SIFt: A Compiler for Streaming Applications

by

Elliot L. Waingold

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

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2000

© Elliot L. Waingold, MM. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly
paper and electronic copies of this thesis and to grant others the right to do so.

Author ....
Department of Electrical Engineering and Computer Science
May 15, 2000

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

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

SIFt: A Compiler for Streaming Applications

by

Elliot L. Waingold

Submitted to the Department of Electrical Engineering and Computer Science
on May 15, 2000, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Due to the increasing popularity of multimedia content and wireless computing, streaming applications have become an important part of modern computing workloads. Several hardware architectures that are geared towards such applications have been proposed, but compiler support for streams has not kept pace. This thesis presents an intermediate format and set of compilation techniques for a stream-aware compiler. The prototype implementation of this compiler targets the MIT Raw architecture. Performance results suggest that certain applications perform better when compiled using these techniques than when compiled using ILP-extraction techniques alone. However, these same results show that further optimizations are necessary to realize the maximum throughput of a streaming application. Three such optimizations are outlined.

Thesis Supervisor: Saman Amarasinghe
Title: Assistant Professor

Acknowledgments

I am grateful to my advisors Saman Amarasinghe and Anant Agarwal for their technical guidance as well as funding and moral support. I also thank the other members of the MIT Raw project for developing the software infrastructure that made much of this research possible. Finally, many thanks to Ernst Heinz, Kath Knobe, and Bill Thies for offering external perspective on the problem of compiling streaming applications.

Contents

1 Problem Statement **9**

2 Background **12**

2.1 Stream Theory 12

2.2 Related Work 14

3 Intermediate Format **18**

3.1 Structure 18

3.2 Informal Semantics 20

3.3 Formal Semantics 23

3.4 Example Program 29

4 Compilation **32**

4.1 Overview 32

4.2 Analysis 33

4.3 Placement 34

4.4 Communication Scheduling 39

4.5 Code Generation 45

5 Optimization **50**

5.1 Propagating the Communication Context 50

5.2 Analyzing Communication Patterns 53

6 Results **55**

6.1 Evaluation Environment 55

6.2 Experimental Results 58

7 Conclusion **62**

7.1 Summary 62

7.2 Future Work 64

A Using the SIFt Compiler **66**

A.1 Installing the Compiler 66

A.2 Compiling a Program 67

B Extended Example: IDEA **69**

List of Figures

1-1 Simplified MPEG video decoding process. 10

2-1 Sieve of Erasthones implemented in Scheme using streams. 13

2-2 Stream transducer network using all four common transducer types. 13

2-3 The Raw microprocessor. 17

3-1 Syntactic domains used to specify the structure of SIFt. 19

3-2 An s-expression grammar of SIFt. 19

3-3 Semantic domains used in the operational semantics of SIFt. 24

3-4 Desugaring of degenerate and complex SIFt forms. 25

3-5 Desugaring of simple SIFt forms. 26

3-6 Rewrite rules for simplified transition relation $\Rightarrow_s$. 27

3-7 Formal definition of substitution. 27

3-8 Rewrite rules for general transition relation $\Rightarrow$. 28

3-9 Structure of the FIR filter program. 30

3-10 SIFt program that implements the FIR filter $[2, 3, 4, 5]$. 31

4-1 The main phases of the SIFt compiler. 33

4-2 Communication graph for FIR filter. 34

4-3 Random initial placement for FIR filter. 39

4-4 Optimized final placement for FIR filter. 39

4-5 Progression of temperature and energy during annealing. 40

4-6 First communication context of FIR filter. 44

4-7 Second communication context of FIR filter. 44

4-8 Routing of second communication context for FIR filter. 46

4-9 Switch code for tile (0,0) of FIR filter. 46

4-10 Forcing the network to proceed to a particular channel's context. 47
4-11 Code generated to force the network to proceed. 48

5-1 Optimized code for forcing the network to proceed. 51
5-2 Inference rules for propagating `ccr`. 52

6-1 Placement of eight-tile buffer program. 56
6-2 Placement of eight-tile adder program. 56
6-3 Placement of sixteen-tile IDEA program. 57
6-4 Performance of buffer benchmark. 59
6-5 Performance of adder benchmark. 59
6-6 Performance of FIR filter benchmark. 60
6-7 Performance of IDEA benchmark. 61

B-1 SIFt code for IDEA benchmark, part 1. 70
B-2 SIFt code for IDEA benchmark, part 2. 71
B-3 SIFt code for IDEA benchmark, part 3. 72
B-4 SIFt code for IDEA benchmark, part 4. 73
B-5 Communication graph of IDEA benchmark. 74
B-6 Placement of IDEA benchmark on 4x4 Raw chip. 74
B-7 Communication context 0 of IDEA benchmark. 75
B-8 Communication context 1 of IDEA benchmark. 75
B-9 Communication context 2 of IDEA benchmark. 76
B-10 Communication context 3 of IDEA benchmark. 76
B-11 Communication context 4 of IDEA benchmark. 77
B-12 Communication context 5 of IDEA benchmark. 77
B-13 Communication context 6 of IDEA benchmark. 78

List of Tables

4.1 Relationship between physical and simulated annealing. 35

6.1 Throughput of SIFt programs (inputs processed per kilocycle). 58

6.2 Throughput of C programs (inputs processed per kilocycle). 58

Chapter 1

Problem Statement

The popularity of multimedia content and wireless computing has resulted in an abundance of computing resources dedicated to streaming applications. In fact, media applications alone have already become the dominant consumer of computing cycles [18], and the growth rate has been phenomenal. Usage of streaming media players, the software that decodes and renders audio and video data on a personal computer, has grown from just a few million users in 1997 to 20 million home users and 9 million business users in 1999 [13]. Wireless computing has grown in popularity as well, including such technologies as personal digital assistants, cellular phones, and telemedicine. The algorithms that form the underpinnings of wireless communication systems are also stream-based.

Such streaming applications are characterized by simple, independent, repetitive operations on elements of an ordered sequence of data values. Therefore, these applications exhibit a large degree of fine-grain parallelism. For instance, MPEG decoding can be pipelined into several stream processing stages to allow multiple data values to be operated upon concurrently [8]. Figure 1-1 shows a simplified version of the MPEG decoding process as a series of stream processing stages. As data is streamed through the pipeline, each stage can operate concurrently on a different portion of the input. This type of decoding process occurs whenever you watch an MPEG-encoded news broadcast or listen to MP3-encoded music over the web. The software radio receiver proposed by Bose et al. [3] has a similar decomposition. Software radio technology allows wireless communication devices to be constructed and maintained more flexibly than traditional all-hardware solutions.

Several processor architectures have already been designed to exploit the available par-

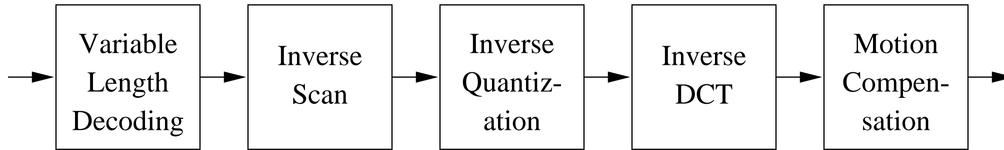

Figure 1-1: Simplified MPEG video decoding process.

allelism in streaming applications. The Cheops video processing system makes heavy use of specialized hardware for computationally intensive stream operations [4]. The Imagine architecture for media processing is also organized around the concept of streams, but without specialized functional units [18]. The Raw architecture, which is the target of our compiler, supports streaming codes through a combination of simple replicated processing elements and a fast compiler-controlled interconnection network [25]. This is not an exhaustive list of stream-oriented architectures; several others exist.

Despite all of this hardware effort, few of these architectures have been matched by languages and compilers that facilitate the programming of stream applications. Such languages should allow stream computation to be expressed naturally and independent of the target machine. The programmer must not bear the burden of orchestrating concurrent execution, either through program structure or annotations. Instead, the compiler should extract an appropriate degree of parallelism from the program based on its knowledge of the target architecture. Most existing streaming systems use a software layer to handle the streams. Although this approach partially achieves the goal of simplifying the software engineering effort, it introduces unnecessary runtime overhead.

This vision presents four major technical challenges:

1. Which language should be chosen for programming streaming applications? There are several dimensions to this question, including imperative vs. declarative and implicit vs. explicit parallelism.

2. How should stream computations be represented by the compiler? The intermediate format should be general enough to accommodate a variety of input languages, simple enough to make code generation straightforward, and precise enough to be amenable to optimization.

3. What techniques are required to compile this intermediate format to efficient machine code on a parallel architecture? In particular, we are interested in architectures that

support the fine-grained parallelism exploitable in streaming applications.

4. What optimizations can gainfully be applied to the intermediate format and during code generation? The most interesting of these will be non-traditional optimizations that take advantage of the streaming structure.

This thesis focuses on the latter three questions.

To address Question 2, we first propose an intermediate format, called the Stream Intermediate Format (SIFt, pronounced "sift"), that represents streaming applications as a fixed configuration of threads and communication channels. The overall structure of SIFt is presented, along with an informal semantics. A formal semantics is also provided, which serves to clear up any ambiguity in the informal description and can be used to verify the correctness of optimization and code generation.

To address Question 3, we then show how this format can be compiled to the Raw architecture. The SIFt compiler transforms a concrete syntax of the intermediate format into Raw machine code through a series of steps. The first phase is placement, in which threads are assigned to processors using techniques borrowed from VLSI layout. The second phase is communication scheduling, which employs compiled communication to leverage Raw's programmable interconnection network. The final phase is code generation for the processor and communication switch of each Raw tile.

To address Question 4, we discuss opportunities for optimization in this back-end compilation process. These include eliminating overhead in individual send and receive operations and more intelligent communication scheduling that utilizes predicted communication volume on individual channels. The details of lowering a high-level stream language to SIFt and performing high-level optimizations is left to future work.

The rest of this thesis is organized as follows. Chapter 2 presents some theoretical background on streams and related work in languages and architectures. Chapter 3 describes the structure and semantics of the intermediate format. Chapter 4 details the algorithms used in compilation, while Chapter 5 proposes optimizations that could be incorporated into the compiler. Chapter 6 describes some experimental performance results. Chapter 7 concludes and discusses potential future work.

Chapter 2

Background

For decades, streams have been used as a programming abstraction for modeling state [1]. In that context, they have been used to model time-varying data values without recourse to assignment and mutable data, thereby simplifying the analysis and optimization of programs that use them. Many algorithms can be implemented quite elegantly and efficiently using the stream paradigm. Even operations on infinite streams can be easily expressed. The sieve of Erasthones, an algorithm for enumerating the prime numbers, is implemented using streams in Figure 2-1. In the figure, `cons-stream`, `stream-car`, `stream-cdr`, and `stream-filter` are delayed versions of the standard list operations. These delayed versions improve space efficiency by not generating the tail of a list until some consumer requires it.

2.1 Stream Theory

For our purposes, a *stream* is simply an ordered collection of data values. In general, these values can be of any type, even themselves streams. Computation on streams take the form of *stream transducers*, which take zero of more streams as input and generate zero or more streams of output. Four basic types of stream transducers occur commonly in practice, including *enumerators*, *mappings*, *filters*, and *accumulators* [14]. These types differ both in the number of input and output streams that they take and their functionality. An enumerator generates values into an output stream from "thin air", not reading from any input streams. A mapping applies a function element-wise to one or more input streams. A filter selectively discards elements from input to output. An accumulator is the dual of an enumerator, reading from an input stream but outputting nothing. These simple types

```
(define (sieve stream)
  (cons-stream
   (stream-car stream)
   (sieve (stream-filter
           (lambda (x)
             (not (divisible? x (stream-car stream))))
           (stream-cdr stream)))))

(define (integers-starting-from n)
  (cons-stream n (integers-starting-from (+ n 1))))

(define primes (sieve (integers-starting-from 2)))
```

Figure 2-1: Sieve of Erasthones implemented in Scheme using streams.

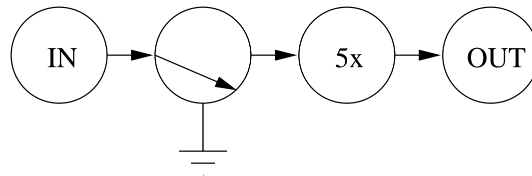

Figure 2-2: Stream transducer network using all four common transducer types.

of transducers can be composed to form arbitrarily complex ones.

Figure 2-2 is an example transducer network that uses all four common transducer types to decimate and scale an input stream. The circles represent transducers and the arrows represent streams flowing between them. The input is an enumerator, while the output is an accumulator. The first processing stage is a filter that drops every other packet, while the second is a mapping that scales each element by five.

Streaming applications exhibit a great deal of potential concurrency, which comes in three major flavors. *Pipeline parallelism* is the ability to execute multiple nested transducers concurrently. After the pipeline is primed, multiple transducers can operate on different portions of the input stream at the same time. In the decimate-and-scale example, this means that the filter can drop elements at the same time that the multiplier operates on elements that have already passed through. *Control parallelism* is the ability to concurrently generate multiple input streams for a single transducer. If the transducers that are generating the inputs do not depend on one another, then they can operate in parallel. A simple

example of control parallelism arises in an expression tree; the inputs to an operator can be evaluated concurrently. Finally, *merge parallelism* is available when the task of generating a stream can be split among multiple independent transducers. This is applicable to streams in which the order of elements is immaterial (i.e., sets).

2.2 Related Work

A handful of languages and architectures have been designed specifically for stream processing. Following is an outline of a few of these systems, with comparison to the approach taken for SIFt.

The SVP data model is a formal framework for collections, which are a generalization of streams and sets [16]. Mappings on SVP collections are specified as recursive functional equations, and an important subset of mappings called SVP-transducers are identified. Each SVP-transducer can be written as a restructuring of the input, mapping of input elements, and recombination of the results. SVP-transducers can be combined to form transducer networks. While SVP is general enough to handle arbitrary collections, SIFt focuses exclusively on streams.

Early work on networks of communicating processes by Gilles Kahn [10] has greatly influenced SIFt. He presents and analyzes the semantics of a simple language for parallel programming. This language, like SIFt, consists of a set of channel and process declarations. Channels are used to communicate values of a single type between processes, and behave like first-in-first-out (FIFO) queues. The main program composes these processes to define the structure of the network. Kahn concludes that the major benefit of this style of language is the vast simplification in analyzing the state of the system. Further work by Kahn and MacQueen draws parallels between such networks of parallel processes and coroutines.

Occam is a language of communicating processes that provides channel communication primitives and both sequential and parallel composition of processes [9]. Further, it offers guarded alternatives, which provide a mechanism for waiting for input from any one of a set of channels. This capability is not present in the current SIFt design.

The π-calculus is to concurrent programming as the λ-calculus is to sequential programming. Once again, computation in the π-calculus contains processes, the active components of the system, and channels, the communication mechanism [17]. In contrast to Kahn's

language described above, channels act as synchronous rendezvous points rather then FIFO queues. This means that neither the sending or receiving process can progress until both have reached the communication point. Pict is an example of a language based directly on the π-calculus.

The ASPEN distributed stream processing environment allows programmers to specify the desired degree of concurrency to be exploited during execution [14]. ASPEN extends the rewrite-rule language Log(F) with annotations that specify when it would be cost-effective to reduce a term in parallel with the rest of the program. A major benefit of this approach is that programs need not be restructured when re-targeted to a different distributed configuration. Rather, only simple annotations need to be adjusted. Although the current SIFt infrastructure requires programs to be hand-structured for parallel execution, future work will obviate the need by extracting the appropriate level of stream parallelism automatically.

Modern general-purpose microprocessors are not prepared to exploit the parallelism available in streaming applications. The major reason for this is the manner in which these systems handle data. The typical memory hierarchy, including the register file, multiple levels of cache, main memory, and disk storage, is optimized for access patterns exhibiting both temporal and spatial locality. Since streaming applications usually access data values in order, they do exhibit a large degree of spatial locality. However, temporal locality is almost completely lacking. Once a data value has passed through a stream processing stage, it will probably never be touched by that stage again. Therefore, one of the major benefits of a memory hierarchy does not apply to stream computation. The processor architectures described below solve this problem by including a communication network that tightly integrates the functional units. A data value is passed rapidly through this network rather than through the memory system, and is kept close to the computing elements only for as long as it is needed.

The Cheops video processing system makes heavy use of specialized hardware for computationally intensive stream operations [4]. The programming model for Cheops is essentially an interface to the runtime scheduler, and requires the user to be aware of what specialized hardware is available in the target configuration. Successively simpler interfaces have been created, culminating in a language for describing the program's data-flow graph.

The Imagine architecture for media processing is also organized around the concept of streams, but does not make use of highly specialized functional units [18]. The programming

model for Imagine consists of a series of stream transducers written in a C-like syntax whose operations are controlled by a top-level C++ program running on the host processor. The transducers themselves are similar to the processes in SIFt, but SIFt allows transducers and control to be expressed in the same framework.

PipeRench is a pipelined reconfigurable architecture for streaming multimedia acceleration [7]. Programs are written in a data-flow intermediate language called DIL, which is mapped to the virtual pipeline of the hardware. PipeRench benefits from exploiting the pipeline parallelism available in streaming applications and customizing the hardware for frequently-occurring operations.

The initial target of the SIFt compiler is the Raw architecture [25]. Raw is a scalable microprocessor architecture consisting of a set of simple compute tiles connected in a two-dimensional mesh. See Figure 2-3 for an illustration. Each tile contains a local register file, a portion of on-chip memory, a unified integer and floating point ALU, and an independent program counter. Communication over the interconnection network can be statically orchestrated by the compiler by programming the static switches in each tile. This allows low-latency transfers of single-word data values between tiles. Since data can flow rapidly from tile to tile, this architecture is a well-suited substrate for streaming applications. Although the general-purpose RawCC compiler can parallelize sequential C and FORTRAN programs, it does not realize all of the potential concurrency in streaming applications [12]. Developing a compiler that will obtain the dramatic speedups possible for streaming applications is one of the major challenges facing the Raw project [2].

