# Chapter 1

# Introduction

## 1.1 The Problem

Media processing applications, such as image-processing, signal processing, and graphics, motivate new processor architectures that place new burdens on the compiler. These applications demand very high arithmetic rates on the order of 10-100 billion operations per second. They also demand correspondingly high data bandwidth, and have little to no data reuse, i.e. most data are read only once after being written. [11][46]

Conventional processors, like the one shown in Figure 1-1, cannot meet the demands of media processing applications. They cannot support enough functional units to achieve the needed arithmetic rates because their register file architecture does not scale. As shown in Figure 1-1, each functional unit input or output is connected by a dedicated bus and register file port to a single register file. The size of the register file is proportional to the cube of the number of functional units [44]. Conventional processors cannot supply the needed data bandwidth because their on-chip memory is in the form of a cache that relies on data reuse to reduce memory traffic. A cache never reduces memory traffic by more than half for an application without data reuse, in which all data is written once and read once. The cache cannot anticipate that the data is never reused so it must propagate the write to memory. Further, the random-access design of a cache limits the amount of bandwidth it can provide to the processor core.

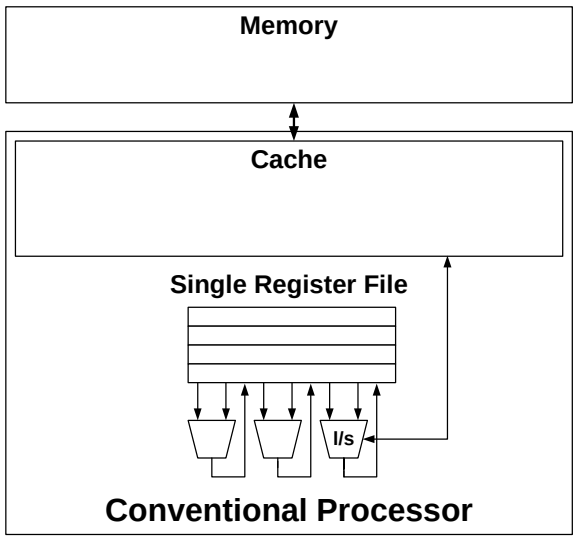



**FIGURE 1-1. Simplified diagram of a conventional processor**

The Imagine Media Processor (Imagine), shown in Figure 1-2, introduces two innovations that enable it to meet the demands of media processing applications. First, Imagine replaces the single register file with distributed register files: multiple two-ported register files connected to the functional units by shared buses and register file ports. These distributed register files can efficiently support a large number of functional units. Each function unit output is connected to a bus that is connected to all shared register file write ports. Second, Imagine uses a Stream Register File (SRF) instead of a cache. The SRF allows the application to explicitly load and store long sequences of data records called streams. Loading and storing streams only when necessary significantly reduces memory traffic. The SRF is optimized for sequential access to these streams, allowing it to provide much higher bandwidth to the processor core than a cache.

Other architectures designed for media processing have incorporated limited support for similar features. For example, the Texas Instruments C6X [48] has two partitioned register files and the Equator MAP-CA [2] has a programmer controlled DMA unit for accessing streams through an existing cache. However, Imagine is designed around the concepts of partitioned register files and streaming memory access: register files are partitioned down

to a single register file per functional unit input and streams are used for all memory accesses.

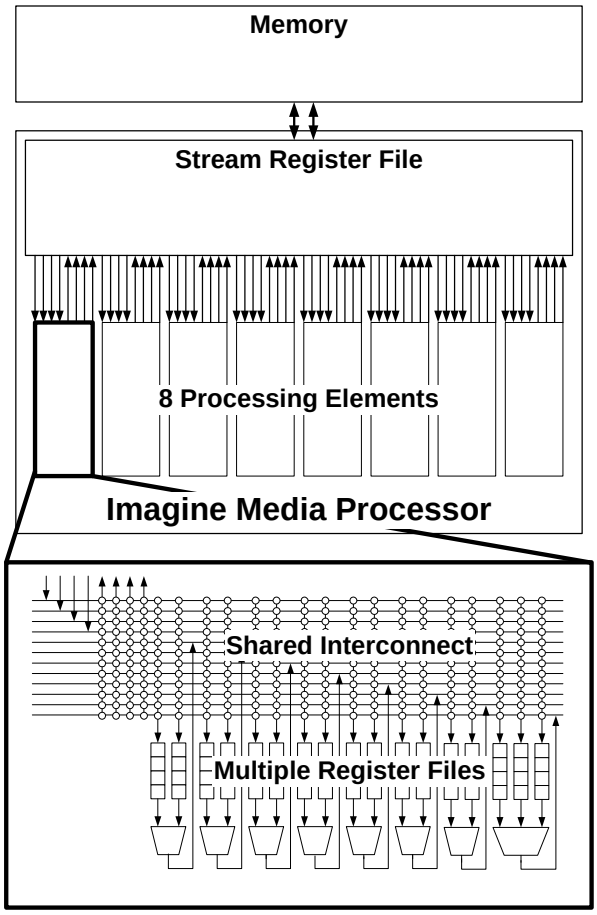



**FIGURE 1-2. Simplified diagram of an Imagine processor**

These architectural innovations place additional burdens on the compiler. Distributed register files require the compiler to allocate the shared buses and register file ports and to manage the movement of data between the multiple register files. A stream register file requires the compiler to explicitly allocate space in the stream register file to hold streams and manage the loading and storing of streams.

This thesis presents a programming system for the Imagine Media Processor that consists two C-like languages called KernelC and StreamC that implement the stream program-

ming model, and two compilers, one for each language. The two compilers introduce new techniques to handle Imagine's architectural innovations.

The stream programming model divides a media processing application into one or more kernels that define each processing step in the application, and a stream program that defines the high-level control- and data- flow between kernels. A kernel, written in KernelC, is a function that operates on streams. Internally, a kernel usually consists of a computation-intensive loop that iterates over the records in the input stream(s) and produces the records in the output stream(s). The stream program, written using StreamC in combination with C++, consists of a series of operations performed on streams, the most common of which are kernels. Figure 1-3 graphically depicts a simplified version of a polygon rendering application written using the stream programming model. The first kernel takes a stream of triangles as input and produces a stream of spans (lines of pixels) as output, the second kernel takes the stream of spans as input and produces a stream of fragments (pixels) as output.

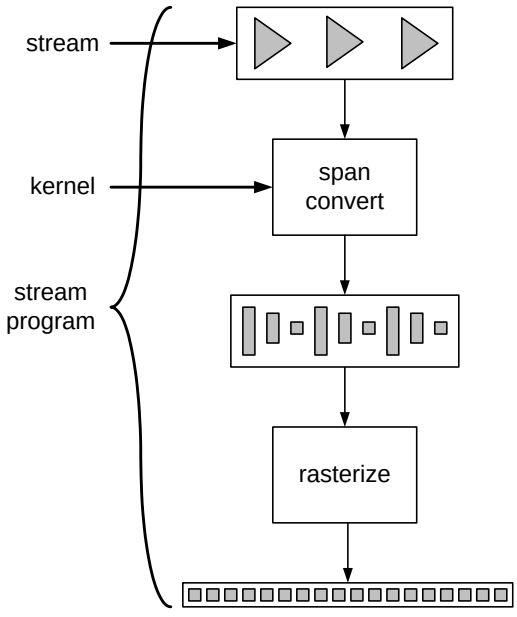



**FIGURE 1-3. Simplified polygon rendering using stream programming model**

The KernelC compiler introduces *communication scheduling* to allocate shared interconnect resources and manage data movement between multiple register files and functional units. With a single register file, all functional units can access all data. With multiple register files this is no longer the case. Communication scheduling ensures that the result of each operation is available to the operations that use that result. It assigns each *communication*, the logical transfer of a value from the operation that computed it to an operation that uses it, to a *route* that defines the resources used to move the value between the functional units that perform the operations, as depicted in Figure 1-4. Communication scheduling composes each route from three components: a *write stub* that defines how the result is written to an initial register file, zero or more *copy operations* to move the value between register files, and a *read stub* that defines how the operand is read from the final register file.

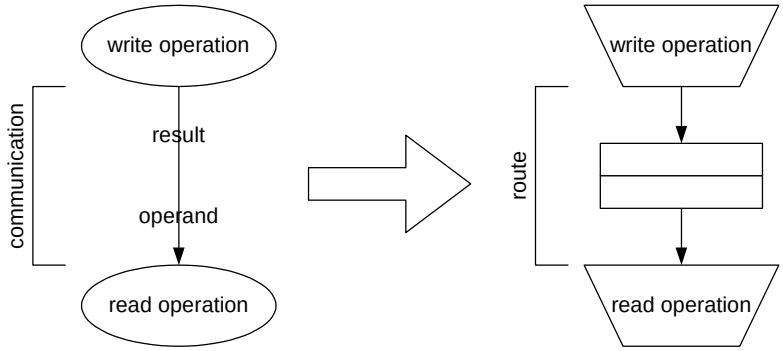



**FIGURE 1-4. Communication scheduling assigns each communication to a route**

The StreamC compiler introduces stream scheduling to allocate the SRF and determine when to load and store streams. It assigns each stream access to a buffer in the SRF as depicted in Figure 1-5. Stream scheduling attempts to minimize memory traffic and maximize parallelism of kernels and memory accesses. Ideally, a stream is kept in the SRF between accesses, reducing memory traffic. When a stream must be loaded or stored, it is allocated a buffer that is disjoint from the buffers used by nearby kernels, enabling the memory access to occur in parallel with those kernels. Stream scheduling allocates the SRF as a two dimensional space. One dimension of this space is the SRF address space,

the other dimension is time, the duration of the stream program. It assigns all stream accesses in the stream program to rectangular buffers with appropriate size (width) and duration (height), then tries to position all of the buffers in the two dimensional space. If all of the buffers do not fit, stream scheduling reduces the size or duration of a buffer and then tries again until it succeeds.

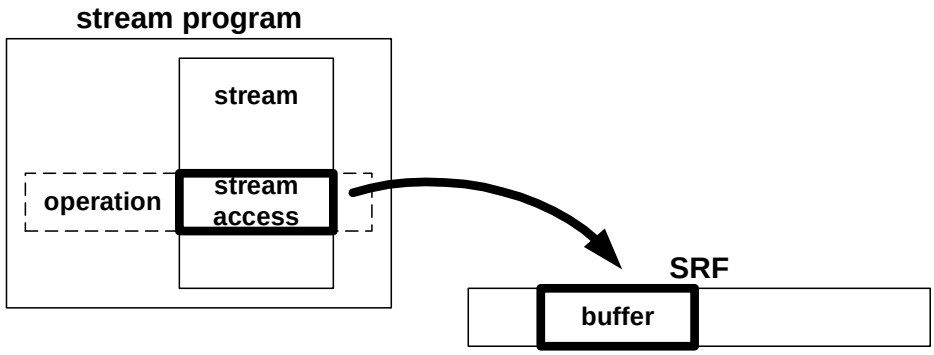



**FIGURE 1-5. Stream scheduling assigns each stream access to a buffer in the SRF**

Experimental results demonstrate that this programming system can be used to implement sophisticated, high-performance applications including stereo depth extraction, MPEG2 encoding, and polygon rendering for the Imagine Media Processor. For a set of benchmarks that includes these applications, communication scheduling delivers performance with shared interconnect and multiple partitioned register files that is comparable to one multi-ported register file and stream scheduling manages the SRF as well as or better than experienced Imagine programmers can by hand.

## 1.2 Contributions

The main contributions of this thesis are:

1. An implementation of the stream programming model which introduces the StreamC and KernelC languages to enable efficient development of stream programs and kernels. This implementation allows programmers to write high-performance applications for stream processors like Imagine without detailed knowledge of the target architecture and facilitates high-level optimization.

2. Communication scheduling, a compiler technique for allocating shared interconnect between functional units and multiple register files. Communications scheduling has several key innovations and supporting optimizations:
   - use of *stubs*, partial routes between functional units, to allow independent assignment of operations to functional units
   - a method for incrementally composing routes from stubs during the scheduling process
   - *communication cost*, a heuristic component for assigning operations to functional units that reflects the impact of copy operations on schedule length
   - heuristics for ordering operations and assigning them to functional units tailored to wide VLIWs with shared interconnect, including the use of a random component to explore several possible schedules

   Communication scheduling can be incorporated into a variety of VLIW scheduling algorithms, extending them to a large class of architectures that includes the very efficient distributed register file architecture.

3. Stream scheduling, a compiler technique for allocating a stream register file and managing the loading and storing of streams. Stream scheduling has several key innovations and supporting optimizations including:
   - a unique allocation process that combines compile-time allocation of on-chip memory with spilling and double-buffering
   - application of data-flow analysis to streams of records
   - use of a profile of the stream program to allow efficient allocation of on-chip memory

- a method for estimating batch size when stripmining stream programs
- a software-pipelining algorithm for covering sequential memory latency in stream programs

Stream scheduling combines the performance benefits of managing on-chip memory explicitly with much of the ease of an implicit on-chip memory like a cache.

## 1.3 Thesis Roadmap

This thesis is organized into a series of chapters that present background material, describe the components of the Imagine programming system, and evaluate their performance. Each of these chapters is briefly summarized in the remainder of this section.

Chapter 2 presents background on VLIW scheduling, streams, and the Imagine Media Processor. In particular, it discusses the limitations of prior VLIW scheduling techniques for multiple register file architectures, describes prior uses of the concept of streams, and provides an overview of the Imagine architecture.

Chapter 3 introduces an implementation of the stream programming model. It describes how the stream programming model divides a media processing application into a stream program and one or more kernels. It provides an overview of StreamC, a programming language extension used in combination with C++ to write stream programs, and KernelC, a C-like language used to write kernels.

Chapter 4 introduces communication scheduling, and describes how it allocates shared interconnect resources and manages data transfers between multiple register files and functional units. Chapter 5 presents the KernelC compiler, which uses communication scheduling to compile KernelC for Imagine.

Chapter 6 introduces stream scheduling, and describes how it allocates space in a stream register file and manages the loading and storing of streams for applications written using the stream programming model. Chapter 7 presents the StreamC compiler, which uses stream scheduling to compile StreamC for Imagine.

Chapter 8 presents a quantitative evaluation of the KernelC and StreamC compilers, with emphasis on communication scheduling and stream scheduling. It describes a testing methodology and a set of benchmarks for each compiler, then presents and analyzes the results obtained by applying the methodology to the benchmarks.

Finally, Chapter 9 summarizes this thesis and discusses future areas of exploration.

# Chapter 2

# Background

This chapter presents background for the programming system described in this thesis. It provides an overview of VLIW scheduling algorithms, and discusses the limitations of existing algorithms when applied to architectures with multiple register files and shared interconnect. It describes prior uses of the concept of streams, which have primarily involved adding hardware to accelerate memory access. Lastly, it presents an overview of the Imagine Media Processor Architecture.

## 2.1 VLIW Scheduling

A very long instruction word (VLIW) scheduler takes a set of operations and produces a schedule that specifies which operations to issue to which functional unit on a given cycle. The key problems in VLIW scheduling are finding enough parallel operations, and scheduling those operations to occur on a particular functional unit on a particular cycle in a way that effectively utilizes this parallelism. There are two approaches to increasing the number of parallel operations: expand the size of the region of operations that can be scheduled beyond a basic block, and transform the operations within a basic block to increase parallelism. The first approach is used by trace scheduling [15], which schedules a series of basic blocks that are likely to occur in sequence, and superblock scheduling [21], which schedules multiple basic blocks with a single entry point but multiple exit points. The second approach is used by loop unrolling [29], which duplicates a loop body allowing multiple iterations to overlap and software pipelining [28][41], which divides a

loop into stages and then overlays the stages so that successive iterations can occur simultaneously.

The second key problem, assigning operations to functional units and scheduling them to be issued as part of a particular cycle, is NP-complete. A variety of heuristic VLIW scheduling methods have been developed. Most algorithms, such as Bottom-Up-Greedy (BUG) perform these tasks using separate phases [5][10][12][32][38]. Typically, each operation is first assigned to a functional unit. Operations are then assigned to cycles using a top-down or bottom-up traversal of an acyclic version of the data dependency graph. The earliest cycle on which an operation can be issued is the cycle after the last operation it is dependent on completes (or visa versa in the case of a bottom-up traversal). However, the multi-phase approach either imposes constraints on the scheduling process by preassigning operations to functional units, or visa versa. For instance, two operations may be assigned to the same functional unit ahead of time. During scheduling, it may be desirable to schedule both operations on the same cycle. Under a multi-phase approach this is not possible even if there is more than one functional unit available. In contrast, Unified Assign and Schedule (UAS) [40] assigns operations to functional units and cycles in a single phase. UAS attempts to assign each operation to the earliest possible cycle. If an appropriate functional unit is available on that cycle then UAS assigns the operation to that functional unit. Otherwise, it delays it until the next earliest cycle, and so on.

Prior VLIW scheduling algorithms have concentrated on architectures with either a single register file architecture or a clustered register file architecture. In a single register file architecture, all functional units are connected to the same register file by dedicated interconnect. Every functional unit always reads from and writes to the same register file, so assigning operations to functional units is sufficient. In a clustered register file architecture, functional units are grouped into clusters and all functional units in a cluster are connected to the same register file by dedicated interconnect. Values can be transferred between cluster register files across global buses by means of copy operations. For these architectures, VLIW scheduling also needs to add and schedule a copy operation when the operation that computes a value and an operation that uses it are assigned to functional

units in different clusters. UAS and the multiphase algorithm presented in [38] both target clustered register file architectures.

Some scheduling algorithms target specific architectures that take incremental steps beyond a clustered register file architecture. The polycyclic compiler [42] targets an architecture that allows functional units to read from/write to multiple register files, but provides a dedicated register file between every functional unit output and every functional unit input to avoid resource conflicts. The Cydra5 [9] compiler targets architectures in which each functional unit input can read from multiple register files, but provides each input with a dedicated bus and a dedicated register file port to access each register file. Both the polycyclic and Cydra5 architectures allow the compiler to consider only functional units when scheduling operations, with all shared interconnect allocated implicitly. The TMS320C6x compiler [48] targets an architecture with two cluster register files that includes a small number of cross-cluster buses, but which still provides each functional unit input and output with a dedicated bus and register file port to access its cluster register file. This architecture guarantees a conflict-free way to read operands and write results, allowing the compiler to use a version of the slack scheduling algorithm for clustered architectures presented in [20] that is modified to be single-phase for better performance. Distributed modulo scheduling [14], another single-phase scheduling algorithm, targets a clustered register file architecture that places special communication queue register files between adjacent clusters. (“distributed” does not refer to the distributed register file architecture used by Imagine). It uses a modified modulo scheduling algorithm that tries to schedule communicating operations on the same or adjacent clusters so that it can use these communication queue register files to avoid the need for copy operations. It adds copy operations if it unable to schedule communicating operations on adjacent clusters, and backtracks if all else fails. The Multiflow compiler [32] targets architectures with some shared interconnect using a multiphase algorithm based on BUG that assigns operations to functional units before scheduling.

A few scheduling algorithms address unique architectures that include shared interconnect but add special purpose hardware to handle conflicts. Transport-triggered architectures

(TTA) [19] place a special register (not a register file) at each input and output of functional unit, then connects these special registers with shared buses. The TTA compiler resolves conflicts by delaying a result in the register at the functional unit output as long as necessary, which stalls other operations in that functional unit's pipeline. The RAW machine [31] consists of a grid of connected tiles each containing one functional unit. It uses small routers between tiles to handle conflicts.

## 2.2 Streams

A stream is a sequence of data records defined by a regular access pattern. For example, the simplest access pattern is constant stride, in which records are separated by a fixed amount. Though some work on automatically converting regular access patterns into streams has been done [3], most implementations require manual specification of streams in the program [2][6][35].

Several hardware optimizations have taken advantage of the concept of streams. At a hardware level streams offer the potential to hide latency by allowing data to be loaded in advance of execution, optimize memory access patterns by reordering accesses, compact data within a cache, and enable higher bandwidth access to on chip-data. The WM architecture [3] separates memory access from computation by routing all loads and stores through FIFOs, then supports stream accesses that load or store all records in a stream to or from a FIFO. The Stream Memory Controller [35] uses FIFO stream buffers in parallel with a cache. The records that compose a stream can be sequentially read from or written to these buffers, bypassing the cache. The Impulse Memory Controller [6] implements two optimizations based on streams that work with a conventional cache. First, it adds an extra stage to address translation that remaps the records in a stream into a sequential series of addresses in unused address space, allowing it to be stored as compactly as possible in the cache. Second, it uses the stream's access pattern to perform prefetching. The DataStreamer [2] in the Equator MAP-CA architecture allows the programmer to load or store a block of data and into or out of the cache independent of the main thread of execution.

## 2.3 Imagine Media Processor

The Imagine Media Processor is designed to process streams [25][43][45]. Imagine works in conjunction with a conventional host processor that executes a scalar application and sends operations to Imagine. Imagine, shown in Figure 2-1, consists of five major components. The stream controller/host interface, numbered 1 in Figure 2-1, receives operations from the host and issues them to the components of Imagine. It also transfers streams between Imagine and the host. The stream register file, numbered 2 in Figure 2-1, contains the current working set of streams. The memory system, numbered 3 in Figure 2-1, loads and stores streams to and from off-chip memory. The processor core, numbered 4 in Figure 2-1, processes streams. Each of these four components is described in more detail below. The fifth component, the network interface, sends and receives streams to and from a high-speed network. It is not shown and is outside the scope of this thesis.

### 2.3.1 Stream controller/host interface

The stream controller/host interface receives *Imagine operations* from the host and issues them to the components of Imagine. Imagine operations include loading, storing, or transferring a stream, executing a kernel (a small program that has streams as inputs and outputs), and reading or writing a control register such as an SDR or MAR (see below). The stream controller contains an operation buffer into which the host processor can write Imagine operations and information about dependencies between operations. The stream controller then issues these operations to the appropriate Imagine component. It can issue the operations out-of-order, subject to dependency constraints. The stream controller/host interface also transfers streams between the host processor and Imagine.

