finishes a task, it looks in its work queue for more work. If the local queue is empty, then the worker thread randomly chooses another worker thread and attempts to steal tasks from that thread’s queue.

The work stealing algorithm for parallel execution described in this dissertation is built on top the algorithm found in Intel Thread Building Blocks (TBB) [6]. In this framework, tasks are C++ objects containing a method named `execute`. Each worker thread contains a double ended work queue (`deque`) and uses the following algorithm to obtain tasks:

1. Get the task returned by method `execute` for the previous task. This rule does not apply if `execute` returned NULL.
2. Pop a task from the bottom of its own deque. This rule does not apply if the deque is empty.
3. Steal a task from the top of another randomly chosen deque. If the chosen deque is empty, the thread tries this rule again until it succeeds.

At a high level the SKIR parallel scheduling algorithm is similar to the single threaded algorithm in Figure 5.6. However, instead of explicitly running the kernel coroutines, the multithreaded SKIR scheduler creates TBB tasks called `KERNELTASKS`, and spawns them for execution. Spawning a task causes the task to be pushed to the top of the currently executing worker thread’s deque. It is also possible for the scheduler to recycle a completed task, which causes it to be pushed onto
the worker thread’s deque as if it were a freshly spawned task, but avoids some of the overheads
associated with allocating and spawning a new task.

Each KernelTask encapsulates one SKIR kernel coroutine. The KernelTask::Execute
method is shown in Figure 5.8. This method is called whenever the task stealing scheduler selects
a kernel to run. It runs the kernel’s work function until it finishes or until it yields a pointer to a
blocking kernel. If a blocking kernel is returned and it is not active in the scheduler, then a new
KernelTask is allocated and spawned. Otherwise an attempt is made to steal the task corre-
sponding to the blocking kernel. The steal may fail (e.g. if another thread stole it first), returning
zero. The current task is then recycled, which causes it to be placed back into the thread’s work
queue and makes it available to be stolen by another thread. If a new task was spawned or stolen,
it is returned to the scheduler which will then run that task immediately, bypassing work stealing.
If Execute returns zero, then the worker thread will revert to the basic work stealing algorithm.

Figure 5.8: The KernelTask::Execute method runs the kernel work function and makes
scheduling decisions in conjunction with a work stealing algorithm. The next task to run is re-
turned.

Figure 5.9 shows the operation of the multithreaded scheduling algorithm running a three
Figure 5.9: An example of two processors executing a three stage pipeline. The large downward arrows represent work being done by a kernel, the dotted lines represent work stealing, and the horizontal arrows represent scheduling events. Time flows down.

The execution continues with K1 moving between processors one and two. The figure indicates each time that a kernel is moved from one processor to the other as a steal. The initial steal
of K0 by Processor 2 is potentially the most expensive, because it was done by randomized task stealing. The steal of K1 by Processor 2 is less expensive, because the K0 kernel task identified K1 as the task to steal, bypassing the randomized task stealing algorithm. The subsequent steals of K1 are potentially the least expensive because they are also targeted steals and because it is more likely that the kernel code is already in the processor’s instruction cache.

Because there are three kernels and two processors in the example, one of the processors is always producing data that the other is consuming. This can increase communication costs because the data must be communicated between processors through a shared cache or through system memory. This can be a much higher latency operation than communicating through the L1 cache as would happen in a single threaded system. However, we can see in the figure that this occurs only part of the time. The rest of the time, data is written and read by kernels on the same processor, thus keeping the data in the L1 cache. Current hardware also uses techniques such as hardware prefetching to hide the cost of accessing higher levels of memory.

We note that when Processor 2 steals K1, the scheduling algorithm is effectively choosing to communicate code instead of data, since there is already a full buffer of data in the processor’s cache. This is often a good choice, because the size of the data buffer is usually larger than the kernel code and because the kernel code will usually enjoy a much higher cache hit rate.

5.4 Discussion

Overall this scheduling scheme has several nice properties. It is fairly simple and makes scheduling decisions in a distributed manner only using information local to the kernel that is executing and its immediate neighbors in the stream graph. It also maintains locality by choosing to run neighboring kernels when possible. Because it is a dynamic algorithm built on randomized task stealing, it naturally performs load balancing.

The use of coroutines for scheduling stream parallel programs was first proposed by Kahn and MacQueen [40]. In their single threaded formulation, the execution of kernels is demand driven. As in SKIR, when a kernel attempts to read from an empty stream, that kernel is suspended
and the producer at the other end of the stream is activated. In addition, the empty stream is marked as *hungry*, indicating that the consumer is waiting for data. When data is written to a hungry stream, execution yields back to the consumer kernel. Demand driven schemes ensure that kernels never produce unnecessary results, but can result in an excessive number of context switches to propagate demand.

To enable parallel execution, Kahn and MacQueen suggest that after writing data to a hungry stream and re-activating the consumer kernel, the producer kernel be allowed to continue executing in anticipation of the consumer needing more data later on. This technique is an alternative to demand driven execution known as *data driven* execution. With data driven scheduling, kernels are allowed to execute as soon as they have enough input to run. That is, they are either blocking waiting for input, or they are running. The problem with this approach is that a kernel or an entire sub-graph of the program may produce unnecessary output which may also lead to unbounded accumulation of stream data [48]. Kahn and MacQueen addressed this problem by adding a parameter called an *anticipation coefficient*, $A$, to their streams. When running in a data driven mode, a kernel was not allowed to write more than $A$ unconsumed items to its output stream. The main problem with this approach is the overhead associated with keeping track of the number of unconsumed items and the overhead of context switching to propagate demand when $A$ is too small.

The SKIR scheduling algorithm improves on Kahn and MacQueen in several ways. First, by using work stealing, SKIR provides the potential for parallel execution, but does not require it. Parallel execution only occurs in a work stealing system if a task is actually stolen. This partially alleviates the problem of performing unnecessary work because the work is only performed if there is excess processing capacity in the system. Second, by using a fixed sized circular buffer for the stream communication implementation, SKIR makes the backpressure mechanism provided by anticipation coefficients implicit, rather than explicit. No additional calculation is needed other than the standard enqueue and dequeue operations to guard against excessive data driven execution. Finally, there is no need to maintain a distinction between hungry and non-hungry streams.
Parks presents a theoretical treatment of demand driven, data driven, and hybrid scheduling methods for Kahn process networks in his Ph.D. thesis [48]. He also gives two requirements that should be met by any scheduling mechanism for Kahn process networks:

1. **Complete Execution** – The scheduler should implement a complete execution of the Kahn process network program. In particular, if the program is non-terminating, then it should be executed forever without terminating.

2. **Bounded Execution** – The scheduler should, if possible, execute the Kahn process network program so that only a bounded number of tokens ever accumulate on any of the communication channels.

Parks proves that limiting the capacity of stream buffers is sufficient to ensure both requirements, with the exception that artificial deadlock may occur. To see how artificial deadlock is possible, simply imagine a SKIR kernel with a peek rate of 101 and a fixed stream buffer size of 100. The input stream will never accumulate enough data for the kernel to become unblocked. Luckily it is easy to distinguish artificial deadlock from real deadlock. If artificial deadlock is present, all kernels will be blocked, and at least one kernel will be blocked trying to write to a full stream buffer. If real deadlock is present, all kernels will be blocked on empty buffers. To eliminate artificial deadlock while maintaining bounded execution (i.e. requirement 2 above), it is sufficient to increase the size of the smallest full buffer whenever artificial deadlock is detected [48].

Because the SKIR scheduling mechanism uses a hybrid of data and demand driven execution, and because it uses bounded stream buffers, SKIR meets both of Parks’ requirements for a dynamic scheduler of process networks. In addition, for programs with a bounded number of kernels, the scheduling algorithm uses a bounded amount of memory for task allocation. This follows from the fact that the algorithm in Figure 5.8 never allocates more tasks than there are kernels in the stream graph. Similarly, kernels have a finite number of input and output streams so the system will never allocate an unbounded number of stream buffers.

We have implemented the D4R deadlock detection algorithm [20] in SKIR for completeness
and to ensure that our scheduling algorithm is not incompatible with low overhead detection. The D4R distributed algorithm detects both local and global deadlock, makes a distinction between artificial and true deadlock, and identifies the bottleneck stream buffer. Like our scheduling algorithm it never requires global data and only requires that kernels share information with their immediate neighbors. Because both artificial and true deadlock are exceptional conditions we can allow the detection algorithm to converge slowly. We take advantage of this fact by executing the algorithm on a sampled basis. That is, we do not run the algorithm each time a kernel blocks. This has no impact on the correctness or convergence of the algorithm and allows it to run with negligible overhead.

The benchmarks we run on SKIR do not suffer from artificial deadlock. For such a situation to occur a program must require more data in flight between two kernels than the stream buffers on the path between them can hold. In the implementation of SKIR we study in this thesis, stream buffers are more than large enough for most applications. In addition, an application that requires a large amount of data in flight often packages such data into its own data buffers or structured packets then communicates pointers to this data between kernels. This is done for reasons of efficiency and is how the dedup and swps3 benchmarks used in our evaluation (Chapter 7) were originally written. Because we do not have a use case for artificial deadlock, we have not implemented artificial deadlock mitigation. Since SKIR is implemented as a JIT compiler, such an implementation would simply JIT compile new kernels to use a larger buffer.
Chapter 6

SKIR Compilation

The SKIR compiler is implemented as a just-in-time (JIT) compiler. Runtime compilation is driven by the execution of the program and can be broadly divided into three parts: *kernel analysis*, which determines static characteristics of kernel work functions; *kernel specialization*, which optimizes kernel work functions using the results of kernel analysis; and *code generation*, which lowers the SKIR representation to vanilla LLVM bitcode, runs standard compiler optimization passes, and performs machine code generation. These phases of compilation are closely tied to the dynamic scheduling mechanisms described in Chapter 5. Together, the dynamic compilation and scheduling techniques described in this thesis provide efficient parallel execution of SKIR programs.

This chapter also describes three additional uses of the JIT compiler for high level code transformation: reduction of parallelism using dynamic kernel fusion, parallel execution of data parallel kernels using dynamic kernel fission, and acceleration of kernels on heterogeneous hardware using an OpenCL backend.

6.1 Static vs. Dynamic SKIR Compilation

For some application domains, it may be preferable to compile some or all of a stream parallel program graph using existing static techniques. Examples of this type of application domain are compiling a synchronous dataflow graph to target a FPGA or compiling a DSP algorithm for embedded hardware. In both of these cases, dynamic JIT compilation makes little sense, due to
very high compilation time (e.g. place and route time for an FPGA), or completely static program and platform characteristics (e.g. an embedded DSP design).

Despite the fact that SKIR is implemented as a dynamic compiler and runtime for this thesis, the representation itself is not restricted to a dynamic model. In most situations the compilation of individual kernels could be implemented as an ordinary ahead-of-time compiler (i.e. not JIT). In general, however, it is not possible for a SKIR compiler to extract a static graph structure from a SKIR program. A simple example of why this is not always possible is a stream graph that is constructed differently depending on runtime parameters. StreamIt programs, for example, contain static program graphs but allow values calculated at runtime to determine the number of stages in a pipeline, the number of children kernels in a split-join, or the input and output rates of kernels. Techniques that perform complex static transformations on StreamIt programs simply choose to ignore this subset of the StreamIt language.