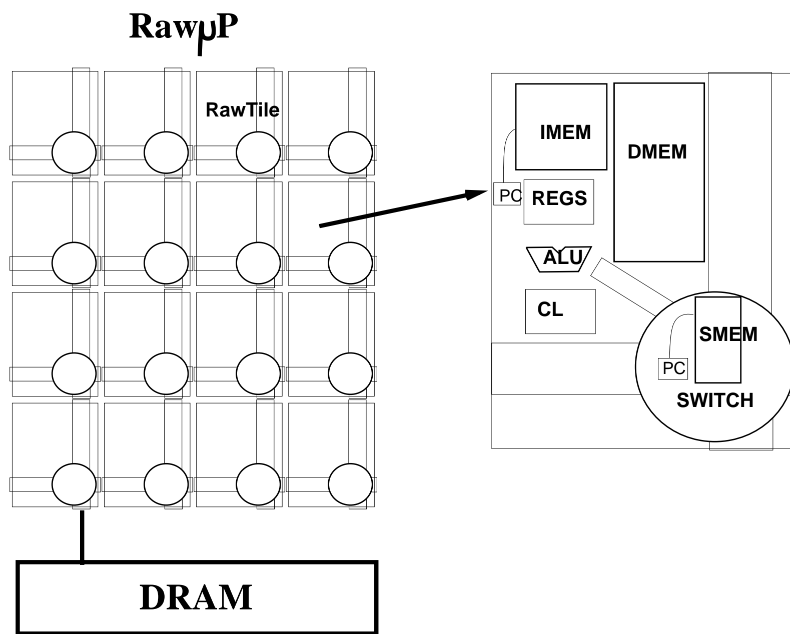

Figure 2-3: The Raw microprocessor.

Chapter 3

Intermediate Format

This chapter describes the structure and semantics of the Stream Intermediate Format (SIFt). As mentioned earlier, SIFt is an intermediate format for the class of streaming applications that can be described as a fixed set of processes and communication channels. SIFt has also been designed to map easily onto parallel architectures that support statically-compiled communication, such as the Raw processor.

The style of presentation is strongly influenced by the description of FL by Turbak, Gifford, and Reistad [24]. Section 3.1 presents an s-expression grammar for SIFt and described the overall structure of the format. Section 3.2 describes the semantics of SIFt programs. Section 3.3 formalizes this description with an operational semantics. Finally, Section 3.4 presents an example SIFt program that implements a four-tap finite impulse response (FIR) filter.

3.1 Structure

A well-formed SIFt program is a member of the syntactic domain Program defined by the s-expression grammar shown in Figure 3-1 and Figure 3-2. Such a program consists of a collection of global bindings, which specifies a set of processes and communication channels that exist at runtime. Each process contains an expression that specifies its behavior, and may communicate through the declared channels.

SIFt expressions are built from literals (booleans, integers, and floating point values), variable references, primitive operations, and conditionals. Local bindings, which associate the value of an expression with an identifier, are introduced with `let` expressions. Expres-

```
P ∈ Program
D ∈ Definition
F ∈ Definable
E ∈ Expression
I ∈ Identifier
S ∈ SimpleType = {int, float, bool}
T ∈ Type
L ∈ Literal = IntLit + FloatLit + BoolLit + UnitLit
      IntLit = {..., -2, -1, 0, 1, 2,...}
      FloatLit = ;; Floating point literals
      BoolLit = {#f, #t}
      UnitLit = {#u}
O ∈ PrimOp = ArithOp + RelOp + BitOp
      ArithOp = {+, -, *, /, %}
      RelOp = {=, !=, <, <=, >, >=}
      BitOp = {&, ~, |, ^, <<, >>}
```

Figure 3-1: Syntactic domains used to specify the structure of SIFt.

```
P ::= (program D*)

D ::= (define I F)              [Global Binding]

F ::= (channel S)               [Internal Channel]
    | (input N S)               [Input Channel]
    | (output N S)              [Output Channel]
    | (process E)               [Process]

E ::= L                         [Literal]
    | I                         [Variable Reference]
    | (primop O E*)             [Primitive Operation]
    | (if E E E)                [Conditional]
    | (let ((I E)*) E)          [Local Binding]
    | (begin E*)                [Sequencing]
    | (label I E)               [Control Point]
    | (goto I)                  [Backward Branch]
    | (send! I E)               [Channel Write]
    | (receive! I)              [Channel Read]

T ::= S                         [Simple]
    | unit                      [Unit]
    | (channel-of T)            [Channel]
```

Figure 3-2: An s-expression grammar of SIFt.

sions with side-effects can be sequenced using the `begin` construct. The `label` and `goto` forms provide backward branching to form loops.

The communication primitives `send!` and `receive!` are used to write to and read from channels. The `!` notation indicates that these expressions have side effects. Three types of channels exist in SIFt – internal, input, and output. Internal channels are used to communicate between processes in the same program, and are declared simply by specifying an element type. Input and output channels are used to communicate with the outside world, which could be external devices or other programs. Such device channels are declared by specifying both an element type and a unique port number.

The simple type system of SIFt consists of the base types `int`, `float`, `bool`, and `unit`, and type constructor `channel-of`. Note that only simple types can be communicated through channels. This restriction exists to simplify the implementation. The lack of functions in SIFt is another such simplifying restriction that should be conceptually painless to add in the future.

3.2 Informal Semantics

Every SIFt expression denotes a value belonging to one of the types described above. The primitive values supported by SIFt include the unit value, boolean truth values, integers, and floating point values. The unit value is the sole member of the singleton type `unit`. In addition, SIFt supports channels. A channel allows communication of values between a pair of processes. Channels and processes are *not* first-class values in SIFt. That is, they can only be defined statically and cannot be passed through channels.

The description of the informal semantics of SIFt expressions will be motivated by example. The notation $E \xrightarrow{\text{SIFt}} V$ indicates that the expression E evaluates to the value V. Values will be written as follows:

unit	Unit value
false, *true*	Boolean values
42, *-12*, *0*	Integer values
3.14159, *2.72*, *-1.0*	Floating point values
error	Errors
∞-*loop*	Non-termination

Literal expressions represent the corresponding constants:

$$\texttt{\#u} \xrightarrow{\text{SIFt}} \textit{unit}$$
$$\texttt{\#f} \xrightarrow{\text{SIFt}} \textit{false}$$
$$\texttt{23} \xrightarrow{\text{SIFt}} \textit{23}$$
$$\texttt{150.23} \xrightarrow{\text{SIFt}} \textit{150.23}$$

The behavior of the primitive operators are the same as in the C language. The expression (primop O $E_1 \ldots E_n$) evaluates to the result of applying O to the values of E_1 through E_n. For example,

$$\texttt{(primop - 10 7)} \xrightarrow{\text{SIFt}} \textit{3}$$
$$\texttt{(primop = 7.5 3.2)} \xrightarrow{\text{SIFt}} \textit{false}$$
$$\texttt{(primop \^{} 95)} \xrightarrow{\text{SIFt}} \textit{-96}$$

Primitive operators are overloaded in a natural manner. For instance, each bitwise operator can also be used as a logical operator on booleans, with the expected results.

Passing the wrong number or types of arguments to a primitive operator is an error. Individual operators have additional error cases, such as division or modulo by zero. Some sample error cases include,

$$\texttt{(primop + 10 4 -9)} \xrightarrow{\text{SIFt}} \textit{error} \quad ;; \textit{ Wrong number of arguments}$$
$$\texttt{(primop - \#f 5)} \xrightarrow{\text{SIFt}} \textit{error} \quad ;; \textit{ Wrong type of argument}$$
$$\texttt{(primop / 4 0)} \xrightarrow{\text{SIFt}} \textit{error} \quad ;; \textit{ Division by zero}$$

The conditional expression (if E_{test} E_{con} E_{alt}) evaluates to the value of E_{con} or E_{alt} based on the value of E_{test}. E_{test} must evaluate to a boolean; *true* selects the consequent E_{con} and *false* selects the alternative E_{alt}. Note that the unchosen expression is not evaluated.

$$\texttt{(if (primop > 5 10) 100 (primop + 1 2))} \xrightarrow{\text{SIFt}} \textit{3}$$
$$\texttt{(if (primop / 1.0 22.0) 0 1)} \xrightarrow{\text{SIFt}} \textit{error} \quad ;; \textit{ Test is not boolean}$$
$$\texttt{(if \#f (primop / 1 0) 0)} \xrightarrow{\text{SIFt}} \textit{0} \quad ;; \textit{ Non-strictness}$$

The local binding expression (let ((I_1 E_1) ... (I_n E_n)) E_{body}) introduces local variables. The body E_{body} is evaluated in an environment in which the values of E_1 through E_n are bound to identifiers I_1 through I_n, respectively. All of the bound expressions are evaluated before the body, regardless of whether the corresponding identifier is used. Identifiers

introduced by let-expressions can shadow those from higher scope, but must not conflict within the same scope.

```
(let ((x 5) (y (primop + 1 2)))
  (primop * x y))                     --SIFt--> 15

(let ((bottom (primop / 1 0)))
  5)                                  --SIFt--> error   ;; Strictness

(let ((popular 10) (popular #t))
  popular)                            --SIFt--> error   ;; Name collision
```

The sequencing expression (`begin` $E_1 \ldots E_n$) evaluates its operands in order, ultimately taking on the value of E_n. In the following example, the sequencing semantics guarantees that `1` will be sent before `x`.

```
(begin (send!  c1 1) (send!  c1 x) #t) --SIFt--> true
```

Values are communicated over channels using `send!` and `receive!` expressions. The expression (`send!` I E) writes the value of E onto the channel denoted by I and evaluates to *unit*. The expression (`receive!` I) consumes the next value from the channel denoted by I and evaluates to that value. Channel contents are buffered, meaning that a send is allowed to proceed before a receive consumes the sent value.

Each channel is meant to communicate values between just two endpoints. Therefore, the following additional restrictions are imposed on SIFt programs:

- Each input channel must have zero writers and a single reader,
- each output channel must have a single writer and zero readers, and
- each internal channel must have a single writer and a single reader,

where "writer" is a process that performs `send!` operations on a channel, and "reader" is a process that performs `receive!` operations on a channel.

The meaning of a complete SIFt program is the observable behavior that evolves as the body of each process is evaluated on its own virtual processor. In particular, the observable behavior is the series of values produced on each of the output channels along with the number of values consumed on each of the input channels.

3.3 Formal Semantics

This section presents a formal semantics for SIFt programs using the theory of structured operational semantics (SOS). An SOS consists of five parts: the set of configurations $\mathcal{C}$, the transition relation $\Rightarrow$, the set of final configurations $\mathcal{F}$, an input function $\mathcal{I}$, and an output function $\mathcal{O}$. A configuration encapsulates the state of an abstract machine, and the transition relation specifies legal transitions between those states. A final configuration is one for which the program has terminated. The input function specifies how an input program is converted to an initial configuration; the output function specifies how a final configuration is converted into a final result.

Figure 3-3 shows the auxiliary semantic domains used in the SOS for SIFt. The semantic domain Value coincides with the syntactic domain Literal, and a ValueList is a sequence of such objects. The ChannelConfig domain contains functions that map channel names to the contents of that channel's FIFO queue. The ProcessConfig domain contains sets of expressions. In a configuration, each expression in the set corresponds to the state of a process.

Before continuing, an explanation of the notation used for sets, lists, and functions is in order. Sets are represented in the traditional fashion. This can be a sequence of comma-separated values enclosed in curly braces (e.g. $\{2, 4, 6\}$), set builder notation that generates the elements according to some predicate (e.g. $\{n \in \mathcal{N} | n < 7 \wedge 2 \| n\}$), or the union of two sets (e.g. $\{2, 4\} \cup \{6\}$). Lists are represented as comma-separated values enclosed in square brackets or as the catenation of two lists. For instance, $[1, 2, 3]$ is the three-element list of the first three natural numbers. Two lists can be catenated using the infix @ operator, as in $[1, 2]@[3]$. Finally, functions are represented using both lambda notation and graphs. Lambda notation shows how the result of a function can be computed based on its arguments. In that way, the increment function can be expressed as $\lambda x.(x + 1)$. The graph of a function f is the set of input-output pairs, that is $\{\langle x, y\rangle | (f x) = y\}$.

With these helper domains and notational conventions in hand, we can now define $\mathcal{C}$, $\mathcal{F}$, $\mathcal{I}$, and $\mathcal{O}$. A configuration is simply a channel configuration paired with a process configuration.

$$\mathcal{C} = \text{ChannelConfig} \times \text{ProcessConfig}$$

The set of final configurations consists of all configurations which cannot transition to any

v	$\in$	Value = Literal
vl	$\in$	ValueList = Value*
cc	$\in$	ChannelConfig = Identifier $\rightarrow$ ValueList
pc	$\in$	ProcessConfig = $\mathcal{P}$(Expression)

Figure 3-3: Semantic domains used in the operational semantics of SIFt.

other configuration. It is therefore defined in terms of $\Rightarrow$.

$$\mathcal{F} = \{c \in \mathcal{C} | \neg\exists c' \in \mathcal{C}, c \Rightarrow c'\}$$

For each channel declared in the program, the input function places an empty entry in the channel configuration. For each process declared in the program, the input function places the process's body in the process configuration. The helper functions $\mathcal{I}_c$ and $\mathcal{I}_p$ handle channels and processes, respectively. The channel configuration passed to $\mathcal{I}$ and $\mathcal{I}_c$ specify the initial contents of the input channels.

$$
\begin{array}{l}
\mathcal{I} : \text{Program} \rightarrow \text{ChannelConfig} \rightarrow \mathcal{C} = \lambda\ \texttt{(program}\ D^*\texttt{)}\ cc.\ \langle(\mathcal{I}_c\ D^*\ cc), (\mathcal{I}_p\ D^*)\rangle \\
\mathcal{I}_c : \text{Definable}^* \rightarrow \text{ChannelConfig} \rightarrow \text{ChannelConfig} = \\
\quad \lambda\ l\ cc.\ \ \textbf{if}\ \ (\textit{empty?}\ l) \\
\qquad\qquad \textbf{then}\ \ \emptyset \\
\qquad\qquad \textbf{else}\ \ \textbf{matching}\ (\textit{head}\ l) \\
\qquad\qquad\qquad \triangleright\ \texttt{(define}\ I\ \texttt{(channel}\ S\texttt{))} \quad \|\ (\mathcal{I}_c\ (\textit{tail}\ l)) \cup \{\langle I, []\rangle\} \\
\qquad\qquad\qquad \triangleright\ \texttt{(define}\ I\ \texttt{(input}\ N\ S\texttt{))} \quad \|\ (\mathcal{I}_c\ (\textit{tail}\ l)) \cup \{\langle I, (cc\ I)\rangle\} \\
\qquad\qquad\qquad \triangleright\ \texttt{(define}\ I\ \texttt{(output}\ N\ S\texttt{))} \quad \|\ (\mathcal{I}_c\ (\textit{tail}\ l)) \cup \{\langle I, []\rangle\} \\
\qquad\qquad\qquad \triangleright\ \textbf{else} \quad \|\ (\mathcal{I}_c\ (\textit{tail}\ l)) \\
\qquad\qquad\qquad \textbf{endmatching} \\
\mathcal{I}_p : \text{Definable}^* \rightarrow \text{ProcessConfig} = \\
\quad \lambda\ l.\ \ \textbf{if}\ \ (\textit{empty?}\ l) \\
\qquad\quad \textbf{then}\ \ \emptyset \\
\qquad\quad \textbf{else}\ \ \textbf{matching}\ (\textit{head}\ l) \\
\qquad\qquad\quad \triangleright\ \texttt{(define}\ I\ \texttt{(process}\ E\texttt{))} \quad \|\ (\mathcal{I}_p\ (\textit{tail}\ l)) \cup \{E\} \\
\qquad\qquad\quad \triangleright\ \textbf{else} \quad \|\ (\mathcal{I}_p\ (\textit{tail}\ l)) \\
\qquad\qquad\quad \textbf{endmatching}
\end{array}
$$

Finally, the output function extracts the visible state from a final configuration, which includes the contents of all of the channels.

$$\mathcal{O} : \mathcal{F} \rightarrow \text{ChannelConfig} = \lambda\langle cc, pc\rangle.cc$$

The real heart of the semantics is the rewrite rules that define $\Rightarrow$, which in turn define how the computation evolves. Each rewrite rule has an antecedent and a consequent, written as follows:

$$\mathcal{D}[\![\texttt{(let () } E_0\texttt{)}]\!] = \mathcal{D}[\![E_0]\!]$$

$$\begin{aligned}&\mathcal{D}[\![\texttt{(let ((}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}\ E_0\texttt{)}]\!] =\\ &\quad \texttt{(let ((}I_1\ \mathcal{D}[\![E_1]\!]\texttt{))}\ \mathcal{D}[\![\texttt{(let ((}I_2\ E_2\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}\ E_0\texttt{)}]\!]\texttt{)}\end{aligned}$$

$$\mathcal{D}[\![\texttt{(begin)}]\!] = \texttt{\#u}$$

$$\mathcal{D}[\![\texttt{(begin } E\texttt{)}]\!] = \mathcal{D}[\![E]\!]$$

$$\begin{aligned}&\mathcal{D}[\![\texttt{(begin } E_1\ E_2\ \ldots\ E_n\texttt{)}]\!] =\\ &\quad \texttt{(begin } \mathcal{D}[\![E_1]\!]\ \mathcal{D}[\![\texttt{(begin } E_2\ \ldots\ E_n\texttt{)}]\!]\texttt{)} \qquad \forall n > 2\end{aligned}$$

Figure 3-4: Desugaring of degenerate and complex SIFt forms.

$$\frac{\text{It is snowing in April.}}{\text{You aren't in San Diego.}}$$

This means "If it is snowing in April, then you aren't in San Diego". In other words, if the antecedent is true, then the consequent follows. Rules without an antecedent are unconditionally true.