### 2.3.2 Stream register file

The stream register file (SRF) contains the current working set of streams. The SRF is a very wide single-ported memory. Despite being single-ported, it effectively supports simultaneous access to different streams by multiple clients by writing or reading a very wide word containing a portion of the stream into or out of a buffer dedicated to a particular client on each cycle. The client then writes or reads data into or out of the buffer at a lower granularity as often as every cycle. The location and length of a stream in the SRF is

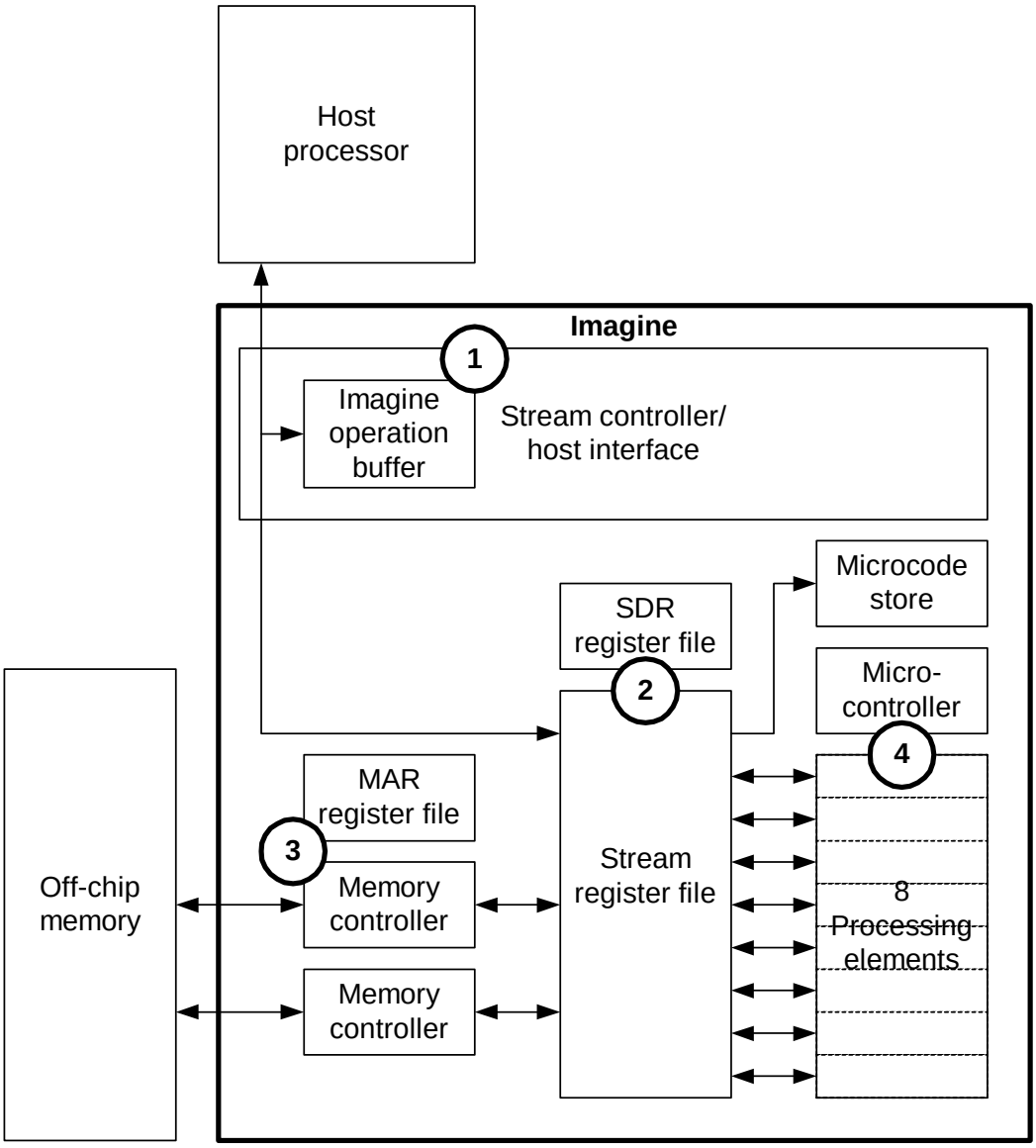



**FIGURE 2-1. Imagine Media Processor**

stored in a stream descriptor register (SDR) in the SDR register file. Imagine operations specify which streams they operate on by referring to the appropriate SDR.

### 2.3.3 Memory system

The memory system loads and stores streams to and from off-chip memory. The memory system consists of the Memory Access Register (MAR) register file and two memory con-

trollers. Each MAR contains the start address in memory and access pattern for a stream. Each memory controller is capable of loading or storing a stream between a location in the SRF described by an SDR and a location in memory described by an MAR.

### 2.3.4 Processor core

Imagine's processor core executes small programs called kernels on eight identical processing elements. It consists of the eight SIMD processing elements, the microcontroller, and the microcode store. The eight processing elements operate in a SIMD fashion; each processing element executes the same set of operations each cycle. However, the processing elements are physically distinct and operate on full word data types. Each processing element contains eight functional units and multiple local register files. The ALUs within a processing element also support segmented operations on 16-bit or 8-bit data. Since the processing elements operate in SIMD fashion, the single microcontroller can decode and issue instructions to all eight processing elements. The microcontroller reads instructions from the on-chip microcode store. It also contains a small register file that holds special microcontroller variables used to pass arguments between the host processor and Imagine.

## 2.4 Summary

This chapter presented background for the programming system described in this thesis. First, it provided an overview of VLIW scheduling algorithms and discussed the limitations of the register file architectures these algorithms target, and some common optimizations. Next, it summarized the concept of streams and the hardware optimizations that have been developed to take advantage of streams. Lastly, it presented an overview of the major components of Imagine Media Processor: the stream controller/host interface, stream register file, memory system, and processor core.

# Chapter 3

# Stream Programming Model

This chapter presents an implementation of the stream programming model [43], a programming model for media processors. General purpose programming languages are not well suited to media processing applications. They emphasize expressiveness and flexibility, allowing development of a wide range of applications. However, their lack of a consistent structure obscures the high-level data flow and memory access patterns, which makes high-level optimizations difficult. Their abundant low-level control divides programs into many small basic blocks, limiting instruction level parallelism. This implementation of the stream programming model provides a consistent structure that is specialized for media processing applications. This structure makes high-level data flow and memory access patterns explicit. It also limits low-level control flow. The stream programming model divides a media processing application into a *stream program* that specifies the high-level structure of the application and one or more *kernels* that define each processing step. Each kernel is a function that operates on *streams*, sequences of records. This chapter describes StreamC, a set of classes and functions used in combination with C++ to write stream programs, and KernelC, a simple programming language used to write kernels. The steam program is executed on a host processor; the computation intensive kernels are executed on Imagine.

This chapter consists of three sections. Section 3.1 presents an overview of the stream programming model. Section 3.2 presents KernelC. Section 3.3 presents StreamC.

## 3.1 Overview

The stream programming model divides a media processing application into one or more *kernels* and a *stream program* as depicted in Figure 3-1. A kernel is a computation intensive function that operates on sequences of records called *streams*. Each kernel takes streams of records as input and produces streams of records as output. Kernels are written using a C-like language called KernelC. The *stream program* declares the streams and defines the high-level control- and data-flow between kernels. Stream programs are written using a programming language extension called StreamC intermixed with C++.

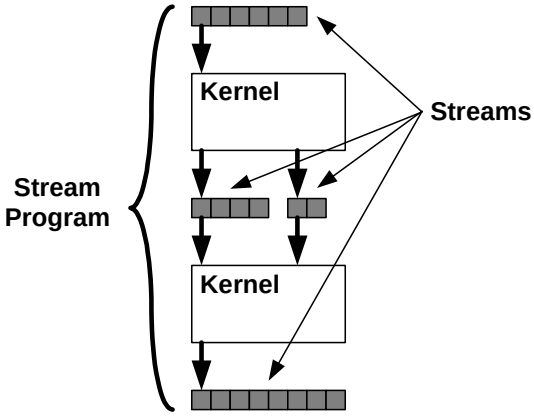



**FIGURE 3-1. Stream programming model**

Figure 1-3 graphically depicts a simplified polygon rendering application written using the stream programming model. The application is structured as a stream program that declares three streams: triangles, spans (lines of pixels), and fragments (pixels), and calls two kernels. The first kernel takes a stream of triangles as input and produces a stream of spans as output, the second kernel takes the stream of spans produced by the first kernel as input and produces a stream of fragments as output.

The implementation of the stream programming model described in this chapter can be thought of as a code transformation on programs that consist of a series of loops that process arrays of records. The access pattern of each loop with respect to each array is

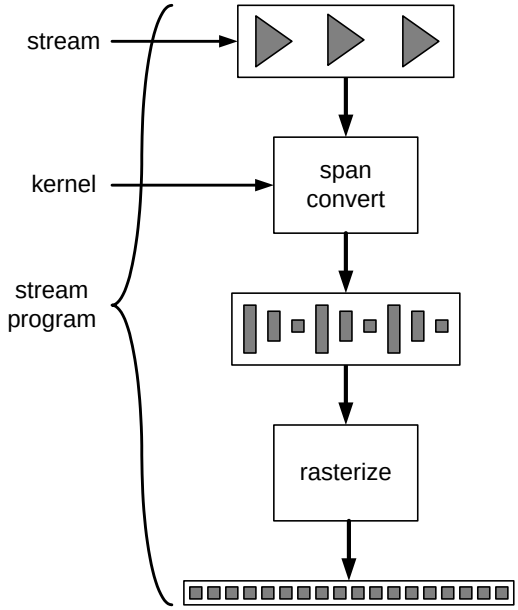



**FIGURE 3-2. Simplified polygon rendering using stream programming model**

extracted into one or more streams, and the computation performed by each loop is encapsulated inside a kernel. The remaining code composes the stream program. In reality, most applications need to be restructured to make efficient use of the stream programming model but this code transformation serves as a useful starting point.

Figure 3-3 shows an example of example of a conventional program and the corresponding stream program and kernel. The conventional code consists of a loop that reads all records from the arrays *a* and *b* and writes the even records in the array *c*. The corresponding stream program declares four streams: three that correspond to the three arrays and a fourth, *cEven*, that refers to the even records in the stream *c*. The stream *cEven* is specified as a subset of the stream *c* that starts with record 0 and ends at record 512, with a stride, or fixed interval between records, of 2. After declaring the streams, the stream program then calls a kernel that processes the streams *a* and *b* to produce the stream *cEven*. Scalar arguments to the kernel such as *uc_amul* and *uc_bmul* are encapsulated as microcontroller (“uc”) variables. The kernel is declared as taking two input streams (“istreams”), one output stream (“ostream”) and two microcontroller variables as arguments. If first reads the values of the microcontroller variables, then loops over the records in the input streams

computing records in the output stream. This stream program and kernel are explained in detail in the remainder of this chapter.

**Conventional program:**

```
void main()
{
  int a[256];
  int b[256];
  int c[512];
  int amul = 2;
  int bmul = 3;
  ...
  for (int i = 0; i < 256; i++) {
    if (a[i] > 0) {
      c[i * 2] = a[i] * amul;
    } else {
      c[i * 2] = b[i] * bmul;
    }
  }
  ...
}
```

**Stream Program:**

```
void main()
{
  stream<int> a(256);
  stream<int> b(256);
  stream<int> c(512);
  stream<int> cEven =
    c(0, 512, FIXED, STRIDE, 2)
  uc<int> uc_amul = 2;
  uc<int> uc_bmul = 3;
  ...
  example1(a, b, cEven,
           uc_amul, uc_bmul);
  ...
}
```

**Kernel:**

```
KERNEL example1(istream<int> a,
                istream<int> b,
                ostream<int> c,
                uc<int> uc_amul,
                uc<int> uc_bmul)
{
  int amul = ucRead(uc_amul);
  int bmul = ucRead(uc_bmul);
  loop_stream(a) {
    int ai, bi, ci;
    a >> ai;
    b >> bi;
    ci = select(ai > 0,
      ai * amul,
      bi * bmul);
    c << ci;
  }
}
```

**FIGURE 3-3. Code transformation to stream programming model**

## 3.2 KernelC

Kernels are written using a language called KernelC, which uses a limited C-like syntax. Figure 3-4 gives an abbreviated definition of KernelC. The purpose of this definition is to summarize the language; it includes minor changes to the actual syntax and omits some non-essential details. KernelC is more restrictive than C. It does not allow global variables, pointers, function calls, or control-flow constructs other than loops. However, it can be compiled more efficiently and retains considerable flexibility. KernelC has four important features:

- **Structured data access:** KernelC only allows global data to be accessed through special arguments passed to the kernel.
- **Limited control flow:** KernelC only allows loops, but supports conditional assignments, stream reads, and stream writes.
- **Packed data types and DSP math operators:** KernelC supports several additional data types and math operators useful for digital signal processing.
- **SIMD processing support:** KernelC supports multiple SIMD processing elements.

The remainder of this section discusses each of these features in detail.

### 3.2.1 Structured data access

KernelC only allows access to global data through arguments to the kernel. All arguments to the kernel are passed by reference. Writing a record in an output stream or setting the value of a microcontroller variable changes that record or variable outside of the kernel. A kernel takes a small number of input streams (“istream”) and output stream (“ostream”) arguments. The kernel sequentially reads records from the input streams and sequentially writes records to the output streams using the << and >> operators, respectively. Kernels also can take special microcontroller (“uc”) variables as arguments, which encapsulate scalar values. This encapsulation is required because the variables are passed from the stream program running on the host to the microcontroller that controls execution of the kernel on Imagine. The kernel reads and writes the microcontroller variables using the ucRead and ucWrite functions.

```
basic-type:
  int
  unsigned int
  half2
  unsigned half2
  byte4
  unsigned byte4
  float


record-type-definition:
  RECORD record-type { field-definition, ... };

field-definition:
  complex-type id

complex-type:
  basic-type
  record-type


kernel-definition:
  KERNEL identifier ( argument, ... ) { statement ... }

argument:
  istream<complex-type> istream-id
  ostream<complex-type> ostream-id
  uc<basic-type> uc-id

statement:
  declaration
  assignment
  input
  output
  loop

declaration:
  basic-type id;
  basic-type id = expression;
  record-type id;
  array<basic-type> array-id(constant);
  uc<basic-type> uc-id;
  uc<basic-type> uc-id = constant;
```

```
assignment:
  lvalue = expression;
  uc-id = ucWrite(PE-index, expression);

lvalue:
  id
  id.id
  array-id[expression]

expression:
  rvalue
  math-expression
  permute(PE-permutation, expression)
  select(expression, expression, expression)
  ucRead(uc-id)

rvalue:
  constant
  id
  id.id
  array-id[expression]

math-expression:
  unary-operator id
  id binary-operator id
  unary-operation(id)
  binary-operation(id, id)


input:
  istream-id >> lvalue;
  istream-id(expression, lvalue) >> lvalue;


output:
  ostream-id << expression;
  ostream-id(expression) << expression;


loop:
  loop-count (uc-id) { statements }
  loop-whileany (expression) { statements }
  loop-whileall (expression) { statements }
  loop-untilany (expression) { statements }
  loop-untilall (expression) { statements }
  loop-stream (istream-id) { statements }
```

**FIGURE 3-4. Abbreviated definition of KernelC**

Figure 3-5 highlights the data access in the example kernel introduced in Figure 3-3. The example takes five arguments: two input streams, *a* and *b*, an output stream, *c*, and two microcontroller variable arguments, *uc_amul* and *uc_bmul*. It reads the two microcontroller variables, *uc_amul* and *uc_bmul,* into local variables which it uses repeatedly in the main loop. On each iteration of the main loop, it reads a record from the input streams *a* and *b* and writes a record to the output stream *c*. This kernel loops over the stream *a*, and assumes that the streams *b* and *c* are the same length.

```
KERNEL example1(istream<int> a,
                istream<int> b,
                ostream<int> c,
                uc<int> uc_amul,
                uc<int> uc_bmul)
{
  int amul = ucRead(uc_amul);
  int bmul = ucRead(uc_bmul);
  loop_stream(a) {
    int ai, bi, ci;
    a >> ai;
    b >> bi;
    ci = select(ai > 0,
      ai * amul,
      bi * bmul);
    c << ci;
  }
}
```

**FIGURE 3-5. Example with data access highlighted**

### 3.2.2 Limited control flow

KernelC limits control flow in order to maximize instruction level parallelism. The only control flow that KernelC allows are loops. In addition to count, while, and until loops, KernelC introduces a new looping construct “loop_stream(*input stream*)” that iterates until all records have been read from the specified input stream. Instead of if-statements, KernelC supports conditional assignment using the “select” function (similar to the C ?: operator) to choose between two values based on a condition. For instance, “select(cond, x, y)” returns the value of *x* if *cond* is true or *y* if *cond* is false. It also supports conditional reads from streams and conditional writes to streams [23]. For instance, “ostream1(cond) << x” only writes the value of *x* to *ostream1* if *cond* is true.

Figure 3-6 highlights the limited control flow in the example kernel. It loops over the input stream *a* until all records in the stream have been read. It uses a select statement to determine the output each iteration. If *ai* is greater than 0, it outputs *ai* multiplied by *amul*, otherwise it outputs *bi* multiplied by *bmul*.

```
KERNEL example1(istream<int> a,
                istream<int> b,
                ostream<int> c,
                uc<int> uc_amul,
                uc<int> uc_bmul)
{
  int amul = ucRead(uc_amul);
  int bmul = ucRead(uc_bmul);
  loop_stream(a) {
    int ai, bi, ci;
    a >> ai;
    b >> bi;
    ci = select(ai > 0,
      ai * amul,
      bi * bmul);
    c << ci;
  }
}
```

**FIGURE 3-6. Example with limited control flow highlighted**

### 3.2.3 Additional data types and math operators

KernelC adds packed data types and DSP math operations. KernelC includes two new packed data types: *(unsigned) byte4*, four bytes packed into one 32-bit word, and *(unsigned) half2*, two 16-bit half-words packed into one 32-bit word. Operations performed on these packed data types affect each of the packed components in a SIMD fashion. KernelC also includes various mathematical operations that are useful for signal- and image processing such as saturating add and subtract.

The simple kernel shown in Figure 3-7 uses packed data types and saturating addition to brighten an 8-bit grayscale image. The single saturating add operation increments four 8-bit pixel values by the value of *mod* (which, in the context of this kernel, must contain the same value in each byte.)

```
KERNEL brighten(istream<unsigned byte4> in,
                ostream<unsigned byte4> out,
                uc<unsigned byte4> uc_mod)
{
  unsigned byte4 mod = ucRead(uc_mod);
  loop_stream(in) {
    unsigned byte4 ini, outi;
    in >> ini;
    outi = addsat(ini, mod);
    out << outi;
}
```

**FIGURE 3-7. Kernel to brighten an 8-bit grayscale image**

### 3.2.4 SIMD processing support

KernelC supports Imagine's eight SIMD processing elements. All eight processing elements execute a kernel in parallel. Every cycle, all processing elements execute the same operations on different data. Each processing element reads every eighth record from the input stream(s) and writes every eighth record to the output stream(s). For example, processing element 3 reads or writes records 3, 11, 19, etc. from or to each stream.

The *permute* operation is used to explicitly interchanges values between processing elements. The permute operation takes a constant describing the permutation and the value to be permuted as arguments. The permutation is an integer in which the *nth* nibble specifies the index of the processing element from which the *nth* processing element gets the value of the permuted variable. For instance, "permute(0x65432107, x)", rotates the value of *x* in each of the processing elements to the left by specifying that processing element 7 gets the value from processing element 6, and so on.

Microcontroller variables, which encapsulate scalar arguments passed from the host processor to the microcontroller, are not replicated across Imagine's eight processing elements. The ucWrite function, which is used to write a value from one of the processing elements to a microcontroller variable takes two arguments, the index of the processing element to read the value from, and the value. For instance, "uc_x = ucWrite(0, x)" writes the value of *x* in processing element 0 to the microcontroller variable *uc_x*.

The simple kernel shown in Figure 3-8 computes the sum of a stream of integers. First, it loops through the stream and sums all of the integers processed by each processing element. Next, it adds the totals in the processing elements together in a tree-like fashion. The kernel adds the total in each processing element to the total one processing element to the left, then adds that total to the total two processing elements to the left, then adds that total to the total four processing elements to the left. Lastly, it writes the final total in processing element 0 to a microcontroller variable.

```
KERNEL sumStream(istream<int> in,
                 uc<int> uc_total)
{
  int total = 0;
  loop_stream(in) {
    int ini;
    in >> ini;
    total = total + ini;
  }
  total = total + permute(LROTATE, total);
  total = total + permute(LROTATE2, total);
  total = total + permute(LROTATE4, total);
  uc_total = ucWrite(0, total);
}
```

**FIGURE 3-8. Kernel to sum a stream of integers**

## 3.3 StreamC

Stream programs are written using a small number of classes and functions collectively called StreamC that are intermixed with arbitrary C++. The StreamC functions, which consist of kernel calls and functions to copy and transfer streams, are compiled for execution on Imagine by the StreamC compiler as described in Chapter 6. StreamC includes the following components:

- stream class
- microcontroller variable class
- kernels exposed as functions
- functions for copying streams and transferring data between streams and arrays
- conventions for specifying data-dependent streams and control flow

Figure 3-9 shows the C++ declarations for these components. The StreamC presented in Figure 3-9 is a cleaner version of the actual implementation of StreamC and is used throughout this thesis for clarity and as a basis for future implementations. Each of these components is discussed in more detail in the remainder of this section.

```
// 1. Streams

enum AccessPatternEnum { SEQUENTIAL, STRIDED, INDEXED };

template<class type>
class stream {
 public:
  stream()
  // basic stream
  stream(int _size,
    DataDependenceEnum _dataDependence = FIXED);

  // sequential derived stream (default)
  stream<type> operator()(
    int _start, int _end, DataDependenceEnum _dataDependence = FIXED,
    AccessPatternEnum _accessPattern = SEQUENTIAL,
    int _recordLengthOverride = 1);

  // strided derived stream
  stream<type> operator()(
    int _start, int _end, DataDependenceEnum _dataDependence,
    AccessPatternEnum _accessPattern, int _stride,
    int _recordLengthOverride = 1);

  // indexed derived stream
  stream<type> operator()(
    int _start, int _end, DataDependenceEnum _dataDependence,
    AccessPatternEnum _accessPattern, stream<int> _index,
    int _recordLengthOverride = 1);

   int getLength();
};

// 2. Microcontroller variables

template<class type> class uc {
 public:
  uc<type>()
  uc<type> operator=(const type rvalue);
}

template<class type> type ucRead(uc<type>& uc);

// 3. Kernels

// kernels exposed as C++ functions, varies with application
// example:
void example1(istream<int>& a, istream<int>& b, ostream<int>& c,
  uc<int>& uc_amul, uc<int>& uc_bmul);
```

```
// 4. Copies and transfers

template<class type>
void streamCopy(stream<type>& streamFrom, stream<type>& streamTo);

template<class type>
void streamLoadBin(stream<type>& stream, type* array, int length);

template<class type>
void streamSaveBin(stream<type>& stream, type* array);


// 5. Data-dependence annotations

enum DataDependenceEnum { FIXED, VARIABLE_LENGTH, VARIABLE_BOUNDS };

// macros used to note entry and exit into/out of control-flow
// that depends on the data being processed
#define if_VARIABLE ...
#define while_VARIABLE ...
#define for_VARIABLE ...
```

**FIGURE 3-9. C++ declarations for StreamC components**

### 3.3.1 Streams

The stream class can be used for two kinds of streams, basic streams and derived streams. A *basic stream* is an array of records. A basic stream is defined by a *size* as shown in Figure 3-10. A *derived stream* is a reference to a subset of the records in a basic stream. A derived stream defined by a basic stream and a *start*, *end*, and *access pattern* within that stream as shown in Figure 3-11. The start is the index of the first record in the stream. The end is the index of the record after the last record that could be in the stream. The access pattern defines which records between the start and the end are in the stream. Both stream definitions also include a data dependence parameter. The data dependence parameter is used to annotate streams with characteristics that vary depending on the data being processed. It is described in Section 3.3.5.

```
stream<type> name(size, dataDependence);
```

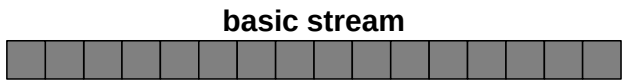



**FIGURE 3-10. A basic stream is an array of records**

```
stream<type> name = basic-stream(start, end, dataDependence, access-
pattern);
```

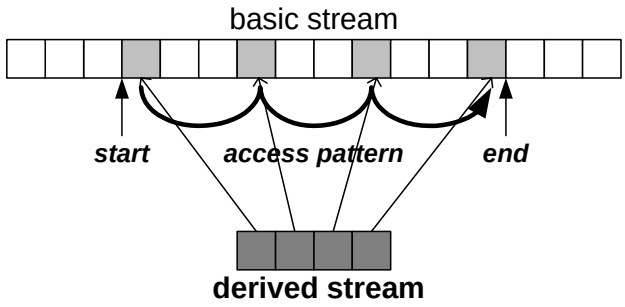



**FIGURE 3-11. A derived stream is a subset of the records in a basic stream defined by a start, end, and access pattern**

Conceptually, an access pattern can define any series of records. In reality, only a few access patterns are useful enough to be supported in hardware. The three most common access patterns are: *sequential*, *strided*, and *indexed*, depicted in Figure 3-12 through Figure 3-14. A sequential stream refers to all records from the *start* record up to, but not including, the end record. A strided stream refers to records separated by a constant offset called the *stride*. It refers to every stride*th* record from the start record up to the end record (i.e. start, start + stride, start + 2*stride, etc.). An indexed stream accesses records at offsets given by a stream of integers called the *index stream*. An indexed stream includes one record for each value in the index stream, at an offset from the start equal to that value (i.e. start + first index, start + second index, etc.). Since the effective access pattern of an indexed stream depends entirely on the index values it can provide any desired access pattern.