To enable static compilation, a front-end compiler or programmer generating SKIR code only needs to make the structure of the stream graph obvious to a static SKIR compiler. Consider for example the SKIR C program shown in Figure 6.1. It should be obvious that this program is easily analyzable and that a compiler would be able to extract the structure of the stream graph by examining the skir.kernel and skir.call operations and the kernels and streams they reference. In fact, a SKIR compiler pass exists to do just that. If we run this pass on SKIR bitcode from the command line, it prints any static graph structures found in the program. For example, we can run it on the code shown in Figure 6.1:

```bash
$ llvm-g++ -emit-llvm -c -o simple_add.bc simple_add.cpp
$ opt -skir-c-to-intr < simple_add.bc > simple_add-skir.bc
$ opt -skir-extract-graph -o=/dev/null < simple_add-skir.bc
digraph {  
  _ZL15simple_0_1_workPvPS_S0_ -> _Z8add_workPvPS_S0_  
  _ZL15simple_0_1_workPvPS_S0_ -> _Z8add_workPvPS_S0_  
  _Z8add_workPvPS_S0_ -> _ZL16int_printer_workPvPS_S0_ }
```

It should also be obvious that code for arbitrarily complex static stream graphs could be written in a style similar to the code in Figure 6.1, thus enabling static extraction of those graph
#include <stdio.h>
#include "skir_intrinsics.h"
#include "simple_kernels.h"

int add_work(void **s, skir_stream_ptr_t ins[], skir_stream_ptr_t outs[])
{
    int a, b, c;
    _SKIR_pop(0, &a);
    _SKIR_pop(1, &b);
    c = a + b;
    _SKIR_push(0, &c);
    return 0;
}

int main(int argc, char *argv[])
{
    if (argc != 2) {
        printf("usage: %s <num samples>\n", argv[0]);
        exit(0);
    }
    int cnt = atoi(argv[1]);
    int cnt0 = cnt;
    int cnt1 = cnt;

    skir_stream_ptr_t in0, in1, out;
    in0 = _SKIR_stream(sizeof(int));
    in1 = _SKIR_stream(sizeof(int));
    out = _SKIR_stream(sizeof(int));

    skir_kernel_ptr_t src0 = _SKIR_kernel((void*)simple_0_1_work, &cnt0);
    skir_stream_ptr_t src0_ins[1] = { 0 };  
    skir_stream_ptr_t src0_outs[2] = { in0, 0 };  
    _SKIR_call(src0, src0_ins, src0_outs);

    skir_kernel_ptr_t src1 = _SKIR_kernel((void*)simple_0_1_work, &cnt1);
    skir_stream_ptr_t src1_ins[1] = { 0 };  
    skir_stream_ptr_t src1_outs[2] = { in1, 0 };  
    _SKIR_call(src1, src1_ins, src1_outs);

    skir_kernel_ptr_t add = _SKIR_kernel((void*)add_work, 0);
    skir_stream_ptr_t add_ins[3] = { in0, in1, 0 };  
    skir_stream_ptr_t add_outs[2] = { out, 0 };  
    _SKIR_call(add, add_ins, add_outs);

    skir_kernel_ptr_t sink = _SKIR_kernel((void*)int_printer_work, new int_printer_t(""));
    skir_kernel_ptr_t sink_ins[2] = { out, 0 };  
    skir_kernel_ptr_t sink_outs[1] = { 0 };  
    _SKIR_call(sink, sink_ins, sink_outs);

    _SKIR_wait(sink);
    return 0;
}

Figure 6.1: An example of a SKIR C program with a statically analyzable stream graph.

structures as well. Furthermore, if each of the kernel work functions found in the static graph
conform to the synchronous dataflow model, then the graph can be statically scheduled and trans-
formed using any number of existing static scheduling and code generation techniques. The point is that the use of SKIR does not preclude the use of existing static techniques. Instead, it extends the reach of those techniques and permits their use where appropriate while allowing more general and dynamic programs. The advantage of a JIT compiler is that we can perform compilation specializing a particular dynamically generated program graph to the hardware that the program is running on. As discussed in Chapter 1, this greatly increases program portability.

6.2 Kernel Analysis

Kernel analysis can begin when a \texttt{skir.kernel} operation is executed by a SKIR program. It is not until this point that the SKIR runtime is given a pointer to a kernel work function. The SKIR JIT compiler maintains an internal mapping from function pointers to LLVM bitcode. When a pointer is obtained by the SKIR runtime by a \texttt{skir.kernel} operation, the bitcode corresponding to the pointer is looked up for analysis and compilation. Because the SKIR JIT operates in a lazy mode of operation, the pointer passed to the \texttt{skir.kernel} operation is actually a stub, and the overhead of re-compilation is avoided.

Kernel analysis has three essential tasks: identify hierarchical kernels; identify data parallel kernels; and attempt to determine the rates at which data is read from a kernel’s input streams and written to a kernel’s output streams.

6.2.1 Detecting Hierarchical Kernels

Determining whether or not a kernel is hierarchical is a straightforward process. We can simply scan all of the instructions in the kernel, looking for SKIR operations. If \texttt{skir.push}, \texttt{skir.pop}, or \texttt{skir.peek} operations are not encountered, then the kernel is marked as hierarchical and the analysis is finished. If stream operations are found, then analysis continues with detection of data parallelism.
6.2.2 Detecting Data Parallelism

To support data parallel execution, a kernel must be free from side-effects. The SKIR compiler can conservatively identify two classes of actions performed by kernel work functions that can cause state modification or other side-effects. The first is calls to functions that cannot be analyzed. This includes system calls, calls to external libraries, or calls to any other function whose code is not available for analysis. If such a function call is found, the kernel is marked as having side-effects. For function calls where the body of the called function is available for analysis, we can simply apply the same analysis as for kernel work functions.

The second class of actions causing state or side-effects are writes to memory locations that are not part of the program stack. That is, writes to memory locations that may be read during subsequent executions of the same kernel instance or by other parts of the program. It is assumed that writes to the program stack are to local variables or other temporary state. If a work function only writes to stack locations, then we can assume that stores executed by the kernel do not cause side-effects. In an implementation based on the LLVM representation, pointers to stack memory locations are always the result of an alloca instruction.

Stores to the stack are identified using a simple and conservative algorithm. For each store instruction in the kernel work function, we check to see if the destination is an alloca instruction. If it is, we can move on to examine the next store instruction. If the destination address is the result of simple pointer arithmetic, then we recursively examine the operands of the pointer arithmetic until we determine whether or not the base pointer in the arithmetic is an alloca instruction. If a pointer is encountered that is defined outside of the kernel work function, then the kernel is marked as having side-effects.

6.2.3 Determining Rates

An important class of stream program kernels are those that perform a fixed number of stream operations each time they are executed. We call these kernels synchronous since they conform to
the synchronous dataflow model. Optimization of this subset of stream parallel programs makes up the bulk of recent research in the area of compiling for stream processing. This is because when the input and output rates of each kernel are constant and known, a static schedule for the program can be computed and the size of buffers between the kernels can be determined – both at compile time. Constant rates also make it easier for a compiler to merge and split kernels based on scheduling or resource constraints.

One of the advantages of SKIR is that stream program graphs can contain a mix of synchronous and non-synchronous kernels; SKIR program kernels are not restricted to have constant input and output rates. Some systems, like StreamIt, choose to make the rate information of kernels explicit using programmer annotation. While SKIR does not currently provide a way to annotate a program with such information, we could easily extended the LLVM based implementation of SKIR to provide such annotations. Similar annotations are used by other language front ends to provide meta-information to backend LLVM passes. Debug information is one example. However, in keeping with a minimalist design we instead choose to extract this information using static analysis. It is our experience that when a high-level language makes input and output rate information available, it is usually not difficult for the front-end compiler to emit SKIR code that is analyzable enough to recover this information. An example of this is our front end compiler for the StreamIt language described in Section 4.3, which generally produces analyzable code.

Kernel analysis to determine work function input and output rates is done by a single compiler pass called SKIRKernelInfoPass. This pass runs once for each kernel work function. For each SKIR stream operation intrinsic in a work function, a simple analysis routine is run to try to determine the input rate (for skir.pop and skir.peek) or output rate (for skir.push) for that single instruction. The routine begins by determining whether or not the stream operation is contained in a loop. If it is not, and the basic block containing the operation post-dominates the entry block of the work function, then we know that the operation executes exactly once. Recall that a basic block $A$ post-dominates another basic block $B$ if all paths from $B$ to an exit block must pass through $A$. Thus, if a block post-dominates the entry block, it is unconditionally executed.
If the stream operation is contained in a loop, then further analysis must be done. If the loop is unconditionally entered and has a computable constant trip count, then we can also compute the number of times the stream operation executes (i.e. the rate). If the loop is contained in another loop, we recursively apply the analysis to the parent loop.

For \texttt{skir.peek} operations, we must also determine the range of the offset parameter. It is not uncommon, for example, to read a windowed region of a stream using a \texttt{skir.peek} inside of a \texttt{for} loop. In this case the offset argument of the \texttt{skir.peek} is a variable. For the rate computation, we want to know the maximum value of that variable. In many cases this is easily obtainable using the Scalar Evolution analysis pass available in LLVM. This analysis pass is used for representing recurrences corresponding to the value of loop oriented expressions, such as loop trip count and induction variables. Once we have a recurrence corresponding to the offset computation, we can perform a search over the space defined by the recurrence, and obtain the maximum. This is possible because we already know the rate of the \texttt{skir.peek} instruction (i.e. the trip count of the loop containing the \texttt{skir.peek}) from the previous analysis and we can evaluate the recurrence at each iteration. If a recurrence which produces constants when evaluated cannot be obtained for a particular \texttt{skir.peek}, then the peek rate for that instruction cannot be determined.

If rates have been determined for each of the SKIR stream operations in a kernel work function, the rates for the kernel as a whole are easily computed. The push rate for a stream is simply the sum of the rate of all the \texttt{skir.push} operations on that stream, and the pop rate of a stream is the sum of the rate of all the \texttt{skir.pop} operations on that stream. The peek rate of a stream is the maximum of the rate of all the \texttt{skir.peek} operations on that stream.

It could be argued that the set of kernel work functions that \texttt{SKIRKernelInfoPass} can compute input and output rates for is fairly small compared to the set of all possible work functions. The counter argument is that most of those aren’t synchronous anyway, and the limitations are not any different for a static synchronous dataflow compiler. The main limitation with the SKIR analysis is that it misses case of dynamic synchronous behavior, where the rate is a runtime variable but either does not change, or changes slowly. Of course a static SDF compiler can’t handle
these cases either, but with a dynamic implementation of SKIR, it should be possible to do better. As an example, if dynamic profiling can determine that a kernel appears to be synchronous, but analysis did not, then maybe it would be beneficial to re-compile the kernel as synchronous, with an abort and re-compile mechanism if the synchronous assumptions are broken. This is possible in a dynamic SKIR environment but would not be feasible with a traditional static ahead of time approach. The reason this type of a scheme has not been explored in this work is that there is not a compelling use case for it among our benchmarks.

6.3 Kernel Specialization

Before machine code generation, SKIR kernel work functions are transformed and optimized. This is necessary to improve performance, to lower SKIR stream operations to LLVM instruction sequences implementing the operation, and to support dynamic scheduling. The results of kernel analysis, the hardware target, and the desired scheduling mechanism all determine how various SKIR compiler passes will be run on the work function. We call this step kernel specialization because the kernel is being specialized with respect to its own characteristics and the characteristics of the target architecture. This section describes the compiler transformations used when compiling SKIR code to run using a single or multiple threads using the scheduling techniques discussed in Chapter 5.

The first step in kernel specialization is to make a copy of the kernel work function. This is done for several reasons. First, it preserves the original procedure to be used by other optimizations, such as dynamic fusion. Second, it modifies the function signature to take an additional parameter, called \texttt{rt\_state}. This parameter can be used by the SKIR runtime to pass additional arguments to the transformed kernel. It is currently used for runtime profiling. Third, the kernel work function must be transformed to return tagged pointers instead of boolean values, in support of the scheduling algorithms discussed in Chapter 5.
6.3.1 Kernel Batching

Many stream parallel programs have kernels that are quite small. For example DSP applications such as the benchmarks distributed with the StreamIt compiler contain kernels that require only a few tens or hundreds of cycles of computation per input item consumed (see Chapter 7, Table 7.1). The reason for this is that the programmers of these benchmarks have tried to expose as much concurrency as possible to the compiler. When kernels are this small the overhead of simply running the kernel via a function call or of incurring a cache miss when reading stream data can be significant. To reduce these overheads, the SKIR compiler performs an optimization called batching on all kernels. Batching simply wraps the body of a kernel work function in a while loop. As a high level example, we can applying this transformation to the SKIR_CONSUMER kernel from Figure 3.2 to produce a new kernel, BATCHED_CONSUMER, shown in Figure 6.2. Performing this

```c
BATCHED_CONSUMER (int *state)
{
  bool r = false
  while(!r) {
    // SKIR_CONSUMER
    int d
    if (*state == 0) {
      r = true; goto ret
    }
    skir.pop(0, &d)
    print(d)
    *state = *state - 1
    r = false
    ret:
  }
  return r
}
```

Figure 6.2: Applying batching to SKIR_CONSUMER produces BATCHED_CONSUMER.