In order to simplify the rewrite rules, we present them in three stages. The first stage consists of a desugaring $\mathcal{D}$ that converts certain SIFt forms into semantically equivalent but simpler forms. The second stage presents a simplified transition relation $\Rightarrow_s$ for expressions that do not involve communication. This obviates the need to show the entire configuration in each rule. The third stage defines the full-blown transition relation $\Rightarrow$ in terms of $\mathcal{D}$, $\Rightarrow_s$, and its own individual rewrite rules that may involve communication.

Figure 3-4 presents the special cases of the desugaring function. A let expressions with no bindings is simply replaced by its body. A let expression with more than one binding is recursively converted into a series of nested single-binding let expressions. An empty begin expression is replaced by the unit value, and a begin expression containing just one expression is replaced with that expression. Finally, a begin expression with more than two subexpressions is recursively converted into a series of nested two-way begin expressions. Figure 3-5 completes the desugaring function. It shows that all other forms are handled by leaving the top-level form intact and desugaring the subexpressions.

Figure 3-6 enumerates the rewrite rules that define $\Rightarrow_s$. The *desugar* rule carries over the desugaring function from above into the transition relation. The *if-true* and *if-false*

$\mathcal{D}[\![L]\!] = L$

$\mathcal{D}[\![I]\!] = I$

$\mathcal{D}[\![\texttt{(primop } O\ E_1\ \ldots\ E_n\texttt{)}]\!] = \texttt{(primop } O\ \mathcal{D}[\![E_1]\!]\ \ldots\ \mathcal{D}[\![E_n]\!]\texttt{)}$

$\mathcal{D}[\![\texttt{(if } E_{test}\ E_{con}\ E_{alt}\texttt{)}]\!] = \texttt{(if } \mathcal{D}[\![E_{test}]\!]\ \mathcal{D}[\![E_{con}]\!]\ \mathcal{D}[\![E_{alt}]\!]\texttt{)}$

$\mathcal{D}[\![\texttt{(let ((}I_1\ E_1\texttt{))}\ E_0\texttt{)}]\!] = \texttt{(let ((}I_1\ \mathcal{D}[\![E_1]\!]\texttt{))}\ \mathcal{D}[\![E_0]\!]\texttt{)}$

$\mathcal{D}[\![\texttt{(begin } E_1\ E_2\texttt{)}]\!] = \texttt{(begin } \mathcal{D}[\![E_1]\!]\ \mathcal{D}[\![E_2]\!]\texttt{)}$

$\mathcal{D}[\![\texttt{(label } I\ E\texttt{)}]\!] = \texttt{(label } I\ \mathcal{D}[\![E]\!]\texttt{)}$

$\mathcal{D}[\![\texttt{(goto } I\texttt{)}]\!] = \texttt{(goto } I\texttt{)}$

$\mathcal{D}[\![\texttt{(send! } I\ E\texttt{)}]\!] = \texttt{(send! } I\ \mathcal{D}[\![E]\!]\texttt{)}$

$\mathcal{D}[\![\texttt{(receive! } I\texttt{)}]\!] = \texttt{(receive! } I\texttt{)}$

Figure 3-5: Desugaring of simple SIFt forms.

rules specify what happens after the predicate of a conditional has been evaluated. In the *primop* rule, $\underline{O}$ stands for the operation denoted by O, which will not be specified in further detail. The *let-done* rule shows that a locally-bound variable is replaced with its computed value in the body of the let expression. This rule uses the substitution operator $[x/y]$, which will be described in more detail below. The *begin-done* rule simply discards the first expression after it has been evaluated down to a value, and proceeds to the next one. The *label* rule shows the expression itself being substituted for every goto expression that targets the particular label. Keep in mind that these rewrite rules specify semantics, but not necessarily implementation.

The substitution operator requires additional explanation. The result of the substitution $[x/y]E$ is E with all free occurrences of y replaced by x. The free occurrences of y in E are the occurrences not bound by a let expression or label expression that also appears in E. Figure 3-7 formally defines substitution. The figure shows only the base-case clauses and clauses necessary to avoid free identifier capture. For illustration,

$$[6/\texttt{x}]\texttt{(primop + x (let ((x 2)) x))} = \texttt{(primop + 6 (let ((x 2)) x))}$$

Note that the second reference to `x` has not been replaced, because it is not free.

$$E \Rightarrow_s \mathcal{D}[\![E]\!] \qquad [\textit{desugar}]$$

$$\texttt{(primop } O \ v_1 \ \ldots \ v_n\texttt{)} \Rightarrow_s \underline{O}(v_1, ..., v_n) \qquad [\textit{primop-done}]$$

$$\texttt{(if \#t } E_{con} \ E_{alt}\texttt{)} \Rightarrow_s E_{con} \qquad [\textit{if-true}]$$

$$\texttt{(if \#f } E_{con} \ E_{alt}\texttt{)} \Rightarrow_s E_{alt} \qquad [\textit{if-false}]$$

$$\texttt{(let ((}I_1 \ v_1\texttt{)) } E_0\texttt{)} \Rightarrow_s [v_1/I_1]E_0 \qquad [\textit{let-done}]$$

$$\texttt{(begin } v \ E\texttt{)} \Rightarrow_s E \qquad [\textit{begin-done}]$$

$$\texttt{(label } I \ E\texttt{)} \Rightarrow_s [\texttt{(label } I \ E\texttt{)}/\texttt{(goto } I\texttt{)}]E \qquad [\textit{label}]$$

Figure 3-6: Rewrite rules for simplified transition relation $\Rightarrow_s$.

$$[E/I]I = E$$

$$[E/I]\texttt{(let ((}I \ E_1\texttt{)) } E_0\texttt{)} = \texttt{(let ((}I \ [E/I]E_1\texttt{)) } E_0\texttt{)}$$

$$[E/I]\texttt{(let ((}I_1 \ E_1\texttt{)) } E_0\texttt{)} = \texttt{(let ((}I_1 \ [E/I]E_1\texttt{)) } [E/I]E_0\texttt{)}, \text{ if } I \neq I_1$$

$$[E/\texttt{(goto } I\texttt{)}]\texttt{(goto } I\texttt{)} = E$$

$$[E/\texttt{(goto } I\texttt{)}]\texttt{(label } I \ E_0\texttt{)} = \texttt{(label } I \ E_0\texttt{)}$$

$$[E/\texttt{(goto } I\texttt{)}]\texttt{(label } I_0 \ E_0\texttt{)} = \texttt{(label } I_0 \ [E/\texttt{(goto } I\texttt{)}]E_0\texttt{)}, \text{ if } I \neq I_0$$

Figure 3-7: Formal definition of substitution.

$$\frac{E \Rightarrow_s E'}{\langle cc, \{E\} \cup pc\rangle \Rightarrow \langle cc, \{E'\} \cup pc\rangle} \quad [\mathit{simple}]$$

$$\frac{\langle cc, \{E_k\} \cup pc\rangle \Rightarrow \langle cc', \{E'_k\} \cup pc\rangle}{\begin{array}{c}\langle cc, \{\texttt{(primop } O\ v_1\ \ldots v_{k-1}\ E_k\ \ldots E_n\texttt{)}\} \cup pc\rangle \Rightarrow \\ \langle cc', \{\texttt{(primop } O\ v_1\ \ldots v_{k-1}\ E'_k\ \ldots E_n\texttt{)}\} \cup pc\rangle \\ \forall\, n > 0, 0 < k \leq n\end{array}} \quad [\mathit{primop\text{-}progress}]$$

$$\frac{\langle cc, \{E_{test}\} \cup pc\rangle \Rightarrow \langle cc', \{E'_{test}\} \cup pc\rangle}{\begin{array}{c}\langle cc, \{\texttt{(if } E_{test}\ E_{con}\ E_{alt}\texttt{)}\} \cup pc\rangle \Rightarrow \\ \langle cc', \{\texttt{(if } E'_{test}\ E_{con}\ E_{alt}\texttt{)}\} \cup pc\rangle\end{array}} \quad [\mathit{if\text{-}progress}]$$

$$\frac{\langle cc, \{E_1\} \cup pc\rangle \Rightarrow \langle cc', \{E'_1\} \cup pc\rangle}{\begin{array}{c}\langle cc, \{\texttt{(let ((}I_1\ E_1\texttt{))}\ E_0\texttt{)}\} \cup pc\rangle \Rightarrow \\ \langle cc', \{\texttt{(let ((}I_1\ E'_1\texttt{))}\ E_0\texttt{)}\} \cup pc\rangle\end{array}} \quad [\mathit{let\text{-}progress}]$$

$$\frac{\langle cc, \{E_1\} \cup pc\rangle \Rightarrow \langle cc', \{E'_1\} \cup pc\rangle}{\langle cc, \{\texttt{(begin } E_1\ E_2\texttt{)}\} \cup pc\rangle \Rightarrow \langle cc', \{\texttt{(begin } E'_1\ E_2\texttt{)}\} \cup pc\rangle} \quad [\mathit{begin\text{-}progress}]$$

$$\frac{\langle cc, \{E\} \cup pc\rangle \Rightarrow \langle cc', \{E'\} \cup pc\rangle}{\langle cc, \{\texttt{(send! } I\ E\texttt{)}\} \cup pc\rangle \Rightarrow \langle cc', \{\texttt{(send! } I\ E'\texttt{)}\} \cup pc\rangle} \quad [\mathit{send\text{-}progress}]$$

$$\begin{array}{l}\langle cc \cup \{\langle I, vl\rangle\}, \{\texttt{(send! } I\ v\texttt{)}\} \cup pc\rangle \Rightarrow \\ \quad \langle cc \cup \{\langle I, (vl@[v])\rangle\}, \{\texttt{\#u}\} \cup pc\rangle\end{array} \quad [\mathit{send\text{-}done}]$$

$$\begin{array}{l}\langle cc \cup \{\langle I, ([v]@vl)\rangle\}, \{\texttt{(receive! } I\texttt{)}\} \cup pc\rangle \Rightarrow \\ \quad \langle cc \cup \{\langle I, vl\rangle\}, \{v\} \cup pc\rangle\end{array} \quad [\mathit{receive}]$$

Figure 3-8: Rewrite rules for general transition relation $\Rightarrow$.

Finally, Figure 3-8 shows the rewrite rules for $\Rightarrow$. The *simple* rule ensures that $\Rightarrow_s$ is incorporated into $\Rightarrow$. Each of the transitions implied by this rule leave the channel context unchanged, because none of them involve communication. The progress rules *send-progress*, *primop-progress*, *if-progress*, *let-progress*, and *begin-progress* show how subexpressions are allowed to proceed and engage in communication. The rules *send-done* and *receive* are the only ones that directly manipulate the channel context. According to *send-done*, a send expression can add its argument to the tail of the channel's FIFO queue after that argument has been fully evaluated. According to *receive*, a receive expression can proceed as soon as the channel's FIFO queue has at least one element, at which point it removes and evaluates to the head element.

3.4 Example Program

This section presents an example SIFt program that implements a four-tap FIR filter. An FIR filter processes a discrete-time input signal $x[\cdot]$ according to the following general form:

$$y[n] = \sum_{m=0}^{N-1} h[m] \cdot x[n-m]$$

The output $y[\cdot]$ is a weighted sum of a finite number N of current and past inputs. The coefficients $h[\cdot]$, also called *taps*, can be chosen to implement a variety of different filters [20]. Software implementations of such filters find application in software radios, which perform most of their signal processing in software.

The SIFt program in Figure 3-10 implements the four-tap FIR filter with coefficients $h[\cdot] = [2, 3, 4, 5]$. The program begins with the definition of the input and output channels as well as various internal communication channels. Figure 3-9 shows the overall structure of the program, with the nodes representing processes and edges representing communication channels. Processes `p1`, `p2`, `p3`, and `p4` form a pipeline which the input values are routed down. They are responsible for scaling the input by the appropriate coefficients. Processes `p5`, `p6`, and `p7` form an addition tree. They are responsible for summing the scaled values and outputting the final result.

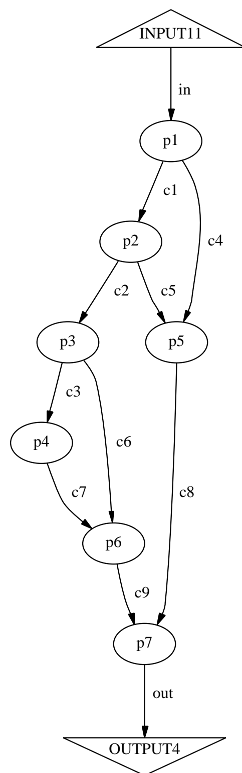

Figure 3-9: Structure of the FIR filter program.

```
(program
 (define in (input 11 int))
 (define c1 (channel int)) (define c2 (channel int)) (define c3 (channel int))
 (define c4 (channel int)) (define c5 (channel int)) (define c6 (channel int))
 (define c7 (channel int)) (define c8 (channel int)) (define c9 (channel int))
 (define out (output 4 int))
 (define p1 (process (label loop
                      (let ((v (receive! in)))
                        (begin (send! c1 v) (send! c4 (primop * 2 v))
                               (goto loop))))))
 (define p2 (process (begin (send! c5 0)
                            (label loop
                             (let ((v (receive! c1)))
                               (begin (send! c2 v) (send! c5 (primop * 3 v))
                                      (goto loop)))))))
 (define p3 (process (begin (send! c6 0) (send! c6 0)
                            (label loop
                             (let ((v (receive! c2)))
                               (begin (send! c3 v) (send! c6 (primop * 4 v))
                                      (goto loop)))))))
 (define p4 (process (begin (send! c7 0) (send! c7 0) (send! c7 0)
                            (label loop
                             (let ((v (receive! c3)))
                               (begin (send! c7 (primop * 5 v))
                                      (goto loop)))))))
 (define p5 (process (label loop (let ((v1 (receive! c4))
                                       (v2 (receive! c5)))
                                   (begin (send! c8 (primop + v1 v2))
                                          (goto loop))))))
 (define p6 (process (label loop (let ((v1 (receive! c6))
                                       (v2 (receive! c7)))
                                   (begin (send! c9 (primop + v1 v2))
                                          (goto loop))))))
 (define p7 (process (label loop (let ((v1 (receive! c8))
                                       (v2 (receive! c9)))
                                   (begin (send! out (primop + v1 v2))
                                          (goto loop)))))))
```

Figure 3-10: SIFt program that implements the FIR filter $[2, 3, 4, 5]$.

Chapter 4

Compilation

This chapter describes the process of compiling a SIFt program to our target architecture, the Raw microprocessor. Section 4.1 offers an overview of compilation, briefly describing each phase and the interfaces between phases. Section 4.2 describes the early analysis performed by the compiler. Section 4.3 details how the compiler chooses on which tile to place each process. Section 4.4 shows how the compiler maps the virtual channels in the source program onto the physical interconnect. Section 4.5 describes the Raw code generation algorithm.

4.1 Overview

The entire compilation process is depicted in Figure 4-1. The compiler begins by reading in and parsing the input SIFt program, generating an abstract syntax tree. This tree is passed to the analysis phase, generating a communication graph that summarizes communication between processes. This graph and a user-specified machine configuration are fed into the placer, which generates an assignment of processes onto tiles. Given a placement, the communication scheduling phase splits the virtual channels into subsets whose elements can be active simultaneously. This information is finally used to emit code for each processor and each switch in the target configuration.

The SIFt compiler is implemented in 6,800 lines of Java code. This includes all of the passes up to and including code generation. The code generators output three-address code in the Stanford University Intermediate Format (SUIF) [26]. This "low SUIF" output is passed through the Raw MachSUIF back-end to generate a Raw binary. This is the

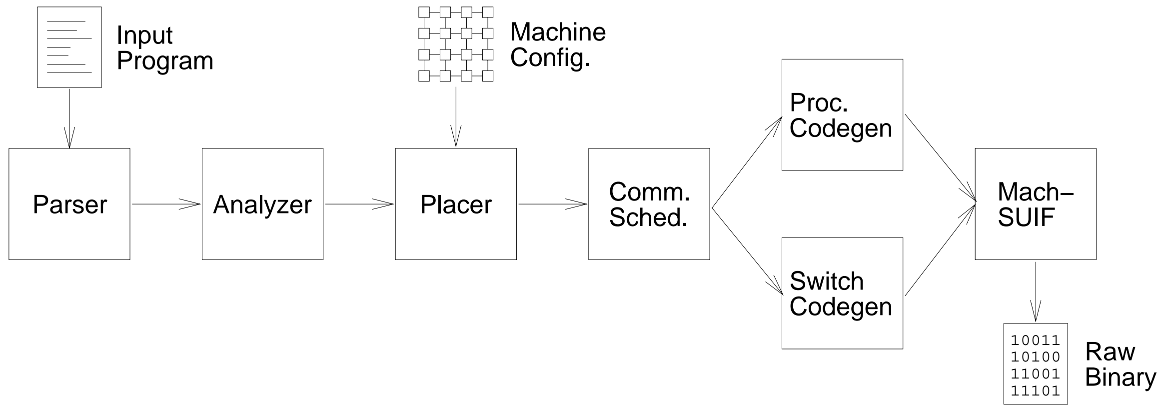

Figure 4-1: The main phases of the SIFt compiler.

same back-end used for RawCC, and performs several traditional compiler optimizations in addition to translation. MachSUIF is an extension to SUIF for machine-dependent optimizations [21].

4.2 Analysis

The analysis phase begins by performing a few static checks. The most notable of these is enforcement of the channel usage restrictions, which guarantee that each channel is used to communicate values in a single direction between just two endpoints. This restriction is does not exist in the formal semantics presented in Chapter 3, but greatly simplifies compilation.

The analysis phase also builds a communication graph $G_c = \langle N_c, E_c \rangle$. For each process in the program, N_c contains a corresponding node. For each channel in the program, E_c contains an edge from the node of the writer to the node of the reader. In order to handle input and output channels properly, each input or output device is also given a corresponding node in N_c. This graph summarizes which pairs of processes communicate with one another and in which direction. It says nothing about the order or rates of communication, and therefore should not be confused with a data-flow graph.