The example stream program introduced in Figure 3-3 contains three basic stream declarations and one derived stream declaration, all highlighted in Figure 3-15. The basic streams *a* and *b* contain 256 records. The basic stream *c* contains 512 records. The derived stream *cEven* refers to the even records in the basic stream *c*. It is specified with a start of 0, an end of 512, and a strided access pattern with a stride of 2. This derived stream replaces a conventional array access in which a loop index is multiplied by two.

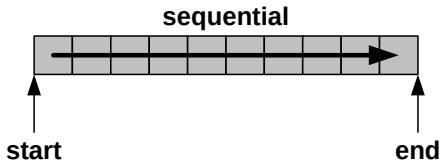



**FIGURE 3-12. Sequential access pattern includes every record**

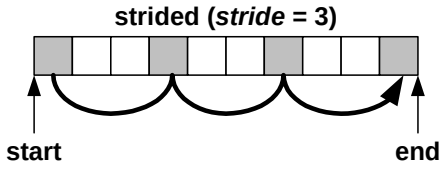



**FIGURE 3-13. Strided access pattern includes every *stride*th record**

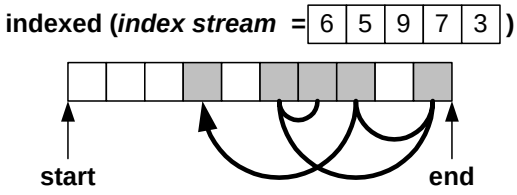



**FIGURE 3-14. Indexed access pattern includes records with positions given by index stream**

```
void main()
{
  stream<int> a(256);
  stream<int> b(256);
  stream<int> c(512);
  stream<int> cEven =
    c(0, 512, FIXED, STRIDE, 2)
  uc<int> uc_amul = 2;
  uc<int> uc_bmul = 3;
  ...
  example1(a, b, cEven,
           uc_amul, uc_bmul);
  ...
}
```

**FIGURE 3-15. Example with streams highlighted**

A derived stream can also be defined in terms of another derived stream. Internally, the start, end, and stride of the new derived stream are recalculated in terms of the underlying

basic stream. As a simple example, if a new stream with a start of 5 is defined in terms of an old stream with a start of 10, it actually has a start of 15 in terms of the underlying basic stream. More specifically, the start of the new stream is multiplied by the stride of the old derived stream since each record in the old stream is separated by the stride, then added to the start of the old stream since the first record in the old stream is offset by that amount. To avoid multiple levels of indirection, indexed streams can only be derived in terms of sequential streams, and streams cannot be derived from indexed streams.

Figure 3-16 shows a graphical table that illustrates this concepts. The first column contains a stream definition. The second column shows how the records in that stream correspond to records in a basic stream. The first declaration is of the basic stream *x*. It refers to an array of eight integer records. The second declaration is of the derived stream *y* that accesses every odd record of *x*. The third declaration is of the derived stream *z* that accesses every odd record in *y*. The stream *z* is remapped to *x*, the underlying basic stream of *y*. The actual start of *z* is equal to the specified start times the stride of *y* plus the start of *y* = (1 * 2) + 1 = 3. The end and stride are remapped similarly. *Z* maps to every fourth record of the underlying basic stream *x*, starting with record 3.

The derived stream definition also takes an optional parameter, *recordLengthOverride*, which overrides the length of a record to force the stream to treat multiple records as a single record. For instance, this parameter can be used to access a 2D block of an image by overriding the record length with the width of the block and specifying a stride equal to the width of the image. Each row of the block is then treated as a single record.

### 3.3.2 Microcontroller variables

StreamC uses microcontroller variables to pass scalar arguments to kernels. A microcontroller variable encapsulates a single integer or floating point variable. The value of a microcontroller variable can be set in a stream program using standard assignment. When a kernel is called with a microcontroller variable as an argument, its current value is sent to the microcontroller on Imagine. The stream program can only read the new value of the

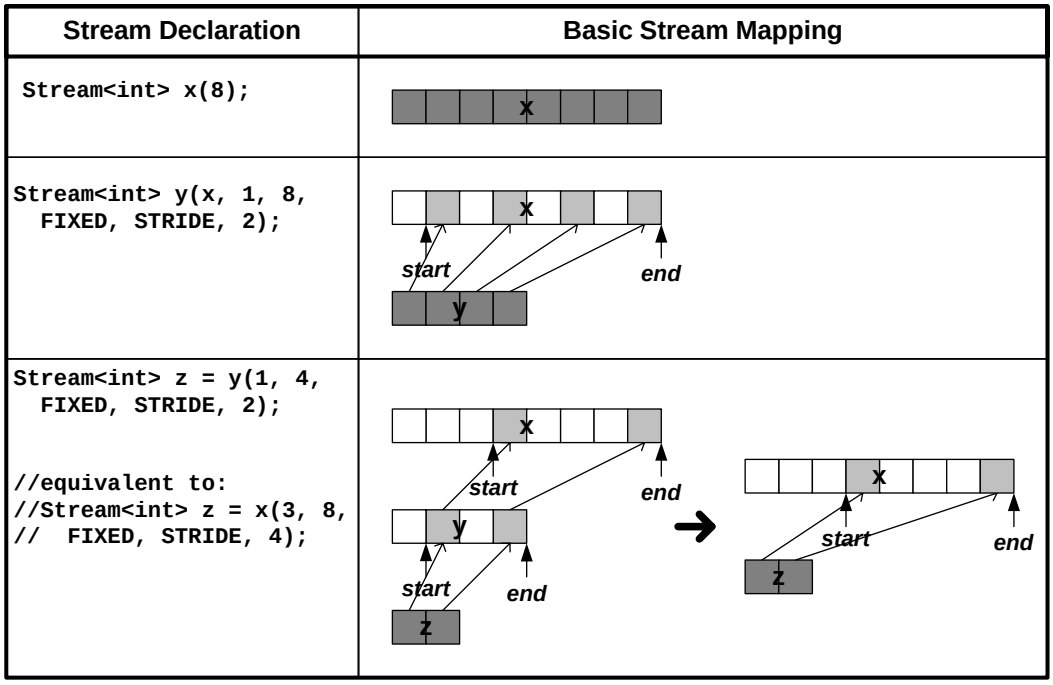



**FIGURE 3-16. Derived streams mapped to the underlying basic stream**

microcontroller variable computed by the kernel using the ucRead function, which pauses execution of the stream program until the kernel finishes.

Figure 3-17 shows an extended version of the example that includes a call to the sumStream kernel defined in Figure 3-8, with the uses of microcontroller variables highlighted. The kernel example1 takes two microcontroller variable arguments *uc_amul* and *uc_bmul*, which are set using standard assignment. The kernel sumStream takes a microcontroller variable argument *uc_sum* which is read using the ucRead function.

### 3.3.3 Kernels

Kernels are called in StreamC just like normal functions that take streams and microcontroller variables as arguments. All arguments to kernels are passed by reference. Figure 3-18 highlights the kernel call in the example.

```
void main()
{
  stream<int> a(256);
  stream<int> b(256);
  stream<int> c(512);
  stream<int> cEven =
    c(0, 512, FIXED, STRIDE, 2);
  uc<int> uc_amul = 2;
  uc<int> uc_bmul = 3;
  ...
  example1(a, b, cEven,
           uc_amul, uc_bmul);
  ...
  uc<int> uc_sum;
  sumStream(c, uc_sum);
  int sum = ucRead(uc_sum);
  ...
}
```

**FIGURE 3-17. Extended example with microcontroller variables highlighted**

```
void main()
{
  stream<int> a(256);
  stream<int> b(256);
  stream<int> c(512);
  stream<int> cEven =
    c(0, 512, FIXED, STRIDE, 2);
  uc<int> uc_amul = 2;
  uc<int> uc_bmul = 3;
  ...
  example1(a, b, cEven,
           uc_amul, uc_bmul);
  ...
}
```

**FIGURE 3-18. Kernel call in example**

### 3.3.4 Stream copies and transfers

In addition to Kernels, StreamC includes several functions used to copy data between streams and transfer data between arrays and streams. The streamCopy function reads all of the records from one stream and writes them to another stream. The streamLoadBin function copies a specified number of records from an array to a stream. The streamSave-Bin function copies records from a stream to an array of records. Figure 3-19 summarizes these functions.

| Function | Description |
|---|---|
| `streamCopy(`<br>`  stream<type> fromStream,`<br>`  stream<type> toStream)` | copies records from one stream to another |
| `streamLoadBin(`<br>`  stream<type> toStream,`<br>`  char* fromPtr, int length)` | copies records from an array to a stream |
| `streamSaveBin(`<br>`  stream<type> fromStream,`<br>`  char* toPtr)` | copies records from a stream to an array |

**FIGURE 3-19. Copies and transfers**

Figure 3-20 shows a simple stream program that copies records from an array to a stream, then to another stream, and finally back to an array.

```
void main()
{
  int array1[256];
  int array2[256];
  stream<int> stream1(256);
  stream<int> stream2(256);
  streamLoadBin(stream1, array1, 256);
  streamCopy(stream1, stream2);
  streamSaveBin(stream2, array2);
}
```

**FIGURE 3-20. Simple example of stream copies and transfers**

### 3.3.5 Data-dependence annotations

The StreamC compiler described in Chapter 6 requires the programmer to set the *data dependence* argument in the definition of a stream if it is used to hold the output of a kernel that produces a varying number of records, or is a derived stream with a start or end that varies depending on the data being processed. The *data dependence* argument can be either *fixed* (the default), *variable length*, or *variable bounds*. A fixed stream has the same definition regardless of the data being processed. A variable length stream is used to hold the output of a kernel that produces a varying number of records. For instance, a kernel might rasterize a stream of triangles into a varying number of pixels. When a kernel writes a variable length stream, it updates the length of the stream to reflect the actual number of records it contains. The getLength() method is used in the stream program to determine

the number of records in such a stream. A variable bounds stream is a derived stream with a start or end that depends on the data being processed. For instance, the location of a reference block within an image might vary, so a stream used to access that block would have variable bounds.

StreamC also requires that all control flow that depends on the data being processed be explicitly annotated. If the number of iterations of a loop or the execution of an if statement depends on the data being processed, that loop or if statement is annotated as shown in Figure 3-21.

*data-dependent loops:*

```
while_VARIABLE(...) {
  // loop body
}

for_VARIABLE(...) {
  // loop body
}
```

*data-dependent if:*

```
if_VARIABLE(...) {
  // if body
}
```

**FIGURE 3-21. Data dependent control flow annotations**

Figure 3-22 shows a simple program that recirculates a stream of records through a kernel until all records have reached a final state. The number of iterations required depends on the actual records, so the while loop is marked as data-dependent. The number of records that still need to be recirculated and the number of records that are finalized each iteration varies, so the streams *remainingRecs* and *newFinalRecs* are variable length. The newly finalized records need to be appended to the end of the records finalized on previous iterations. Thus, the stream *newFinalRecs* has a data-dependent start within the accumulated stream *finalRecs* and is also variable bounds.

```
stream<Rec> remainingRecs(MAX_RECS, VARIABLE_LENGTH);
stream<Rec> finalRecs(MAX_RECS);
...
int finalRecCount = 0;
while_VARIABLE(remainingRecs.getLength() > 0) {
  stream<Rec> newFinalRecs = finalRecs(finalRecCount, MAX_RECS,
    VARIABLE_LENGTH | VARIABLE_BOUNDS);
  processRecs(remainingRecs,                  // inputs
              newFinalRecs, remainingRecs);   // outputs
  finalRecCount += newFinalRecs.getLength();
}
...
```

**FIGURE 3-22. Example of data dependence annotations**

## 3.4 Summary

This chapter presented an implementation of the stream programming model. It described how the stream programing model structures a media processing application as a stream program and one or more kernels that operate on streams of records. It introduced StreamC, a set of classes and functions used in combination with C++ to write stream programs, and KernelC, a simple programming language used to write kernels.

Though designed for stream processors, this implementation of the stream programming model provides a consistent structure for media processing applications that can be efficiently mapped to other platforms. It makes the high-level control and data flow explicit, opening the door for powerful compiler optimizations even on general purpose processors.

# Chapter 4

# Communication Scheduling

This chapter presents communication scheduling [34], a new component of VLIW scheduling that enables scheduling to s*hared interconnect architectures* in which some or all functional unit inputs or outputs are connected to multiple registers files by shared buses and register file ports. Communication scheduling assigns each *communication*, the logical transfer of the result of one operation for use as a specific operand of another operation, to a *route* that defines the interconnect resources used to transfer the value between functional units.

Scheduling to shared interconnect architectures is difficult because it requires simultaneously allocating functional units to operations and buses and register file ports to the communications between operations. Communication scheduling solves this problem by incrementally composing a route for each communication from three components: a *write stub* that defines how the result is written to an initial register file, zero or more *copy operations* to move the value between register files, and a *read stub* that defines how the operand is read from the final register file. When the first operation is scheduled, the first stub is tentatively allocated. When the second operation is scheduled, both stubs are allocated and any required copy operations are scheduled. This composition allows the communicating operations to be scheduled independently while ensuring that the interconnect resources will be available to complete the communication.

This chapter is divided into four sections. Section 4.1 presents the motivation for communication scheduling. Section 4.2 presents an overview of the communication scheduling process and defines required terminology. Section 4.3 presents the communication scheduling algorithm. Section 4.4 discusses implementation of communication scheduling.

## 4.1 Motivation

A VLIW scheduler assigns each operation to a functional unit and schedules it on a particular cycle; communication scheduling allocates the interconnect resources the operation uses to read its operands and write its result. In a conventional, single register file architecture every functional unit input and output is connected to the same register file by a dedicated bus and register file port. All functional units read operands from and write results to the same register file, and can always do so without bus or register file port conflicts. The scheduler only needs to assign each operation to a functional unit and schedule it on a cycle. In a shared interconnect architecture, multiple functional unit inputs or outputs share buses connected to multiple register files. The scheduler needs to specify which register file to read each operand from, which register file (or multiple register files) to write each result to, and which shared buses and register file ports to use for doing so. Sometimes, the scheduler needs to insert copy operations to move a value from one register file to another. Communication scheduling extends a VLIW scheduler to handle these additional requirements. It ensures that each result is available to operations that use it as an operand, and avoids bus or register file conflicts when reading operands and writing results.

Scheduling the code fragment shown in Figure 4-1 for the single register file architecture shown in Figure 4-2 only requires assigning operations to functional units and scheduling them on cycles. Figure 4-3 is a graphical schedule[1] that illustrates all functional unit, interconnect, and register file activity on each cycle. It shows how each functional unit always reads its inputs from and writes its outputs to the same register file using the same dedi-

1. For illustrative purposes, all operations have unit latency.

cated interconnect. All functional units access the same register file, so every functional unit can access all data.

```
1:  a = load ...
2:  b = ... + ...
3:  c = ... + ...
4:  ... = a + b
5:  ... = a + c
```

**FIGURE 4-1. Example code fragment**

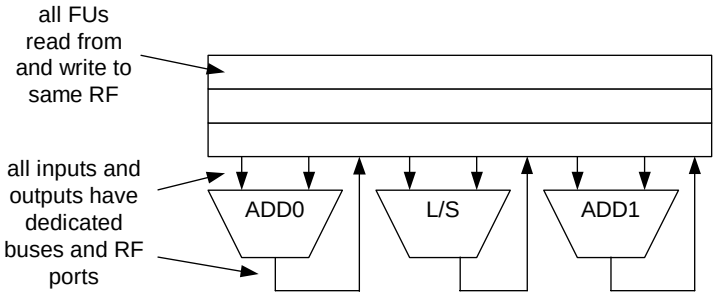



**FIGURE 4-2. Single register file architecture**

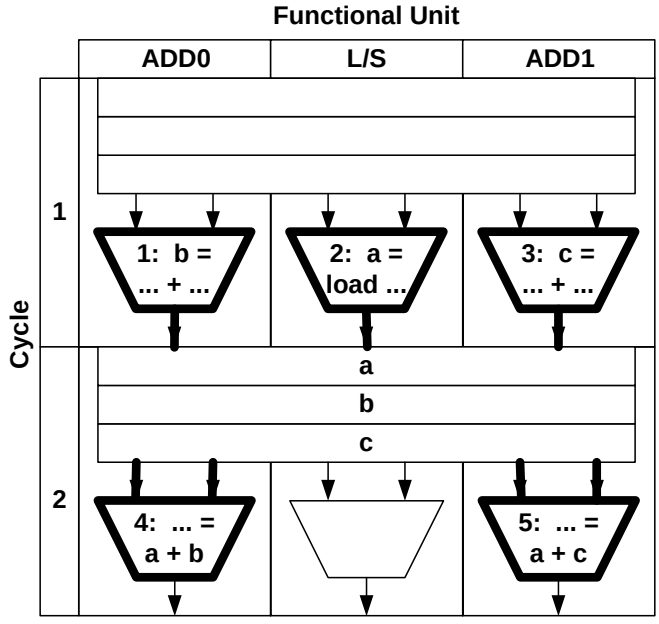



**FIGURE 4-3. Schedule for single register file architecture**

Now consider scheduling the same code fragment on the shared interconnect architecture shown in Figure 4-4. Though simple and purposefully non-optimal, the architecture includes all of the features of a shared interconnect architecture that require communication scheduling. Each adder output and the load/store unit output is connected by a shared bus to two register files. Both of the shared buses can drive the shared write port of the center register file. The scheduler specifies which driver drives each shared interconnect resource. Unlike a single register file architecture, all functional units cannot access all register files, so copy operations may be required to move values between register files.

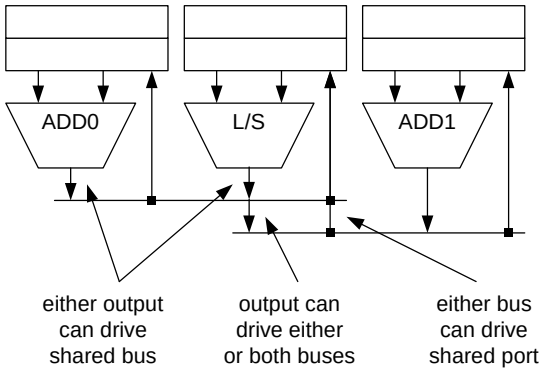



**FIGURE 4-4. Shared interconnect architecture**

The graphical schedule shown in Figure 4-5 illustrates these differences. On each cycle, the scheduler specifies which functional unit outputs drive the shared buses, and which bus drives the shared register file port. For example, on cycle 1 adder 0 uses the top shared bus to write to the left register file, while the load/store unit uses the bottom shared bus to write to the other two register files (using the shared write port of the middle register file). Note that operation 3 could not be scheduled on cycle 1 because the three functional unit outputs share only two output buses. On cycle 2, the scheduler schedules a copy operation to move a the result of operation 1 from the middle register file to the left register file so that operation 4 can use it as an operand.

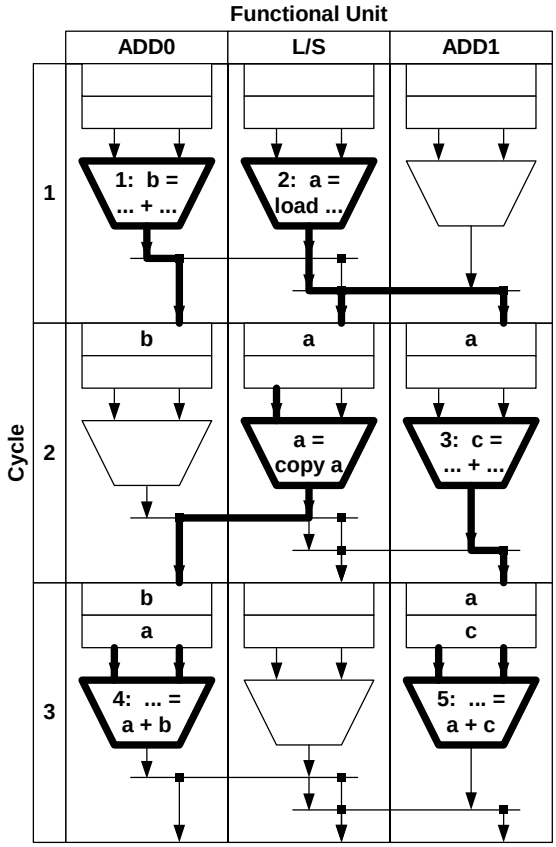



**FIGURE 4-5. Schedule for shared interconnect architecture**

# 4.2 Overview

To handle the additional requirements of a shared interconnect architecture, communication scheduling assigns each *communication* between operations to a *route* between the functional units that perform those operations as depicted in Figure 4-6. A communication is a scheduler abstraction for the use of the result of one operation as an operand of another operation. A communication exists from the *write operation* that computes a result to each *read operation* that could use the result as an operand. If multiple operations could use the result as an operand, or one operation could use the result as multiple operands, then a separate communication exists for each such *read operand*. If an operation could use one of several results as an operand due to different control flows then a separate communication exists for each such result.