This optimization ensures that the kernel will remain running as long as there is data or free space available in the kernel’s stream buffers. This reduces scheduling and communication overhead and may improve the performance of hardware optimizations such as memory prefetching and loop stream detection.

Figure 6.2 shows the batching transformation applied to pseudo-code. The actual transforma-
tion operates on SKIR bitcode using a code transformation pass called SKIROuterLoopPass, so named because it adds an outer loop to a kernel. The technique used by the outer loop pass is simple, flexible, and powerful. When the pass is instantiated by the code generator, it is given a work function suffix. In the case of the simple batching described in this section, the suffix is “loop”. The suffix is used to lookup a work function template from a bitcode library. Work function templates have names of the form _SKIRRT_workfn_suffix. The C source code corresponding to the template function _SKIRRT_workfn_loop is shown in Figure 6.3.

```c
extern "C"
void * __SKIRRT_workfn_loop(skir_rt_state_t *rt_state, void *kernel_state,
                          skir_stream_t *ins[], skir_stream_t *outs[])
{
    void *k = 0;
    while (k == 0) {
        k = __SKIRRT_workfnExtern(rt_state, kernel_state, ins, outs);
        rt_state->niter++;
    }
    return k;
}
```

Figure 6.3: Kernel work function template implementing batching.

The SKIROuterLoopPass makes a copy of the work function template bitcode, replaces the call to __SKIRRT_workfnExtern with a call to the kernel being compiled, then inlines that call. The result is a batched kernel of the form shown in Figure 6.2. This approach of writing the template function in C, compiling it to bitcode, then using that bitcode as a template for code transformation is pragmatic and powerful. Not only does it reduce bugs trying to, for example, generate proper loops, it also allows for easy experimentation and optional insertion of additional code (e.g. profiling code) using conditional compilation.

### 6.3.2 Generating Kernels as Coroutines

As discussed in Chapter 5, it is necessary to suspend the execution of kernel work functions when a blocking condition is encountered during the execution of skir.push, skir.pop, and skir.peek operations. This is because dependent kernels might share the same operating system
thread and blocking the execution of the thread for a stream operation can easily cause deadlock.
To avoid this situation, kernel work functions are transformed to voluntarily yield control of the
processor to the dynamic scheduler when such a condition occurs. This is done by transforming
kernel work functions into coroutines.

Kernel work functions can be transformed into coroutines by inlining the yielding stream
operations from Figure 5.4 into the work function body. The actual inlining of stream operations
takes place in several steps and will be described in Section 6.3.5. Briefly, functions like the
skir.pop implementation shown in Figure 6.4 are inlined into the kernel work function for each
of the stream operations. Once this has been accomplished, it is easy to find the potential block-
ing points in the kernel work function. They are simply the calls to _SKIRRT_would_block. Like the call to _SKIRRT_workfnExtern in the kernel work function template in Figure 6.3,
_SKIRRT_would_block is never emitted as generated code. It is simply a placeholder used
by the SKIR compiler to locate certain points in the program. For coroutine scheduling, calls
to _SKIRRT_would_block get replaced with calls to the SKIR yield operator skir.yield
and all return statements get replaced with the coroutine return operator skir.return. The
skir.yield and skir.return operations are implemented as SKIR operations (i.e. LLVM
intrinsics), but are only used internally. They are not exposed as part of the SKIR instruction set.

```c
void __SKIRRT_inline_pop (skir_stream_t *p[],
     skir_stream_idx_t idx,
     skir_stream_element_t e)
{
    skir_stream_t *s = p[idx];
    size_t tail = s->tail;
    while (s->head == tail) {
        __SKIRRT_would_block(s->dst, s->src);
    }
    memcpy(e, &s->buf[tail], e->elem_size);
    s->tail = (tail + e->elem_size) % STREAM_BUFFER_SIZE;
}
```

Figure 6.4: An implementation of the skir.pop operation.

During code generation, the SKIR skir.yield and skir.return operators are low-
ered to calls to the assembly routine shown in Figures 6.6 and 6.7. Additionally, the runtime
scheduler does not call coroutine kernel work functions directly. Instead it calls them through the assembly routine `koro_workfn_64` shown in Figure 6.5. It is not important to understand the routines shown in the figures, only to understand what they do. The primary task of the assembly routines is to perform context switching for the coroutines. Recall that coroutines do not unwind the stack when they yield. Instead, they suspend their execution and jump to another routine. This means that each coroutine executes using a separate program stack. During a context switch three things must be saved and restored: the stack pointer, the program counter, and the callee save registers. The three assembly routines perform these tasks.

```c
extern "C" void *koro_workfn_64(void *rt_state, void *state,
    void *impl_ins, void *impl_outs);
asm( \
    "text
    " global koro_workfn_64\n    "type koro_workfn_64 @function\n    "align 16\n    "koro_workfn_64:\n    "movq 0(%rdi), %r10\n    "movq 0(%r10), %r8\n    "movq 8(%r10), %r11\n    "pushq %rbx\n    "pushq %rbp\n    "pushq %r12\n    "pushq %r13\n    "pushq %r14\n    "pushq %r15\n    "movq %rsp, 24(%r10)\n    "movq %r8, %rsp\n    "popq %r15\n    "popq %r14\n    "popq %r13\n    "popq %r12\n    "popq %rbp\n    "popq %rbx\n    "jmp *%r11\n    "ud2\n);
```

Figure 6.5: SKIR coroutine helper function for x86-64.

To the runtime scheduler, `koro_workfn_64` (Figure 6.5) looks like an ordinary work function. The first thing this assembly routine does is load the saved program counter from the kernel’s state. Initially, this is simply the ordinary entry point for the kernel work function being called. The routine then saves the current execution context, loads the rest of the coroutine execution context, and jumps to the saved program counter. The kernel work function then executes normally until it encounters a `skir.yield` or `skir.return`. 
To a yielding coroutine, the `skir.yield` operation looks like an ordinary procedure call. The operation takes two arguments: the yielding kernel and the blocking kernel. These pointers correspond to the runtime kernel objects returned by `skir.kernel` and are stored inside in the SKIR runtime stream object (see Figure 5.2). Because `skir.yield` executes inside of a stream operation, this information is readily available. The implementation of the `skir.yield` operation (Figure 6.6) saves the kernel work function’s execution context context and restores the scheduler context saved by `koro.workfn_64`. It then emulates the return from `koro.workfn_64`. The value returned is the blocking kernel. Thus program control ends up back in the scheduler as if the kernel work function was called directly then returned a blocking kernel using an ordinary return statement.

Similar to the `skir.yield` operation, the implementation of `skir.return` (Figure 6.7) emulates a return from `koro.workfn_64` back into the scheduler. Instead of returning a pointer
to a blocking kernel, however, it returns whatever the kernel work function wants to return. That is, it returns a tagged pointer representing a boolean value. The difference from skir.yield is that the kernel work function actually wants to return. This means that skir.return implements the semantics of a return statement by unwinding the stack. In addition, it resets the context information saved for the coroutine, so that next time koro_workfn_64 calls the work function, it is entered from the top like an ordinary procedure call.

It should be apparent that the coroutine scheme described in this section adds some overhead to the execution of kernel work functions when compared to statically scheduling their execution. There are a small number of additional memory operations and procedure calls involved. More subtly, the coroutine routines don’t follow normal call and return semantics. This can impact hardware branch prediction and therefore overall performance. For kernel work functions that require many thousands of cycles per iteration, this overhead is negligible. For simpler kernels, the overhead may be more significant. In any case, the overhead of coroutine context switching is less than the overhead of some other dynamic methods such as signaling and context switching at the
level of OS threads.

6.3.3 Coroutine Elimination

For synchronous kernels, it is possible to do better than dynamic scheduling using coroutines without relying on static scheduling. In fact, we can eliminate the use of coroutines completely for these kernels, while still using coroutines for other kernels in the program. Recall that the motivation for using coroutines is that the SKIR compiler cannot know, in general, how much data a kernel will read or write each time it is executed. However, for static kernels this information is often available via static compiler analysis. We could obtain this information from other sources as well. For instance, runtime profiling or programmer annotation are realistic options which are not explored in this thesis.

The coroutine elimination optimization pass eliminates the use of coroutines for synchronous kernels by computing, each time the kernel is scheduled, how many times the kernel work function can execute. It does this by computing how much data is in each input stream and how much space is available in each output stream then dividing the results by the kernel’s input and output rates. The baseline work function template used by this transformation is shown in Figure 6.8. In the figure the \_SKIRRT\_compute\_niter\_s function computes the number of iterations. The result of this function is either the number of iterations \texttt{niter}, or a blocking kernel. The coroutine elimination transformation is implemented using the same SKIROuterLoopPass compiler pass used for the batching transformation, just with different runtime parameters to use the “nocheck” template function.

We can observe that the use of \_SKIRRT\_compute\_niter\_s makes the conditional checks performed by the SKIR stream operations (Figure 5.1) redundant. By precomputing the number of iterations, it is guaranteed that there will be data or buffer space available in the input and output streams when stream operations execute. We can think of this as hoisting the conditional code out of the stream operations, out of the template function while loop, and into the \_SKIRRT\_compute\_niter\_s procedure. As a result, when coroutine elimination is used we
void * __SKIRRT_workfn_nocheck(skir_rt_state_t *rt_state, void *kernel_state, skir_stream_t *ins[], skir_stream_t *outs[])
{
    void *v;
    size_t niter = __SKIRRT_compute_niters(&v, ins, 1, outs, 1);
    if (v) return v;
    // start timing
    START_TSC(rt_state->cycles);
    void *k = 0;
    while (niter--)
    {
        rt_state->niter++;
        k = __SKIRRT_workfnExtern(rt_state, kernel_state, ins, outs);
        if (k) break;
    }
    // stop timing
    GET_TSC(rt_state->cycles);
    return k;
}

size_t __SKIRRT_compute_niters(void **v, skir_stream_t *ins[], int nins, skir_stream_t *outs[], int nouts)
{
    int i;
    int n = STREAM_BUFFER_SIZE;
    for (i=0; i<nouts; i++) {
        skir_stream_t *s = outs[i];
        size_t space = (__SKIRRT_push_space(s)) / s->elem_size;
        long long npush = space / s->push_rate;
        if (npush <= 0) { *v = s->dst; return 0; }
        n = min(n,npush);
    }
    for (i=0; i<nins; i++) {
        skir_stream_t *s = ins[i];
        size_t space = (__SKIRRT_pop_space(s)) / s->elem_size;
        long long npop = (space - s->peek_rate) / s->pop_rate;
        if (npop <= 0) { *v = s->src; return 0; }
        n = min(n,npop);
    }
    *v = 0;
    return n;
}

Figure 6.8: A coroutine elimination work function template.

can generate a stream operation implementation that is much simpler. Instead of inlining stream
operations of the form shown in Figure 6.4, the compiler inlines code of the form shown in Figure
6.9.

Besides eliminating the branch and conditional checking associated with the baseline SKIR
stream operations, using stream operations of the form shown in Figure 6.9 has another potential
```c
void __SKIRRT_inline_pop_nocheck (skir_stream_t *p[],
       skir_stream_idx_t idx,
       skir_stream_element_t e)
{
    skir_stream_t *s = p[idx];
    size_t tail = s->tail;
    memcpy(e, &s->buf[tail], s->elem_size);
    s->tail = (tail + s->elem_size) % STREAM_BUFFER_SIZE;
}
```

Figure 6.9: An implementation of skir.pop for use with coroutine elimination.

benefit. Giacamoni, et. al. [30] argue that not reading both the head and tail each time the stream data structure is accessed improves the performance of the operation. This is because while pop operations only write tail and push operation only write head, each operation must read both. Thus the cache line(s) corresponding to head and tail must be frequently communicated between the caches of concurrently executing kernels. In FastForward, this is solved by storing a sentinel value in the queue to indicate an empty condition [30]. Because FastForward communicates only pointers, this is achievable by using a sentinel of NULL. In SKIR, it is impossible to pick such a value because each application communicates using its own internal data types. Using coroutine elimination partially solves this problem without a sentinel value. For each input (output) stream, the head (tail) index is read only once in the __SKIRRT_compute_niters function. The cost of the access is then spread across many iterations of the kernel work function. In addition, in SKIR programs dependent kernels might not always execute concurrently.