The communication graph for the FIR filter from Figure 3-10 is shown in Figure 4-2. This image was automatically generated by the compiler at the end of the analysis phase. Circles represent process nodes, triangles represent input nodes, and inverted triangles represent output nodes. The numbers at the end of "INPUT11" and "OUTPUT4" are the port numbers specified in the declarations of the input and output channels. How they are

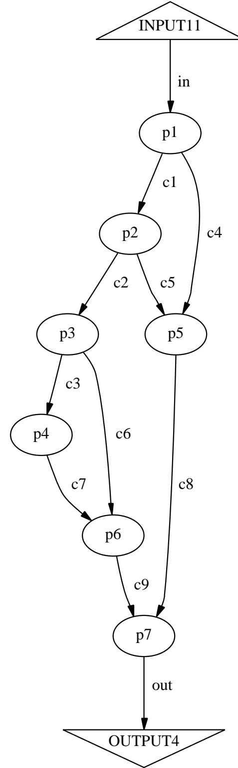

Figure 4-2: Communication graph for FIR filter.

translated to physical ports on the Raw machine will be discussed in Section 4.3.

4.3 Placement

The input to the placement phase is a target machine configuration and the communication graph that was generated by the analysis phase. The target configuration for Raw is specified as the number of rows and columns in the two-dimensional mesh. This mesh is modeled as a graph in which each tile is directly connected to its four nearest neighbors (except for boundary tiles). Each node in the communication graph must be assigned to a single processor in the target configuration. Further, each tile in the target configuration can be assigned at most one process. The responsibility of the placer is to find such an assignment that leads to the best performance. How to measure predicted performance is discussed below.

Placement algorithms have been studied extensively in the context of VLSI layout [11]. These algorithms fall into two categories, both of which optimize the placement based on a cost function. *Constructive initial placement* builds a solution from scratch, using the first

Physical Concept	Simulated Concept
energy function	cost function
particle states	parameter values
low-energy configuration	near-optimal solution
temperature	control parameter

Table 4.1: Relationship between physical and simulated annealing.

complete placement that it generates as the final result. An example is an algorithm that partitions the graph based on some criterion and recursively places the subgraphs onto a subset of the target topology. *Iterative improvement* begins with a complete placement, and repeatedly modifies it in order to minimize the cost function. For instance, RawCC uses a swap-based greedy algorithm for instruction placement [12]. Beginning with an arbitrary placement, the algorithm swaps pairs of mappings until it is no longer beneficial to do so. The SIFt compiler uses simulated annealing, which is a more sophisticated type of iterative improvement. Simulated annealing was chosen because of the flexibility it permits in choosing a cost function, and its widespread success in VLSI placement algorithms.

Simulated annealing is a stochastic computational technique for finding near-optimal solutions to large global optimization problems [5]. The physical annealing process attempts to put a physical system into a very low energy state. It accomplishes this by elevating and then gradually lowering the temperature of the system. By spending enough time at each temperature to achieve thermal equilibrium, the probability of reaching a very low energy state is increased. In simulating this process, each physical phenomenon has a simulated analogue, as shown in Table 4.1.

The following prose-code describes the placement algorithm in detail.

SimAnnealPlace(G_c, m) performs placement of communication graph G_c onto machine model m using simulated annealing. The communication graph was described above. The machine model encapsulates the details of the target architecture, which is used in generating a random initial placement and evaluating the cost function.

1. (Initial configuration.) Let C_{init} be a random placement of G_c on m. Set $C_{old} \leftarrow C_{init}$.

2. (Initial temperature.) Let T_0 be the initial temperature, determined by algorithm **SimAnnealMaxTemp**, described below. Set $T \leftarrow T_0$.

3. (Final temperature.) Let T_f be a final temperature, determined by algorithm **SimAnnealMinTemp**, described below.

4. (Perturb configuration.) Holding the temperature constant at T, the placement is repeatedly perturbed. Perturbations which decrease energy are unconditionally accepted. Those which increase energy are accepted probabilistically based on the Boltzmann distribution. The energy function $E(C)$ is shorthand for **PlacementCost**(C, m). The following steps are repeated 100 times.

   (a) Let C_{new} be C_{old} with the placements of a randomly-chosen pair of processes swapped.

   (b) If $E(C_{new})$ is less than $E(C_{old})$, then the replacement probability is $P = 1$. Otherwise, the replacement probability is $P = e^{\frac{E(C_{old})-E(C_{new})}{T}}$.

   (c) Randomly choose a number $0.0 \leq R \leq 1.0$.

   (d) If $R < P$, then accept the new configuration $C_{old} \leftarrow C_{new}$. Otherwise, keep the old configuration.

5. (Cool down.) Set $T \leftarrow \frac{9}{10} \cdot T$. If $T > T_f$, then go back to step 4.

6. (Return.) Return C_{old} as the final placement.

The initial and final temperatures are generated by the following algorithms.

SimAnnealMaxTemp$(C_{init},\ m)$ calculates an appropriate initial temperature for simulated annealing. It searches for a temperature high enough that the vast majority of perturbed configurations are accepted.

1. (Initial temperature.) Set $T \leftarrow 1.0$.

2. (Probe temperatures.) Repeat the following steps until the fraction of new configurations that are accepted in Step 2c is at least 90% or the steps have been repeated 100 times.

   (a) Set $T \leftarrow 2 \cdot T$.

   (b) Set $C_{old} \leftarrow C_{init}$.

(c) Perform Step 4 of **SimAnnealPlace**.

3. (Return.) Return T as the initial temperature.

SimAnnealMinTemp(C_{init}, m) calculates an appropriate final temperature for simulated annealing. It searches for a temperature low enough that the vast majority of perturbed configurations are rejected.

1. (Initial temperature.) Set $T \leftarrow 1.0$.

2. (Probe temperatures.) Repeat the following steps until the fraction of new configurations that are accepted in Step 2c is at most 1% or the steps have been repeated 100 times.

   (a) Set $T \leftarrow \frac{1}{2} \cdot T$.

   (b) Set $C_{old} \leftarrow C_{init}$.

   (c) Perform Step 4 of **SimAnnealPlace**.

3. (Return.) Return T as the final temperature.

All placements are constrained so that input and output device nodes are placed at their assigned locations. For each such node, this location is given by the port number specified in the declaration in the source program. For the Raw architecture, I/O devices are located on the periphery of the mesh, and the ports are numbered clockwise starting from the slot above the upper-left tile.

There are several constants embedded in the placer's simulated annealing algorithms. Some examples are the number of perturbations per temperature (100), the temperature decay rate ($\frac{9}{10}$), and the acceptance thresholds for choosing initial and final temperatures (90% and 1%, respectively). These values were chosen arbitrarily, and then adjusted based on the qualitative results of placement. Future research could investigate the impact of these constants on performance of the compiler and the generated code.

There are multifarious options for the cost function **PlacementCost**(C, m). Three of these alternatives were considered. The first option evaluates the average distance in number of hops between directly communicating processes. This is primarily a communication latency optimization. Low latency is important so that the lag between input and output does not become excessive. The second option calculates the number of channel pairs that

interfere when laid out dimension-ordered on the interconnection network. This is primarily a communication throughput optimization. High throughput is important because many streaming applications must handle input from high-bandwidth devices. The final option, and the one that the placement algorithm uses, is a combination of the first two. It begins by laying out the placement on the interconnection network using dimension-ordered routing. It then calculates the sum over all tiles of the square of the number of channels passing through it. This cost function increases with the distance between directly communicating processes *and* the predicted contention at the communication switches. This is the best of the three choices because it more precisely captures the level of interference between channels.

The correctness of this algorithm is quite simple to argue. Every step in **SimAnnealPlace** leaves a valid placement in C_{old}. This is because we begin with a valid initial placement, and swapping the location of two processes cannot invalidate a placement. Therefore, if the algorithm terminates, it does so with a valid placement. Since each of the loops is guaranteed to eventually exit (each is controlled by an induction variable that will reach its final value), the entire algorithm is guaranteed to terminate.

PlacementCost has worst-case running time of $O(|E_c|(r_m + c_m) + r_m c_m)$, where $|E_c|$ is the number of edges in G_c, and r_m and c_m are the dimensions of the machine model. Evaluation of the cost function dominates the loop body of Step 4 in **SimAnnealPlace**. This loop iterates a constant number of times, making its worst-case running time also $O(|E_c|(r_m + c_m) + r_m c_m)$. Examining Step 5 of **SimAnnealPlace**, the temperature decay causes Step 4 to be repeated $O(\log \frac{T_0}{T_f})$ times. The total worst-case running time of the placement algorithm is $O((|E_c|(r_m + c_m) + r_m c_m) \log \frac{T_0}{T_f})$. In practice, the placement phase is the most time-consuming portion of compilation. The worst-case space complexity is the space required to hold a single placement, $O(|N_c|)$, where $|N_c|$ is the number of nodes in G_c.

Figure 4-3 shows a random initial placement for the example FIR filter. This happens to be a fairly decent placement, with a cost of just 110, but it can be improved. Figure 4-4 shows the optimized final placement, which has a cost of 71. Among other things, this layout has fewer long channels and places both of the processes that use device channels right next to their respective devices. Figure 4-5 demonstrates the evolution of energy and temperature in a typical simulated annealing. Towards the beginning, the high temperature allows the

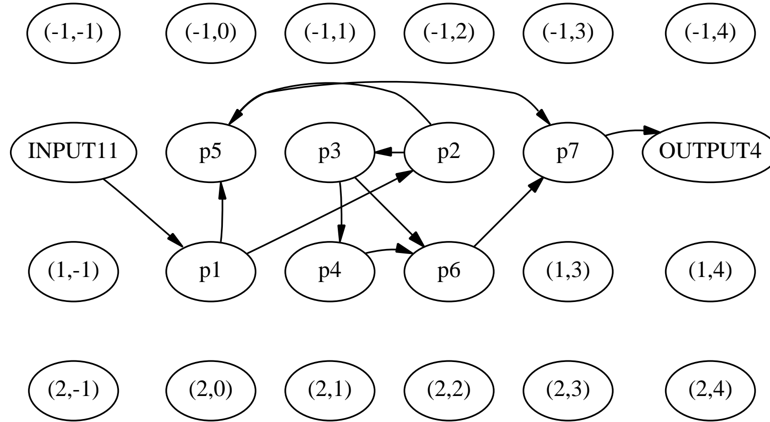

Figure 4-3: Random initial placement for FIR filter.

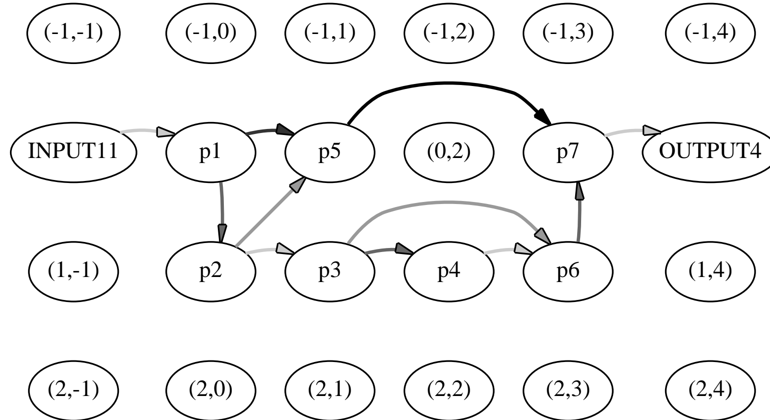

Figure 4-4: Optimized final placement for FIR filter.

configuration to jump around the space. As the temperature decreases, the configuration drifts towards a low energy state, where it finally freezes.

4.4 Communication Scheduling

The placement phase has placed the nodes of the communication graph onto the tiles of the target architecture. The next phase of the compiler, communication scheduling, must map the virtual channels to something that can be realized on the interconnection network. The input to this phase is the communication graph and placement. The set of all channels in the program must be partitioned such that there are no routing conflicts between channels in the same partition. A *routing conflict* exists among a set of channels if the routing algorithm cannot route all of them simultaneously. A partition that is free of routing conflicts is called a *communication context*. Therefore, the communication scheduling algorithm produces a

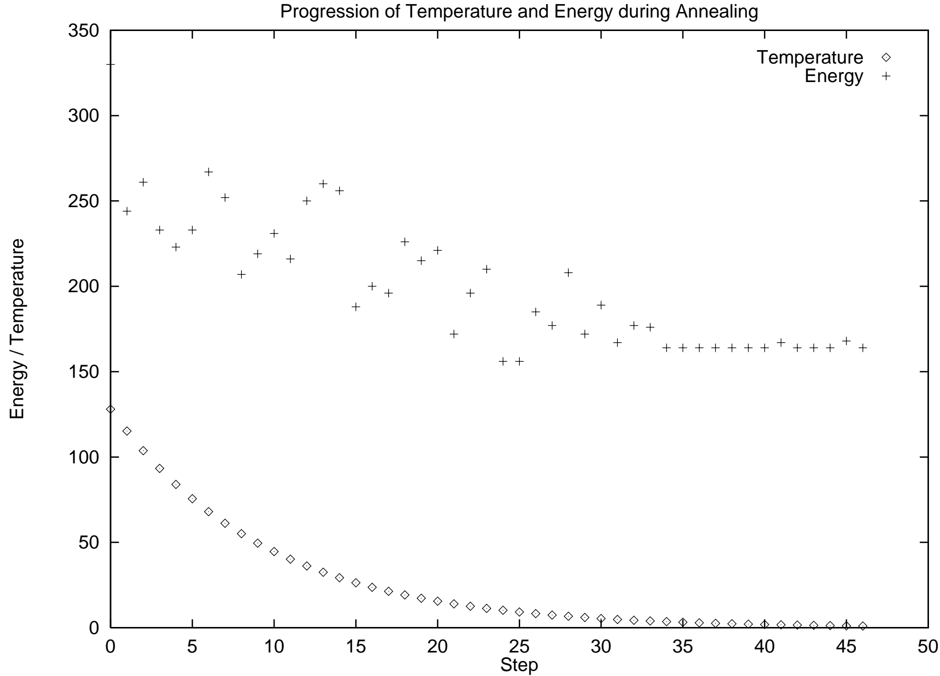

Figure 4-5: Progression of temperature and energy during annealing.

sequence of communication contexts. At runtime, the interconnection network can cycle through these communication contexts in order to support communication over the virtual channels.

The Raw architecture has two interconnection networks for communication between tiles [22]. The *dynamic network* supports communication of small packets of data between tiles whose identities are not necessarily known at compile time. It uses dimension-ordered wormhole routing with hardware flow-control. The *static network* supports communication of single-word values between tiles whose identities are known at compile time. Communication over this network is controlled by switch processors that are programmed by the compiler. Each switch has an instruction stream that tells it how to route values among its four neighbors and processor. Since streaming programs typically have regular communication patterns that can be analyzed at compile time, the SIFt compiler uses just the static network. On the static network, a set of channels exhibits a routing conflict when one of the switches is involved in routing more than one of the channels.

The Raw static network is a suitable target primarily because of the low communication overhead for single-word data transfers. It can take just one cycle for a processor to inject a word into or extract a word from the static network. The instruction set even allows an

ALU operation to be combined with a send or receive operation. In contrast, doing the same operation using the Raw dynamic network would take several more cycles for administrivia, including constructing a packet header and launching the contents of the network queue. Another benefit of the static network is the programmability of the switches, which admits flexibility in routing data between tiles. For instance, bandwidth-efficient multicasting could be implemented without additional hardware.

However, the static network also has its share of drawbacks. The most serious of these is that bandwidth must be statically reserved. In order to fully leverage the static network, a compiler must therefore have an intimate understanding of communication patterns.

Previous work has explored techniques for compiling to such statically orchestrated communication networks. The ConSet model used in iWarp's communication compiler splits an arbitrary set of connections between processors into phases [6]. Each phase contains a set of connections that can simultaneously be active given the hardware resources. Their algorithm uses sequential routing based on shortest path search, followed by repetitive ripping up and rerouting of connections. The *compiled communication* technique has been used in all-optical networks to minimize the overhead of dynamically establishing optical paths. Three algorithms described by Yuan, Melhem, and Gupta [27] for this purpose were considered for the SIFt compiler.

The greedy algorithm is the simplest of the three. It builds a communication context by adding channels until no more can be added without conflict. It then starts a new context, and repeats until all of the channels have been assigned. The coloring algorithm uses a graph-coloring heuristic to partition the channels. It builds an interference graph in which nodes correspond to channels and edges are placed between channels that cannot be routed simultaneously. After coloring the graph such that adjacent nodes have different colors, nodes of the same color are lumped into the same communication context. The ordered all-to-all personalized communication (AAPC) algorithm is an improvement on the greedy algorithm for dense communication patterns. Before performing the greedy algorithm, it sorts the channels in a manner that puts an upper bound on the number of necessary communication contexts. This guarantees that the number of contexts does not exceed what is required for a complete communication graph (i.e., one in which all pairs communicate).

The coloring algorithm is not suitable because it assumes that if all pairs of channels

in a set contain no conflicts, then the set itself contains no conflicts. This is true for dimension-ordered routing, but not true for the adaptive routing used in the SIFt compiler. The ordered AAPC algorithm is not suitable because streaming applications tend to have sparse communication graphs. Therefore, the communication scheduler uses the greedy algorithm, which is described by the following prose-code:

GreedyCommSched(G_c, p, m) partitions the set of edges E_c of communication graph G_c into communication contexts. Each such context has no conflicts when its constituent channels are routed on machine model m with placement p. The communication graph and machine model are described above. The placement is a function from the set of communication nodes N_c to locations on the machine model. S is a sequence of communication contexts representing the communication schedule. C is the set of channels yet to be assigned to a context.