A route defines the resources used to transfer a value from a functional unit output to a functional unit input. A route consists of resources to write the value to a register file from the functional unit that computes it, resources to read it from a register file to the functional unit that uses it, and, if necessary, copy operation(s) to move the value between register files.

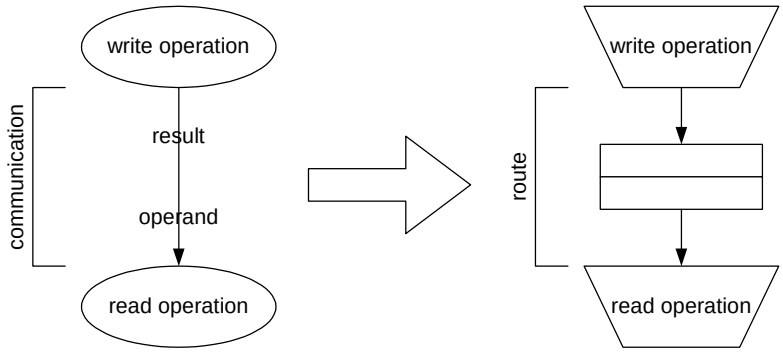



**FIGURE 4-6. Communication scheduling assigns each communication to a route**

The motivating example contains four communications as shown in Figure 4-7 and four routes, one for each communication, as shown in Figure 4-8. For example, operation 1 computes the value *a*, which is used by operation 4 and operation 5. There are two communications from operation 1, one to operation 4 and one to operation 5, each of which is assigned to a route.

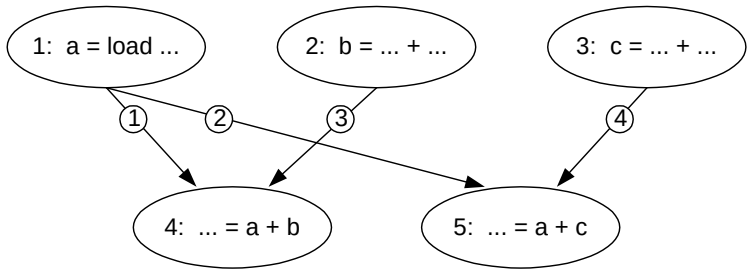



**FIGURE 4-7. Communications in motivating example**

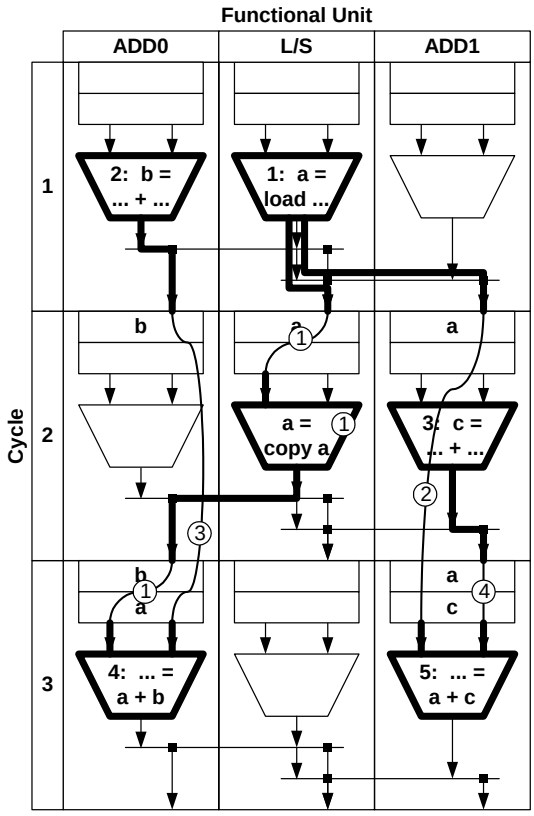



**FIGURE 4-8. Routes for communications in motivating example**

Communication scheduling composes a route for each communication as shown in Figure 4-9. The *write stub* consists of the functional unit output, bus, and register file write port allocated to write the result. The write stub is allocated on the cycle that the writing operation completes. The *read stub* consists of the register file read port, bus, and functional unit input allocated to read the operand. The read stub is allocated on the cycle that reading operation issues. If the write stub and read stub access the same register file, they form a route. Otherwise, one or more copy operations are used to move the value from one register file to another to connect the stubs and form a route.

Figure 4-10 shows the write stub, read stub, and copy operation that compose the route for the communication of *a* from operation 1 to operation 4 in the motivating example.

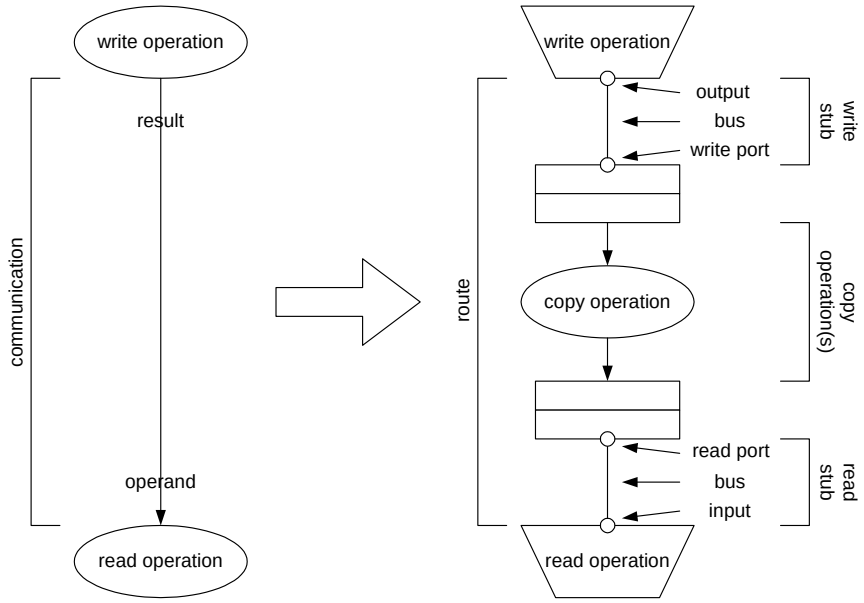



**FIGURE 4-9. Composition of a route**

Communication scheduling composes routes such that stubs on the same cycle do not conflict. Read stubs for different operands or write stubs for different results conflict if they use the same resource, such as a functional unit input or output, bus, or register file port. An operand can only be read from one register file, so two read stubs for the same operand conflict if they are not identical. A result can be written to multiple register files, so two write stubs for the same result only conflict if they write to the same register file using different buses or register file ports.

### 4.2.1 Role in a VLIW scheduler

The communication scheduling algorithm presented in this chapter is a general technique that works as a drop-in addition to a variety of VLIW scheduling algorithms. The VLIW scheduler is responsible for assigning operations to functional units and scheduling them on cycles, communication scheduling simply accepts or rejects each placement as shown in Figure 4-11. The only assumption communication scheduling makes is that repeatedly rejecting an operation placement will eventually force that operation into an otherwise empty region of the schedule. However, considering communication scheduling when placing operations results in better performance. Chapter 5 describes the KernelC com-

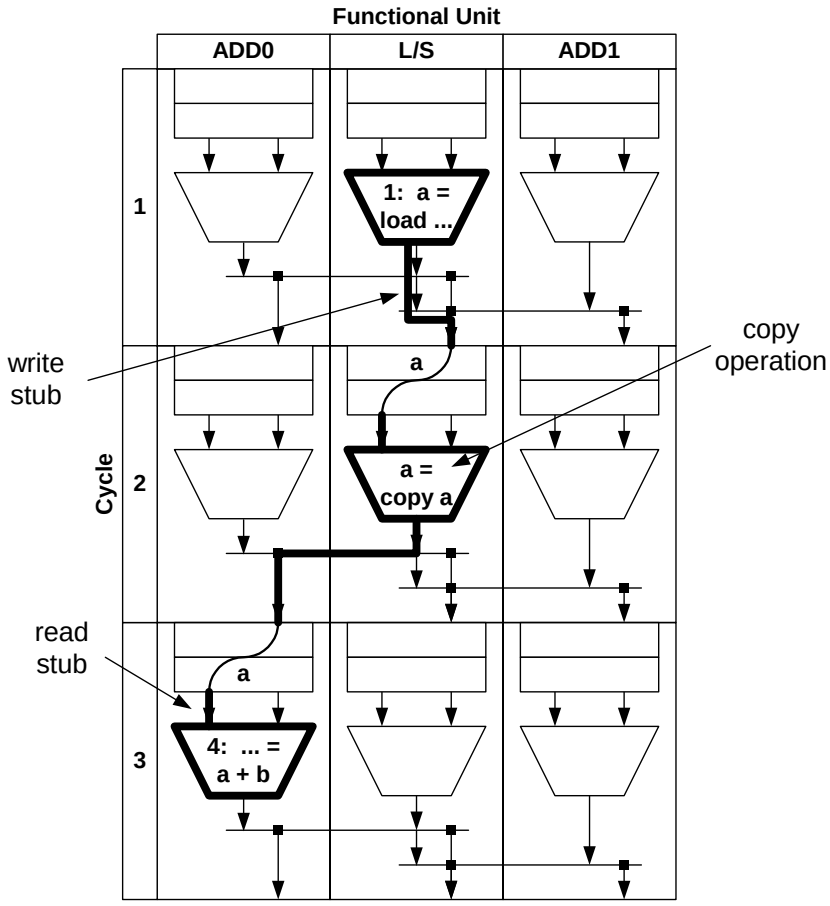



**FIGURE 4-10. Composition of route for communication of *a* from operation 1 to operation 4**

piler, a VLIW scheduler that uses communication scheduling, and discusses the ramifications of communication scheduling for the scheduling process as a whole.

As the VLIW scheduling algorithm selects and schedules operations, communication scheduling incrementally composes a route for each communication. Figure 4-12 shows how communication scheduling composes a route for a communication between two arbitrary operations, *operation 1* and *operation 2*. In this example, operation 1 is scheduled before operation 2, the process is the same if the order were reversed. When operation 1 is being scheduled, the communication is *opening*: communication scheduling determines the valid stubs and selects a stub that does not conflict with other stubs on the same cycle. If it cannot find a stub that does not conflict, it rejects the placement until it succeeds.

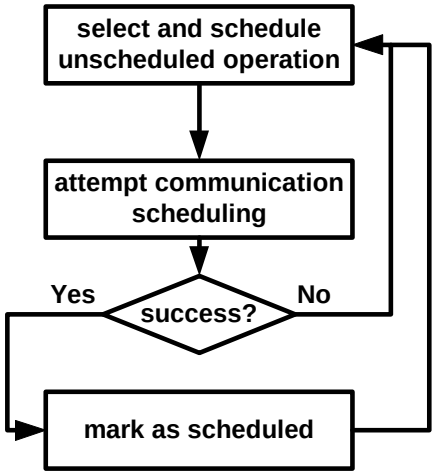



**FIGURE 4-11. Flowgraph for a simple scheduler with communication scheduling**

Other operations may be scheduled before operation 2 is scheduled depending on the scheduling algorithm, and often must be scheduled when an operation communicates with multiple operations since only one can immediately follow it. As each such operation is scheduled, communication scheduling may change the stub assigned to the *open* communication to allow stubs to be found for other communications. When the operation 2 is being scheduled, the communication is *closing*: communication scheduling tries to find a write stub and a read stub that access the same register file to form a route. If necessary, it tries to insert and schedule copy operations to connect the stubs and form a route. If it is unable to do so, it unschedules all copy operations and rejects the placement of operation 2. Once a communication has been assigned to a route, the stubs and any copy operations that compose the route cannot be changed and it is called *closed*. Once all operations are scheduled, all communications are closed and have all been assigned to routes.

## 4.3 Algorithm

For each potential operation placement, communication scheduling performs the following steps for the operation that is being scheduled, hereafter called the current operation:

1. determine the valid read stubs for each communication to the current operation and the valid write stubs for each communication from the current operation

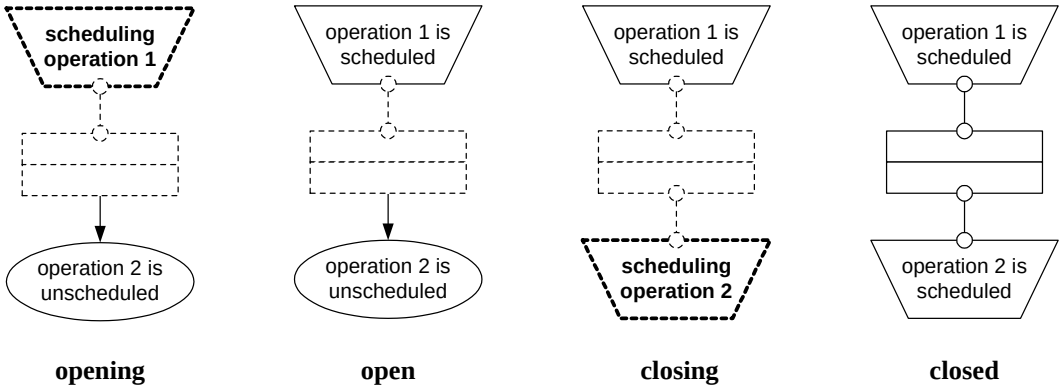



**FIGURE 4-12. Incremental composition of a route (left to right)**

2. find a non-conflicting permutation of read stubs for communications to operations on the cycle the current operation issues on
3. find a non-conflicting permutation of write stubs for communications from operations on the cycle current operation completes on
4. for each closing communication, if the read stub and write stub form a route then assign the communication to that route
5. for each closing communication, if the read stub and write stub do not form a route then insert and attempt to schedule copy operation(s) to connect the stubs

Each of these steps is described in detail in the remainder of this section.

## Step 1. Determine valid stubs

First, communication scheduling determines the valid read stubs for each communication to the current operation and the valid write stubs for each communication from the current operation. A read stub connects the read port of a register file to an appropriate input of the functional unit that the current operation is assigned to. A write stub connects the output of the functional unit to which the current operation is assigned to a write port of a register file. For a communication from operation $o_1$ to operation $o_2$, zero or more copy operations can be used to move a value from any register file written to by a valid write stub for $o_1$ to any register file read from by a valid read stub for $o_2$, regardless of which functional units the operations are assigned to.

Figure 4-13 shows all four valid write stubs for the communication from operation 1, scheduled on the load/store unit on cycle 1, to operation 4. The four stubs are, described

left to right then top to bottom: using the left bus to write to adder 0's register file, using the right bus to write to adder 1's register file, using the left bus to write to the load/store unit's register file, and using the right bus to write to the load/store unit's register file.

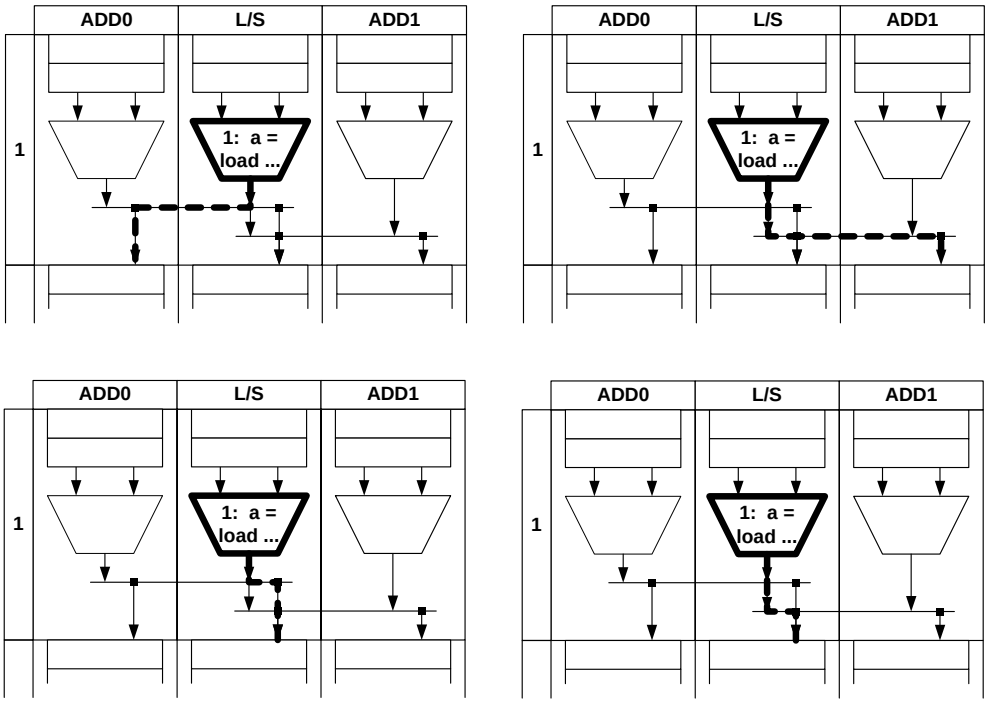



**FIGURE 4-13. Valid write stubs**

Figure 4-14 shows the valid read stubs for operation 4 when, after scheduling several other operations, it is scheduled on adder 0 on cycle 3. Since addition is a commutative operation, adder 0 can read the value of *a* from its register file using either input port. Zero or more copy operations can be used to connect any write stub in Figure 4-13 to any read stub in Figure 4-14.

## Step 2. Find permutation of read stubs

Second, communication scheduling attempts to find a permutation of read stubs for all communications to the current operation and previously scheduled operations that are issued on the same cycle. This set of communications is called $C_{to}$. Since communication scheduling can't change the read stub assigned to a closed communication, the stubs it

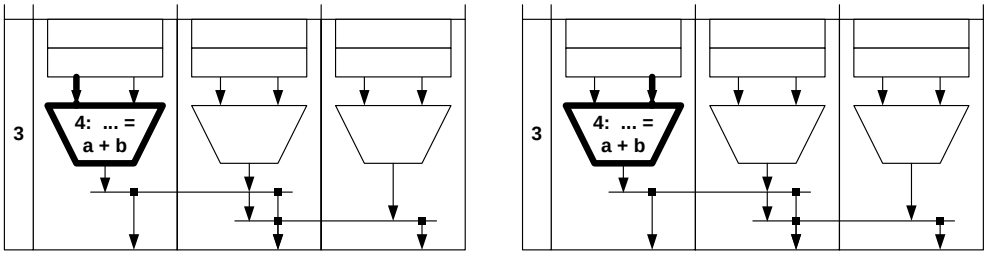



**FIGURE 4-14. Valid read stubs**

finds for other communications must not conflict with those stubs. Therefore, it removes all closed communications in $C_{to}$ and eliminates all valid stubs for the remaining communications that conflict with any read stub assigned to a closed communication. Communication scheduling then attempts to find a valid stub for each communication remaining in $C_{to}$. It can choose any stub for each open communication, but tries to choose a read stub for each closing communication that forms a route. When selecting a read stub for a closing communication $c$ from a scheduled operation $o_s$, communication scheduling also attempts to find a permutation of write stubs for communications to operations that complete on the same cycle as $o_s$ such that the write stub for $c$ accesses the same register file as the read stub and forms a route.

## Step 3. Find permutation of write stubs

Third, communication scheduling analogously attempts to find a permutation of write stubs for all communications from the current operation or operations that complete on the same cycle as the current operation. This set of communications is called $C_{from}$. If communication scheduling cannot find a permutation of read stubs or a permutation of write stubs, it rejects the current operation placement.

In the motivating example, communication scheduling finds different permutations of write stubs for the communications from operations on cycle 1 as each of the first two operations are scheduled. Communication scheduling chooses the permutation of write stubs shown in Figure 4-15 when operation 1 is scheduled, then changes to the permutation shown in Figure 4-16 when operation 2 is scheduled. Operation 3 cannot be sched-

uled on cycle 1 because a permutation of write stubs cannot be found due to stub conflicts as shown in Figure 4-17.

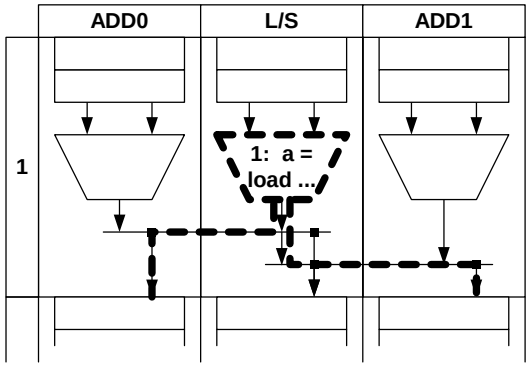



**FIGURE 4-15. Permutation of write stubs when scheduling operation 1**

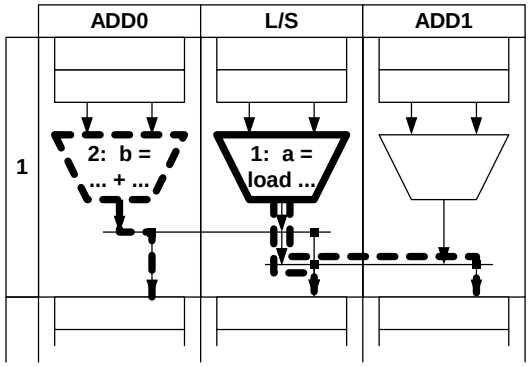



**FIGURE 4-16. Permutation of write stubs when scheduling operation 2**

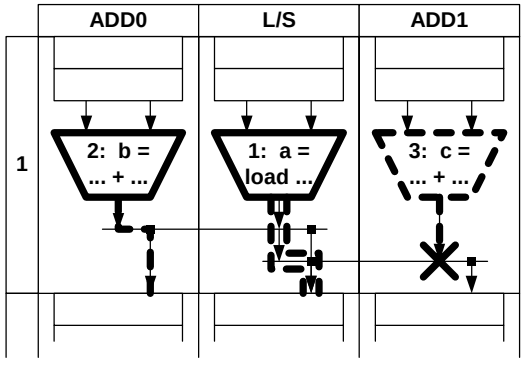



**FIGURE 4-17. Operation 3 cannot be scheduled due to stub conflicts**

## Step 4. Assign routes

Fourth, communication scheduling examines each closing communication and assigns a route if possible. If the read stub and write stub access the same register file and form a route, communication scheduling immediately assigns the communication to that route.

When scheduling operation 4 in the motivating example, the write stub and a read stub form a route for the closing communication of *b* from operation 2, so communication scheduling immediately assigns it to that route as shown in Figure 4-18. The stubs for the closing communication of *a* from operation 1 do not form a route.