In summary, the coroutine elimination pass does two things. It eliminates the overheads associated with coroutines: separate stacks for each kernel work function, context switching overhead, and indirect branching. And, it eliminates some of the overhead associated with the SKIR stream communication mechanism: additional branch instructions and inter-cache communication.

### 6.3.4 Specialization Templates

In the code transformations discussed in the previous sections, a common technique is to use compiled C++ code as a bitcode template for the transformation. For example, this technique is used
with the SKIROutLoopPass pass to implement the kernel blocking and coroutine elimination loop transformations. Within the work function templates, the __SKIRRT_workfnExtern is a parameter replaced by the body of the kernel work function that is being compiled.

The template technique is also used when lowering SKIR stream operations. For example, when generating yielding versions of stream operations, the __SKIRRTWouldBlock procedure call is a stream operation template parameter, replaced during the code transformation with the desired functionality.

This mechanism is taken a step further in the SKIR implementation by using bitcode templates which are themselves parameterized at the source code level using C++ templates. Figure 6.9 showed a skir.pop implementation that can be used by the coroutine elimination pass. We can note two things that are undesirable about the code in that figure. One is that the size of the stream buffer is hard coded to be the maximum size:

\[
s->tail = (tail + s->elem_size) \mod \text{STREAM\_BUFFER\_SIZE};
\]

The other is that the read of data out of the stream buffer requires a lookup of the size of the data contained in the stream:

\[
\text{memcpy}(e, &s->buf[tail], s->elem_size);
\]

In practice, however, it is preferable to have a version of this function that is specialized with respect to the size of the data items in the stream and which allows other stream buffer sizes to be used. The parameterized versions of skir.pop actually used by the coroutine elimination pass are shown in Figure 6.10. It can be seen in this figure that the version of skir.pop shown in Figure 6.9 is generated as a special case. It can be used when there isn’t a specialized version of the operation available. The __SKIRRT_INLINE_POP_NOCHECK macro in the figure is used to statically instantiate different versions of the operation at compile time (i.e. when the compiler library is compiled, not when a user program is compiled).

Using versions of the bitcode templates parametrized in this way results in better code generation by the compiler. For the example above, the use of a constant in the memcpy operation
template< int ELMSZ, int BUFSZ >
void __SKIRRT_inline_pop_nocheck_E_B_(skir_stream_t *p[],
    skir_stream_idx_t idx,
    skir_stream_element_t e)
{
    skir_stream_t *s = p[idx];
    size_t tail = s->tail;
    memcpy(e, &s->buf[tail], ELMSZ);
    s->tail = (tail + ELMSZ) % BUFSZ;
}

template<>
void __SKIRRT_inline_pop_nocheck_E_B_< 0, 0 > (skir_stream_t *p[],
    skir_stream_idx_t idx,
    skir_stream_element_t e)
{
    skir_stream_t *s = p[idx];
    size_t tail = s->tail;
    memcpy(e, &s->buf[tail], s->elem_size);
    s->tail = (tail + s->elem_size) % STREAM_BUFFER_SIZE;
}

void (* __SKIRRT_inline_pop_nocheck)(skir_stream_t **, skir_stream_idx_t, skir_stream_element_t) = \
    &__SKIRRT_inline_pop_nocheck_E_B_< 0, 0 >;

#define __SKIRRT_INLINE_POP_NOCHECK(ELEMENT_SIZE, BUFFER_SIZE) \
    void (* __SKIRRT_inline_pop_nocheck_##ELEMENT_SIZE##_##BUFFER_SIZE)(skir_stream_t **,
        skir_stream_idx_t, \
        skir_stream_element_t) = \
        &__SKIRRT_inline_pop_nocheck_E_B_< ELEMENT_SIZE, BUFFER_SIZE >;

__SKIRRT_INLINE_POP_NOCHECK(4,16384);
__SKIRRT_INLINE_POP_NOCHECK(8,16384);
__SKIRRT_INLINE_POP_NOCHECK(4,32768);
__SKIRRT_INLINE_POP_NOCHECK(8,32768);
...

Figure 6.10: Generating specialized skir.pop templates with meta-programming.

ensures that the compiler generates a move instruction, instead of an expensive call to memcpy. Most of the bitcode templates used by the SKIR compiler are parameterized in this way with respect to stream buffer size and data size. As another example of how this can improve performance, consider the __SKIRRT_compute_niters procedure shown in Figure 6.8. One of the expressions in this function is a division based on the size of the data elements in the stream. When the parameterized version of __SKIRRT_compute_niters is used, the divisor in this expression is a constant power of two instead of a value loaded from memory. As a result, the optimizer can use strength reduction to change the operation into a shift operation which is much cheaper than division.
6.3.5 Summary

This section has described most of the code transformations that run during kernel specialization. These can be summarized by describing the compiler passes implementing this functionality and describing the order in which they run. The SKIR kernel specialization passes run in the order shown in Figure 6.11.

The first pass that runs is the outer loop pass, SKIROuterLoopPass, which was partially described above. This pass takes a kernel work function template representing the desired scheduling methodology (e.g. coroutines or coroutine elimination), and inlines the actual kernel work function into that template. At the same time, SKIROuterLoopPass replaces any skir.push, skir.pop, or skir.peek operations with calls to their desired implementation. As with the kernel work function template, an optional suffix string determines which version of the operation gets used (for example the “nocheck” suffix seen in Figure 6.10). After SKIROuterLoopPass, a set of standard LLVM passes run to clean up any messes left behind after inlining.

The stream specialization pass, called SKIRStreamOptsPass, runs next. This function further specializes stream operations to the size of data in the stream and the size of the stream buffer by replacing function calls of the form,
With function calls similar to the form,

\[
\text{__SKIRRT\_inline\_pop\_suffix} (\ldots)
\]

In this case 4 is the size of data stored in the stream and 32768 is the size of the stream buffer. This transformation can take place for all stream operations because we always know both of these numbers.

After stream specialization, the stream operation inlining pass, SKIRStreamInlining, can run. As its name implies, this transformation pass simply inlines the implementation of the specialized stream operations into the kernel work function. It also inlines \text{skir.become} as a call into the SKIR runtime. After inlining, we are done for synchronous kernels if we are running them under coroutine elimination. That is, all SKIR operations have been completely lowered to stock LLVM operations. For other kernels or scheduling options, we still have to handle the blocking condition of the stream operations by replacing \text{__SKIRRT\_would\_block}.

For coroutine scheduling, we handle the blocking condition with a pass called SKIRKoroPass. This pass is very simple. It replaces \text{__SKIRRT\_would\_block} with \text{skir.yield} and it replaces LLVM \text{return} statements with \text{skir.return}. These two operations are lowered during code generation to calls to the appropriate assembly routines. This is done during code generation instead of in a kernel specialization pass because for architectures other than x86, it may be preferable to emit machine code implementing \text{skir.yield} and \text{skir.return} directly into the kernel work function. This cannot be performed in a LLVM level transformation pass because the functionality of \text{skir.yield} and \text{skir.return} are architecture specific and because the LLVM JIT does not support inline assembly. For x86, this is not an issue because we must use a procedure call to obtain the program counter, allowing the \text{skir.yield} and \text{skir.return} routines to be written in assembly and compiled into the SKIR runtime.
6.4 Code Generation

After kernel specialization there will not be any SKIR operations remaining in the kernel work function except for possibly some skir.yield and skir.return operations. The rest of code generation can be performed using the ordinary LLVM JIT compiler. Before this is done, a set of standard optimization passes is run on the specialized kernel. This cleans up any dead code or other messes left by the transformations of kernel specialization and takes advantage of any optimization opportunities they created. The set of passes is roughly equal to LLVM’s −O2 command line flag. Many JIT compilers use a multi-stage approach to optimization, where the code is initially JIT compiled as quickly as possible, without much optimization. Optimization might be applied later, but only after runtime profiling has identified a section of code as hot. In SKIR, the assumption is made that kernels are always long running and hot. LLVM is not known for its fast JIT times, so the trade-off between latency and throughput with respect to JIT time is a potential area for future research, both in SKIR specifically and in LLVM in general.

Code generation (i.e. from LLVM bitcode to machine code), along with all of the other kernel transformation passes, takes place the first time the kernel is picked by the dynamic scheduler for execution. That is, it is done lazily. This allows the code generation to potentially take place in parallel with the rest of the program. The main limitation to this parallelism is that access to the JIT itself must be serialized, because that is what LLVM currently requires. Up to $n$ kernels could contend for access to the JIT during the beginning phase of execution, where $n$ is the number of worker threads running the program.

6.5 Compiling for GPUs

Stream parallelism is known to be an appropriate programming model for non-CPU hardware such as graphics processors [23][38][54]. This is largely due to the stream communication abstraction, good encapsulation of kernel state, and abundant parallelism. A stream parallel program representation like SKIR coupled with a dynamic compilation environment is ideally suited
for heterogeneous computing. Not only does SKIR enable existing static program transformations and scheduling techniques for heterogeneous platforms, it allows dynamic re-targeting of code to these same types of systems. Furthermore SKIR allows for applications where only select parts of the program are designed around stream parallelism, and with only parts of those stream graphs running on hardware accelerators. This section demonstrates this ability by describing how SKIR can take advantage of a heterogeneous platform consisting of a CPU coupled with a discrete GPU to accelerate data parallel kernels.

Data parallel portions of stream programs, like many data parallel computations, are suitable for acceleration by graphics processors. GPU acceleration is supported by the SKIR compiler using an OpenCL backend. OpenCL defines programming, execution, and memory models tailored for heterogeneous computing. The primary programming model for OpenCL applications is to specify data parallel kernels in a kernel language derived from C. These kernels execute on one or more OpenCL compute devices which are physically or logically separate from the host processor. The execution model defines how many instances of a particular kernel execute on the compute device. Using the OpenCL API, the host program defines a 1, 2, or 3 dimensional iteration space for each kernel. This iteration space is called a \textit{NDRange}. One instance of the kernel runs for each point in the NDRange. Each NDRange is also divided into sub-spaces called \textit{work groups}. The OpenCL memory model provides four levels of memory to an OpenCL kernel: global, constant, local, and private. Global memory is large and accessible from all kernel instances. Constant memory is read only and initialized by the host program. Local memory is smaller and faster than global memory and is shared by all kernel instances within a work group. Private memory is a small region of memory private to each kernel instance in the iteration space.

The SKIR compiler maps data parallel SKIR kernels onto the OpenCL execution model by defining one dimensional NDRanges. Each time an OpenCL version of a SKIR kernel executes, it computes the amount of input data and output space available in its stream buffers. The result and the kernel’s rates determine how many iterations of the kernel can run. This is similar to the calculation performed in kernels transformed by coroutine elimination. To run the kernel work function...
a one dimensional NDRange with size equal to the number of computed iterations is created. This ensures that the OpenCL runtime will execute one instance of the kernel work function for each of the necessary iterations. Each kernel instance uses its point in the NDRange to read and write the appropriate region of the stream buffer. This execution strategy is illustrated in Figure 6.12.

Figure 6.12: An example of the one work function instance per iteration execution strategy used by the OpenCL backend. In this example the kernel has an input rate of 2 and an output rate of 3.

To emit a kernel as OpenCL code, the runtime compiler must be able to statically determine all read-only state used by the kernel. This is because all such state must be packaged up and transferred to the GPU device before the kernel can execute there. In the current implementation, the SKIR compiler is able to obtain this information if the program gives a typed pointer as state information to the `skir.call` operation and it can be determined that a shallow copy is sufficient to transfer the state. A shallow copy is sufficient when the kernel state does not contain any pointers to other memory. These are similar to the requirements placed on Sluice kernels in Section 4.4 only instead of using SKIR as a compiler backend to accelerate JavaScript, we are using OpenCL as a compiler backend to accelerate SKIR.
Figure 6.13: The structure of OpenCL offload within the SKIR runtime.