1. (Initial schedule.) Set $S \leftarrow []$.
2. (Initial channels.) Set $C \leftarrow E_c$.
3. (Generate contexts.) A new communication context is created by adding channels until no more can be added without creating a routing conflict. X is the context being constructed, which is a set of channels. r is the summary of the routing of these channels, which is the set of switches that are occupied. The following steps are repeated while $C \neq \emptyset$.
    (a) (Initial context.) Set $X \leftarrow \emptyset$.
    (b) (Initial routing.) Set $r \leftarrow \emptyset$.
    (c) (Build context.) Insert as many channels into X as possible. The following steps are repeated for each $c \in C$.
        i. (Attempt routing.) Set $r' \leftarrow$ **RouteChannel**(r, c, m).
        ii. (Add channel.) If **RouteChannel** succeeded, then perform the following steps.
            A. Set $r \leftarrow r'$.
            B. Set $X \leftarrow X \cup \{c\}$.
            C. Set $C \leftarrow C - \{c\}$.

(d) (Extend schedule.) Set $S \leftarrow S@[X]$.

4. (Return.) Return S as the communication schedule.

This algorithm treats the contexts as if only one will be active on the network at a given time. However, there is no global synchronization barrier between the contexts at runtime, so different parts of multiple contexts actually will be active at the same time. For instance, consider two channels, one going from the upper-left corner of the chip to the lower-right corner of the chip, and the other going from the upper-right corner of the chip to the lower-left corner of the chip. These channels will be placed in separate contexts because they have an unavoidable conflict. However, the parts of the two channels that do not conflict can be simultaneously active, and only the switch at which they conflict will sequentialize them.

The routine **RouteChannel** is used as an approximation to the compiler's routing algorithm, which is presented as part of code generation in Section 4.5. Given the set of already-occupied switches, the channel to route, and a machine model, it attempts to route the given channel. If it succeeds, it also returns the new set of occupied switches. A switch is occupied with respect to a communication context if it is involved in routing some channel in that context.

The correctness argument must show that, assuming termination, each channel is assigned to exactly one communication context and there are no routing conflicts within a context. Assume that the algorithm does terminate. Each channel is placed into at most one context by virtue of being removed from C immediately after being added to X. Each channel is placed into at least one context because Step 3 does not terminate until C is empty. There can be no conflicts in any of the resulting contexts because of the routability check at Step 3(c)ii.

The termination argument must show that each of the loops runs for a finite number of iterations. The inner loop at Step 3c satisfies this because C is a finite set. The outer loop at Step 3 also satisfies this because C is finite and at least one element of C is removed for each iteration. This is guaranteed by the fact that any channel is routable in isolation, so that the routability test will succeed for at least $r = \emptyset$.

The outer loop is run for at most $|E_c|$ iterations. For each iteration of the outer loop, the inner loop is run for $|C|$ iterations. In the worst case, C is diminished by only one

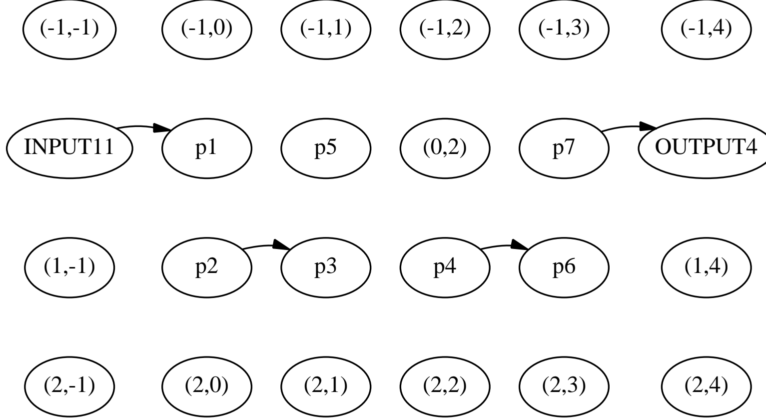

Figure 4-6: First communication context of FIR filter.

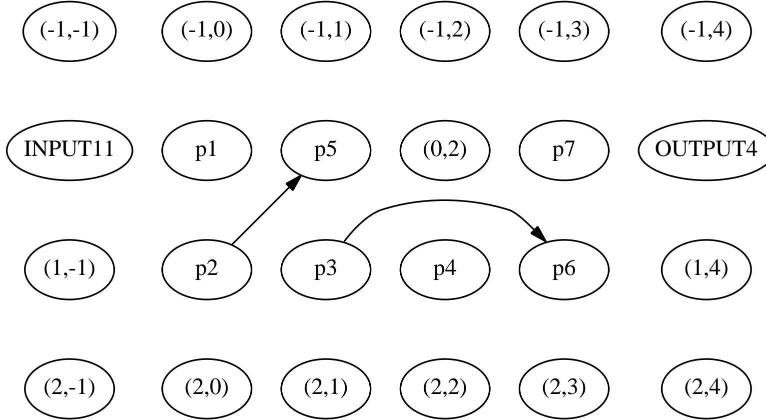

Figure 4-7: Second communication context of FIR filter.

element per iteration of the outer loop. In this case, the body of the inner loop is run $O(|E_c|^2)$ times. The worst-case running time of **RouteChannel** is $O(r_m + c_m)$. Since all other operations can be implemented in constant time, the total worst-case running time of **GreedyCommSched** is $O(|E_c|^2(r_m + c_m))$. In practice, communication scheduling takes much less time than placement. The worst-case space complexity is the space required to hold the temporary set of occupied switches and the final schedule, which is $O(r_m c_m + E_c)$.

Figures 4-6 and 4-7 show the results of communication scheduling for the example FIR filter. This example ends up having five contexts, but the figures show just the first two. For the complete picture, see Figure 4-4, in which channels of the same color belong to the same context. Note that channels within the same context have no routing conflicts.

4.5 Code Generation

The final task of the SIFt compiler is to generate code for each of the switches and processors. Standard techniques for converting expression trees into three-address code are well-understood, and will not be described here in detail. Rather, this section focuses on code generation for communication channels and communication primitives `send!` and `receive!`. The algorithms for generating switch code and generating processor code share the same input, but are otherwise independent. The common input is the communication graph, placement, and communication schedule.

At runtime, the switches must cycle through the contexts of the communication schedule. For each context, a switch involved in routing channel c must perform its part of the route. The Raw switch processor has a very basic instruction set consisting of moves and branches. In addition to this, each instruction has a routing component. This routing component specifies pairs of ports between which a single word should be transferred before the instruction completes. For instance, the switch instruction "`move $3, $cEi route $cNi -> $cSo`" stores the incoming value from the switch's eastern neighbor to a register and transfers the value from its northern neighbor to its southern neighbor.

Each channel is routed using an adaptive shortest-path routing algorithm. Starting at the source tile, the route is built one hop at a time, subject to three restrictions. First, the route cannot use a switch that is already involved in routing a channel within the same communication context. Second, each hop must bring the route closer to the destination tile. Third, when given a choice between moving horizontally and vertically, the horizontal hop is preferred. Only the first restriction is strictly necessary. The routing of the context shown in Figure 4-6 is trivial, since each channel requires just one hop. The non-trivial routing of Figure 4-7 is depicted in Figure 4-8.

The static network cycles through the communication contexts without regard to when a channel will actually be used. Therefore, there must be some mechanism for the processor to inform the network that a channel will not be used this time around. This is accomplished by pre-sending a boolean value over the channel's route indicating whether an actual value will be sent. The code in a switch first routes and examines this indicator, only routing the actual value if it was non-zero. Figure 4-9 shows the final switch code for tile (0,0) of the FIR filter. This switch is involved in routing three different channels.

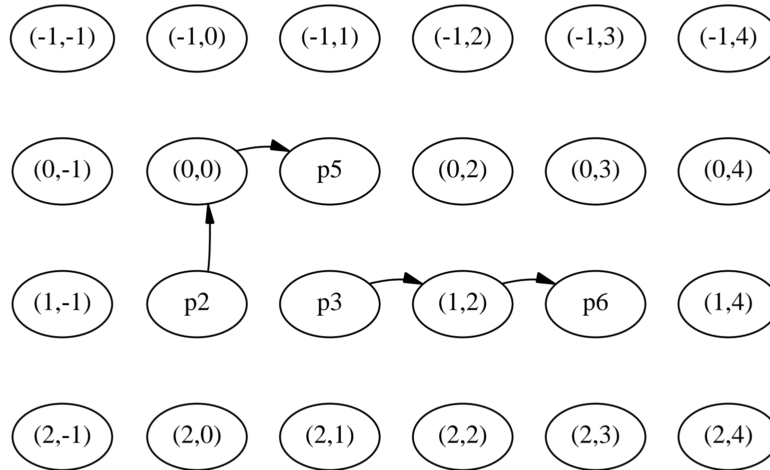

Figure 4-8: Routing of second communication context for FIR filter.

```
label0:
        move    $3, $cWi        route $cWi -> $csti
        beqz    $3, label1
        nop
        move    $0, $0          route $cWi -> $csti
label1:
        move    $3, $csto       route $csto -> $cSo
        beqz    $3, label2
        nop
        move    $0, $0          route $csto -> $cSo
label2:
        move    $3, $csto       route $csto -> $cEo
        beqz    $3, label3
        nop
        move    $0, $0          route $csto -> $cEo
label3:
        j       label0
        nop
$
```

Figure 4-9: Switch code for tile (0,0) of FIR filter.

```
while (ccr != context(c)) {
   if (isChannelSource(ccr)) {
      // Pre-send a false indicator.
      writeSwitch(0);
   }
   if (isChannelDest(ccr)) {
      // Check the pre-sent indicator.
      if (readSwitch()) {
         // Read and buffer this value.
         enqueue(ccr, readSwitch());
      }
   }
   ccr += 1;
}
```

Figure 4-10: Forcing the network to proceed to a particular channel's context.

The code generated on a processor for each `send!` form must first ensure that the static network has reached the appropriate communication context before writing a value to the switch. The code generated for a `receive!` must do the same before reading a value from the switch. The logic that does this is shown in Figure 4-10; this is placed inline into the generated code.

Assume that `c` is the channel specified in the communication primitive. Each processor maintains a variable `ccr` that tracks the current communication context of the network. `isChannelSource` and `isChannelDest` are predicates that tell whether a given context contains a channel that has an endpoint at the current tile. If there is a channel with its source at the current tile, the processor informs the network that nothing will be sent over it. If there is a channel with its destination at the current tile, the processor checks whether a value is being sent over it, and if so, buffers that value. Separate buffers are kept for each context.

In reality, the while loop shown in the figure will be fully unrolled in the generated code. This allows the increment of `ccr` and the branch at the end of the loop to be eliminated. It also facilitates elimination of the redundant checks implicit in `isChannelSource` and `isChannelDest`. These checks are redundant because the value of `ccr` on entry to the code completely determines the sequence of reads and writes to the switch that need to be performed. The unrolled code simply checks `ccr` on entry and branches to the appropriate

```
sl0:
        move    $13, $0
        beq     $12, $13, sl2
        nop
        addiu   $9, $0, 1
        beq     $12, $9, sl3
        nop
        addiu   $10, $0, 2
        beq     $12, $10, sl4
        nop
        addiu   $11, $0, 3
        beq     $12, $11, sl5
        nop
sl3:
        ori     $csto, $0, 0
sl4:
        ori     $csto, $0, 0
sl5:
sl2:
        addiu   $12, $0, 1
        or      $14, $csti, $0
        beq     $14, $0, sl6
        nop
        or      $14, $csti, $0
        j       __done17
        nop
sl6:
```

Figure 4-11: Code generated to force the network to proceed.

point in this sequence. Figure 4-11 shows an example of this code sequence from tile (0,0) of the example FIR filter. This tile is the destination of a channel in context 0, and the source of a channel in contexts 1 and 2. The code sequence attempts to proceed to context 0, assuming that register 12 is used as `ccr`. Another alternative is to use a jump table rather than a series of branches.

The code generated for a `send!` consists of this preprocessing step followed by pre-sending the value 1 and then sending the argument value. The code generated for a `receive!` is more complicated. It first checks whether any values have been buffered for the given channel. If so, it returns the oldest such value. Otherwise, it enters a loop consisting of the preprocessing step followed by reading the pre-sent indicator. If the indicator is non-zero, then it reads and returns the actual value. Otherwise, it repeats the loop,

waiting for the next time that the appropriate context is reached.

Note that this requires I/O devices to be aware of indicator values. In particular, input devices must generate them and output devices must accept them. This onus could be removed by using the dynamic network for device channels, relying on the hardware flow-control to determine when values are available.

Chapter 5

Optimization

The compilation scheme presented in Chapter 4 correctly transforms a SIFt program into code for a given Raw architecture. However, the quality of that code could be vastly improved. This chapter discusses three optimization opportunities aimed at improving both the processor and switch communication code. Section 5.1 shows how the code for `send!` and `receive!` on the processors can be improved by propagating information about the communication context register. Section 5.2 argues that analyzing the communication patterns of the entire program will allow the compiler to make better decisions about placement and communication scheduling.

5.1 Propagating the Communication Context

Even with loop unrolling, the code sequence used to force the network to a particular context (Figure 4-10) is fairly heavy-weight. This is unacceptable because streaming applications engage in a lot of communication. The expense of communication primitives directly limits how much pipeline parallelism can be gainfully exploited. However, this expense can be reduced by making compile-time inferences about the dynamic state of the network. In particular, the value in the communication context register `ccr` can be propagated throughout the body of a process. Wherever it is known that `ccr` can only carry a restricted set of values, the forcing code can be pared down to check for just those values. For instance, consider the code shown in Figure 4-11. If the compiler can infer that the value of `ccr` is always 2 at this program point, then the code can be optimized as shown in Figure 5-1.

The value analysis of `ccr` is presented as a set of inference rules in Figure 5-2. As in

```
        ori     $csto, $0, 0
        addiu   $12, $0, 1
        or      $14, $csti, $0
        beq     $14, $0, sl6
        nop
        or      $14, $csti, $0
        j       __done17
        nop
sl6:
```

Figure 5-1: Optimized code for forcing the network to proceed.

Section 3.3, each inference rule consists of an antecedent and consequent separated by a horizontal line. The rules summarize how expressions affect the set of possible values for `ccr`. The notation $A \vdash E : A'$ means "given that the set of possible current values for `ccr` is A, the set of possible values for `ccr` after evaluating E is A'". Lack of anything to the left of $\vdash$ means that the prior approximation has no bearing on the new one. The helper function *context-of* maps a channel identifier to the context to which it belongs, and *next-context* maps a context to the next one in the cycle. The number of communication contexts is represented by n_c.

According to the *literal* and *variable* rules, trivial expressions have no effect on `ccr`. The *primop*, *let*, and *begin* rules show that these types of expressions indirectly affect `ccr` through their subexpressions, and based on sequentially chaining the effects of those subexpressions. The *if* rule is similar, except that the approximations from the two sides are joined via the union operator. The *label* and *goto* rules are described below. The *send!* and *receive!* rules are the only ones that add information to the approximation, since the communication context is precisely known following every `send!` and `receive!` expression.

The analysis is conservative, meaning that the approximation at a given program point may contain values of `ccr` that are not realizable at that program point. However, the approximation will never fail to include a possible value. An example of this conservatism is found in the *label* and *goto* rules. They do not allow feedback from a `goto` to the entry of the corresponding `label` expression. Rather, nothing is assumed about `ccr` at the entry of the body of the `label` expression. This sacrifices precision in exchange for simplicity and efficiency of the analysis.

These inference rules serve as a roadmap for an analysis algorithm. The actual imple-

$$A \vdash L : A \qquad [\textit{literal}]$$

$$A \vdash I : A \qquad [\textit{variable}]$$

$$\frac{A \vdash E_1 : A_1; \quad A_1 \vdash E_2 : A_2; \quad \ldots; \quad A_{n-1} \vdash E_n : A_n}{A \vdash (\texttt{primop}\ O\ \ E_1\ \ E_2\ \ \ldots\ \ E_n) : A_n} \qquad [\textit{primop}]$$

$$\frac{A \vdash E_{test} : A_{test}; \quad A_{test} \vdash E_{con} : A_{con}; \quad A_{test} \vdash E_{alt} : A_{alt}}{A \vdash (\texttt{if}\ \ E_{test}\ \ E_{con}\ \ E_{alt}) : (A_{con} \cup A_{alt})} \qquad [\textit{if}]$$

$$\frac{A \vdash E_1 : A_1; \quad A_1 \vdash E_0 : A_0}{A \vdash (\texttt{let}\ ((I_1\ \ E_1))\ \ E_0) : A_0} \qquad [\textit{let}]$$

$$\frac{A \vdash E_1 : A_1; \quad A_1 \vdash E_2 : A_2; \quad \ldots; \quad A_{n-1} \vdash E_n : A_n}{A \vdash (\texttt{begin}\ E_1\ \ E_2\ \ \ldots\ \ E_n) : A_n} \qquad [\textit{begin}]$$

$$\frac{\{0, 1, ..., n_c\} \vdash E : A'}{\vdash (\texttt{label}\ \ I\ \ E) : A'} \qquad [\textit{label}]$$

$$\vdash (\texttt{goto}\ \ I) : \emptyset \qquad [\textit{goto}]$$

$$\frac{A \vdash E : A'}{A \vdash (\texttt{send!}\ \ \ I\ \ E) : \{(\textit{next-context}\ (\textit{context-of}\ I))\}} \qquad [\textit{send!}]$$

$$\vdash (\texttt{receive!}\ \ \ I) : \{(\textit{next-context}\ (\textit{context-of}\ I))\} \qquad [\textit{receive!}]$$

Figure 5-2: Inference rules for propagating `ccr`.

mentation would annotate each `send!` and `receive!` expression with the set of possible values of `ccr` at the program point immediately preceding the expression. The processor code generator could emit code that only handles the values in this approximation. In the best case, the approximation is a singleton set, and the code need not branch on `ccr` at all.

5.2 Analyzing Communication Patterns

As described, the SIFt compiler does very little to understand the communication patterns of the input program. The analysis phase simply builds a communication graph, which can only tell later phases which pairs of processes engage in communication. This level of abstraction ignores both the order in which channels are used and the amount of communication over a given channel. This more detailed information could improve the heuristics used during placement and communication scheduling.