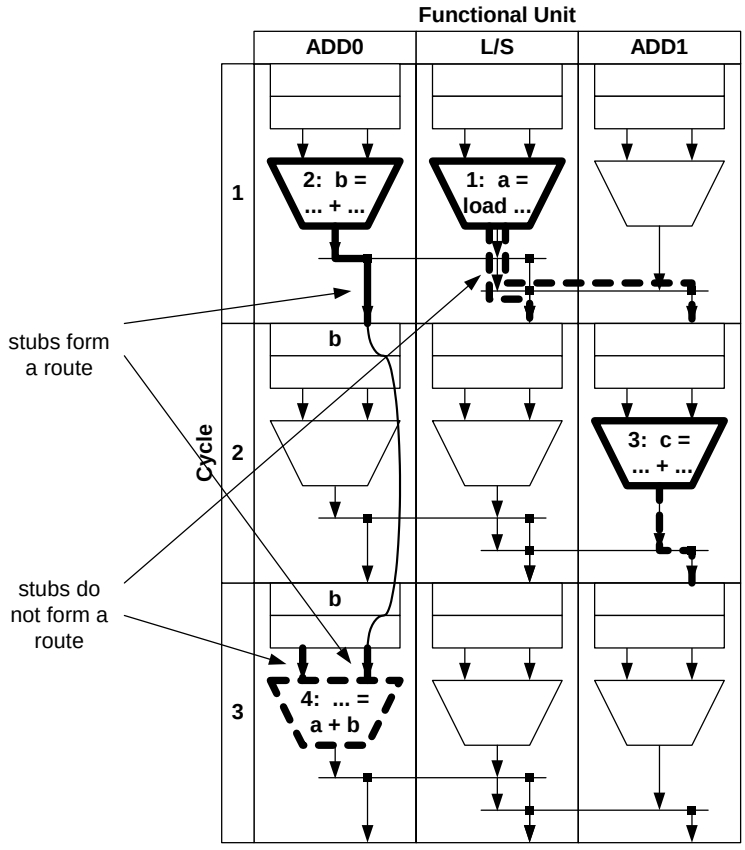



**FIGURE 4-18. Route for communication of *b* from operation 2 to operation 4**

## Step 5. Insert copy operations

Fifth, communication scheduling inserts and attempts to schedule a copy operation to connect the stubs and form a route for each remaining closing communication. Inserting a copy operation is equivalent to the code transformation shown in Figure 4-19.

```
x = ...                          x = ...
...                              x' = copy x
... = x ...                      ...
                                 ... = x' ...
```

**FIGURE 4-19. Copy operation code transformation**

Effectively, this transformation splits the original communication into two communications, one from the write operation to the copy operation, and one from the copy operation to the read operation as shown in Figure 4-20. Communication scheduling then calls on the scheduler to schedule the copy operation.

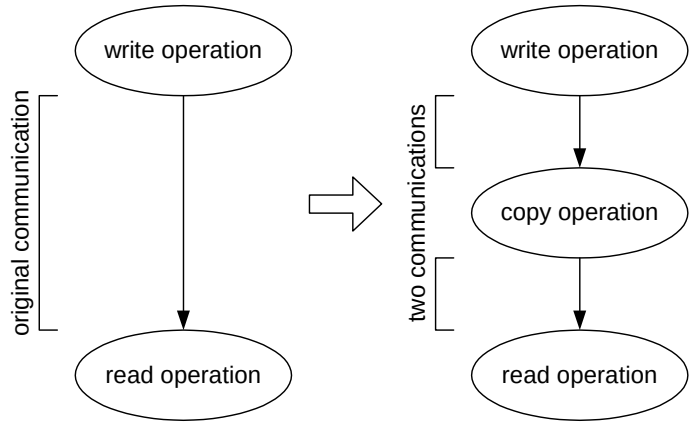



**FIGURE 4-20. A copy operation effectively splits original communication into two communications**

The copy operation is scheduled just like any other operation, except that it must be scheduled on a cycle in the *copy range* of the original communication. If the write operation is before the read operation in the same basic block, the copy range is all cycles between the cycle on which the write operation completes and the cycle on which the read operation

issues. Otherwise, the copy range is all cycles in the write operation's basic block after the write operation. These two cases are shown in Figure 4-21.

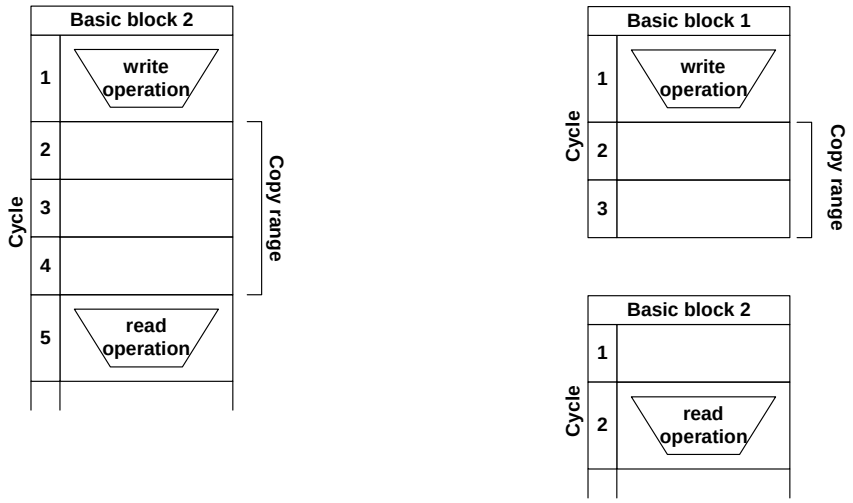



**FIGURE 4-21. Copy ranges based on location of read operation**

Copy operations for communications between operations in different basic blocks (or from a write operation to a read operation earlier in the same block) are restricted to the write operation’s basic block so that they do not overwrite the result of other write operations. Multiple write operations could compute an operand depending on control flow. A copy operation for a communication from one such write operation scheduled in the read operation’s basic block would overwrite the result of any other write operation, regardless of the actual control flow. If necessary, the scheduler inserts additional cycles at the end of the write operation’s basic block to accommodate copy operations.[1]

Communication scheduling treats the copy operation just like any other operation, so communication scheduling can recursively insert additional copy operations as needed.

1. The implementation of communication scheduling used in the evaluation section backtracks to the basic block containing the write operation rather than adding additional cycles, but backtracking is a costly way to handle a rare special case and not recommended for future implementations.

Returning to the motivating example, the stubs for the closing communication of *a* from operation 2 to operation 4 do not form a route, so communication scheduling inserts and attempts to schedule a copy operation as shown in Figure 4-22.

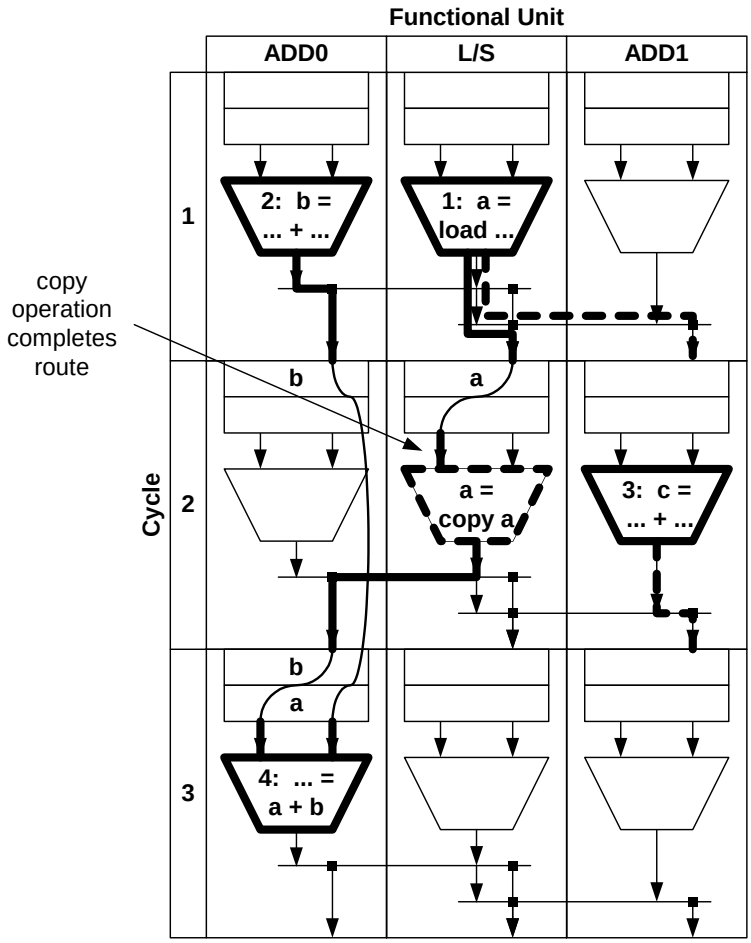



**FIGURE 4-22. Route for communication of *a* from operation 1 to operation 4**

Communication scheduling succeeds if it finds a permutation of write stubs and a permutation of read stubs, and assigns each closing communication to a route. If communication scheduling fails, any routes assigned to communications to/from the current operation are unassigned, and any copy operations are unscheduled. The scheduler then reschedules the operation and attempts communication scheduling again. Once all operations have been scheduled successfully, communication scheduling has assigned all communications to routes.

## 4.4 Implementation

This section discusses implementing three key components of the communication scheduling algorithm: determining the valid stubs for a communication and finding a permutation of stubs for a set of communications, and efficiently scheduling copy operations.

### 4.4.1 Determining valid stubs for a communication

To determine the valid stubs for a communication, communication scheduling first finds all possible stubs using a transversal of the architecture's interconnect. In the case of read stubs, it first determines which functional unit input(s) can be used to read the operand. Then, for each input, it enumerates all the buses the input is connected to and all the register file read ports each such bus is connected to. Each combination of a connected input, bus, and register file port is a possible read stub.

However, not all possible stubs are valid because communication scheduling requires that, for a given communication, it must be possible to use copy operations to complete a route from any valid write stub to any valid read stub. This constraint is fundamental to communication scheduling because it allows two communicating operations to be scheduled independently. Regardless of which stub is chosen when scheduling the first operation, the second operation can still be assigned to any functional unit.

This constraint can only be met for a *copy-connected* architecture, such as Imagine. A register file, *rf1*, is copy-connected to another register file, *rf2*, if zero or more copy operations can be used to move a value from *rf1* to *rf2* (a register file is also considered copy-connected to itself). An architecture is copy-connected if, given any pair of operations, *o1* and *o2*, and a specific operand of *o2*, *operand*, such that the result of *o1* can be used as that operand of *o2*, it is possible to find two sets of register files, *RFwrite* and *RFread* such that:

- The output of any functional unit that can perform *o1* is connected to at least one register file in *RFwrite*
- Every input that can be used to read *operand* by any functional unit that can perform *o2* is connected to at least one register file in *RFread*

- Every register file in *RFwrite* is copy-connected to every register file in *RFread*

Figure 4-23 illustrates this constraint.

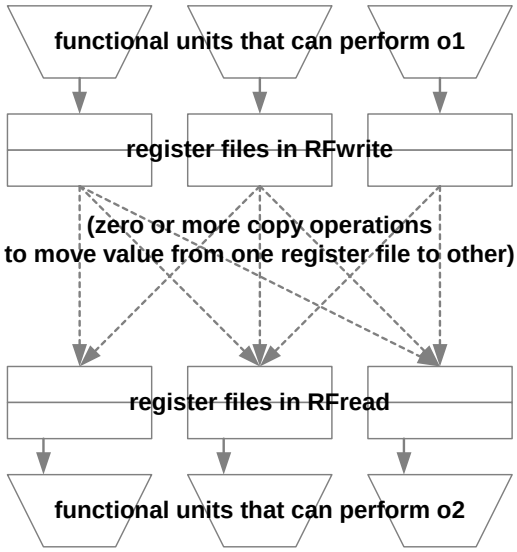



**FIGURE 4-23. Copy connected architecture constraint**

The *RFwrite* and *RFread* sets for each *o1*, *o2*, *operand* triplet can be precomputed using several methods. The most flexible method is an exhaustive search of all permutations of register files. Since these sets are used to limit valid stubs, it is desirable to find the largest sets possible. If *rf1* is copy-connected to *rf2* and *rf2* is also copy-connected to *rf1*, the two register files can be treated as one register file for the purpose of this search. Any set that contains *rf1* also contains *rf2*. For many architectures, every register file is copy-connected to every other register file, so the search is trivial and the *RFread* and *RFwrite* sets always contain all register files.

Communication scheduling determines the valid stubs for a communication by using the *RFwrite* set and *RFread* set for the communication's write operation, read operation, read operand triplet. The valid write stubs are the possible stubs that write to a register file in *RFwrite*. The valid read stubs are the possible stubs that read from a register file in

*RFread*. Since every register file in *RFread* is copy connected to every register file in *RFwrite*, copy operations can be used to complete the route between any two valid stubs.

### 4.4.2 Finding a permutation of stubs

Finding a non-conflicting permutation of stubs is computationally expensive, but the search does not need to be exhaustive. The number of permutations of stubs is exponential with the number of communications. However, the search will complete if:

- It can find a read/write stub for all communications to/from an operation in the absence of other communications
- It can always find a permutation of stubs for a given set of communications if it ever finds a permutation of stubs for that set of communications (i.e. it is repeatable)

The first requirement ensures that an operation can always be scheduled, even if only by scheduling it to issue and complete on cycles without any other scheduled operations. The second requirement ensures that, once an operation has been scheduled it will remain possible to find a permutation of stubs for the cycles on which it issues and completes, even if only by scheduling no additional operations on those cycles.

One search algorithm that meets these requirements orders the communications, then finds a stub for each communication in turn. The algorithm orders the communications so that the communications for which it is most important to complete a route come first. All closing communications are ordered before all open or opening communications. Closing communications are ordered by smallest copy range first, so that the communications with the fewest cycles to schedule copy operations on have preference in choosing stubs to form routes. Once the communications are ordered, the algorithm selects the first stub for each communication that does not conflict with the stub found for a previous communication. If all stubs for a communication conflict with stubs found for a previous communications, the search falls backs to the first such communication and chooses a new stub. The search terminates when a stub has been found for each communication or after an arbitrary, relatively large, number of partial permutations have been tried.

Figure 4-24 shows detailed pseudocode for a function that implements this search algorithm.[1] Note that this function is used both to find a permutation of read stubs in Step 2 of the communication scheduling algorithm, and to find an opposite read stub for a closing communication in Step 3. The corresponding function used to find a permutation of write stubs in Step 3 and an opposite write stub for a closing communication in Step 2 mirrors this function exactly.

```
Boolean FindReadStubs(Operation o, Communication cFindOpposite = NULL)
{
   Integer i, j, k;

   Set<Operation> O = GetOperationsIssuedOnCycle(o.issue);
   Set<Communication> CTo = GetCommunicationsToOperations(O);
   Set<Communication> CClosed = GetClosedCommunications(CTo);
   Set<Communication> CNonClosed = GetNonClosedCommunications(CTo);

   // sort non-closed communications so that all closing communications come
   // first, in order of ascending copy range size
   PrioritizeCommunications(CNonClosed);


   // remove valid stubs that conflict with stubs of closed communications
   for (i = 0; i < CNonClosed.count; i++) {
      Communication c = CNonClosed[i];
      for (Integer j = 0; j < CClosed.count; j++) {
         RemoveConflictingStubs(c.validReadStubs, CClosed[j].readStub);
      }
      c.readStub = NULL;
   }


   // with each non-closed communication...
   i = 0;
   Integer permutationCount = 0;
   while (i >= 0 && i < CNonClosed.count &&
          permutationCount  < MAX_PERMUTATIONS) {

      Communication c = CNonClosed[i];
      Stub firstNonConflictingStub;
      Boolean conflict = TRUE;

      // find the first stub that does not conflict with a previous stub
      Integer prevConflictMaxIdx = -1;
      for (j = GetIndexOfNextStub(c.validReadStubs, c.readStub);
             j < c.validReadStubs.count && conflict; j++) {
         c.readStub = c.validReadStubs[j];
         conflict = FALSE;
         for (k = i - 1; k >= 0 && !conflict; k--) {
            if (CheckStubConflict(c.readStub, CNonClosed[k].readStub) {
```

1. Pseudocode assumes all object variable are reference-counted pointers

```
                conflict = TRUE;
                prevConflictMaxIdx = Maximum(prevConflictMaxIdx, k);
            }
        }

        // if a stub is found and the communication is closing,
        // try and find the opposite stub
        // unless this function is being used to find an opposite stub,
        // then try and form a route
        if (!conflict && c.status == CLOSING) {
            if (cFindOpposite == NULL) {
                conflict = !FindWriteStubs(c.writeOp, c));
            } else {
                conflict = !CheckStubsFormRoute(c));
            }
            // note the first non-conflicting stub
            if (firstNonConflictingStub == NULL) {
                firstNonConflictingStub = c.readStub;
            }
        }
    }

    // if the communication is closing and a route cannot be formed
    // use the first non-conflicting stub, copy operations will be added later
    // unless trying to find the opposite stub for this communication
    if (conflict && firstNonConflictingStub != NULL && c != cFindOpposite) {
        c.readStub = firstNonConflictingStub;
        conflict = FALSE;
    }

    // if a stub is found, advance to the next communication
    // otherwise fallback to the first previous communication with a stub
    // that conflicts with any stub of this communication
    if (!conflict) {
        i++;
    } else {
        while (i > prevConflictMaxIdx) {
            CNonClosed[i].readStub = NULL;
            i--;
        }
    }
    permutationCount++;
}

// return true if a stub is found for all non-closed communications
return (i == CNonClosed.count);
}
```

**FIGURE 4-24. Pseudocode for stub permutation search**

### 4.4.3 Scheduling copy operations

The scheduler merges copy operations for different communications of the same result to make more efficient use of resources. Suppose one operation computes a result that is communicated to two other operations, and both communications require copy operations to form routes. The scheduler schedules the copy operation for the first communication, *copy1*, normally. If it schedules *copy1* in the copy range of the copy operation for the second communication, *copy2*, it schedules *copy2* on the same cycle and functional unit as *copy1*, then attempts communication scheduling. Communication scheduling treats stubs for communications with either operation as stubs of communications with the same operation for the purpose of determining conflicts. If communication scheduling succeeds for *copy2*, the scheduler merges the two copy operations into one copy operation. Otherwise, it schedules *copy2* normally.

## 4.5 Performance

Architectures with shared interconnect and multiple register files impose additional constraints on VLIW scheduling; communication scheduling contributes to good performance on these architectures by limiting the impact of these constraints on scheduling. Communication scheduling introduces an incremental method for composing routes from shared interconnect resources during scheduling that does not need to know which operations are assigned to which functional units prior to scheduling. This allows the use of a single-phase scheduling algorithm which assigns operations to functional units during scheduling. Most multi-phase algorithms rely on constructing an approximate schedule before constructing the actual schedule in order to assign parallel operations to different functional units. This approximation becomes less accurate in the presence an additional resource constraint, such as complex shared interconnect. Further, the effects of a poor approximation are magnified when scheduling kernels with excess instruction level parallelism on architectures with many functional units. When more operations can occur in parallel than there are available functional units, the operations cannot be assigned to functional units such that they can always be scheduled in parallel. More functional units increases the chance that for a set of operations with an effectively random assignment to functional units, one or more operations will be assigned to the same functional unit.

The effectiveness of communication scheduling depends on the topology of the shared interconnect. It is designed for architectures with a high-degree of connectivity and mostly equivalent interconnect resources, such as Imagine. In architectures with very limited connectivity among functional units, there are few decisions for a communication scheduling algorithm to make. On such architectures, operation placement is the determining factor in performance. Communication scheduling as presented in this chapter assumes that most interconnect resources are equivalent. In an architecture in which some interconnect resources are more connected that others, a naive algorithm for choosing stubs could wastefully assign highly-connected resources to communications that could use less-connected resources. However, using an algorithm that simply weighted stubs based on connectivity would largely avoid this problem. The scheduling process for a VLIW architecture is already NP-complete and shared interconnect introduces additional, non-orthogonal resources to the allocation problem so an exact approach is not possible.

## 4.6 Summary

This chapter described communication scheduling, a new component of VLIW scheduling that allocates shared interconnect resources such as buses and register file ports by assigning communications between operations to routes that define the resources used to transfer values between functional units. It presented the communication scheduling algorithm used to assign communications to routes, and discussed implementing key portions of the algorithm.

Communication scheduling is a general technique that can be incorporated as part of a variety of scheduling algorithms and applied to a large class of shared of architectures. Communication scheduling can be added to a scheduler simply by allowing communication scheduling to accept or reject each operation placement. Communication scheduling is not architecture specific. It can be used to explore novel register files architectures without implementing a custom compiler for each architecture.

# Chapter 5

# KernelC Compiler

This chapter describes the Imagine KernelC compiler, which compiles KernelC for the processing elements of the Imagine media processor. This chapter provides an overview of the KernelC compiler as a whole and concentrates on the design choices and innovations motivated by the new hardware concepts introduced in the Imagine processor architecture and the characteristics of media processing kernels.

The KernelC compiler supports the Imagine media processor architecture's multiple register files with shared interconnect, sequential interface to the stream register file, and addressable scratchpad memory. The KernelC compiler uses communication scheduling, a new compiler technique described in detail in Chapter 4, to support multiple register files with shared interconnect. This chapter describes the analysis used to construct the *communication graph* used for communication scheduling, and presents a scheduling algorithm optimized for communication scheduling. The KernelC compiler introduces *stream input/output ordering*, a pre-scheduling step that ensures that memory accesses can be ordered sequentially, and modifies the dependency graph to ensure that they are ordered sequentially. The KernelC compiler handles scratchpad accesses in the same manner as arithmetic operations with additional dependencies.

The KernelC compiler is optimized for high performance media processing kernels that usually consist of a single, computation-intensive loop. The performance critical nature and relative simplicity of media processing kernels motivate the use of relatively expen-

sive scheduling heuristics. Since kernel performance is dominated by the loop, the KernelC compiler incorporates a variation of modulo software pipelining[28].

The Imagine KernelC compiler compiles kernels written in KernelC into machine code executable on the Imagine media processor. The compilation process is separated into three steps, each described in a section of this chapter. Pre-scheduling translates source code into primitive operations and analyzes and modifies those operations as described in Section 5.1. Scheduling assigns the primitive operations to functional units and schedules them on cycles as described in Section 5.2. Post-scheduling allocates registers and generates machine code as described in Section 5.3.

## 5.1 Pre-scheduling

The pre-scheduling process translates a kernel written in KernelC into primitive operations augmented with all the information necessary for scheduling. The KernelC compiler parses the source code into operations, separates the operations into basic blocks, generates a communication graph that is the basis for communication scheduling, and produces the dependency graph used to order operations for scheduling.

### 5.1.1 Parsing

The KernelC compiler translates KernelC into primitive operations using a standard lexical analyzer and parser. The KernelC compiler adds operations to compute constants, and performs any loop unrolling specified by the programmer.

Figure 5-1 shows a simple kernel in KernelC and the corresponding primitive operations. The simple kernel performs a coordinate transformation with two explicit dimensions (x and y) and one implicit dimension (z). This kernel will be used throughout the remainder of this section to illustrate features of the KernelC compiler.