Because the buffers allocated for stream communication must fit in the processor’s L1 cache (for performance reasons), they can be much smaller than the amount of data needed to profitably execute a computation on a discrete GPU. For small amounts of data, the communication overhead between the GPU and the CPU can dwarf the performance gains. The kernel to be accelerated is therefore replaced by a stub kernel that primarily reads data from the input and output streams into and out of larger device buffers. These buffers are mapped into the OpenCL runtime. When enough data has been buffered, the kernel state is copied to the device and the OpenCL version of the kernel is executed. Output data is pushed to the output stream of the original kernel as space
becomes available. This enables enough data to be buffered to profitably execute on the GPU while
decoupling the execution of the OpenCL kernel from the rest of the program. The decoupling effect
is similar to manual double-buffering and allows the upstream and downstream kernels to execute
without knowing that the accelerated kernel is executing on an OpenCL device. The structure of a
SKIR kernel running using OpenCL acceleration is depicted graphically in Figure 6.13.

6.6 Stream Graph Level Transformations

There are two basic stream graph transformations that are building blocks for stream graph
manipulation. They are kernel fusion and kernel fission. These transformations can be applied stat-
tically at compile time based on the output of a static scheduling algorithm, or they can be applied
dynamically at runtime, based on the results of runtime profiling. In this section we describe how
SKIR supports these transformations in a dynamic environment.

Kernel fusion is simply the merging of two kernels into one kernel. When the number of
program kernels is larger than the number of worker threads or when kernels are excessively fine
gained, it can be beneficial to apply kernel fusion to eliminate unnecessary stream communication
between program kernels and reduce the number of kernels to be scheduled. After kernels are
merged, the stream communication between them can be optimized by replacing stream operations
with reads and writes to registers or to the stack. If the original kernels are part of a stream graph,
then they can be replaced in the graph with the new fused kernel.

Kernel fission is simply the splitting of a data parallel kernel in to many kernels. When a
stream graph is dominated by one or a few coarse grained kernels, it can be beneficial to increase
parallelism by splitting data into multiple kernels. As with any data parallel computation, there
are many ways this can be done. One method, described in Section 6.5, is to use data parallel
hardware. Another method, described below, is to leverage the stream parallel environment and
replace the data parallel kernel with a split-join, provisioning a split kernel to divide the work to
be done among copies of the original kernel, and a join kernel to recombine the results.
6.6.1 Fusion: Reducing Parallelism

We implement fusion as a dynamic runtime optimization. Dynamic fusion is desirable because it is not until runtime that good decisions on the profitability of fusion can be made. If there are 16 processors available we may make very different decisions about when to employ fusion compared to when there is just one processor available. At runtime we have better profile information through the use of cycle counters, we have better information about stream graph structure, and we have better information about available hardware resources.

Our basic fusion algorithm for merging two SKIR kernels is shown in Figures 6.14 and 6.15. The algorithm begins by computing the set of streams that are common to the two kernels, the set of streams that will be inputs to the fused kernel, and the set of streams that will be outputs to the fused kernel. The set of common streams is the set of streams that the fusion algorithm is trying to eliminate.

```
procedure FUSEKERNELS(K₀, K₁)
    S_C = ComputeCommonStreams(K₀, K₁)
    S_IN = ComputeInputStreams(K₀, K₁)
    S_OUT = ComputeOutputStreams(K₀, K₁)
    RenumberStreamOps(K₀, S_IN, S_OUT, S_C)
    RenumberStreamOps(K₁, S_IN, S_OUT, S_C)
    K_new = new KERNEL()
    (niter₀, niter₁) = MatchRates(K₀, K₁, S_C)
    Inline K₀ into K_new with niter₀ iterations
    Inline K₁ into K_new with niter₁ iterations
    for s ∈ S_C do
        Reserve stack space for in s in K_new
        Replace all pop(s) in K_new with stack reads
        Replace all peek(s, ...) in K_new with stack reads
        Replace all push(s, ...) in K_new with stack writes
    end for
end procedure
```

Figure 6.14: SKIR kernel fusion algorithm.
After determining the set of common streams, the input and output rates of the common streams must be matched so that we can allocate small bounded buffers for stream communication internal to the fused kernel. We compute the minimum number of iterations that each of the original kernels must execute to make this happen. This is just the least common multiple of the rates. The rate matching process is a step that is required in a fully static scheduling algorithm and it is sometimes convenient to think of kernel fusion as local static scheduling for two kernels.

After matching rates we perform a step called stream renumbering. Recall from Chapter 3 that stream operations identify the input or output stream to read or write using an integer index. This index must be replaced with one corresponding to the new set of input or output streams for the new fused kernel. The operations referencing common streams are renumbered by adding a large constant value to the stream index. After renumbering, the bodies of the two kernel work functions are inlined into a new function, creating the new fused kernel work function. This is
easily accomplished using kernel work function templates designed for kernel fusion. The algorithm then reserves space on the stack for the common streams and replaces `skir.pop` and `skir.push` operations on the common streams with references to the stack. These operations are easily located in the fused kernel because they are the ones with large stream indexes (due to renumbering).

The fusion algorithm as described only works on the synchronous subset of SKIR kernels. However, these are often the exact kernels we wish to fuse, since they tend to be small simple kernels where the cost of stream communication is large relative to the execution time of the kernel. For the more general case we could employ a weaker form of fusion, where the two kernels are always scheduled together but the inter-kernel stream communication mechanism is left intact. Although this weaker form of fusion may give benefits with respect to data locality, it is not explored in this thesis.

In Chapter 7 we evaluate a simple dynamic optimization using this fusion algorithm. At runtime, we profile kernel execution time using processor time stamp counters. As the program runs, we sample blocked kernels to determine two properties. First, we check to see if the kernel is blocked on an empty input stream; we only fuse across empty streams. Second, we check the profile information to see if the blocked kernel and the blocking kernel have runtimes under some threshold (we used 512 cycles). If both these conditions are true, the optimizer applies fusion to obtain a new kernel and removes the old kernels from the stream graph. The optimizer then invokes `skir.call` on the new kernel. This causes the new kernel to be inserted into the stream graph, JIT compiled, and passed to the scheduler for execution. For this simple dynamic optimization, dynamic fusion remains active as long as there are more than $2 \times \text{(number of worker threads)}$ kernels in the stream graph.

### 6.6.2 Fission: Increasing Parallelism

As with kernel fusion, we can and do perform fission at runtime with SKIR. As mentioned above, the SKIR compiler does this by leveraging the streaming environment and simply replacing
the data parallel kernel with a split-join. The split-join consists of a split kernel that distributes the input of the original kernel to some number of duplicates of the original. The output of the duplicate kernels is recombined using a join kernel and written to the original output stream. The computation is parallelized among the duplicate kernels with the added overhead of the split and join kernels. This is illustrated in Figure 6.16.

![Figure 6.16: SKIR kernel fission.](image)

Because this transformation adds edges to the stream graph instead of destroying them, it has a simpler implementation than kernel fusion. All it has to do is remove the original kernel from the stream graph, create the duplicates, connect everything together using special split and join kernels provided in the compiler bitcode library, then call each of the kernels with `skir.call`. This is not much different than constructing a hierarchical kernel from a user program.

A variety of runtime strategies for choosing the number of threads (i.e. the width of the split-join) to assign to a data parallel kernel are possible depending on the power, throughput, and latency goals for the system [49][51]. All of these strategies result in the assignment of some number of worker threads to each data parallel kernel. This is equivalent to selecting the fission width of our split-join transformation. In this thesis, the focus is on evaluating the use of the split-join as a static or dynamic mechanism for data parallelism and not on developing new allocation strategies. For the benchmarks used in Chapter 7 we try to match the allocation strategy of the
original application as closely as possible. In each case, this means simply using the number of available processors as the fission width. Although this is not the best strategy for all use cases, it is the most commonly implemented.
Chapter 7

Performance Evaluation

One of the stated goals of this work is to demonstrate a stream parallel compilation and scheduling framework more general and more flexible than existing approaches without sacrificing performance. Chapters 3 and 4 presented the SKIR representation for stream parallelism which achieves the goals of generality and flexibility. Chapters 5 and 6 presented scheduling and compilation techniques which we claim can be used to execute SKIR computation efficiently. In this chapter we validate these claims of efficient execution using existing benchmarks taken from several application domains.

7.1 Benchmarks

We use a set of six benchmarks distributed with the StreamIt compiler. They are beamformer, channel vocoder, dct, fft, filterbank, and fm. Most of these are applications from the area of digital signal processing. We choose these benchmarks because they represent algorithms that have been over-decomposed. That is, the programmers of the benchmarks have expressed most or all of the parallelism present in the application without regard to the number of processors in the system that they will run on. For these benchmarks, it important that the compiler or runtime system coarsen granularity and/or schedule parallel execution efficiently.

Although we can compile StreamIt programs directly to SKIR, we have also ported these benchmarks to C++. This was done to separate the concerns of frontend and backend compiler development and to enable better comparison of SKIR performance with the performance of C++
frameworks. We also take the C++ versions of the benchmarks and port them to the GNU Radio dynamic scheduling framework. The most significant changes required of the kernels are to add explicit looping to the kernel work functions and to replace SKIR style stream operations with direct memory accesses. We also have to implement SWIG bindings for each of the C++ kernels so that they are accessible from the GNU Radio Python programming environment. Finally, we have to implement stream graph construction for each of the benchmarks as a GNU Radio Python program.

We found during this porting process that implementing a stream parallel program in GNU Radio when the kernels (processing blocks) do not already exist in the GNU Radio libraries is more work implementing the same program in low level SKIR using compiler intrinsics, even for a programmer experienced in both systems. This productivity gap is due to the two language design of GNU Radio, the requirement for explicit looping in kernels, and the fact that the programmer must manually interface with the stream communication implementation. Again, this illustrates the advantage of the abstractions provided by SKIR.

We note that SKIR and GNU Radio are at a slight disadvantage to StreamIt with respect to split-join patterns found in StreamIt benchmarks. In StreamIt, these are language constructs and are not instantiated as individual kernels as they are in SKIR C code, in the StreamIt to SKIR compiler, or in GNU Radio. This allows the StreamIt compiler to better reason about the memory access patterns implied by split-joins and to better optimize across them.

All StreamIt programs have a single source kernel and a single sink kernel. We measure the performance of the StreamIt benchmarks by measuring throughput at the sink kernel. In all cases, this measurement is taken by counting the number samples consumed by the sink filter during an approximately one second interval and reporting the rate. The results shown in this section are the average of 30 such measurements. This implies that we are only measuring the steady state performance of each of the three systems. In particular, we are not measuring the JIT compilation costs of SKIR or costs of using Python in GNU Radio. We are only measuring the performance of the compiled code, the schedulers, and stream communication.
We also evaluate our system with a set of coarse-grained pipeline parallel benchmarks. These are *dedup*, *black-scholes*, *swps3*, and *nbody*. *dedup* and *black-scholes* are from the PARSEC 2.1 benchmark suite [21]. *swps3* is an implementation of the Smith-Waterman sequence alignment algorithm originally ported to the stream parallel model by the FastFlow project [18]. *nbody* is a n-body simulation taken from the NVIDIA CUDA SDK. In these benchmarks the challenge is not too much parallelism, it is not enough; the system must efficiently execute copies of the data parallel kernels in parallel. Each of these benchmarks are ported to SKIR with little effort.

We evaluate the performance of the Sluice approach to parallel JavaScript execution using a set of compute intense JavaScript benchmarks that can be written using the stream parallel model. Four of the benchmarks are taken from the Pixastic library of image processing routines [10]. We also include a JavaScript version of the *nbody* benchmark. This evaluation is presented in Section 7.3.

*dedup* is organized as a five stage pipeline with 3 data parallel stages. The original version uses Pthreads and assigns a number of threads equal to the number of processors to each parallel stage. We easily port *dedup* to SKIR by replacing the programmer managed queues between each of the pipeline stages with streams. We rely on the SKIR JIT to perform runtime fission on the data parallel stages using the same allocation strategy as original code (i.e. fission width equal to the number of processors). We note that the resulting application is a much simpler than the original with many lines of code removed. We run the benchmark using the *native* input set.

Black-Scholes is essentially a single data parallel kernel parallelized using a TBB parallel_for construct. We port this benchmark to SKIR by introducing an input kernel and a split kernel. The input kernel emits the iteration space of the original parallel for loop to its output stream and the split kernel distributes this work to copies of the original data parallel kernel. That is, we perform manual fission. We run the benchmark using the *native* input set.