For instance, consider the example four-tap FIR filter listed in Figure 3-10. Process `p1` repeatedly reads from input channel `in`, writes to channel `c1`, and writes to channel `c4`, in that order. If the communication schedule produced by the compiler places `in`, `c1`, and `c4` in contexts that are scheduled in some other order, then network bandwidth will be wasted as channel slots are regularly and repeatedly canceled at runtime. Assuming that the channels are scheduled in the order `in`, `c4`, `c1` (and input is always available on `in`), then half of the slots reserved for these three channels will be canceled.

How can this inefficiency be remedied? The first step is to extract ordering information on the use of channels within the same process. This would include such assertions as "a receive on channel `c` is always immediately preceded by a send on channel `d`". The second step is to use this information to guide communication scheduling. Such an algorithm could take one of two approaches. It could construct from scratch a schedule that closely matches the ordering information. Alternatively, it could iteratively improve a schedule using a cost function based on the ordering information.

Beyond the order in which channels are utilized, there is also the issue of the volume of data communicated over a channel. The cost function for placement assumes that all channels are created equal, requiring the same amount of network bandwidth. This can be a very detrimental assumption for a program that heavily reduces or expands its input streams. If the programmer had intimate knowledge of this deficiency, then she could com-

pensate by splitting the heavily-used channel into multiple channels. The communication along the original channel could then be spread out across several channels.

Requiring the programmer to engage in such trickery is reprehensible. Not only does it unnecessarily complicate program construction, but it also relies on properties of the compiler's internals that might change over time. Therefore, the compiler itself should infer which channels are used most heavily and adjust its heuristics accordingly. Given relative utilization information for the channels in a program, the compiler could apply it to two simple optimizations.

The first optimization would use the information to place weights on the edges of the communication graph. The weight of an edge would be proportional to its relative utilization metric. In the placement cost function, this weight would be used as a multiplier on the cost of routing the corresponding channel. The heavily used channels would thereby have more importance in placement decisions.

The second optimization would allow a single channel to exist in multiple communication contexts. This is essentially an automatic version of the channel splitting manually performed by the programmer in the above scenario. The number of contexts that a given channel is placed in would be proportional to its relative utilization metric. At runtime, each `send!` or `receive!` on that channel would use whichever of these contexts is first encountered.

Of course, these optimizations assume that this relative utilization information is available. Actually extracting this information from the source program is the difficult part. A detailed solution to this analysis problem is not presented herein. However, a successful algorithm will likely involve global analysis, since the relative rates of communication on different channels are highly interdependent.

This section has focused on static analysis of the input program. However, runtime profiling can also provide the information needed for these optimizations. For ordering information, the relevant data to collect is the pattern of channel slot cancellations. For relative utilization information, the relevant data to collect is the actual volume of data transferred over each channel.

Chapter 6

Results

This chapter evaluates the results of the compilation scheme presented in Chapter 4. In particular, these results do not include the optimizations discussed in Chapter 5. Section 6.1 describes the benchmark programs and the target architecture used for the experiments. Section 6.2 presents and discusses the actual results.

6.1 Evaluation Environment

Four types of benchmark programs were used to evaluate the correctness and performance of the SIFt compiler. These applications were chosen for their streaming structure.

Buffer simply passes elements of the input stream to the output stream unmodified. It does this by passing the elements down a long chain of processes. Since this type of program performs no computation, it really stresses the communication code. Four such programs were used, each one tailored to one, two, four, or eight tiles. Figure 6-1 shows the placement of the eight-tile version.

Adder implements an adder tree in which each process produces its output stream by pairwise addition of two input streams. Again, four such programs were used, each one tailored to one, two, four, or eight tiles. Each of the four programs takes a different number of input streams. Figure 6-2 shows the placement of the eight-tile version.

FIR filter implements a finite impulse response filter as described in Section 3.4. There are different versions for four-, eight-, and sixteen-tile configurations. These programs

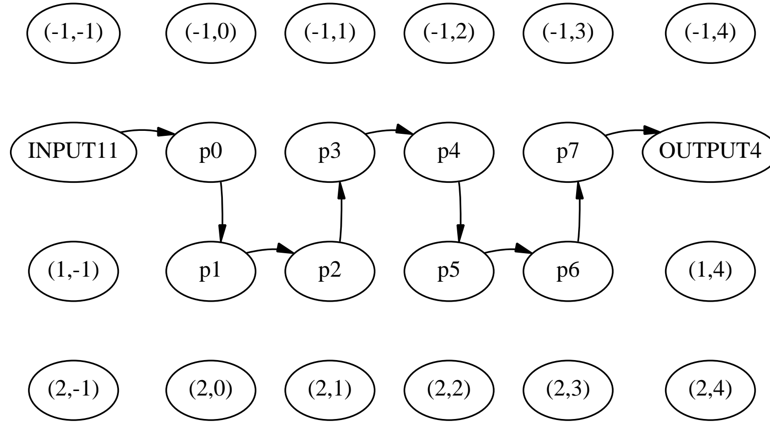

Figure 6-1: Placement of eight-tile buffer program.

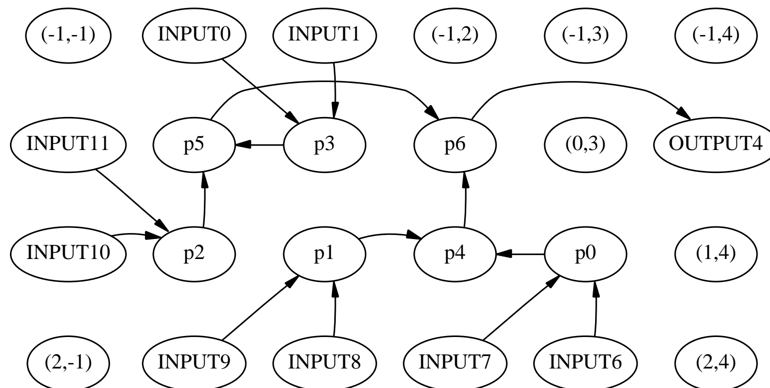

Figure 6-2: Placement of eight-tile adder program.

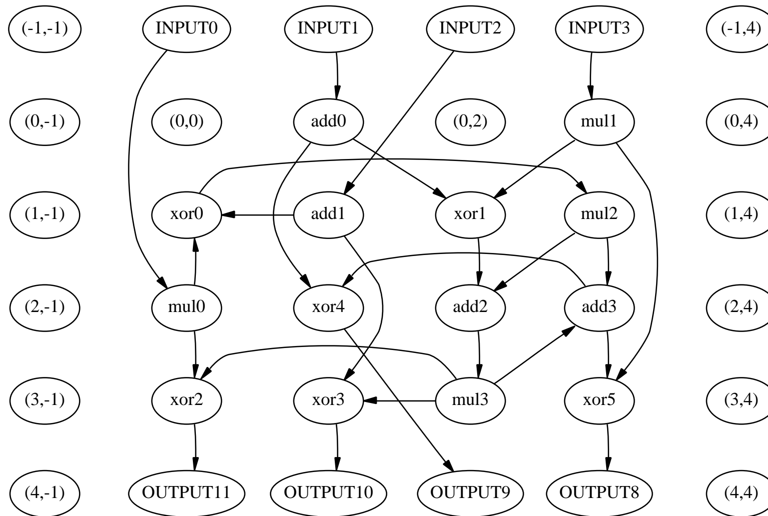

Figure 6-3: Placement of sixteen-tile IDEA program.

implement two-, four-, and nine-tap filters, respectively. One possible placement of the eight-tile version was shown in Figure 4-4.

IDEA implements a single round of the International Data Encryption Algorithm [19], whose full implementation consists of eight such rounds. The encryption key is a compile-time constant. This sixteen-tile program consumes four input streams and produces four output streams. It consists of a complicated network of exclusive-or, add, and multiply operations. Figure 6-3 shows a placement of the IDEA program.

Each program was compiled with the SIFt compiler and run on **bug**, the extensible Raw debugger. **bug** takes a Raw binary as input and simulates its execution on the appropriate Raw configuration. Further, it provides support for external devices with user-defined behavior. This facility was used to implement input and output streams for device channels. The end result of each execution is a set of files containing the sequence of values on each output channel, as well as the number of cycles executed.

For comparison purposes, an equivalent sequential C program was written for each of the SIFt programs. The C programs were compiled using RawCC, the Raw parallelizing compiler that exploits instruction-level parallelism (ILP). The resulting programs were run on the Raw debugger, and their outputs were used to verify the correctness of the SIFt programs.

Program	Number of Tiles				
	1	2	4	8	16
buffer	32.8	31.6	27.2	26.4	–
adder	39.5	57.7	65.4	92.1	–
FIR	–	–	16.2	16.7	10.5
IDEA	–	–	–	–	24.0

Table 6.1: Throughput of SIFt programs (inputs processed per kilocycle).

Program	Number of Tiles				
	1	2	4	8	16
buffer	162.1	163.9	165.3	165.6	–
adder	245.4	294.1	326.5	397.0	–
FIR	–	–	80.0	79.1	29.2
IDEA	–	–	–	–	19.7

Table 6.2: Throughput of C programs (inputs processed per kilocycle).

6.2 Experimental Results

Figure 6.1 shows the throughput of our benchmarks, in terms of number of input words processed per 1000 cycles. Figure 6.2 shows the analogous results for the sequential C programs compiled using RawCC.

The SIFt buffer benchmark degrades in performance as the number of tiles increases (Figure 6-4). In a perfect world, the number of tiles would affect the latency of this benchmark but not the throughput. The observed phenomenon is caused by the greedy communication scheduling algorithm. All four versions of the buffer benchmark can be scheduled using just two communication contexts. However, the algorithm comes up with suboptimal schedules in the four- and eight-tile versions that each use three contexts.

The SIFt adder benchmark improves in performance as the number of tiles increases (Figure 6-5). This is because the larger versions of the program add greater number of input streams, thereby increasing the merge parallelism that can be exploited. The processes representing adders at the leaves of the adder tree can all operate concurrently.

The SIFt FIR filter benchmark has fairly flat performance on four and eight tiles, but degrades sharply on sixteen tiles (Figure 6-6). One possible reason for this is that the four and eight tile configurations are a better match for an FIR filter's communication

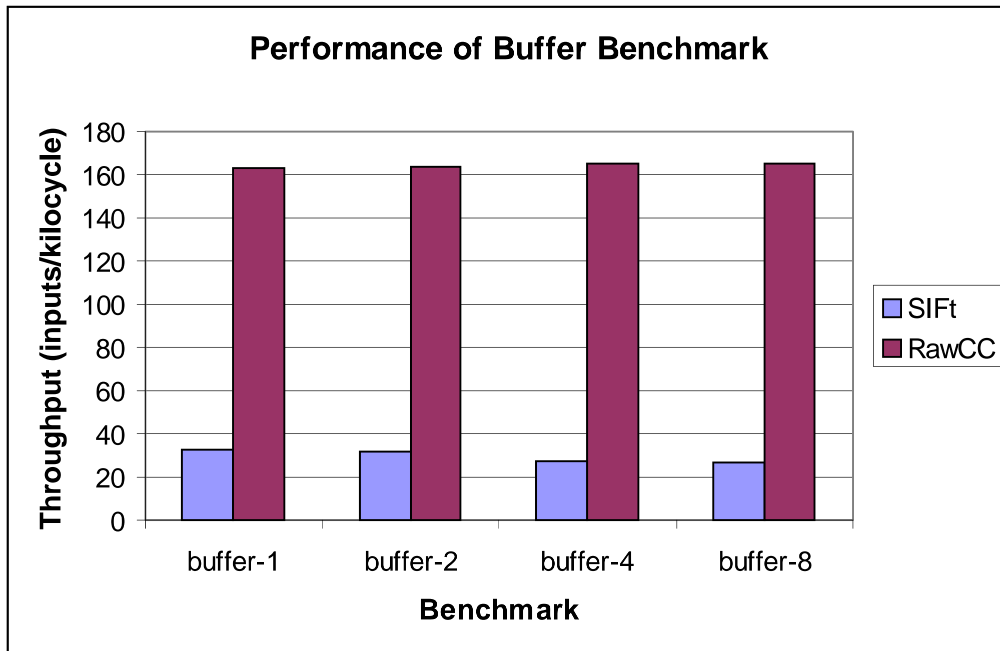

Figure 6-4: Performance of buffer benchmark.

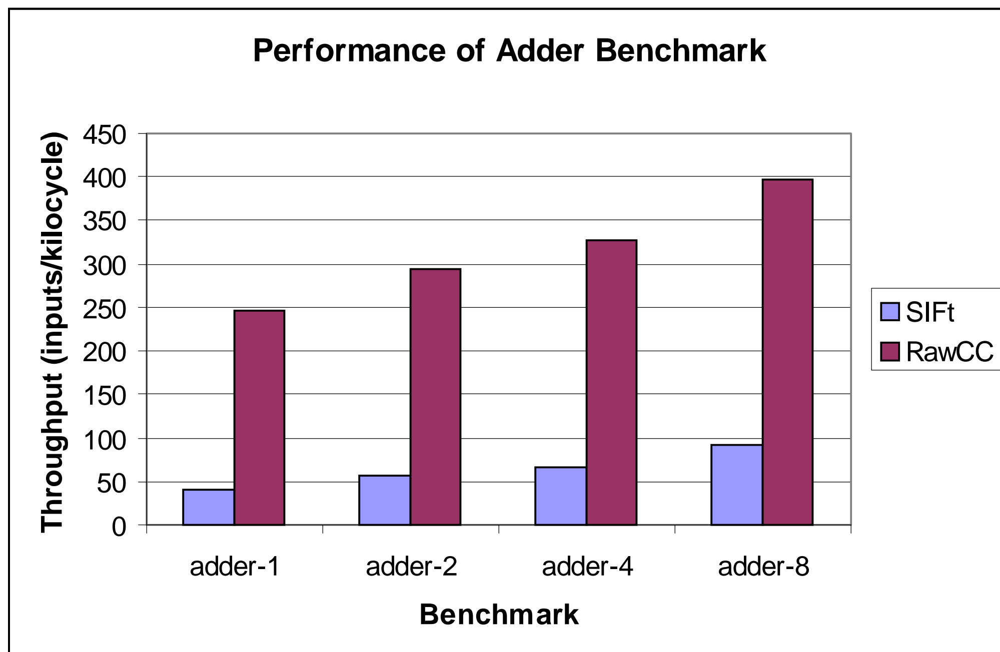

Figure 6-5: Performance of adder benchmark.

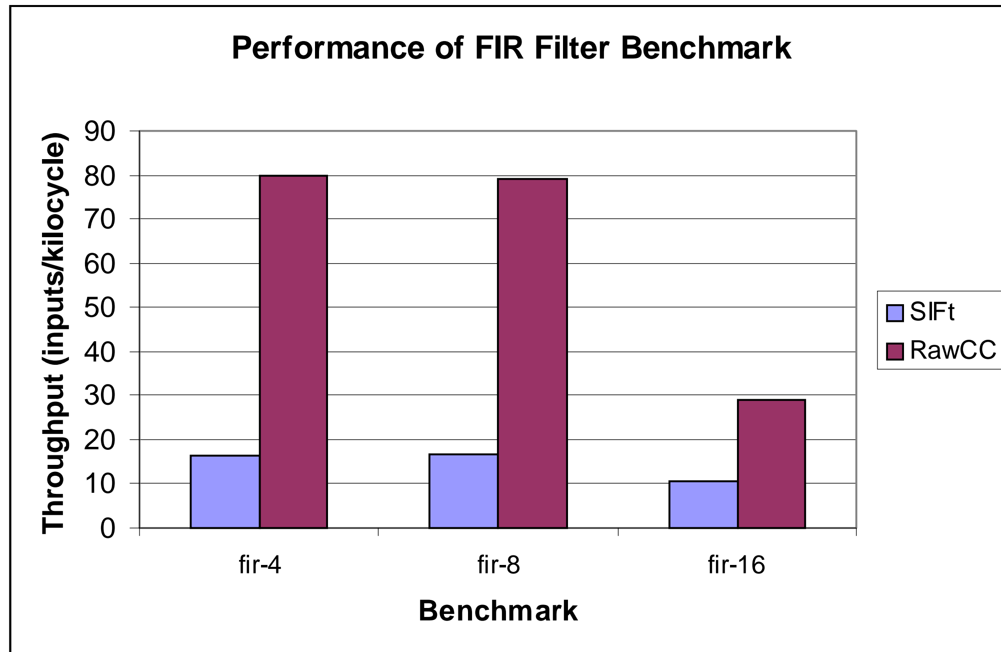

Figure 6-6: Performance of FIR filter benchmark.

graph. The sixteen-tile benchmarks all use a four-by-four configuration, whereas the more elongated two-by-eight configuration might yield better performance in this case.

Most of the throughput figures for the auto-parallelized programs are appallingly better than those for the SIFt programs. This is not at all surprising for the buffer and adder benchmarks, which are very communication-heavy. The negative result for the FIR filter is more enlightening, and suggests that the optimizations of Chapter 5 are absolutely necessary to surpass what can already be achieved by exploiting ILP.

The comparative results for the IDEA benchmark are much more encouraging (Figure 6-7). The throughput of the SIFt version exceeds that of the C version by 22%. This is because the IDEA benchmark is not as much of a “toy” benchmark as the others, particularly because it has nontrivial control flow within each process. The SIFt compilation scheme allows the processors and switches to have entirely decoupled control flow. On the other hand, RawCC’s compilation scheme requires that all processors and switches be at roughly the same control point at all times. This incurs the overhead of distributing the results of each branch instruction across the entire chip. Another contributing factor is that RawCC uses a less sophisticated instruction placement algorithm than the SIFt compiler (i.e., hill-climbing rather than simulated annealing).