### 5.1.2 Control flow analysis

The KernelC compiler separates the operations into basic blocks and constructs a control flow graph containing all basic blocks in the kernel with a directed edge from each basic

```
kernel CoordinateTransform(
  istream<float> inXYs,
  ostream<float> outUVs)
{
  float z = 0;                      1: z = 0.0

  loop_stream(inXYs) {

    // load inputs
    inXYs >> x;                     2: in0 >> x

    inXYs >> y;                     3: in0 >> y


    // coordinate transform
    v = -((y + z)^2);               4: a = y + z
                                    5: b = a * a
                                    6: v = -b

    u = x^2 + v;                    7:  c = x * x
                                    8:  u = c + v

    // store outputs
    outUVs << u << v                9:  out0 << u
                                    10: out0 << v

    // increment implicit z
    z = z + 1.0;                    11: z = z + 1.0

  }
}
```

**FIGURE 5-1. Example kernel in KernelC and corresponding primitive operations**

block to every basic block that could be executed immediately after it. Since the only control flow structures in KernelC are explicit loops, all basic blocks are delineated by the start or end of a loop and the control flow analysis is trivial. Figure 5-2 shows the control flow graph for the example.

### 5.1.3 Data flow analysis

The KernelC compiler constructs a data flow graph for the entire kernel containing all operations with directed edges representing each communication between operations. This type of graph is a standard intermediate representation, but specifics vary from description to description. For clarity, this specific form, which is used as the basis for communication scheduling, is called a communication graph. The communication graph contains a

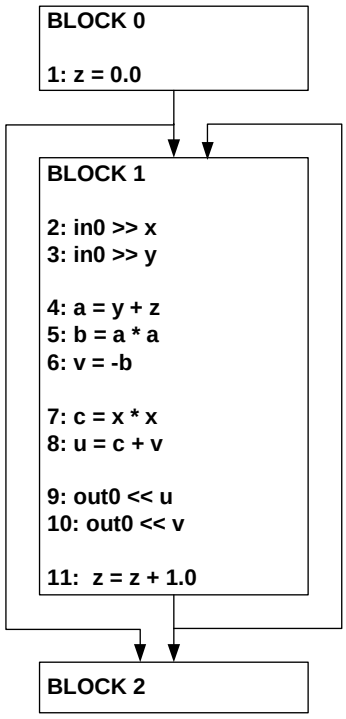



**FIGURE 5-2. Control flow graph**

directed edge from each operation that computes a result to each operation that uses the result as an operand, for each such operand. A communication exists between two such operations regardless of their relative location: communications can exist between operations in different basic blocks, or from a later operation to an earlier operation that uses its loop carried result as an operand. Each edge is labeled with the result used and the operand it is used as. This information is used by communication scheduling to determine which functional unit ports to use for the communication.

The communication graph can be constructed using one of several data flow analysis methods. The KernelC compiler uses a slot-wise approach [37], so named because it considers each result separately, but this method is an arbitrary implementation choice. The KernelC compiler iterates over each operation, *o*. For each result of *o*, *r*, it performs the following analysis: starting with the operation immediately after *o*, it adds an edge from *o* to the current operation for each use of the *r* as an operand, then moves to the next operation in the current basic block. If the KernelC compiler encounters the end of a basic

block, it adds all unreached basic blocks that succeed that basic block in the control flow graph to a worklist. It then removes a basic block from the worklist, marks it as reached, and moves to the first operation in that basic block. If it encounters an operation that computes a result which is assigned to the same variable as *r*, it stops traversing the current basic block and obtains a new basic block from the worklist.

Figure 5-3 shows the communication graph for the sample kernel. The first four edges, shown in bold, are added as follows. Operation 1 ("z = 0.0") produces one result assigned to *z*. There are no more operations in block 0, so block 1 and block 2 are pushed onto the worklist. Block 1 is popped off the work list and traversed. Operation 4 uses *z* as operand two, so an edge is added from operation 1 to operation 4 and labeled "1, 2" (result 1 used as operand 2). Operation 11 also uses *z*, so another edge is added. However, the result of operation 11 is assigned to *z*. Traversal of block 1 stops and block 2 is popped off the worklist. Block 2 is empty, not followed by any blocks in the control graph, and no blocks remain on the worklist, so all edges for operation 1's only result have been added. Operation 2 also produces one result, assigned to *x*. Operation 7 uses *x* for two operands, so *two* edges are added from operation 2 to operation 7, labeled "1, 1" and "1, 2".

### 5.1.4 Dependency analysis

The KernelC compiler constructs a directed acyclic graph (DAG) for each basic block containing all operations in the basic block with an edge from each operation to every operation that depends on it. The initial dependency graph for a basic block is derived from the communication graph and the source code order of the operations, and contains a subset of the edges in that graph. Each edge that connects two operations within the basic block is considered in turn, all edges connecting operations in other basic blocks are ignored. If the edge is from an earlier operation to a later operation, then a corresponding edge annotated as read-after-write (RAW) is added to the dependency graph. If the edge from a later operation to an earlier operation, then a corresponding but reversed edge annotated as write-after-read (WAR) is added to the dependency graph. For simplicity, redundant edges, edges of the same type between the same operations, are omitted. Later analysis steps add edges that impose additional ordering constraints.

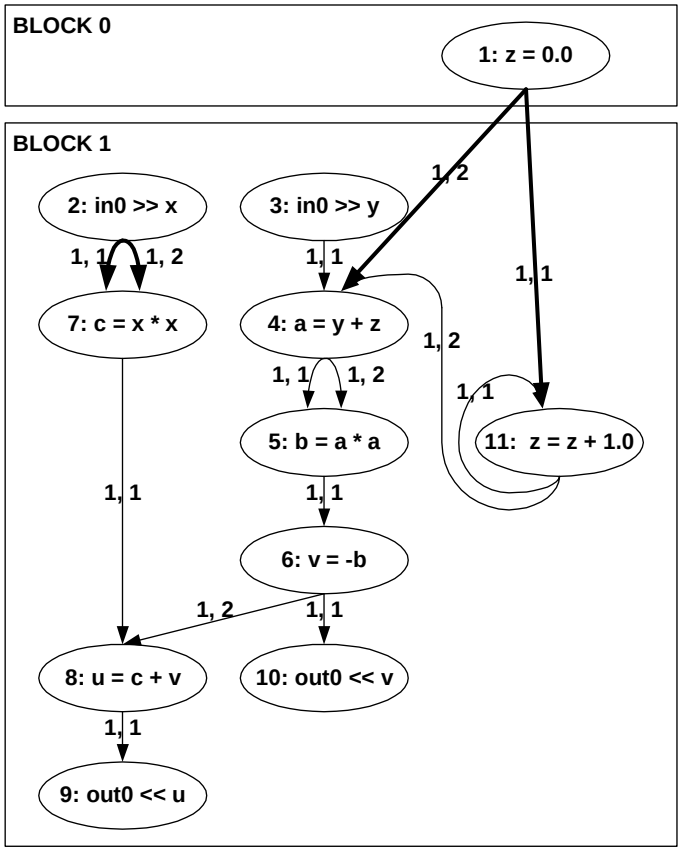



**FIGURE 5-3. Communication graph**

Figure 5-4 shows the dependency graph for the example kernel, along with a thumbnail of the communication graph. Two representative edges in the dependency graph are highlighted. The first highlighted edge is from operation 2 to operation 7. It was added to the dependency graph because there is an edge from operation 2 to operation 7 in the communication graph. Although there are two such edges in the communication graph, the redundant edge is omitted. The second highlighted edge is from operation 4 to operation 11. It was added to the dependency graph because there is an edge from operation 11 to operation 4 in the communication graph. Since the edge in the communication graph is from a later operation to an earlier one, it is reversed and annotated as a WAR edge in the dependency graph.

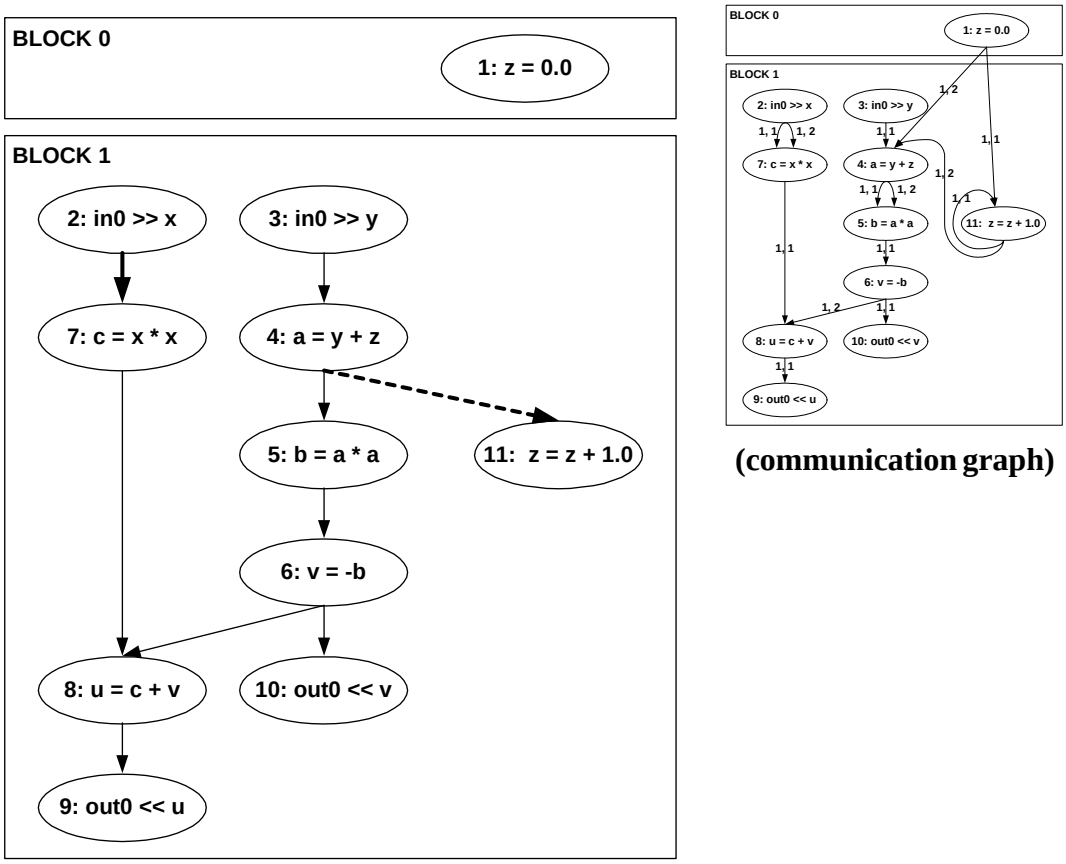



**FIGURE 5-4. Dependency graph**

### 5.1.5 Stream input/output ordering

The KernelC compiler must preserve the order of the operations used to read data from or write data to a stream so that the records within the stream are accessed in the expected order. To enforce this restriction, the KernelC compiler adds a dependency between each input operation that reads data from a stream and the next input operation that reads data from that stream in the same basic block. It similarly adds dependencies between output operations that write data to the same stream.

No register file exists to stage the values written to a stream so the operations that compute the values to be output, hereafter called the output computation operations, need to occur in the same order as the output operations. In most cases, the KernelC compiler adds a dependency from each output computation operation to the next output computation oper-

ation that computes data that is written to the same stream. However, sometimes all the output computation operations are not in the same basic block as the corresponding output operation. Other times there are dependency relationships among the output computation operations such that they cannot occur in the same order as the output operations. For example, one output computation operation may compute a value that is both written to a stream and used to compute another value that must be written to the same stream earlier. To resolve these situations, the KernelC compiler inserts a copy operation that copies the value to be output before some or all of the output operations. These copy operations become the new output computation operations for those output operations, and effectively stage the data through an existing register file.

Figure 5-5 shows the block 1 dependency graph before and after stream input and output ordering. For this example, the KernelC compiler adds an edge from operation 2 to operation 3 to order the input operations, and from operation 9 to operation 10 to order the output operations. It cannot add an edge from operation 8, the output computation operation for operation 9, to operation 6, the output computation operation for operation 10 since a contrary dependency already exists between those operations such that operation 6 must occur before operation 8. Instead, the KernelC compiler inserts a copy operation that copies the value of *v* just before the output. This operation becomes the new output computation operation for operation 10, and the KernelC compiler adds an edge to it from operation 8.

These copy operations can also be inserted using a source code transformation prior to constructing the communication and dependency graphs, which alleviates the need to update those graphs. Using this alternative, the KernelC compiler determines which output operations to insert copy operations before by examining each output operation for a stream in source code order. If the output computation operation for the current output operation is not in the same basic block, or appears before the output computation operation for the previous output operation, the KernelC compiler inserts a copy operation. This approach is overly conservative: output computation operations only need to be sup-

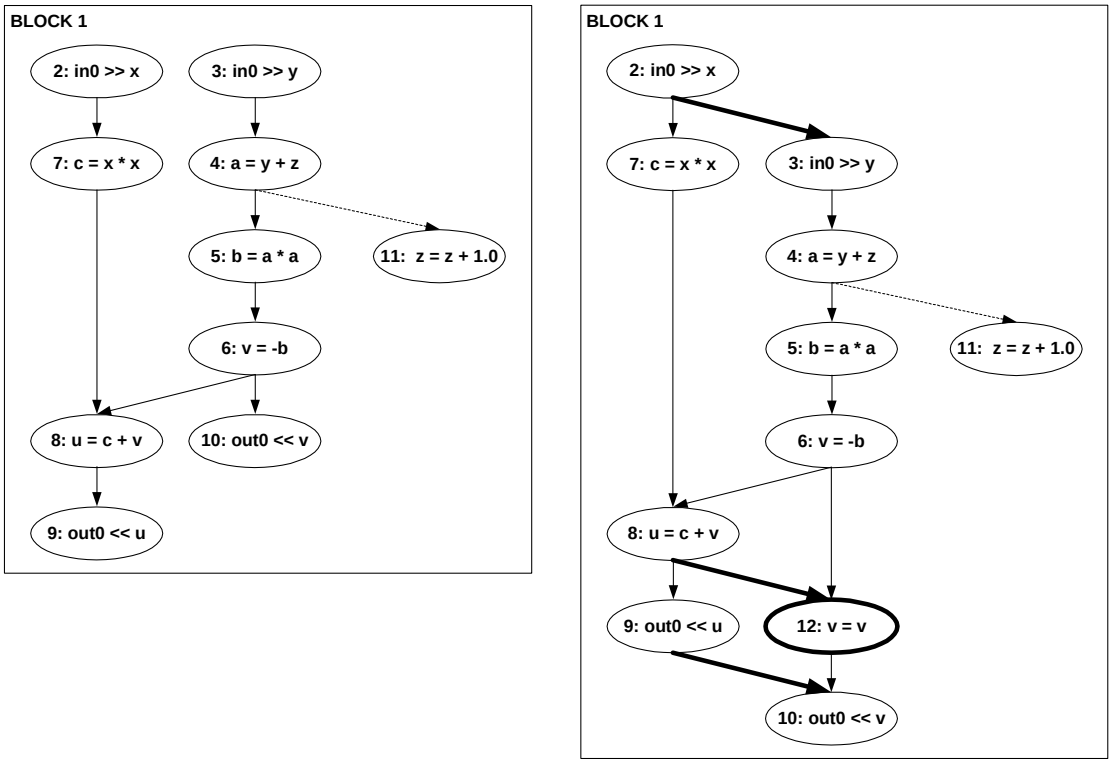



**FIGURE 5-5. Dependency graph before and after stream input/output ordering.**

planted by copy operations if they are ordered incorrectly in the dependency graph, not just in the source code.

### 5.1.6 Scratch pad access ordering

The KernelC compiler adds dependencies to order all scratch pad accesses that may read or write the same data. The scratch pad is used to hold small arrays. To avoid false dependencies between accesses to the same array, the KernelC compiler disambiguates such accesses based on their indices. The KernelC compiler disambiguates accesses with different constant indices, or with the same index variable and additions or subtractions of constants with a non-zero sum between the two accesses.

## 5.2 Scheduling

The scheduling process assigns each operation to a functional unit and schedules it on a cycle. The scheduling algorithm is optimized for use with communication scheduling and

a large number of functional units. Those optimization considerations dictate the order in which basic blocks are scheduled, the general structure of the scheduling algorithm, the order in which operations are scheduled, and the functional unit each operation is assigned to.

The examples in this subsection are taken from scheduling the coordinate transform kernel on the architecture presented in Figure 5-6. This simple architecture captures the primary features of a media processor like Imagine: a distributed register file architecture and stream input and output units.

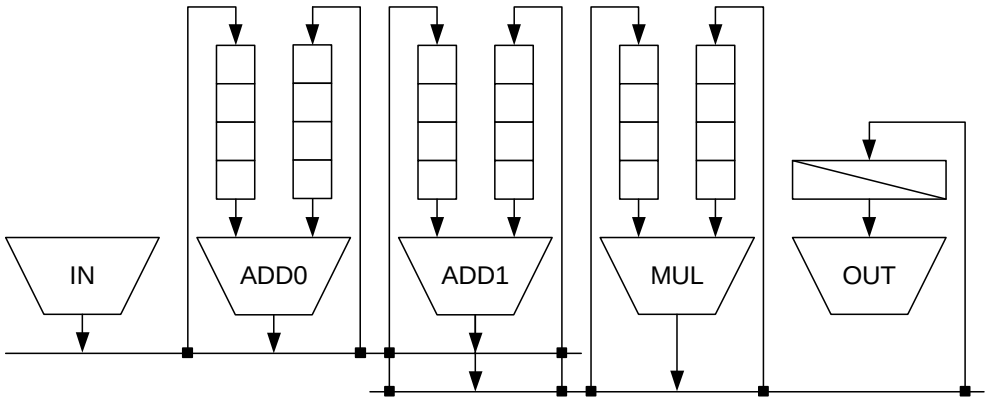



**FIGURE 5-6. Example architecture**

### 5.2.1 Basic block ordering

The KernelC compiler schedules the basic blocks that dominate execution time first so that those blocks have the greatest freedom for communication scheduling. Communication scheduling influences operation placement based on communications between operations in different basic blocks, so basic blocks cannot be scheduled independently. The basic blocks that are scheduled earlier do not need to close communications to operations in basic blocks that are scheduled later, so those basic blocks have the fewest constraints on operation placement. The KernelC compiler orders basic blocks based on deepest nested depth then largest number of operations. Assuming each loop is executed the same

large number of times, this order reflects the influence of each basic block on execution time.

In the example introduced in Figure 5-1, block 1 is more deeply nested than block 0, so it is scheduled first. Block 0 has more operations that block 2, the empty block after the loop, so it scheduled second. If block 0 was scheduled first, the placement of operation 1 (“z = 0.0”) would influence communication scheduling for operations 4 (“a = y + z”) and 11 (“z = z + 1”) in block 1, which could result in an inferior schedule for the inner loop.

### 5.2.2 Scheduling algorithm

The KernelC compiler uses the scheduling algorithm depicted in Figure 5-7. This algorithm is similar to the algorithm described in [40]. The KernelC compiler selects an operation based on a heuristic that considers whether the operation is on the critical path (how much slack it has) and a several other factors, and schedules it on the first possible cycle with an available functional unit. It then assigns the operation to one of the available functional units and attempts communication scheduling. If communication scheduling succeeds, the operation is scheduled. If communication scheduling fails, the KernelC compiler assigns the operation to a different functional unit, or delays it until a later cycle, until it succeeds.

The KernelC compiler is operation-driven rather than cycle-driven: it selects an operation and schedules it on the earliest possible cycle, rather than scheduling as many operations as possible on the current cycle before advancing to the next cycle. An operation-driven scheduler is better than a cycle-driven scheduler for use with communication scheduling because it ensures that communication between operations on the critical path are scheduled first. Consider a communication between two adjacent operations on the critical path, *o1* and *o2*. Using cycle order, the scheduler schedules *o1* then as many operations as possible on the current cycle before moving on to the next cycle. Those additional operations may occupy the interconnect resources needed to find an efficient route for the critical communication. When attempting to schedule *o2* on the next cycle, communication scheduling may be forced to delay it to insert a copy operation. Other operations that can

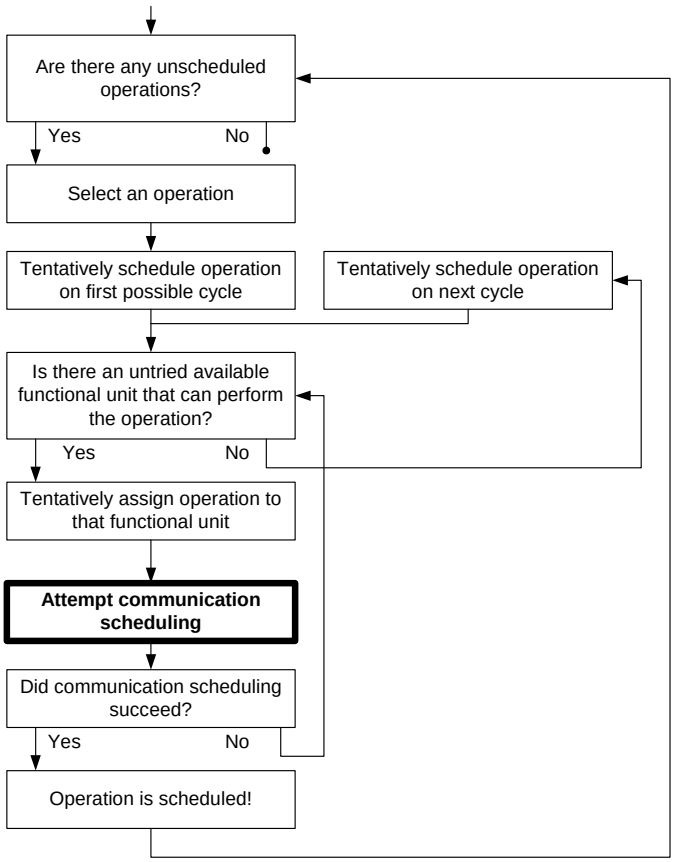



**FIGURE 5-7. Scheduling algorithm flowchart**

be scheduled on that cycle may occupy all functional units that could perform the copy operation, causing *o2* to be delayed even further. Using operation order, after scheduling *o1* the scheduler can immediately schedule *o2* on the next cycle since *o2* does not depend on any other operations. This order allows communication scheduling to assign the communication to an efficient route.