For the Smith-Waterman benchmark, we use the version of *swps3* already ported to the stream parallel model by the authors of the FastFlow framework. It has already been shown that the FastFlow version outperforms the original Pthreads version of the program [18]. It is straight-
forward to port FastFlow programs to the SKIR C++ library since they already use the stream parallel model in a C++ framework. As in the original FastFlow version, we use manual fission. We run the benchmark using a query sequence of length 4000 against the UniProtKB/Swiss-Prot protein database.

We port the *nbody* benchmark to SKIR as a three stage pipeline using the SKIR C++ library with either dynamic fission or the OpenCL backend. The second and third pipeline stages correspond to the two main kernels in the original benchmark. Most of the work is in the second stage which is parallelized by the SKIR JIT compiler using runtime fission. In this stage the program computes the force on a single particle in the simulation due to the gravitational pull of all the other particles in the simulation. The first pipeline stage drives the simulation by emitting a list of particles to its output stream. The *nbody* benchmark is slightly different from the other benchmarks we use. The *nbody* stream graph runs inside of a larger OpenGL program which displays the state of the simulation at each step. Instead of executing as a single long running stream graph, the stream graph is constructed once then run to completion for each iteration of the *nbody* simulation (i.e. for each frame of OpenGL output). The C++ class implementing the main kernel of *nbody* is shown in Figure 7.1. We run *nbody* with a simulation size of 10240 particles.

Table 7.1 summarizes the number of kernels and the distribution of kernel size for each of the benchmarks. The sizes are given in cycles per kernel iteration and were recorded during a single threaded execution of the benchmarks optimized using coroutine elimination. It is easy to see that the StreamIt benchmarks contain a large number of very small kernels. Any overhead introduced into these kernels will be large compared to the amount of work performed. The table also shows that the non-StreamIt benchmarks are dominated by single data parallel kernels.

### 7.2 Results

We evaluate the effectiveness of the SKIR compiler optimizations relative to our baseline scheduling algorithm using the StreamIt benchmarks. The StreamIt benchmarks are used because of their sensitivity to scheduling and communication overhead. Figure 7.2 shows an overview of
class CalculateForces : public Kernel<CalculateForces>
{

public:

    float m_pos_rd[4*NBODES];
    float m_softeningSquared;

    void interaction(float *accel, int pos0, int pos1)
    {
        float r0, r1, r2;
        r0 = m_pos_rd[pos0+0] - m_pos_rd[pos1+0];
        r1 = m_pos_rd[pos0+1] - m_pos_rd[pos1+1];
        r2 = m_pos_rd[pos0+2] - m_pos_rd[pos1+2];

        float distSqr = r0 * r0 + r1 * r1 + r2 * r2;
        distSqr += m_softeningSquared;

        float invDist = 1.0f / sqrtf(distSqr);
        float invDistCube = invDist * invDist * invDist;

        float s = m_pos_rd[pos1+3] * invDistCube;
        accel[0] += r0 * s;
        accel[1] += r1 * s;
        accel[2] += r2 * s;
    }

    CalculateForces(float &softeningSquared)
    :
        m_softeningSquared(softeningSquared)
    {}

}

static int work(CalculateForces *me,
                 StreamPtr ins[], StreamPtr outs[])
{
    Stream<int> in(ins[0]);
    Stream<float> out(outs[0]);

    float force[3] = {0.0f, 0.0f, 0.0f};

    int i = in.pop()*4;
    int N = in.pop()*4;

    for (int j=0; j<N; j+=4) {
        me->interaction(force, j, i);
    }

    float f = i/4;

    out.push(f);
    out.push(force[0]);
    out.push(force[1]);
    out.push(force[2]);

    return 0;
};

Figure 7.1: A SKIR version of the CalculateForces kernel from the *nbody* benchmark. The code is written in C++ using a class library interface to the SKIR intrinsics.
Table 7.1: The number and size of kernels for each benchmark. We bin the kernels in each benchmark according to their execution time in cycles. The table shows the number of kernels in each bin.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total</th>
<th>&lt; 100</th>
<th>101-500</th>
<th>501-1000</th>
<th>1001-5000</th>
<th>5001-20000</th>
</tr>
</thead>
<tbody>
<tr>
<td>beamformer</td>
<td>56</td>
<td>23</td>
<td>6</td>
<td>27</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ch. vocoder</td>
<td>57</td>
<td>4</td>
<td>52</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>dct</td>
<td>40</td>
<td>4</td>
<td>2</td>
<td>32</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>fft</td>
<td>17</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>filterbank</td>
<td>85</td>
<td>52</td>
<td>1</td>
<td>32</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fm</td>
<td>43</td>
<td>30</td>
<td>0</td>
<td>13</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total</th>
<th>&lt; 100</th>
<th>1000-10000</th>
<th>10,001-100,000</th>
<th>100,001-1e6</th>
<th>&gt; 1e6</th>
</tr>
</thead>
<tbody>
<tr>
<td>bs</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>dedup</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>swps3</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>nbody</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The result of running the StreamIt benchmarks on a two-socket, 8-core, Intel Xeon X5550 system using four different configurations and 1, 2, 4, 6, or 8 processor cores. For configurations using less than eight cores, we pin the benchmarks to a specific subset of cores using the `numactl` command line tool. For 1, 2, and 4 core configurations, we make sure that the benchmark runs on only one processor socket. For the 6 core configuration, we make sure benchmarks use 3 cores on each socket. For every configuration, we allow the operating system (Linux version: 2.6.32-33-server #70-Ubuntu SMP) to schedule threads within the subset of cores being used. That is, we do not pin worker threads to specific cores.

The `no-elim` configuration uses the coroutine scheduling method with the simple batching transformation. `cpp` is the baseline configuration plus the coroutine elimination optimization running the C++ source code of each benchmark and `str` is the same configuration using StreamIt source code as input. Finally, `fuse` adds dynamic kernel fusion to the `cpp` configuration. The results shown in the figure indicate that C++ and StreamIt versions of the same benchmark have similar performance. Examining the data more closely, we see that the C++ source code generally outperforms the StreamIt source code by a small amount. This is not surprising since the front-end
Figure 7.2: Throughput obtained by the SKIR compiler and scheduler for the StreamIt benchmarks using a varying number of threads. Error bars indicate one standard deviation.
Figure 7.3: The results of performing coroutine elimination on the StreamIt benchmarks using the SKIR just-in-time compiler.

used with the C++ code (llvm-gcc) is much more mature than our StreamIt to SKIR compiler and performs optimization. We also see in the results that coroutine elimination results in significantly better performance and that dynamic fusion often gives a slight performance improvement. For the two most fine-grained benchmarks (fft and dct), the configuration without coroutine elimination does not scale beyond a single processor socket.

Figure 7.3 gives a more detailed view of the data shown in Figure 7.2 with respect to performance improvements resulting from coroutine elimination. It shows the speedup obtained by the coroutine elimination optimization compared to the baseline configuration. The data shows that coroutine elimination results in significant performance improvements for programs with the most fine-grained kernels, typically 2-4x. For the fft benchmark, the improvement is over 12x. This is because of the high number of skir.peek operations in this benchmark, each of which requires offset calculation and bounds checking, and a low amount of work per input item consumed. Hoisting the computation associated with the skir.peek operations out of the kernel loop is sig-
The speedup is the least significant for the beamformer program which is dominated by kernels taking more than 500 cycles per input item consumed.

![Speedup Due to Fusion](image)

Figure 7.4: The results of performing dynamic fusion on the StreamIt benchmarks using the SKIR just-in-time compiler.

Figure 7.4 gives a more detailed view of the data shown in Figure 7.2 with respect to the performance improvement obtained using the dynamic fusion optimization. The figure does not show results for fft and channel vocoder. Due to extensive use of skir.peek operations, these benchmarks receive no benefit from fusion as we do not yet implement fusion of peeking kernels. dct sees the biggest gains from fusion at roughly 10-15%, depending on the number of threads used. We also see a general trend that fusion is more useful when more threads are used. This is likely due to the fact that fusion reduces parallelization overheads that are also reduced by simply running with a smaller number of threads. All of the results are statistically significant except for beamformer with one and two threads and filterbank with one thread.

Interestingly, only two fusions are ever performed when running dct. They are the fusion of two type conversion filters (e.g. int to float) with their neighbors in the stream graph.
From a programming perspective this type of filter boosts productivity by avoiding the need to modify existing components (e.g. by not rewriting a filter to use float instead of int). From a computation perspective these filters are essentially a noop and having them in a graph may be of little concern when using a static scheduling scheme. However, the results for SKIR show that from a dynamic scheduling perspective these types of filters can sometimes add a huge amount of scheduling overhead. This observation supports the claim made by this thesis that an optimizable representation of stream parallelism is useful when dynamic scheduling is used.

We also compare the performance of the SKIR dynamic scheduler to the performance of two mature stream processing systems: StreamIt and GNU Radio. We use a version of StreamIt pulled from a mirror the StreamIt code repository [14] with a last commit date of September 2, 2011. We use the SMP StreamIt backend which performs aggressive fusion followed by fission with a width equal to the number of threads. This scheduling transformation is described in [33]. The StreamIt compiler generates a statically scheduled stream graph as multithreaded C code. Our porting of the StreamIt benchmarks to GNU Radio was discussed above. GNU Radio uses a scheduling algorithm very similar to the one implemented by Parks to execute KPNs in the Ptolemy framework [48]. It uses one operating system thread per kernel. The threads sleep when their kernel is blocked and signal sleeping threads when a blocking condition is lifted.

We compile GNU Radio and the C code generated by StreamIt with llvm-gcc using the -03 optimization flag. Using the LLVM toolchain ensures that the same C frontend and the same backend optimization passes are used for SKIR, GNU Radio, and StreamIt. Like we did when running the SKIR benchmarks, we pin benchmarks to a subset of available cores using the numactl command line tool when targeting a number of threads less than 8. We compile StreamIt programs to use the target number of threads. We use the SKIR results from the previous figures with coroutine elimination enabled and without dynamic fusion.

The results of the comparison of the three schedulers is shown in Figure 7.5. We can see that the performance of SKIR is very competitive with both of the other scheduling schemes. For five of the benchmarks, SKIR performs the best when targeting 8 threads. GNU Radio performs
Figure 7.5: Throughput obtained by the SKIR dynamic scheduler, the StreamIt static scheduler, and the GNU Radio dynamic scheduler for the StreamIt benchmarks using a varying number of threads. Error bars indicate one standard deviation.
the worst except for filterbank where StreamIt performs poorly. This experiment validates our assertion that dynamic scheduling with support for SDF style optimization can perform as well as static scheduling for SDF style programs, at least for this set of benchmarks on this 8-core machine.

The limits of both dynamic and static scheduling are illustrated when we run a similar set of experiments on a larger number of cores. Figure 7.6 shows the results of running three different versions of each StreamIt benchmark on 1, 6, 12, 18, and 24 cores of a 48-core system containing four AMD Opteron 6174 processors. We run the benchmarks using SKIR with coroutine elimination but without dynamic fusion or fission, using the StreamIt compiled executables for eight threads from the previous experiments, and using StreamIt compiled executables targeting the actual number of threads. As in the previous experiments, we restrict the worker threads to the minimum number of multi-core processors and physical sockets using the Linux `numactl` utility. In this system, there are two six core processors per die for a total of twelve processors per socket.

The first thing to notice in Figure 7.6 is the lack of portability seen in the executables compiled by the StreamIt compiler targeting eight threads. Not only does the measured throughput plateau after the number of available processors exceeds eight, the performance under performs the executable compiled to target the actual number of threads in all but one case. As one would expect from a dynamic scheduler, SKIR generally outperforms StreamIt when the actual number of threads is different from the number of threads targeted at compile time. As in the previous experiments, the StreamIt compiled versions of filterbank perform poorly.