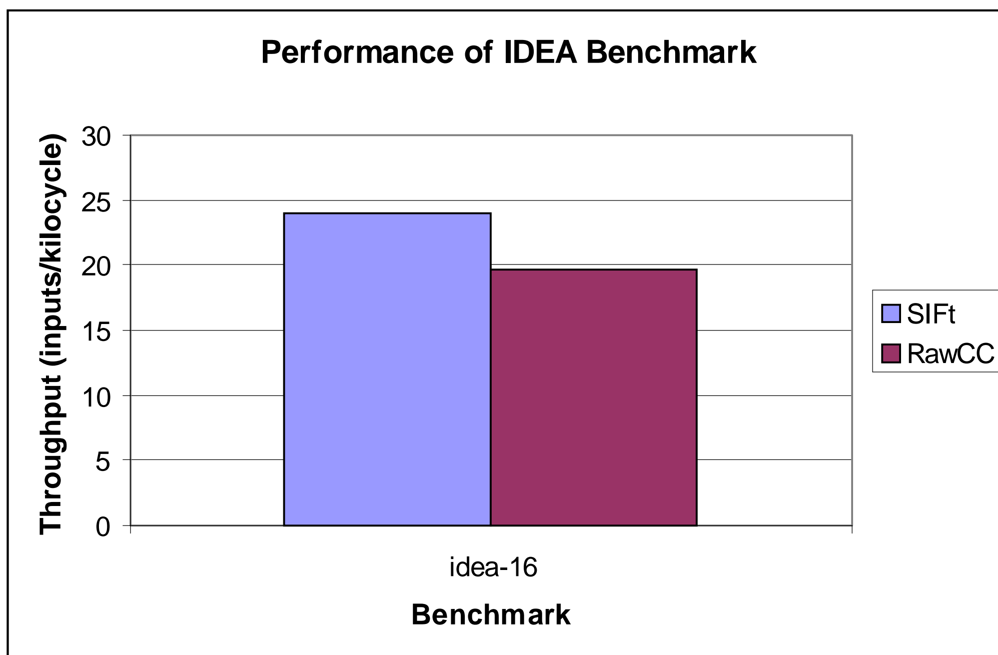

Figure 6-7: Performance of IDEA benchmark.

Chapter 7

Conclusion

7.1 Summary

With the proliferation of multimedia applications and wireless computing, streaming applications have become an important component of typical computing workloads. Programs written using streams exhibit a large degree of parallelism that can be exploited by novel hardware and software designs. Research in hardware geared towards streaming applications has progressed rapidly over the past several years. However, specialized compiler techniques for such programs has not matched this progress. To that end, this paper has presented an intermediate format and compilation technique for streaming applications, and musings about possible low-level optimization techniques.

The computer science concept of a stream has been around for some time. A stream is simply an ordered sequence of values, typically operated upon in order and just a few values at a time. Several theoretical and practical languages have been designed for streaming applications. The most influential theoretical language is the π-calculus, which characterizes a concurrent program as a set of processes communicating with one another through channels. The compiler intermediate format presented in this paper, SIFt, follows this high-level paradigm.

A SIFt program consists of a fixed set of processes communicating through a fixed set of channels. The body of a process is composed of expressions, which include many of the constructs found in a modern programming language such as C. In addition, communication primitives are provided for sending and receiving values through channels. The base types in SIFt include integers, booleans, and floating point values. Richer types such as arrays,

pointers, and structures are not present in this prototype. The semantics of computation in SIFt is fairly standard and intuitive. The semantics of communication are more interesting. Channels are unidirectional and static; each channel can communicate values between exactly two processes in one direction only. Conceptually, each channel is a FIFO queue; send operations add a value to the tail, while receive operations consume a value from the head.

The prototype SIFt compiler consists of five major phases: parsing, analysis, placement, communication scheduling, and code generation. The end result is code that can be fed into the Raw MachSUIF backend to generate a Raw binary. The analysis phase performs some static semantic checks and builds a communication graph of the program. The placement phase uses an optimization technique called simulated annealing to find an efficient assignment of processes to tiles. The communication scheduling phase uses a greedy algorithm to partition the set of channels into subsets (called "communication contexts") that can be trivially scheduled on the interconnection network. The code generation phase generates an instruction stream for each processor and switch in the target configuration. Most notably, it routes the channels and handles the details of cycling the interconnection network through the communication contexts. The entire compiler is implemented in fewer than 7000 lines of Java code.

The code generated by this basic compilation scheme is far from optimal. By propagating static information about the current communication context, the code for send and receive operations on the processor can be vastly simplified. The propagation algorithm was summarized in a small set of inference rules that should be quite simple to implement. Further gains could be achieved by analyzing and leveraging the static communication patterns in a program. In particular, the order in which channels are typically used and the relative volume of communication on different channels would come in handy. The placement and communication scheduling phases could use this information to make better heuristic decisions.

The correctness and performance of the compiler (without the optimizations) were evaluated using four types of hand-written benchmarks. Output and performance numbers were obtained by simulating the compiled programs on the Raw debugger. These were compared to the output and performance of equivalent C programs compiled with RawCC, the auto-parallelizing compiler for Raw. All of the SIFt programs generated correct output. The

throughput of most of the SIFt programs were abysmal compared to their auto-parallelized counterparts. This suggests that the aforementioned optimizations are necessary in order to surpass what is already achievable by exploiting instruction-level parallelism in these toy programs. The largest and most realistic program in the benchmark suite, IDEA encryption, outperformed the auto-parallelized version by 22%. After applying the optimizations, such programs should show much more dramatic improvement, perhaps increasing throughput by integer factors.

7.2 Future Work

Judging from the performance results, the first piece of future work should be implementing and evaluating the optimizations described in Chapter 5. Propagation of the communication cycle register is the low-hanging fruit here, since the analysis algorithm has already been laid out. With the results of that analysis available, eliminating unnecessary overhead in the code generated for `send!` and `receive!` operations should be straightforward. The optimizations based on compile-time analysis of communication patterns will require much more fleshing out.

Almost as important is filling out the feature space of the intermediate format. The lack of functions, globally-accessible data, structures, and arrays, makes it difficult to implement large real-life applications. Such applications include the full IDEA encryption and decryption algorithms, a software radio receiver, and a pipelined Internet Protocol router a la the Click modular router [15]. Adding all of these language features to the current SIFt infrastructure would be prohibitively labor-intensive. Rather, a well-known intermediate format such as SUIF, extended to include communication primitives, could be used for the body of SIFt processes. This would have the added benefit of integrating nicely with the RawCC infrastructure.

The construction of almost any software system comes with difficult tradeoffs that must be resolved. The design of the SIFt compiler was no exception. Throughout this paper, several design decisions have been mentioned, both with and without justification. Exploring alternatives to these decisions will be an important component of future work on the system. The parameters of the placement algorithm (i.e., temperature decay rate, number of iterations per temperature, etc.) could be adjusted to elucidate their effects on

performance. Going further, one could experiment with entirely different placement algorithms. The greedy algorithm used for communication scheduling should almost certainly be replaced. Examining its output reveals some very poor scheduling choices for even the simplest communication graphs.

The explanations for the observed performance have thus far been mostly speculation. Studying in more detail the source of performance bottlenecks would help to guide efforts to eliminate them. For instance, it would be helpful to know the fraction of network bandwidth used for indicator bits versus actual data. This could be accomplished by augmenting the simulator or instrumenting the program to emit such information. This profile-gathering framework could also be used as a source for profile-directed optimizations.

Some of the most exciting and challenging future work will be transforming a sequential program with an implied streaming structure into SIFt's explicitly-streaming format. Preliminary research has shown that even some traditional scientific applications such as LU decomposition have an underlying streaming structure [23]. High-level optimizations on streams, of which retiming and repartitioning are just two examples, have also yet to be explored.

Appendix A

Using the SIFt Compiler

The SIFt compiler runs in the operating environment of the Computer Architecture Group at the MIT Laboratory for Computer Science. This appendix enumerates the steps for installing the compiler and using it to compile your own SIFt programs.

A.1 Installing the Compiler

Since there is no global installation of the compiler, the user must checkout his own private copy of the source tree from the CVS repository.

```
~> cvs -d /projects/raw/cvsroot checkout sift
```

This creates a local copy of the source tree in the "sift" directory. The next step is to compile the compiler, which must be done using a Java 1.2-compliant compiler.

```
~> setenv JAVA_HOME /usr/uns/jdk1.2.1
~> setenv CLASSPATH ~/sift:$CLASSPATH
~> make -C sift
```

This creates the classfiles for packages `sift.format`, `sift.parser`, `sift.analysis`, `sift.codegen`, and `sift.compiler`. The entry-point for the Java portion of the compiler is `sift.compiler.Main`, which is invoked by the top-level driver `siftc.pl`, located in `sift/util/`.

The SIFt compiler uses the same backend as the RawCC compiler. Therefore, the environment variables required for RawCC must be initialized. As of this writing, the following should be sufficient:

```
~> setenv RAWCCDIR /projects/raw/current/rawcc
~> setenv SUIFDIR /projects/raw/current/suif
~> setenv PATH $RAWCCDIR/compiler/bin:$SUIFDIR/bin:$PATH
~> setenv LD_LIBRARY_PATH $RAWCCDIR/compiler/lib:$SUIFDIR/lib
```

If you have checked out the system into a directory other than `$HOME/sift/`, be sure to set the `SIFT_HOME` environment variable to that directory's pathname. Alternatively, you can create a symbolic link from `$HOME/sift/` to the installed directory.

A.2 Compiling a Program

The compiler requires two files as input, the first being the SIFt program. The other file describes the location of I/O devices on the periphery of the chip. The compiler needs this information in order to set up the simulation directory. The syntax of the SIFt program file follows the grammar shown in Figure 3-2; extra whitespace is ignored. This program file must be named `<progname>.sift`. Likewise, the device file must be named `<progname>.dev`. The device file consists of zero or more device specifications. Each device specification occupies a single line and contains four whitespace-separated fields.

The first field gives the kind of device, which can be "input" or "output". The second field contains the port number, which specifies the location of the device. Ports are numbered starting with zero immediately to the north of tile (0,0) and moving counter-clockwise around the periphery of the mesh. The third field specifies the type of data, which can be "int", "float", or "bool". The final field is the pathname of the text file that backs this stream. For an input device, this file contains a series of whitespace-separated data values that will be emitted by the device. For an output device, this file will contain the data values output by the program on that port during execution. An example device file for the program in Figure 3-10 looks like the following:

```
input 11 int input.txt
output 4 int output.txt
```

The program file, device file, and input data files should all be placed in a single directory. The SIFt compiler can then be invoked from within that directory:

```
% siftc.pl <progname> <nrows> <ncols>
```

The `siftc.pl` script performs four major steps. It begins by running the SIFt compiler `sift.compiler.Main`, which generates a raft of C++ code that is tailored to output the low SUIF code for the compiled program. This convoluted intermediate step is required because there is no easy way to emit the SUIF binary format from a Java program. The script then compiles and runs the C++ program, which generates a single file containing the low SUIF. This file is fed through a series of SUIF and MachSUIF passes similar to the code generator for RawCC, resulting in a Raw binary. Finally, the script copies the binary, the input stream data files, and the bC startup code for initializing the attached devices into a subdirectory called `run`.

The resulting program can be run using the Raw simulator, `bug`. This can be accomplished by entering the `run` directory and invoking the simulator as follows:

```
% bug -f code.raw
```

Note that the program will wait for input on its input ports even after the data that you provided has been exhausted. After quitting the simulator, all output should be available in the files specified in the device file.

The future holds two major changes for the SIFt compiler. The first change will be the ability to write programs in a superset of C that is augmented with processes and channels. This will much easier construction of real-life large-scale streaming applications. It will also allow closer integration with the RawCC infrastructure. The second change will make the output of the compiler compatible with `btl`, the new Raw simulator.

Appendix B

Extended Example: IDEA

This appendix presents an extended compilation example using the IDEA benchmark described in Section 6.1. For a description of IDEA, refer to Schneier [19]. This benchmark takes four 16-bit input streams and applies one round of IDEA encryption to them to generate four 16-bit output streams. The encryption key is a compile-time constant.

Figure B-1 shows the first part of the SIFt source code, which declares the four input channels, a multitude of internal channels, and the four output channels. The port numbers assigned to the device channels have the inputs coming into the top of the chip and the outputs coming out the bottom.

Figure B-2 shows the SIFt code for the multiplier processes. Multiplication is done modulo $2^{16}+1$, with 0 representing 2^{16}. The `z?` variables are 16-bit chunks of the encryption key.

Figure B-3 shows the SIFt code for the adder processes. Addition is done modulo 2^{16}, with the modulo operation implemented as a bitwise AND.

Figure B-4 shows the final portion of SIFt code, which implements the exclusive-OR processes. Note that many of the processes have the same structure, resulting in a lot of code duplication. This could be alleviated quite simply by adding functions or macros to the language.

Sifting through the code does little to elucidate the structure of the program. The communication graph in Figure B-5 does a better job of that. This graph was automatically generated by the analysis phase of the SIFt compiler.

Figure B-6 shows the results of the placing this communication graph onto a 4x4 Raw configuration. This graph was automatically generated by the placement phase of the SIFt compiler.

Finally, Figures B-7, B-8, B-9, B-10, B-11, B-12, and B-13 show the results of communication scheduling. These graphs were automatically generated by the communication scheduling phase of the SIFt compiler. Note that the final three communication contexts are very sparse, a result of the greedy communication scheduling algorithm. A more sophisticated algorithm could probably find a schedule consisting of one or two fewer contexts.

The performance of the compiled program on the Raw simulator is shown in Section 6.2.

```
(program

 (define x1 (input 0 int))
 (define x2 (input 1 int))
 (define x3 (input 2 int))
 (define x4 (input 3 int))

 (define mul0.in (channel int))
 (define mul1.in (channel int))
 (define mul2.in (channel int))
 (define mul3.in (channel int))

 (define add0.in0 (channel int)) (define add0.in1 (channel int))
 (define add1.in0 (channel int)) (define add1.in1 (channel int))
 (define add2.in0 (channel int)) (define add2.in1 (channel int))
 (define add3.in0 (channel int)) (define add3.in1 (channel int))

 (define xor0.in0 (channel int)) (define xor0.in1 (channel int))
 (define xor1.in0 (channel int)) (define xor1.in1 (channel int))
 (define xor2.in0 (channel int)) (define xor2.in1 (channel int))
 (define xor3.in0 (channel int)) (define xor3.in1 (channel int))
 (define xor4.in0 (channel int)) (define xor4.in1 (channel int))
 (define xor5.in0 (channel int)) (define xor5.in1 (channel int))

 (define y1 (output 11 int))
 (define y2 (output 10 int))
 (define y3 (output 9 int))
 (define y4 (output 8 int))
```

Figure B-1: SIFt code for IDEA benchmark, part 1.

```
(define mul0 (process
 (let ((z1 22896))
   (label loop
    (let ((v (receive! x1)))
      (let ((m (primop * (if (primop = v 0) 65536 v) z1)))
        (let ((r (primop & (primop % m 65537) 65535)))
          (begin
            (send! xor0.in0 r)
            (send! xor2.in0 r)
            (goto loop)))))))))

(define mul1 (process
 (let ((z4 34761))
   (label loop
    (let ((v (receive! x4)))
      (let ((m (primop * (if (primop = v 0) 65536 v) z4)))
        (let ((r (primop & (primop % m 65537) 65535)))
          (begin
            (send! xor1.in1 r)
            (send! xor5.in0 r)
            (goto loop)))))))))

(define mul2 (process
 (let ((z5 34014))
   (label loop
    (let ((v (receive! mul2.in)))
      (let ((m (primop * (if (primop = v 0) 65536 v) z5)))
        (let ((r (primop & (primop % m 65537) 65535)))
          (begin
            (send! add3.in0 r)
            (send! add2.in0 r)
            (goto loop)))))))))

(define mul3 (process
 (let ((z6 39231))
   (label loop
    (let ((v (receive! mul3.in)))
      (let ((m (primop * (if (primop = v 0) 65536 v) z6)))
        (let ((r (primop & (primop % m 65537) 65535)))
          (begin
            (send! add3.in1 r)
            (send! xor2.in1 r)
            (send! xor3.in1 r)
            (goto loop)))))))))
```

Figure B-2: SIFt code for IDEA benchmark, part 2.

```
(define add0 (process
 (let ((z2 52540))
   (label loop
    (let ((v (receive! x2)))
      (let ((a (primop + v z2)))
        (let ((r (primop & a 65535)))
          (begin
            (send! xor1.in0 r)
            (send! xor4.in0 r)
            (goto loop)))))))))

(define add1 (process
 (let ((z3 41445))
   (label loop
    (let ((v (receive! x3)))
      (let ((a (primop + v z3)))
        (let ((r (primop & a 65535)))
          (begin
            (send! xor0.in1 r)
            (send! xor3.in0 r)
            (goto loop)))))))))

(define add2 (process
 (label loop
  (let ((v0 (receive! add2.in0))
        (v1 (receive! add2.in1)))
    (let ((a (primop + v0 v1)))
      (begin
        (send! mul3.in (primop & a 65535))
        (goto loop)))))))

(define add3 (process
 (label loop
  (let ((v0 (receive! add3.in0))
        (v1 (receive! add3.in1)))
    (let ((a (primop + v0 v1)))
      (let ((r (primop & a 65535)))
        (begin
          (send! xor4.in1 r)
          (send! xor5.in1 r)
          (goto loop))))))))
```

Figure B-3: SIFt code for IDEA benchmark, part 3.

```
(define xor0 (process
 (label loop
  (let ((v0 (receive! xor0.in0))
        (v1 (receive! xor0.in1)))
    (begin
      (send! mul2.in (primop ^ v0 v1))
      (goto loop))))))

(define xor1 (process
 (label loop
  (let ((v0 (receive! xor1.in0))
        (v1 (receive! xor1.in1)))
    (begin
      (send! add2.in1 (primop ^ v0 v1))
      (goto loop))))))

(define xor2 (process
 (label loop
  (let ((v0 (receive! xor2.in0))
        (v1 (receive! xor2.in1)))
    (begin
      (send! y1 (primop ^ v0 v1))
      (goto loop))))))

(define xor3 (process
 (label loop
  (let ((v0 (receive! xor3.in0))
        (v1 (receive! xor3.in1)))
    (begin
      (send! y2 (primop ^ v0 v1))
      (goto loop))))))

(define xor4 (process
 (label loop
  (let ((v0 (receive! xor4.in0))
        (v1 (receive! xor4.in1)))
    (begin
      (send! y3 (primop ^ v0 v1))
      (goto loop))))))

(define xor5 (process
 (label loop
  (let ((v0 (receive! xor5.in0))
        (v1 (receive! xor5.in1)))
    (begin
      (send! y4 (primop ^ v0 v1))
      (goto loop))))))

)
```

Figure B-4: SIFt code for IDEA benchmark, part 4.