Figure 5-8 shows how a cycle-driven scheduler schedules the operations in basic block 1. On cycle 1, it schedules operation 2. On cycle 2, it schedules operation 3 then tries but fails to schedule operation 7 because a copy operation is required between operation 2 and operation 7. On cycle 3, it schedules operation 4, then operation 7 and the required copy operation. It also tries but fails to schedule operation 11 since both shared buses are occupied. On cycle 4, it attempts to schedule the critical-path operation 5, but fails because

every possible write stub for operation 7 occupies the only bus that can write into the multiplier's register file as shown by Figure 5-9. Delaying the critical path operation 5 results in an inferior schedule. This problem is exasperated if, after failing to schedule operation 11 on cycle 3, it schedules operation 11 on cycle 4 on adder 1, the only unit that can perform the copy operation for the communication from operation 4 to operation 5, making the schedule even worse.

| | IN | ADD0 | ADD1 | MUL | OUT |
|---|---|---|---|---|---|
| 1 | 2: in0 >> x | | | | |
| 2 | 3: in0 >> y | | 13: x = x | | |
| 3 | | | 4: a = y + z | 7: c = x * x | |
| 4 | | | | ~~5: b = a * a~~ | |

**FIGURE 5-8. Cycle-driven schedule**

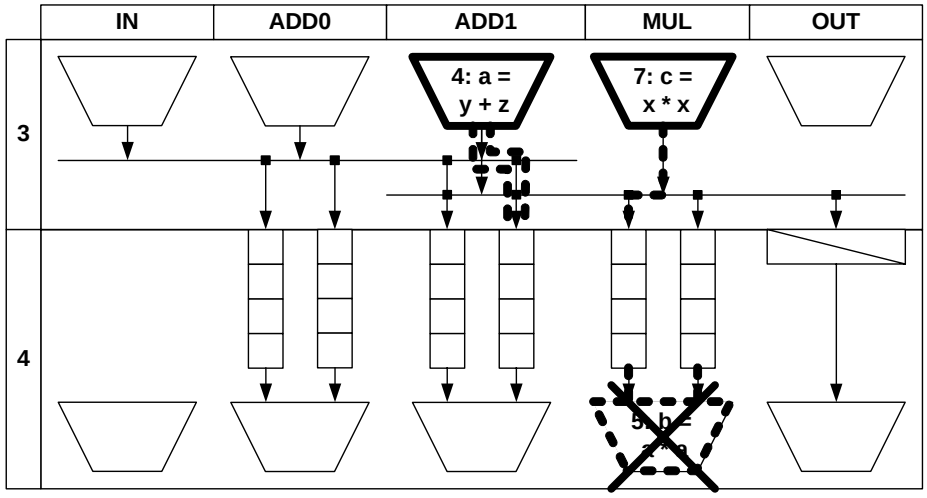



**FIGURE 5-9. Operation 5 can't be scheduled due to communication conflict**

In contrast, Figure 5-10 shows how the first several operations in basic block 1 are scheduled by an operation-driven scheduler. It schedules operation 2 on cycle 1, operation 3 on cycle 2, operation 4 on cycle 3, then operation 5 on cycle 4. Thus, it can close the communications from operation 4 to operation 5 as shown by Figure 5-11.

| | IN | ADD0 | ADD1 | MUL | OUT |
|---|---|---|---|---|---|
| 1 | 2: in0 >> x | | | | |
| 2 | 3: in0 >> y | | | | |
| 3 | | | 4: a = y + z | | |
| 4 | | | | 5: b = a * a | |

**FIGURE 5-10. Operation-driven schedule**

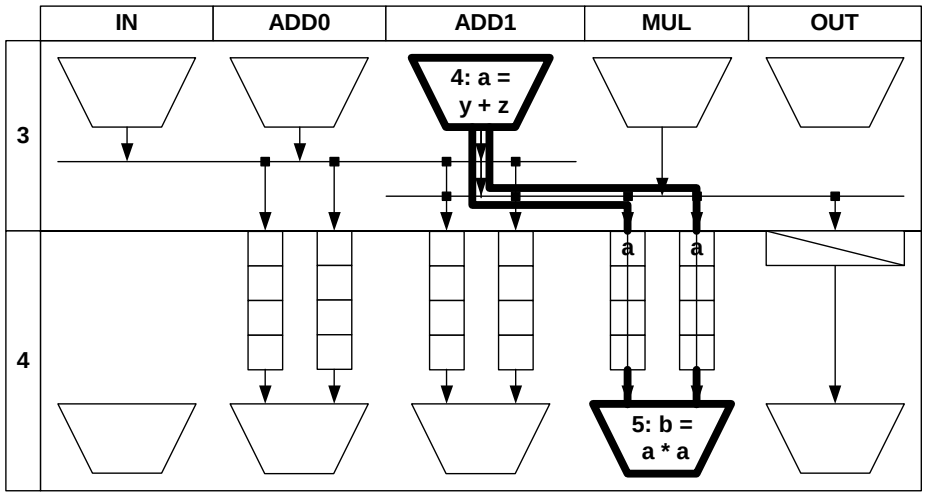



**FIGURE 5-11. Operation 5 can be scheduled without conflict**

The KernelC compiler uses a single phase that assigns each operation to a functional unit at the time it is scheduled on a cycle. Most VLIW schedulers use two phases that assign all operations to functional units, then schedule them on cycles. However, a two-phase scheduler delays an operation until a later cycle if the functional unit it is assigned to is already occupied or bus or register filer port conflicts prevent the use of that functional unit, even if another functional unit could perform the operation on that cycle. Shared bus and register file ports make effectively assigning operations to functional units ahead of time difficult. The single-phase KernelC compiler assigns each operation to an available functional unit on the earliest possible cycle. Further, with shared buses and register file ports, the functional unit an operation is assigned to influences communication scheduling for other operations. The KernelC compiler assigns each operation to a functional unit that allows for good communication scheduling (see the discussion of communication cost in Section 5.2) based on the actual schedule to date.

The optimal schedule for the example kernel is shown in Figure 5-12. Using a two-phase scheduler, operation 12 could reasonably be pre-assigned to the multiplier (since no other multiply operation could possibly conflict with it), but doing so would result in a communication conflict on cycle 5 as shown in Figure 5-13. This communication conflict results from operation 6 and operation 7 occurring on the same cycle. This conflict is almost impossible to predict statically given the complexity of scheduling operation 7 as described above. Using a single-phase scheduler, operation 12 is assigned to a functional unit at the time it is scheduled, after operations 6 and 7 have already been scheduled.

| | IN | ADD0 | ADD1 | MUL | OUT |
|---|---|---|---|---|---|
| 1 | 2: in0 >> x | | | | |
| 2 | 3: in0 >> y | | 13: x = x | | |
| 3 | | 11: z = z + 1 | 4: a = y + z | | |
| 4 | | | | 5: b = a * a | |
| 5 | | | 6: v = -b | 7: c = x * x | |
| 6 | | | 8: u = c + v | | |
| 7 | | | 12: v = v | | 9: out0<<u |
| 8 | | | | | 10: out0<<v |

**FIGURE 5-12. Optimal schedule**

### 5.2.3 Operation prioritization

The KernelC compiler uses a heuristic to determine the order in which operations are scheduled. This order is doubly important on an architecture with shared interconnect because operations compete not only for issue slots on particular functional units, but also for shared buses and register file ports. The earlier an operation is scheduled, the more interconnect resources are available to it. The KernelC compiler prioritizes operations based on a heuristic that considers a weighted combination of slack, latency, average usage of the functional units that can perform the operation, and distance from the edge of the dependency graph.

The KernelC compiler prioritizes operations with low slack above operations with high slack to keep the critical path as short as possible. Slack is the number of cycles between

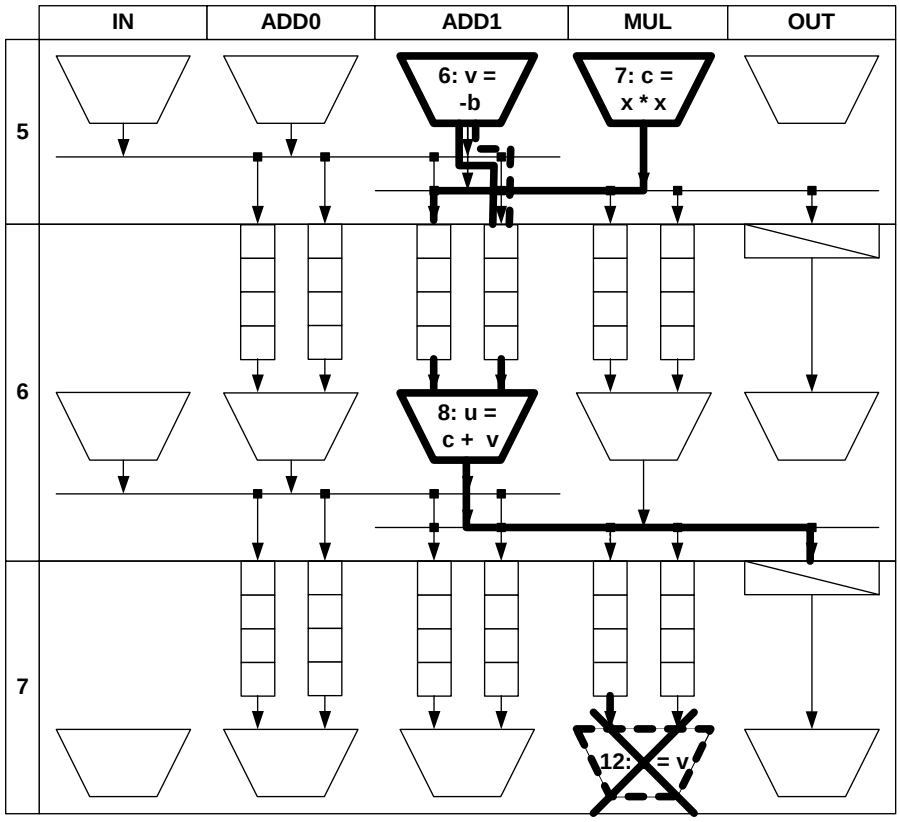



**FIGURE 5-13. Possible communication conflict with a two-phase scheduler**

the earliest possible cycle and the latest possible cycle an operation could be scheduled on in a minimum length schedule given infinite resources. The lower the slack, the greater the chance that delaying an operation will increase the schedule length. The scheduler recomputes the slack of unscheduled operations after each operation is scheduled to reflect the actual cycle on which that operation is issued.

The KernelC compiler prioritizes operations with high latency above operations with low latency so that the low latency operations can fill in the gaps between the high latency operations. A good analogy for this concept is that it is easier to pour sand into a bucket full of rocks than to pour rocks into a bucket full of sand.

The KernelC compiler schedules operations that can only be performed on busy functional units before operations that can be performed on relatively unused functional units. This

policy prevents issue slots on those busy functional units from being occupied by operations that could be performed by many kinds of functional units (such as copy operations), or lost due to bus and register file port conflicts with operations that could be scheduled differently. Before scheduling each basic block, the KernelC compiler first computes a measure of how busy each functional unit is for that basic block, called functional unit usage. The "raw" functional unit usage for a unit is the expected number of operations that would be assigned to the unit if each operation were randomly assigned to a unit that supports that operation. Raw functional unit usage is computed using Equation 1:

*O(u)*, the set of operations in the block that are supported by the functional unit *u*

*U(o)*, the set of functional units that support operation *o*

$$rawFunctionalUnitUsage(u) = \sum_{o \in O(u)} \frac{1}{|U(o)|} \quad (1)$$

The raw functional unit usage values are then normalized relative to the highest raw functional unit usage. For the example, functional unit usage is calculated as shown in Figure 5-14.

| | **IN** | **ADD0** | **ADD1** | **MUL** | **OUT** |
|---|---|---|---|---|---|
| 2: in0 >> x | 1.00 | | | | |
| 3: in0 >> y | 1.00 | | | | |
| 4: a = y + z | | 0.50 | 0.50 | | |
| 5: b = a * a | | | | 1.00 | |
| 6: v = -b | | 0.33 | 0.33 | 0.33 | |
| 7: c = x * x | | | | 1.00 | |
| 8: u = c + v | | 0.50 | 0.50 | | |
| 9: out0 << u | | | | | 1.00 |
| 10: out0 << v | | | | | 1.00 |
| 11: z = z + 1.0 | | 0.50 | 0.50 | | |

| | | | | | |
|---|---|---|---|---|---|
| 12: v = v | | 0.33 | 0.33 | 0.33 | |
| **raw functional unit usage(u)** | **2.00** | **2.17** | **2.17** | **2.67** | **2.00** |
| **functional unit usage(u)** | **0.75** | **0.81** | **0.81** | **1.00** | **0.75** |

**FIGURE 5-14. Functional unit usage calculation**

The KernelC compiler then prioritizes operations that can only occur on units with high functional unit usages. More specifically, it computes the average functional unit usage for all units that support each operation using Equation 2:

$$averageFunctionalUnitUsage(o) = \frac{\sum_{u \in U(o)} functionalUnitUsage(u)}{|U(o)|} \quad (2)$$

It then prioritizes operations that are supported by functional units with high average functional unit usage. For instance, in the example multiply operations are prioritized because the average functional unit usage for the functional unit(s) that support those operations, the multiplier, is 1.0.

Lastly, the KernelC compiler prioritizes operations that are close to the bottom edge of the dependency graph over those that are close to the top. This component of the heuristic counters the fact that slack considers the number of cycles an operation could be scheduled on, but ignores resource conflicts on those cycles. As high priority (low slack) operations are scheduled they occupy resources which could have been used by low priority (high slack) operations. If all high priority operations are scheduled first without regard for distance from the edge of the dependency graph, a long chain of low priority operations left until the end may be unable to fit into the body of the basic block due to resource conflicts, resulting in a “tail” of operations that significantly increases schedule length.

The kernel scheduler combines these four factors (slack, latency, average functional unit usage, and distance from the edge of the dependency graph) into a single weight using Equation 3, with high priority operations having the lowest weight.[1]

$$weight = (1 + slack) \times \left(1 - 0.8 \times min\left(\frac{latency}{10}, 1\right)\right) \times (1 - 0.8 \times averageFunctionalUnitUsage) \times (1 + 0.2 \times distFromEdge) \quad (3)$$

The KernelC compiler not only considers the intrinsic priority of an operation, it also considers the priority of the operations that the operation must be scheduled before. If an operation with high priority must be scheduled before an operation with a low priority, the low priority operation should be scheduled so that the high priority operation can be scheduled. Thus, the KernelC compiler gives each operation a final weight equal to the geometric mean of its own intrinsic weight and that of the lowest weight operation that it depends on (in the case of top-down scheduling) or that depends on it (in the case of bottom-up scheduling). For simplicity, weights are not updated transitively.

### 5.2.4 Functional unit assignment

The KernelC compiler determines which available functional unit to schedule an operation on using an heuristic that considers communication cost, functional unit usage, and least recent use. Functional unit selection is also more important in an architecture with multiple register files or shared interconnect because it determines how communications between that operation and other operations can be scheduled.

The KernelC compiler tries to assign the operation to a functional unit with low *communication cost*. Communication cost reflects the likelihood that assigning an operation to a functional unit will require copy operations to complete open communications, and the likelihood that those copy operations will increase schedule length. Assigning an operation to a particular functional unit can require copy operations to complete communications to or from that operation, to operations issued on the same cycle, or from operations that complete on the same cycle. This set of communications is the union of the sets *Cto*

1. The weight of each individual factor was determined experimentally.

and *Cfrom* introduced in Section 4.3. As shown in Equation 4, communication cost is calculated by taking the sum over this set of the minimum number of copy operations required to complete a route for each communication divided by the estimated size of the *copy range* for that communication, the cycles on which those copies could be scheduled.

$$communication\ cost = \sum_{communications\ in\ Cto\ and\ Cfrom} \frac{requiredCopies}{1 + |copyRange|} \quad (4)$$

The KernelC compiler uses communication scheduling to estimate the number of required copy operations by finding permutations of stubs for the open and closing communications in *Cto* and *Cfrom* (see Section 4.3) as though the operation were assigned to the functional unit in question, then counting the number of stubs that cannot form a route without a copy operation regardless of which functional units the unscheduled operations are assigned to. The copy range for each open communication is estimated by assuming that all unscheduled operations are scheduled on the latest possible cycle without increasing schedule length.

In the example, communication cost is critical when assigning operation 4 to an adder. Since no other operations are scheduled on cycle 3 when operation 4 is being scheduled, *Cto* and *Cfrom* only contain the communications to and from operation 4, shown in Figure 5-14. Operation 4 can be scheduled on either adder. However, only adder 1 is connected to the second shared bus. All communications to operation 4 are from operations that connect to the first shared bus, so none require a copy operation regardless of which adder operation 4 is scheduled on. However, both communications from operation 4 (shown in bold) are to operation 5, a multiply, and the multiplier is only connected to the second bus. Both communications require a copy operation if operation 4 is scheduled on adder 0. Since operation 5 is on the critical path immediately after operation 4, the copy range is estimated as 0 cycles. The communication cost for adder 0 is 2 * (1 / (1 + 0)) = 2. For adder 1, no communications require a copy operation so the communication cost is 0.

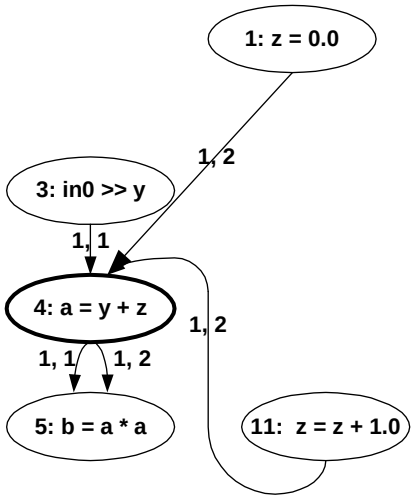



**FIGURE 5-15. Communications to and from operation 4**

The KernelC compiler also tries to assign the operation to a functional unit with low functional unit usage, as defined in Equation 1 in the previous section. This factor is crucial when assigning an operation that can be performed by many kinds of functional units to a functional unit, such as a copy operation. If a basic block contains a large number of operations that only one type of functional unit can perform, it is important that other operations be scheduled on other types of functional units whenever possible.

Returning to the example, operation 6, a negation operation, can be performed by either an adder (as “0 - v”) or a multiplier (as “-1 * v”). Since an adder has lower functional unit usage than a multiplier, it is scheduled on adder 1. This heuristic helps produce the optimal schedule, since a multiply scheduled later must occur on the same cycle.

Lastly, the KernelC compiler tries to assign the operation to a less recently used functional unit. This factor is weighted so as to be insignificant given a difference in communication cost or functional unit usage, but serves to distribute operations in the absence of such differences.

The KernelC compiler combines these three factors, communication cost, functional unit usage, and least recent use into a single weight, with the best functional unit having the lowest weight:[1]

$$weight = (1 + 10 \times communicationCost) \times (1 + 2 \times functionalUnitUsage) \times (1 + 0.01 \times leastRecentUseRank) \quad (5)$$

### 5.2.5 Randomization

By introducing a random component to the heuristics, the KernelC compiler uses multiple iterations of the scheduling process to explore the solution space for the best schedule. The KernelC compiler first schedules the kernel without any randomization, then schedules it again with the final weight of each heuristic multiplied by a random value between 0.5 and 1.5. The best schedule, selected by summing the schedule length of the most deeply nested basic blocks, is chosen as the final schedule. This simple technique is impractical for a standard compiler. It reduces the schedule length by at most one or two cycles, but that difference can be important for small, performance critical media processing kernels.

## 5.3 Post-scheduling

The post-scheduling process translates scheduled operations into machine code executable on the Imagine processing elements. The KernelC compiler allocates registers then generates the machine code.

### 5.3.1 Register allocation

The KernelC compiler allocates registers for each register file separately using conventional techniques [37]. It constructs webs, collections of uses of a variable that can be assigned to the same register, directly from the communication graph, then uses graph coloring to assign webs to registers.

1. Again, the weight of each individual factor was determined experimentally.

To construct the webs for a register file, the KernelC compiler only considers communications assigned to routes through that register file. It iterates through all such communications. If it encounters a communication that has not been assigned to a web, it initializes a new web and an empty worklist. It then pushes the communication onto the worklist. As long as the worklist is not empty, it pops a communication, *c*, off the top of the worklist. If *c* has not been added to a web then the KernelC compiler adds *c* to the web. It also pushes every communication that is either of the same result from the same operation as *c*, or of the same operand to the same operation as *c*, that has not been added to a web onto the worklist. When the worklist is empty, the web is complete.

Returning to the example, consider constructing the web for the left register file of adder 0. The KernelC compiler only considers the communications shown in Figure 5-16, since those are the only communications assigned to routes through that register file as shown in Figure 5-17. First, it selects the communication, *c1*, from operation 1 to operation 11, initializes a web and worklist, and adds *c1* to the worklist. Next, it pops *c1* off the worklist and adds it to the web. It then pushes the communication from operation 11 to itself, *c2*, onto the worklist. It then pops *c2* and adds it the web.

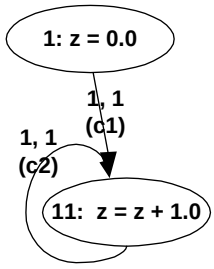



**FIGURE 5-16. Communications with routes through left register file of adder 0**

Once the KernelC compiler has constructed the webs for a given register file, it assigns webs to registers using a standard interference graph and graph coloring.

Since the KernelC compiler does not consider registers during the scheduling process it is possible for more registers to be required than are available in a register file. Register pres-

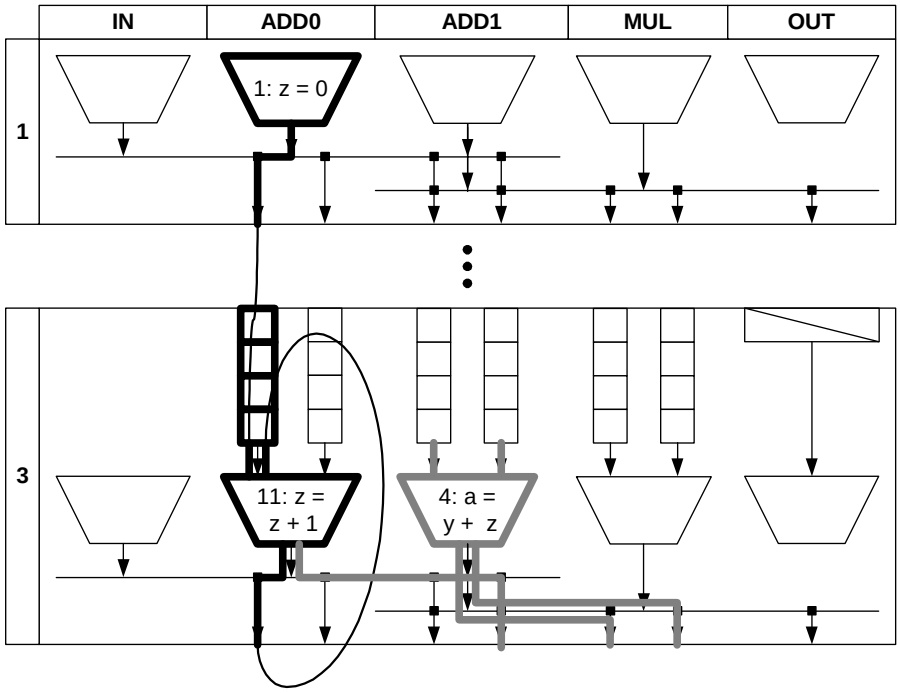



**FIGURE 5-17. Routes through left register file of adder 0**

sure is not as important for a media processor because the working set of most media processing kernels is relatively small and a distributed register file architecture supports a large number of registers. As implemented, the KernelC compiler does not incorporate a spilling mechanism. However, more complicated kernels (e.g. graphics kernels) and optimizations that increase the size of the working set such as software pipelining make consideration of register pressure during the scheduling process and/or spilling desirable.