The SKIR results in Figure 7.6 show the limits of dynamic scheduling without kernel fusion or kernel fission when compared to the StreamIt static scheduling algorithm. As shown in Table 7.1, the fm benchmark is dominated by 13 kernels. Thus a system without kernel fission cannot scale much beyond 13 threads and this is clearly seen in the figure. A similar effect is seen in the fft results, where the total number of kernels is 17. When we force SKIR to use 24 threads for this benchmark, the result is excessive work stealing overhead from TBB. Of course in a more realistic scenario, the number of threads would not be allowed to grow beyond the number of active kernels. For the dct benchmark, Table 7.1 shows that the benchmark contains 32 kernels large
Figure 7.6: Throughput obtained by the SKIR dynamic scheduler, the StreamIt compiler targeting 8 threads, and the StreamIt compiler targeting the actual number of threads. Error bars indicate one standard deviation.
enough that we might expect reasonable scaling. However, this number does not capture the fact that each invocation of the kernel work function produces and consumes 16 data items on the input and output streams. This fact along with two 16-wide split-joins leads to high communication to computation requirements in a dynamic scheduling scheme. On our AMD Opteron system, this communication requirement leads to poor scaling beyond a single multi-core processor and performance losses beyond a single processor socket. In contrast, the static scheduling algorithm of StreamIt transforms the dct computation into a single split-join using aggressive static fission and fusion [33]. This reduces communication requirements by at least 50% compared to a dynamic schedule.

The results for dct in Figure 7.6 indicate a need for further research into dynamic optimization of very fine grained programs when dynamic scheduling is used with a large number of processing cores. We have already shown that a simple dynamic fusion optimization is useful for reducing communication and scheduling overhead when the number of kernels is larger than the number of cores. And, for benchmarks like fm, we can apply similar techniques involving kernel fission to increase program granularity. The performance of simple forms of manual and automatic fission is described below. For the benchmark dct, however, it appears that a combination of fusion and fission is more effective. That is, perform fusion to create a single data parallel kernel, then employ data parallel execution using kernel fission or other means to regain granularity without requiring all of the communication present in the original form of the program graph.

We evaluate the performance of black-scholes, dedup, and swps3 against their original parallelization using TBB, Pthreads, and FastFlow, respectively. We run the programs using 1, 2, 4, 6, and 8 threads. For each benchmark, we match the thread allocation strategy used in the original non-SKIR version. In each case, this means simply using the number of available processors as the fission width. For dedup, dynamic fission is run on the two most compute intense kernels. For the other benchmarks, manual fission is used on the single compute intense kernel. The results of this evaluation are shown in Figure 7.7. Results are normalized to the single threaded execution time of the original benchmark. Execution time is obtained by running the programs with the Linux
Figure 7.7: A comparison of SKIR performance to the performance of coarse-grained pipeline parallel applications parallelized using Pthreads, TBB, and FastFlow. Performance is measured as execution time.

time command. Thus the SKIR results include JIT compilation overhead. The mean of seven runs is reported. The variance of these results is very small, with standard deviations less than one half of one percent of execution time for all swps3 and black-scholes experiments. For dedup more variance is seen, with standard deviations of less than one percent of running time in almost all cases. In no case is the variance large enough to statistically impact the conclusions described below.

The biggest difference in performance is seen for dedup. SKIR performs better than the Pthreads implementation for all but the two thread configuration. The performance difference is 2% for one and two threads and roughly 8%, 18%, and 11% percent for four, six, and eight threads, respectively. This benchmark shows how, for even a simple 5 stage pipeline passing packets of information, using a stream parallelism framework can be beneficial. Not only did porting this benchmark to SKIR simplify the implementation by removing custom code related to queue and thread management, it also improved performance by using a scheduler specifically designed for pipeline parallel computation.
One would expect the performance of black-scholes on SKIR to be very similar to the performance running with TBB since SKIR is implemented on top the TBB scheduler and indeed this is the case. SKIR was 0-1% faster than TBB for black-scholes using 1, 2, 4, or 8 threads. For 6 threads, SKIR was 11% faster than TBB. While the exact reason for the difference in this last case is unclear, it may be due to imbalance in the division of work among tasks. In TBB this division is done by recursively splitting the iteration space of the data parallel kernel using a parallel for loop. In SKIR this same division is done using split-join, which emits much finer grained chunks of work than the recursive splitting of TBB.

SKIR is 0-2% slower than FastFlow for swps3. Most of the computation in this benchmark is performed by hand written SIMD assembly routines running millions of cycles per kernel iteration (Table 7.1). Thus the difference in performance is almost completely due to JIT compilation overhead and differences in scheduling and communication. Using detailed profiling, we can attribute about half of the performance difference to fixed JIT compilation costs. We can also observe that as the runtime of the benchmark increases due to increased input size or decreased parallelism, this cost approaches zero as a percentage of execution time. The remainder of the difference is in scheduling and communication costs. FastFlow has a scheduler and stream communication optimized to use only the split-join pattern (called a Farm pattern in [19]). In SKIR, runtime fission maps the same pattern onto a much more general scheduler using a compiler library of split and join kernels. The lack of specialization in SKIR for this specific pattern likely causes the remainder of the performance difference.

We evaluate our OpenCL backend using nbody. We run the benchmark with the compute intense kernel executing on a GPU using the OpenCL backend and compare the results with ordinary kernel fission using 1, 2, and 4 threads. The CPU for this experiment is a 4-core Intel i7 920 processor and the GPU is an AMD Radeon HD 5750. The results are shown in Figure 7.8. The benchmark reports the performance of the simulation in GFLOPS. The program scales well with kernel fission using 1-4 cores but performance increases significantly when the kernel is moved onto the GPU. The kernel contains manual loop unrolling (as does the original NVIDIA version),
but our OpenCL backend does not perform any other optimizations specific to the GPU. Based on the performance of a hand tuned OpenCL kernel, it appears that the performance could be increased another 2x, based largely on GPU-centric memory optimization. The results also suggest that we are leaving performance on the table in the CPU version, largely due to lack of vectorization.

Figure 7.8: Performance of the nbody benchmark running on SKIR using kernel fission on the CPU and OpenCL on the GPU.

### 7.3 Sluice

In this section we evaluate the characteristics and performance of our Sluice JavaScript acceleration system built on top of SKIR. We use compute intense JavaScript benchmarks that can be expressed in the stream parallel model. Four of the benchmarks are taken from the Pixastic library of image processing routines [10]. We also include a nbody benchmark adapted from the SKIR version of the CUDA benchmarks mentioned above. This benchmark is similar to code that might be found in a game engine.

As written, the code found in the Pixastic library is already very close to the form required by Sluice. We must perform only a few modifications to turn the image processing routines into Sluice kernel objects. Because these routines operate at the granularity of an entire image, we identify parameters that only change between images, such as the image itself. These parameters are re-written as kernel state that is initialized when the kernel is constructed. The rest of the function
body is placed within a kernel work function. Because the Pixastic functions operate on an image at a time, there is little need for stream communication in these benchmarks. Nevertheless, SKIR requires that a kernel has at least one input or output stream. We fulfill this requirement by passing the image width and height to the kernel using streams.

An example from Pixastic is shown in Figure 7.9. In this figure, the original Pixastic code as well as the Sluice version are shown for the invert routine. We run all of the Pixastic benchmarks on image data that is read from disk into memory before timing begins. All images are in RGBA format with dimensions of 2592x1944 (5 Megapixels).

```javascript
function invert_pixastic(params) {
    var data = Pixastic.prepareData(params);
    var invertAlpha = !!params.invertAlpha;
    var rect = params.options.rect;
    var p = rect.width * rect.height;
    var pix = p*4, pix1 = pix+1;
    var pix2 = pix+2, pix3 = pix+3;
    while (p--) {
        data[pix-=4] = 255 - data[pix];
        data[pix1-=4] = 255 - data[pix1];
        data[pix2-=4] = 255 - data[pix2];
        if (invertAlpha)
            data[pix3-=4] = 255 - data[pix3];
    }
    return true;
}
```

```javascript
function invert_sluice(data, invert_alpha) {
    this.data = data;
    this.invertAlpha = invert_alpha;
    this.work = function () {
        var w = this.pop();
        var h = this.pop();
        var p = w * h;
        var pix = p*4, pix1 = pix + 1;
        var pix2 = pix + 2, pix3 = pix + 3;
        while (p--) {
            this.data[pix-=4] = 255-this.data[pix];
            this.data[pix1-=4] = 255-this.data[pix1];
            this.data[pix2-=4] = 255-this.data[pix2];
            if (this.invertAlpha)
                this.data[pix3-=4]=255-this.data[pix3];
        }
        this.push(w); this.push(h);
        return false;
    }
}
```

Figure 7.9: Comparison of Pixastic code (left) with the same code ported to Sluice (right). The code that is shown inverts all the pixels in an image.

The nbody physics benchmark is written as a three stage pipeline. Almost all the computation takes place in the middle stage, shown in Figure 7.10. Each execution of this stage’s kernel work function computes the forces on single particle due to all other particles in the system. The computed forces are pushed to the output stream. Each execution of the entire three stage pipeline corresponds to a single iteration of the nbody simulation. That is, it computes the force and updates the positions and velocities (in the third stage) for all the particles in the system.

We run our implementation of Sluice on node.js, a non-browser JavaScript environment
function CalculateForces(pos_rd, softeningSquared) {
    this.m_pos_rd = pos_rd;
    this.m_softeningSquared = softeningSquared;
    this.work = function () {
        var force = [0.0,0.0,0.0];
        var i = this.pop()*4;
        var N = this.pop()*4;
        for (var j=0; j<N; j+=4) {
            var r0, r1, r2;
            r0 = this.m_pos_rd[j+0] - this.m_pos_rd[i+0];
            r1 = this.m_pos_rd[j+1] - this.m_pos_rd[i+1];
            r2 = this.m_pos_rd[j+2] - this.m_pos_rd[i+2];
            var distSqr = (r0 * r0) + (r1 * r1) + (r2 * r2);
            distSqr += this.m_softeningSquared;
            var invDist = 1 / Math.sqrt(distSqr);
            var invDistCube = invDist * invDist * invDist;
            var s = this.m_pos_rd[i+3] * invDistCube;
            force[0] += (r0 * s);
            force[1] += (r1 * s);
            force[2] += (r2 * s);
        }
        var f = i/4;
        this.push(f);
        this.push(force[0]);
        this.push(force[1]);
        this.push(force[2]);
        return false;
    };
}

Figure 7.10: The CalculateForces kernel found in the nbody benchmark. Ported from the SKIR C++ version of the benchmark shown in Figure 7.1.

build on top of the V8 JavaScript execution engine. Node is typically used in the development of network applications. The single threaded scheduling algorithm implemented by Sluice also makes use of node-fibers [9], a package providing fibers/coroutines for node.js. This is necessary because generators (i.e. yield) are not yet in the language. All experiments are run on a four core Intel i7-920 processor under Ubuntu 10.10.

7.3.1 Single Threaded Offload

We first evaluate the performance of our system when offloading a Sluice kernel to a single threaded SKIR runtime. Because the Sluice and SKIR layers run as separate processes, the offloaded kernel still runs in parallel with the rest of the Sluice program, but we do not attempt to further parallelize the offloaded kernel. The results of this experiment for the Pixastic kernels can be seen in Figure 7.11. We measure three cases: ref shows the performance of the original image processing routine called as an ordinary JavaScript function whereas skir cold and skir warm show
Figure 7.11: Performance of Pixastic image processing routines coded as ordinary JavaScript compared to the performance of the same routines running as Sluice kernels using the SKIR runtime. The results of this experiment for the \textit{nbody} benchmark can be seen in Figure 7.12. The Figure shows the average time per simulation iteration when using a procedural JavaScript version of the benchmark, the three stage pipeline Sluice version, and the Sluice version with the \texttt{CalculateForces} kernel offloaded to SKIR. The results are similar to those for the Pixastic benchmark, with significant performance improvements due to our specialized code generation in most cases. For the smallest test, a system of only 100 particles, the overhead of acceleration overwhelms the benefit.
7.3.2 Task Parallel Execution

Although we did not parallelize the Pixastic benchmarks internally, we can still execute multiple instances of a particular kernel in parallel. This is the simplest form of task parallelism exposed by Sluice programs. To test this, we created a Sluice program that sequentially (because JavaScript is single threaded) creates, compiles, and executes a varying number of image kernels using the SKIR runtime. We give the runtime 8 worker threads (equal to the number of hardware threads) to run the kernels, so up to 8 kernel instances can run in parallel. This scenario is similar to what might be encountered in a compute intense server-side JavaScript application.