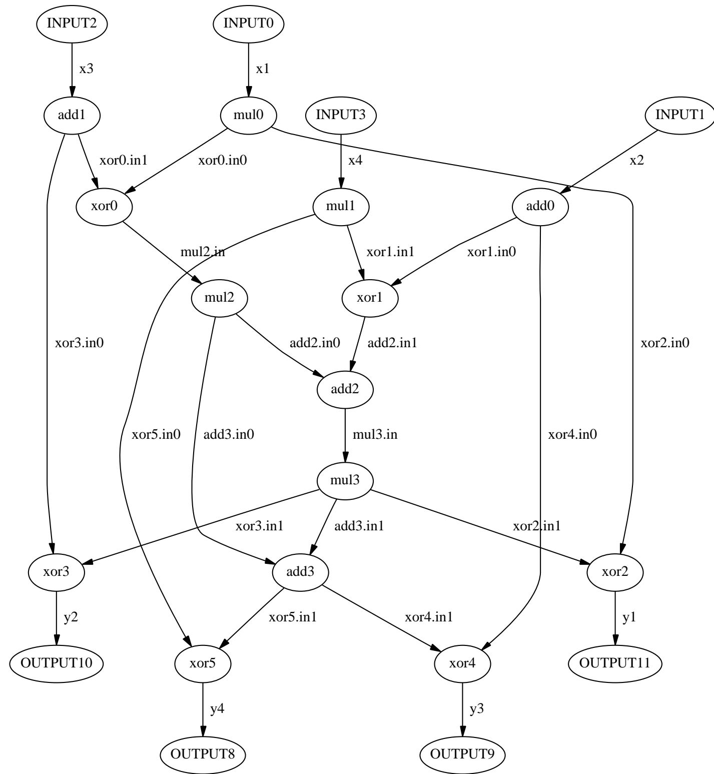

Figure B-5: Communication graph of IDEA benchmark.

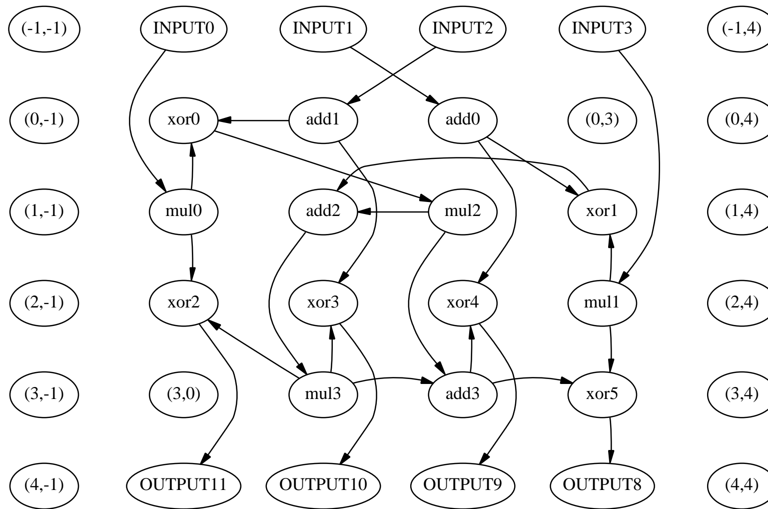

Figure B-6: Placement of IDEA benchmark on 4x4 Raw chip.

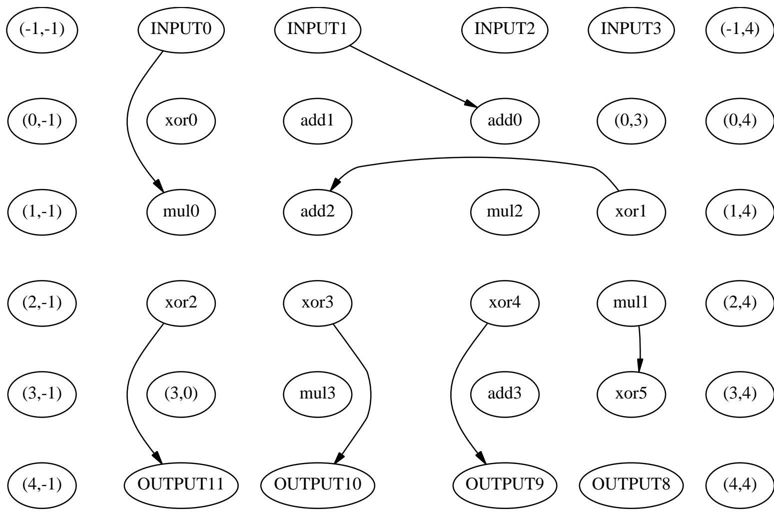

Figure B-7: Communication context 0 of IDEA benchmark.

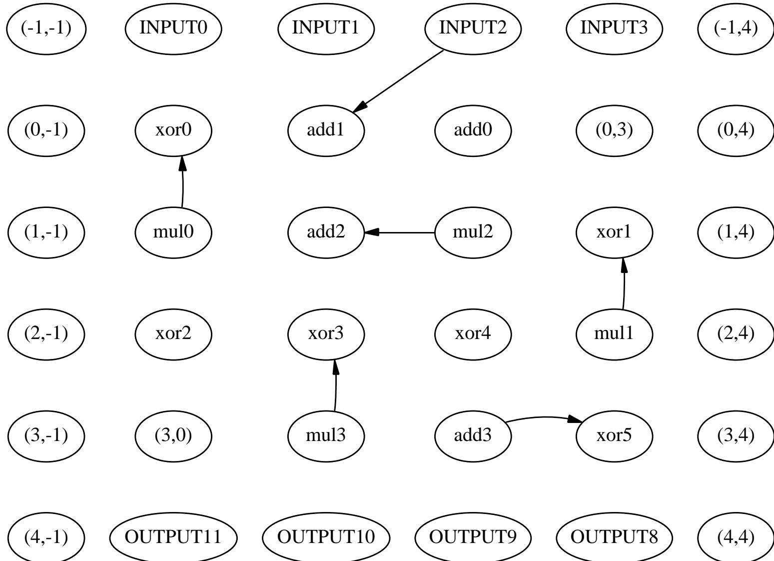

Figure B-8: Communication context 1 of IDEA benchmark.

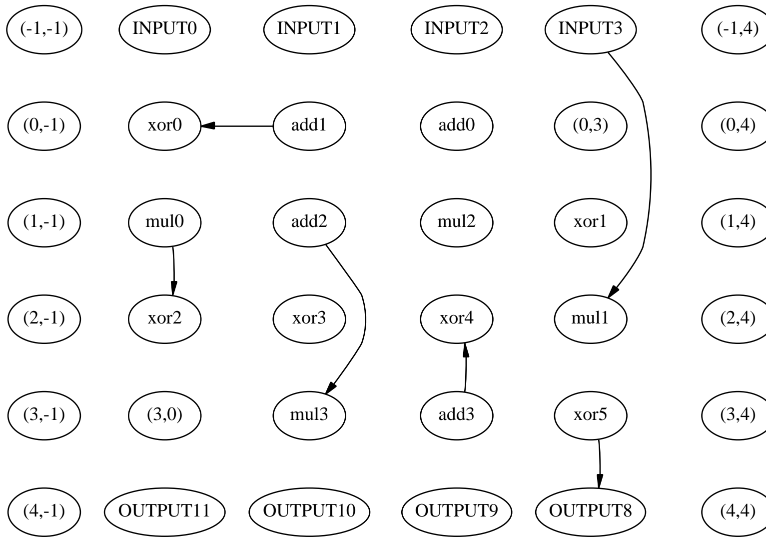

Figure B-9: Communication context 2 of IDEA benchmark.

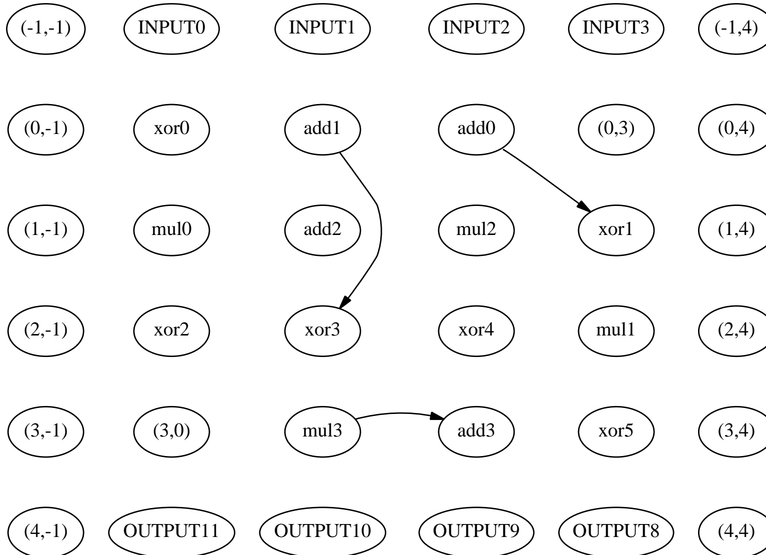

Figure B-10: Communication context 3 of IDEA benchmark.

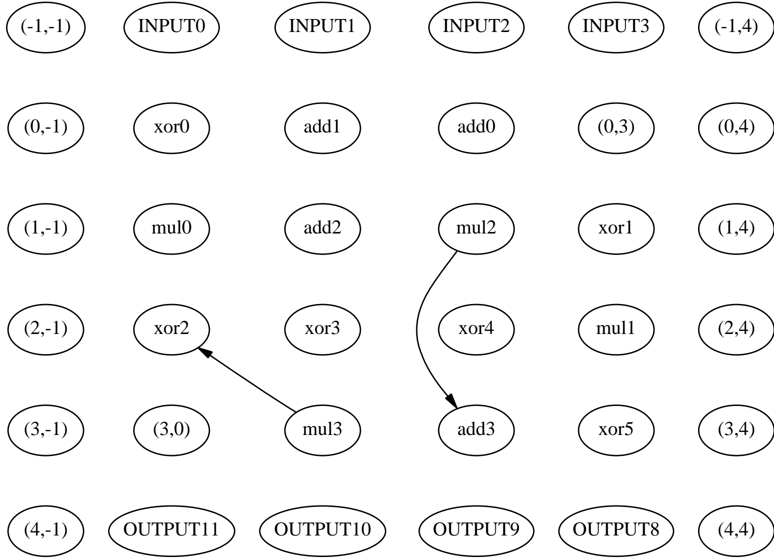

Figure B-11: Communication context 4 of IDEA benchmark.

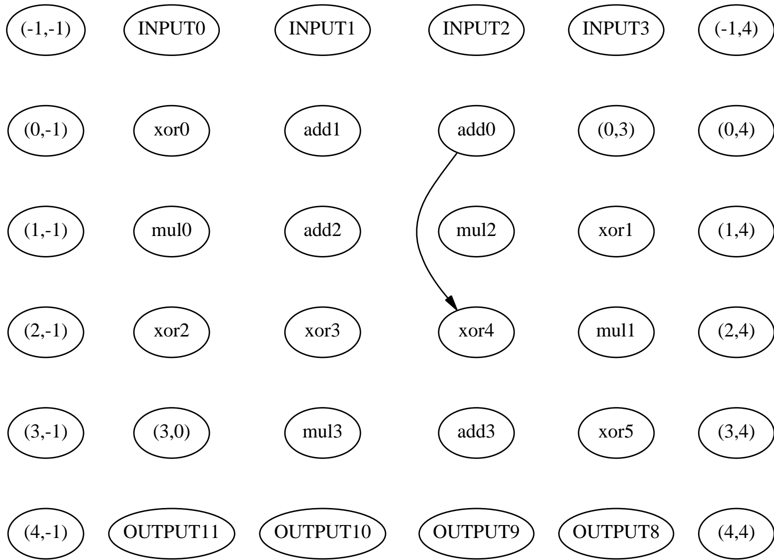

Figure B-12: Communication context 5 of IDEA benchmark.

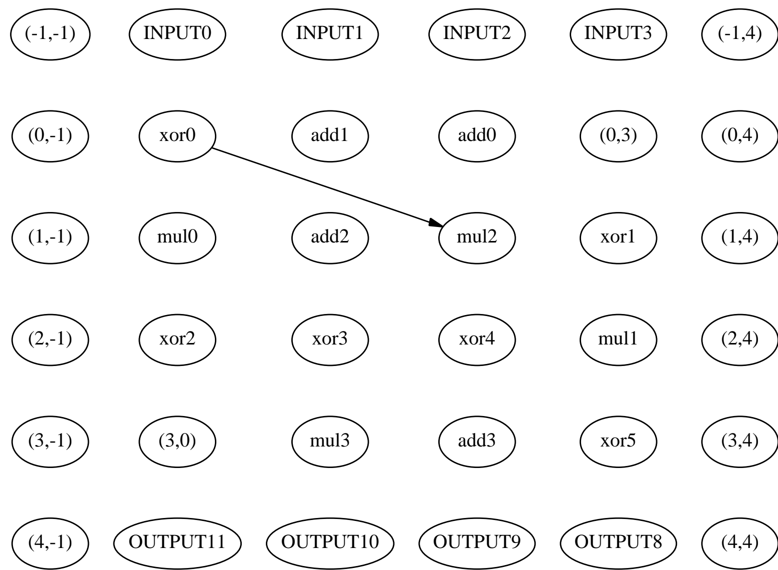

Figure B-13: Communication context 6 of IDEA benchmark.

Bibliography

[1] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, MA, second edition edition, 1996.

[2] Anant Agarwal. Raw computation. *Scientific American*, 281(2):60–63, August 1999.

[3] Vanu Bose, Mike Ismert, Matt Welborn, and John Guttag. Virtual radios. *IEEE Journal on Selected Areas in Communication, Special Issue on Software Radios*, 1999.

[4] V. Michael Bove, Jr. and John A. Watlington. Cheops: A reconfigurable data-flow system for video processing. *IEEE Transactions on Circuits and Systems for Video Technology*, 5:140–149, April 1995.

[5] Lawrence Davis, editor. *Genetic Algorithms and Simulated Annealing.* Research Notes in Artificial Intelligence. Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1987.

[6] Anja Feldmann, Thomas M. Stricker, and Thomas E. Warfel. Supporting sets of arbitrary connections on iWarp through communication context switches. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 203–212, Velen, Germany, June 1993.

[7] Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. Piperench: A coprocessor for streaming multimedia acceleration. In *Proceedings of the Twenty-Sixth International Symposium on Computer Architecture*, pages 28–39, Atlanta, GA, June 1999.

[8] International Organisation for Standardization, Seoul. *Generic Coding of Moving Pictures and Associated Audio*, November 1993. Recommendation H.262, ISO/IEC 13818-2.

[9] He Jifeng, Ian Page, and Jonathan Bowen. Towards a provably correct hardware implementation of occam. In *Proceedings of IFIP WG10.2 Advanced Research Working Conference, CHARME '93*, pages 214–225. Springer-Verlag, May 1993.

[10] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proceedings of IFIP Congress 74*, pages 471–475. North-Holland Publishing Co., 1974.

[11] Andrea S. LaPaugh. Layout algorithms for VLSI design. *ACM Computing Surveys*, 28(1):59–61, March 1996.

[12] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-time scheduling of instruction-level parallelism on a raw machine. In *Proceedings of the Eighth International Conference*

on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, October 1998.

[13] Stacie P. Leone and Max Kalehoff. Media Metrix to cover streaming media content and address digital convergence. *Media Metrix Press Release*, December 1999. Available electronically at http://www.mediametrix.com/PressRoom/Press_Releases/12_07_99.html.

[14] Brian Kevin Livezey. The ASPEN distributed stream processing environment. Master's thesis, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, December 1988.

[15] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The click modular router. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 217–231, Charleston, NC, December 1999.

[16] D. Stott Parker, Eric Simon, and Patrick Valduriez. SVP – a model capturing sets, streams, and parallelism. In *Proceedings of the 18th Conference on Very Large Data Bases*, pages 115–126, Vancouver, British Columbia, August 1992.

[17] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. CSCI Technical Report #476, Indiana University, 107 S. Indiana Ave. Bloomington, IN 47405-7000, March 1997. Available electronically at http://www.cs.indiana.edu/pub/techreports/TR476.html.

[18] Scott Rixner, William J. Dally, Ujval J. Kapani, Brucek Khailany, Abelardo Lopez-Lagunas, Peter R. Mattson, and John D. Owens. A bandwidth-efficient architecture for media processing. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, Dallas, TX, November 1998.

[19] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* John Wiley & Sons, Inc., New York, NY, second edition, 1996.

[20] William McC. Siebert. *Circuits, Signals, and Systems.* The MIT Electrical Engineering and Computer Science Series. The MIT Press, Cambridge, MA, 1986.

[21] Michael D. Smith. Extending SUIF for machine-dependent optimizations. In *Proceedings of the First SUIF Compiler Workshop*, pages 14–25, Stanford, CA, January 1996.

[22] Michael Bedford Taylor. Design decisions in the implementation of a Raw architecture workstation. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, September 1999.

[23] William Thies. Personal communication, February 2000.

[24] Franklyn Turbak, David Gifford, and Brian Reistad. Applied semantics of programming languages. Draft, September 1999.

[25] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(9):86–93, September 1997.

[26] Robert P. Wilson, Robert S. French, Chris S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An infrastructure for research in parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

[27] Xin Yuan, Rami Melhem, and Rajiv Gupta. Compiled communication for all-optical TDM networks. In *Supercomputing '96*, Pittsburgh, PA, 1996.