Figure 7.13 shows the results of running a varying number of Pixastic kernels on different images concurrently using Sluice task parallelism. It shows the mean time required to process a single image. We observe that as the number of executing kernel instances increases, the system starts to effectively mask much of the overhead associated with running Sluice kernels under SKIR. Eventually, however, the lines on the graph flatten, because the system cannot go faster than the
Figure 7.13: Per image processing time for the Pixastic benchmarks implemented when a varying number of images are processed using Sluice task parallelism.

sequential parts of the program (the individual kernels and the Sluice runtime). The larger edge detection and sharpen kernels show the best results as they contain the most computation to overlap with other work. Likewise, invert and sepia show the worst improvement, because their execution is dominated by sequential overhead.

7.3.3 Data Parallel Execution

Stream program kernels are often written so that they read, but do not modify, their internal state. When this is true – as it is for the CalculateForces kernel in the nbody benchmark – it may be profitable to run multiple copies of a single kernel instance on different portions of the input stream. This is how the stream programming model exposes data parallelism.

Figure 7.14 shows the results of executing the nbody benchmark with the CalculateForces kernel offloaded and with data parallelism enabled in the SKIR runtime. The figure shows results for 1024, 2048, 3072, and 4096 particles in the simulated system while using 2, 4, or 8 worker threads in the SKIR runtime. The benchmark runs several hundred iterations of the simulation
Figure 7.14: Speedup of the *nbody* benchmark due to data parallelism when the CalculateForces kernel is run on a multi-threaded SKIR runtime compared to the same benchmark using a single-threaded SKIR runtime.

and reports mean time per iteration. The speedup versus using a single threaded SKIR runtime is shown. All of the simulation sizes show performance improvement of 2-2.5x due to parallelism when utilizing the entire test machine. We point out that because JavaScript fully utilizes one of the cores in our four core test machine, and because this experiment measures scaling in the non-JavaScript portion of execution, we don’t expect that the best case speedup is much greater than 3x.
8.1 Compiler Representations for Stream Parallelism

There is little existing work in the area of source language and target architecture independent program representations for stream parallelism. However, there have been at least three variations of the Stream Virtual Machine (SVM) proposed [35][42][45]. The general goal of SVM is similar to one of SKIR’s goals: to provide an architecture independent target for high level stream programming languages such as Brook and StreamIt. As in SKIR, a two-level approach to compilation is used. First, a high level compiler (HLC) generates a SVM program from a high level stream parallel language. Then, a low level compiler (LLC) lowers the program to machine code. The virtual machine model consists of local memories for storing stream data, kernel processors for executing kernels, DMA engines for data transfer, and a control processor to control the entire machine. In all of the proposals the SVM language targeted by the HLC consists of C plus a version of the SVM API. SVM programs are structured as a single thread of control which calls the SVM API to issue kernel execution, memory movement, and dependency description operations to virtualized stream parallel hardware. An example of SVM-C pseudo code from [35] is shown in Figure 8.1. For each iteration of the inner loop in the figure, the program must issue loads and stores for chunks of stream data, spawn two executions of kernel work functions, and describe to the runtime the dependencies between these operations.

It should be clear from the figure that the division of responsibility between the high level and low level compilers are very different in SKIR and SVM. In the SVM model of Labonte et.
for (ns = 0; ns < NUM_STrips; ns += 2) {
  for (i = 0; i < 2; i++) { //Double buffering
    streamLoad (as[i], a, start_idx, Ns, sizeof (Type a));
    streamLoad (idxs[i], idx, start_idx, Ns, sizeof(int));

    addDep (Gcs[i], Lidxs[i]);
    streamGather (cs[i], c, idxs[i], Ns, sizeof (Type c));

    addDep (K1, Las[i], Lidxs[i], Gcs[i]);
    kernelCall (K1, as[i], cs[i], ds[i]);

    streamLoad (xs[i], x, start_idx, Ns, sizeof (Type x));

    // K2 depends on K1, loads to as[i] and xs[i]
    addDep (K2, K1, Las[i], Lxs[i]);
    kernelCall (K2, ds[i], as[i], xs[i], zs[i]);

    addDep (Szs[i], K2);
    streamStore (zs[i], z, start_idx, Ns, sizeof (Type z));
    start_idx += Ns;
  }
}

Figure 8.1: An example of SVM-C pseudo code.

al. [42] and Mattson et. al. [45], the HLC is responsible for parallelism detection, load balancing, coarse-grained scheduling of kernels, and memory management of streams while the LLC is only responsible for code generation and optimization within kernels. The Streamware SVM implementation adds another layer of abstraction [35]. The high-level compiler still performs stream parallel transformations but leaves the emitted C code in a parameterized form. The run-time system fills in the appropriate values based on the architecture on which the program is executing. This gives slightly better portability than previous instances of SVM. The SVM approach is to separate compilation related to the stream parallel model from traditional compiler functionality whereas in SKIR the goal is to handle both in the same low level compiler.

Similar to the SVM, the Multicore Streaming Layer (MSL) was introduced in [55] as a stream processing runtime for the Cell processor. Like with the SVM, the MSL is used as a C API to abstract away the details of data movement and kernel execution. A modified backend for the the StreamIt compiler was designed to generate stream programs targeting the MSL and the kernel work emitted were compiled with the Cell GCC backend. The resulting compiler stack was used to
evaluate static and dynamic scheduling schemes for the Cell. As one might expect, they found that the dynamic scheduler was more general (e.g. it can handle unpredictable stream programs), but in their case carried a higher overhead than a the static scheduler. With SKIR we show how dynamic scheduling overheads can be reduced to be competitive with static scheduling while maintaining flexibility.

The design of the Erbium system [47] is much closer to the design philosophy of SKIR than SVM. Like SKIR, the intermediate representation designed for Erbium can be used by compiler front-ends and by low level programmers. The key difference between SKIR and the Erbium IR is that in the Erbium IR the data structure implementing stream communication is purposely exposed to the program through a rich API. The stated reasons for this are to provide peek and poke operations as well as multiple producers and consumers. In SKIR, the implementation of stream communication is purposely hidden to promote compiler flexibility and portability. This means that the semantics of SKIR stream operations would not need to change if multiple producers and consumers were desired. The way similar functionality is provided in SKIR today is with split-joins. At one time SKIR supported poke operations but we did not find a need for them and they were removed.

8.2 Dynamic Scheduling of Stream Parallelism

Dynamic scheduling for process networks in the Ptolemy system is implemented using bounded FIFO buffers, POSIX threads, monitors, and condition variables [48]. The operation of the scheduler is quite simple. For each kernel, a POSIX thread is created. The main procedure of the thread repeatedly executes the kernel work function. Whenever a blocking read or write is encountered, the thread is put to sleep. When data or space becomes available, the thread is woken up. This is also the basic scheduling algorithm used in the GNU Radio system [4]. We compare the performance of SKIR to GNU Radio in Chapter 7.

The authors of Streamware also describe a dynamic task queuing scheduler that is quite different than the SKIR dynamic scheduler [35]. Because of the design of SVM, the Streamware
scheduler relies on the high level stream language compiler to decompose the execution of a stream program into a long series of kernel execution operations, dependency description operations, and data movement operations, whereas in SKIR the front-end compiler leaves kernels intact, maintains dependency information in its original form (as edges in the stream graph), and uses demand/data driven execution. Streamware also uses a stream parallelism specific queuing scheme with a dedicated kernel execution queue and a dedicated data movement queue whereas SKIR is built on a general purpose work stealing scheme and kernels perform their own data movement.

A stream specific queuing system more closely related to the SKIR dynamic scheduler is found in the GRAMPS scheduler [50]. The GRAMPS scheduler uses fairly complex task stealing system specific to stream parallelism with per-kernel task queues of varying priority on each worker thread and an explicit backpressure mechanism. In contrast, our simpler scheduling algorithm uses an implicit backpressure mechanism and runs on a general purpose task stealing system (TBB) with only minor modification to support targeted task stealing (about 40 lines of code). This is important because programs may contain forms of parallelism other than stream parallelism and using a common scheduling infrastructure is desirable to avoid interference and over-subscription. TBB itself contains mechanisms to handle these situations by fairly allocating work threads among different parallel contexts executing in the same process.

The Flextream system [37] provides a small amount of dynamic adaptation by using an approach similar to SVM. At compile time, a static compiler finds an optimal schedule for virtualized stream parallel hardware which is a superset of the target hardware. At runtime, the schedule can be refined for more constrained hardware or when available hardware resources change. The motivation for Flextream is that static scheduling algorithms are too slow for runtime re-compilation. This is also a motivation for exploring a fully dynamic scheduling mechanism for SKIR.

Two other recent dynamic scheduling work related to SKIR are DoPE and FDP [49][51]. These are high level software frameworks built in C++. The primary goal of these systems is to choose the best allocation of threads to data parallel kernels found in pipeline parallel programs given a certain goal. This programming pattern is a subset of stream parallelism. In FDP the goal is
power reduction while the authors of DoPE provide APIs to build schedulers with different goals. Both of these works are orthogonal to the dynamic fission performed for SKIR (Section 6.6.2) and could be used to help choose fission width.

8.3 Stream Parallelism for Heterogeneous Hardware

This thesis describes one way SKIR can be compiled to execute on graphics hardware. This functionality is motivated by a large amount of prior work showing that stream parallelism in general and the SDF model specifically are appropriate programming models for heterogeneous computing. Any number of techniques could be taken from the existing literature on the subject and applied to static regions of SKIR stream graphs.

The Brook language [23] described in Chapter 2 as well as the related CUDA and OpenCL frameworks are specifically designed to execute on data parallel hardware such as GPUs. Hor-mati, et. al. [38] describe techniques to generate optimized CUDA code from StreamIt kernels. These techniques could be used to extend the methodology described for SKIR. Udupa et. al. [54] describe a more comprehensive approach to statically dividing the work of a StreamIt program between a GPU and CPU using an Integer Linear Program formulation. Approximations of such methods could be used to perform runtime work partitioning in a dynamic system with load balancing such as SKIR.
Chapter 9

Conclusion

This thesis extends the state of the art in stream parallelism by unifying static and dynamic approaches to the construction, compilation, and scheduling of stream parallel computation. Previous work in the area of static compilation and scheduling has shown the power of restricted stream parallel models such as Synchronous Dataflow. Using these models compilers can compute efficient static schedules, target a variety of hardware types, and perform sophisticated transformations of stream graphs. But, because past work relies on specialized languages and restricted programming models, few of these techniques apply to parallel programming in general purpose languages. Previous systems for stream parallelism in general purpose languages can implement efficient dynamic scheduling but lack the ability to apply the aggressive optimization available to stream parallel compilers.

The Stream and Kernel Intermediate Representation (SKIR) addresses these issues by providing a compiler level representation for stream parallel computation. It is a general representation which can be used to express arbitrary stream graphs containing pipeline and data parallelism. Chapter 4 explored four different uses for SKIR as a compilation target: as low level primitives for C; as an implementation language for object oriented stream parallelism in C++; as a target for the specialized stream parallel language StreamIt; and as an acceleration layer for the dynamic JavaScript language.

This thesis also introduced several techniques to support efficient execution of SKIR programs. In Chapter 5 simple dynamic scheduling and stream communication mechanisms for par-
allel execution of SKIR kernels were described. Chapter 6 showed how we can perform compiler optimizations in support of these scheduling and communication mechanisms both in general and when program kernels conform to stricter stream parallel models. Chapter 6 also demonstrated the ability to perform stream graph level optimizations on SKIR program graphs by describing dynamic kernel fusion and fission transformations and by describing a SKIR to OpenCL backend. Compiling C++ to run on a GPU is not an easy task, but this was shown to be feasible using the SKIR abstractions.

In Chapter 7, we showed that the SKIR program representation combined with our scheduling and compilation techniques obtains performance competitive with mature systems for stream parallelism. Our fine-grained SDF-style StreamIt benchmarks generally perform equal to or better than the same code compiled by the StreamIt compiler or running under the GNU Radio dynamic scheduler. For coarse grained data and pipeline parallel programs, the SKIR system again performs at or beyond the level of existing systems. The SKIR compiler and runtime achieves this performance while allowing far more flexibility and portability than prior systems.

This thesis demonstrates that the SKIR program representation along with dynamic scheduling and compilation is a useful tool for implementing stream parallel programming tools. In the future we look forward to using the SKIR platform to explore more optimization and code transformations opportunities for heterogeneous multi-core hardware.
Bibliography


international conference on Architectural support for programming languages and operating systems, ASPLOS ’11, pages 381–392, New York, NY, USA, 2011. ACM.