### 5.3.2 Machine code generation

The KernelC compiler generates an instruction word encoding which operation to perform on each functional unit, which register to read or write through each register file port, and which driver drives each shared interconnect resource (bus or register file write port). This last component of the instruction word is unique to shared interconnect architectures. The instruction word can encode which driver drives each shared interconnect resource (*resource encoding*), or which resource is driven by each driver (*driver encoding*). The choice of encoding method is driven in part by instruction word size and in part by hardware implementation of the switches used to connect multiple drivers to each shared resource.

The example architecture contains four shared interconnect resources: the two shared buses and the shared register file port of each of two register files connected to adder1. Each shared bus can be driven by one of several functional unit outputs, and each shared register pile port can be driven by either bus. Consider cycle 3 of the example, shown in Figure 5-18. Adder0 drives the top shared bus (denoted bus0), adder1 drivers the bottom shared bus (denoted bus1), and bus0 drives the right register file of adder1 (denoted add1.rf1). The shared interconnect component of the instruction word can be encoded using driver encoding as shown in Figure 5-19 or using resource encoding shown in Figure 5-20.

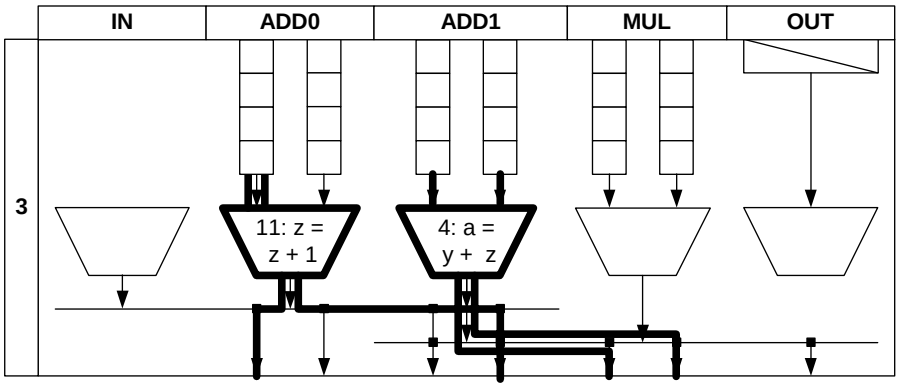



**FIGURE 5-18. Cycle 3 of final schedule**

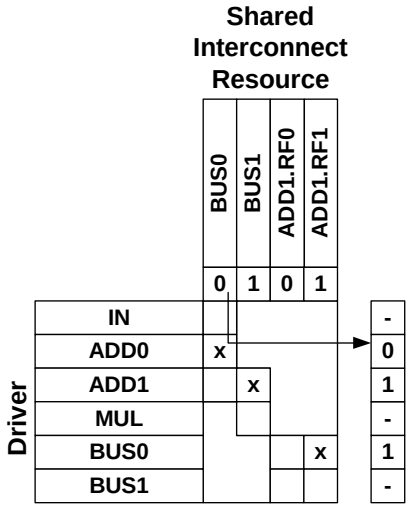



**FIGURE 5-19. Driver encoding**

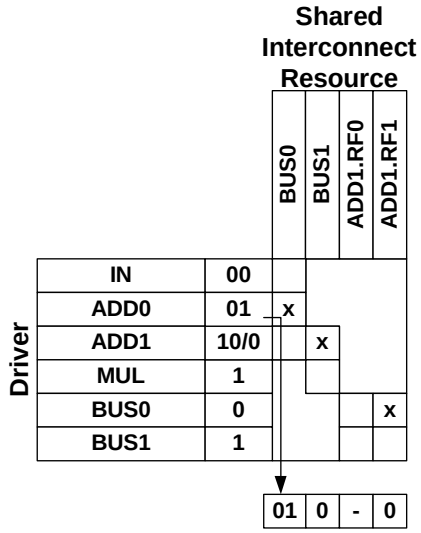



**FIGURE 5-20. Resource encoding**

## 5.4 Summary

This chapter described the KernelC compiler, a VLIW scheduler for the processing elements of the Imagine media processor. The KernelC compiler incorporates communication scheduling to allocate Imagine's shared interconnect, and stream input/output ordering to handle the sequential access requirement of a Imagine's stream register file. It uses heuristics optimized for scheduling small, performance critical media processing kernels to a target architecture with many functional units and shared interconnect to prioritize operations and assign each operation to a functional unit.

In comparison to general-purpose applications, media processing kernels motivate substantially different compiler design tradeoffs. Most media processing kernels center around a single computation intensive loop. Performance of that loop is critical, leading some programmers to resort to hand-placed operations. However, by focusing on operation placement and incorporating techniques such as randomization, the KernelC compiler yields results comparable to hand placement. The KernelC compiler allows programmers to concentrate on exploring algorithmic optimizations, and makes it easy to retarget high-performance code to new architectures.

# Chapter 6

# Stream Scheduling

This chapter presents stream scheduling, a compiler extension that efficiently manages the Stream Register File, an on-chip memory used by a stream processor like Imagine instead of a cache. Stream processors are optimized for media processing applications written using the steam programming model presented in Chapter 3. *Stream programs* consist of a series of operations on streams, sequences of records. Stream programs typically operate on large numbers of records with little data reuse, but access memory in a very predictable fashion. These characteristics make a hardware-managed cache unsuitable for a stream processor. Instead, a stream processor uses a large software-managed memory called a Stream Register File (SRF).

Stream scheduling buffers streams in the SRF based on a profile of the stream program in order to maximize performance by reducing the impact of memory access time. The profile captures the size and access pattern of each stream access and the data flow between stream operations. Stream scheduling assigns each stream access in the profile to a buffer in the SRF. Ideally, the results of one operation are buffered until used by another operation, eliminating memory accesses. Stream scheduling also arranges buffers to allow memory accesses to occur in parallel with execution. If the streams required for a single operation will not fit in the SRF, stream scheduling resorts to double-buffering to cycle the streams from memory through the SRF.

This chapter is divided into four sections. Section 6.1 presents the motivation for stream scheduling. Section 6.2 presents an overview of the stream scheduling algorithm. Section 6.3 describes the stream scheduling algorithm in detail. Section 6.4 discusses several important special cases for the stream scheduling algorithm.

## 6.1 Motivation

Efficient management of the SRF is essential for good performance because stream programs often demand more data bandwidth than the available memory bandwidth. Figure 6-1 shows a simplified diagram of Imagine annotated with available bandwidth. Imagine's eight processing elements can each read data from or write data to the SRF through special hardware buffers at an effective rate of 2 words/cycle. However, Imagine's two memory controllers can only transfer data between memory and the SRF at a rate of approximately 1 word/cycle each. Though most practical stream programs use less data bandwidth than the processing elements' maximum aggregate bandwidth of 16 words/cycle, most demand more than the memory system's aggregate bandwidth of 2 words/cycle. Thus, the SRF needs to satisfy a significant portion of most stream programs' data accesses in order to avoid performance degradation.

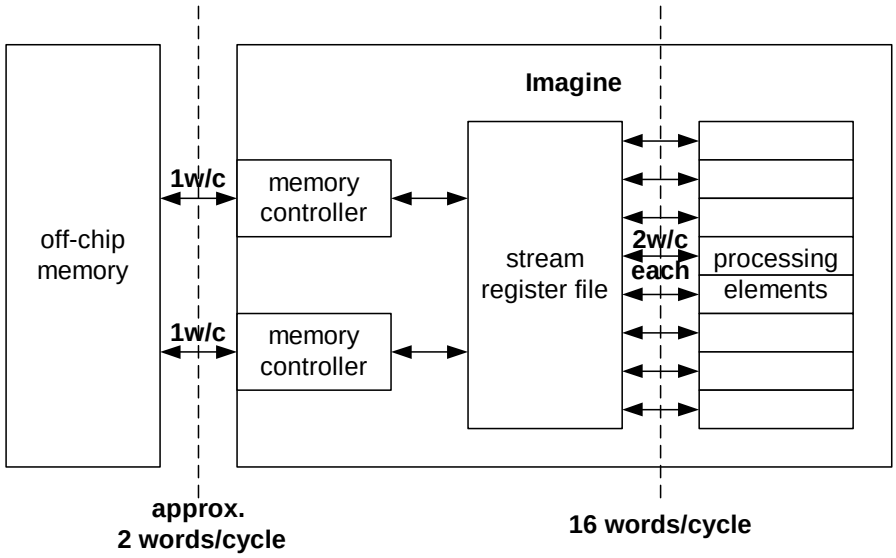



**FIGURE 6-1. Available bandwidth in Imagine**

Figure 6-2 shows a stream program that will be used to motivate the need for stream scheduling to efficiently manage the SRF.[1] The program consists of six kernels, labeled Kernel1 through Kernel6. The kernels read and write streams of integers, each of which is labeled with either a single letter or a letter and a "subscript." Each stream labeled with a single letter is a basic stream. Each stream labeled with a letter and a subscript (e.g. *a0*) is a derived stream that refers to some or all of the records in the basic stream labeled with that letter. The program contains several important cases. Kernel2 and Kernel3 both write to derived streams that refer to half of the basic streams *d* and *e*. Kernel4 then reads all of *d* and *e*. Kernel5 writes to the derived stream *fx*, a portion of *f* that is accessed using a stride of 2. Kernel6 writes to the large stream *g*, which is bigger than the SRF.

Consider managing the SRF at run-time, using a policy called *stream caching* that treats the SRF like a cache that holds streams instead of cache lines. When a stream operation is issued, stream caching allocates spaces for its input and output streams, in their order as arguments. If there is not enough space for a stream, stream caching ejects the least-recently-used stream from the SRF until there is enough space. More complex run-time strategies are possible, but would be difficult to implement given the relatively short execution time of most kernels on a high-performance processor like Imagine.

Stream caching suffers from four main inefficiencies. First, all output data must be stored back to memory because it might be ejected from the SRF and later reused. Second, data that is reused but has been accessed less recently is often ejected in favor of data that is never reused but has been accessed more recently. Third, because buffers are allocated at the first available space in the SRF, buffers can be arranged very inefficiently. For example, a small, recently used buffer in the middle of the SRF divides the SRF until it and all less-recently-used streams are ejected. Fourth, derived streams that compose a basic stream for later use can be assigned to buffers that are not adjacent in the SRF forcing the whole stream to be reloaded in order to be used.

1. For illustrative purposes, this example uses a 9 kiloword SRF. Imagine has a 16 kiloword SRF.

```
// This example assumes a 9 kiloword SRF (actual SRF is 16 kilowords)

Const int N = 2048;

// declare stream variables (indented variables are derived streams)
stream<int> a(N);
stream<int> b(N*2);
stream<int>       b0(b, 0, N);
stream<int>       b1(b, 1, N);
stream<int> c(N);
stream<int> d(N);
stream<int>       d0(d, 0, N/2);
stream<int>       d1(d, N/2, N);
stream<int> e(N);
stream<int>       e0 = e(0, N/2);
stream<int>       e1 = e(N/2, N);
stream<int> f(N);
stream<int>       fx(f, 0, N, FIXED, STRIDE, 2);
stream<int> g(N*10);

// call kernels
//       INPUTS    OUTPUTS
Kernel1( a,        b0, b1, c );

Kernel2( b0,       d0, e0    );

Kernel3( b1,       d1, e1    );

Kernel4( c, d, e,  f,        );

Kernel5( c,        fx        );

Kernel6( a, f,     g         );
```

**FIGURE 6-2. Example stream program**

Stream caching allocates the SRF for the example program as shown by the *SRF allocation graph* in Figure 6-3. Each horizontal bar indicates a kernel. The large vertical bar represents the SRF. The intersection of a kernel bar and the SRF bar contains the state of the SRF for that kernel. Shaded rectangles inside such an intersection indicate a stream access to that portion of the SRF by that kernel. Light shading indicates a read and dark shading indicates a write. Unshaded areas indicate streams that are preserved in the SRF between accesses. Diagonal cross-hatching indicates memory accesses. For example, the stream *a* is loaded from memory and read by Kernel1 but ejected to make room for the output of Kernel2.

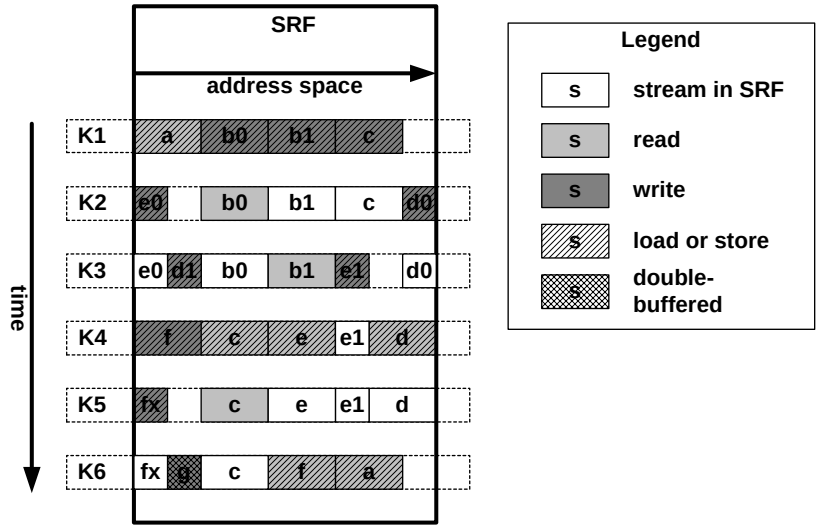



**FIGURE 6-3. SRF allocation graph for stream caching**

This example demonstrates all four of the main inefficiencies of stream caching. First, stream caching stores all three outputs of Kernel1 back to memory because they might be reused later. However, *b0* and *b1* are never reused. Second, when allocating space for the outputs of Kernel3, stream caching ejects *c* instead of *b0* to make room for *e1* because *c* was accessed less recently *b0*, despite the fact that *c* is reused and *b0* is not. Third, the small stream *e1* is allocated in the middle of the SRF. If Kernel4 needed a large buffer, *e1* would make it impossible to allocate one without ejecting *e1* and all streams accessed less recently than *e1*. Fourth, because stream caching allocates the derived streams that compose *d* and *e* when issuing separate kernels, the derived streams are not adjacent in the SRF so both *d* and *e* are reloaded for Kernel 4.

Stream caching also inefficiently sizes buffers for streams that are double-buffered because they are too large for the SRF. Double-buffering, described in more detail in the next section, cycles portions of a stream through two alternating halves of a buffer. The larger the buffer, the fewer times double-buffering needs to swap half-buffers, reducing overhead. However, that overhead is usually less costly than reloading an ejected stream. Using stream caching, either the double-buffered stream is buffered in the first available space, often resulting in a very small buffer, or older streams that might be reused are

ejected to make room for a larger buffer. In the example, stream caching uses the first policy and allocates *g* to an unnecessarily small buffer.

Stream scheduling is a compiler extension that manages the SRF better than a run-time technique such as stream caching because it allocates buffers ahead of time based on a profile of the program that enables it to consider all stream accesses, not just past accesses. Stream scheduling eliminates all of the inefficiencies of stream caching described above. Only output data that is ejected from the SRF and reused later needs to be stored to memory. Stream scheduling determines which streams to buffer in the SRF based on all uses, not just past uses. It arranges streams to make efficient use of the SRF and tries to assign derived stream accesses to adjacent buffers.

Stream scheduling allocates the SRF as shown in Figure 6-4, eliminating all unnecessary memory accesses. Only necessary memory accesses remain. The initial input *a* is loaded, and the final output *g* is stored. The streams *f* and *fx* are stored in order to combine data written with different access modes, then *f* is reloaded. Lastly, the stream *a* is stored and reloaded because it cannot fit in the SRF at the same time as the inputs and outputs of Kernel4. Stream scheduling requires a total of 18.5 kilowords of memory accesses. Stream caching requires a total of 34.5 kilowords of memory accesses. Further, because stream scheduling allocates a larger buffer for *g* than stream caching, *g* needs 5 double buffer swaps instead of 18. Using stream scheduling to produce the allocation shown in Figure 6-4 provides examples throughout the rest of this chapter.

## 6.2 Overview

To allocate the SRF for a program, stream scheduling assigns each *stream access* in a profile of the program to a *buffer* in the SRF as depicted in Figure 6-5. The profile lists the series of operations that compose the program, such as kernels, copies, and transfers to and from the host and network. For each operation, the profile records all *stream accesses*, reads from or writes to a stream, made by that operation. For each stream access, the profile records the start address, end address, and access pattern of the stream. The profile also notes which stream accesses are to streams that are variable length (access a data-

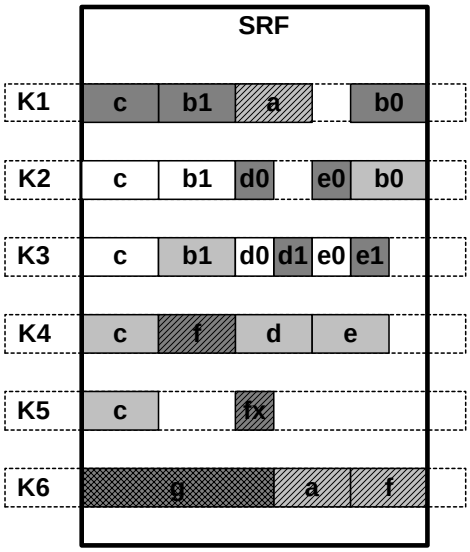



**FIGURE 6-4. SRF allocation graph for stream scheduling**

dependent number of records), or variable bounds (have a data-dependent start and end addresses).

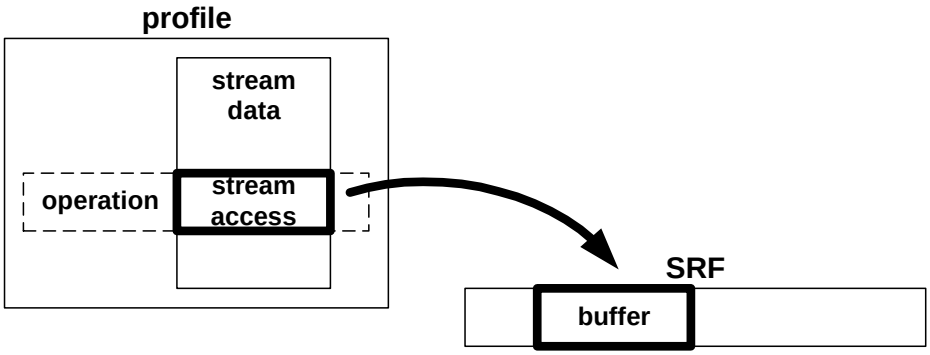



**FIGURE 6-5. Stream scheduling assigns each stream access to a buffer in SRF**

Figure 6-6 graphically depicts the profile of all the stream accesses in the example program. Each horizontal bar in Figure 6-6 represents a kernel. Each vertical bar represents a basic stream. Shaded rectangles where a kernel bar intersects a data bar represent a stream access to that data by that kernel. Light shading indicates a stream read, dark shading indicates a stream write. Vertical cross-hatching indicates stream access with a strided access

pattern. Figure 6-7 shows the SRF buffer each stream access in the example program is assigned to. The first stream access to the stream *a* is highlighted in Figure 6-6, and the buffer it is assigned to is highlighted in Figure 6-7.

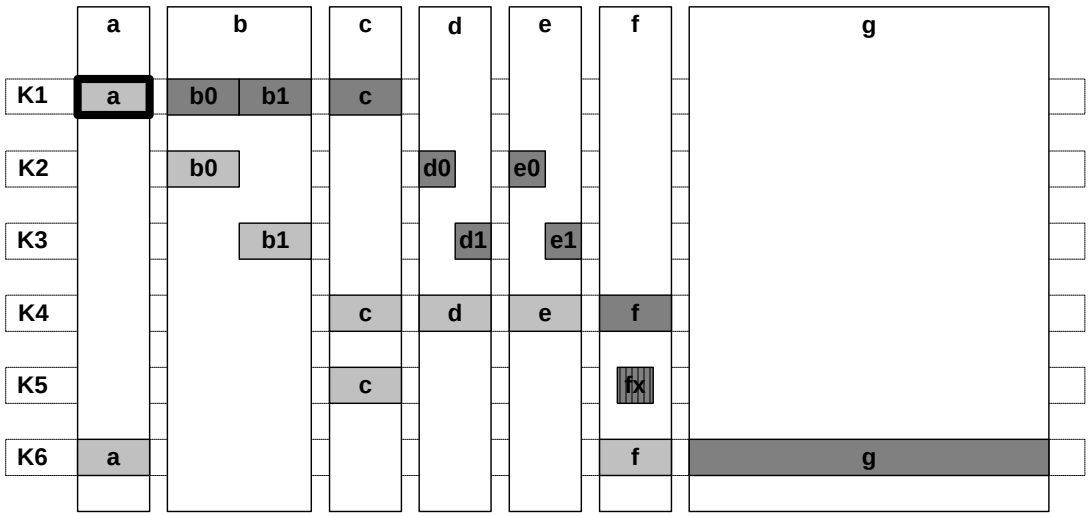



**FIGURE 6-6. Profile of all stream accesses for the example program**

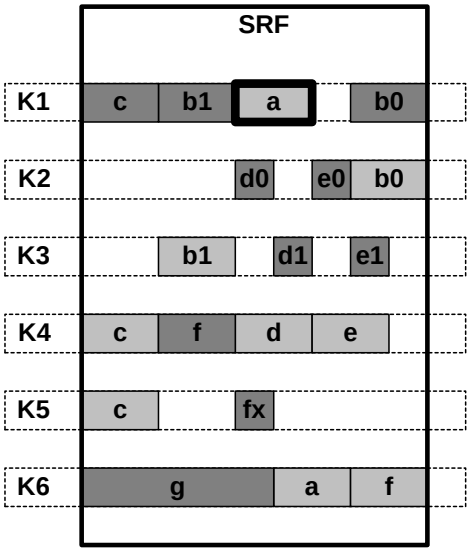



**FIGURE 6-7. Buffers for each stream access in the example program**

To avoid resource conflicts, stream scheduling assigns all stream accesses for a given stream operation to disjoint buffers (except accesses to the same stream). Stream scheduling attempts to assign each stream access to a buffer with a size equal to the size of the stream. If all of the streams accessed by a given stream operation cannot fit in the SRF, stream scheduling assigns one or more large streams to smaller buffers. At run time, *double-buffering* is used to cycle those streams through their buffers.

Double-buffering cycles portions of a large stream through two halves of a smaller buffer. Figure 6-8 shows the double buffering cycle used to read a stream. Initially, the first portion of the stream is loaded into half buffer 1. Then, the processing elements read the contents of half buffer 1 while a new portion of the stream is loaded into half buffer 2. When the processing elements are done with the portion of the stream in half buffer 1, the half-buffers swap roles. A new portion of the stream is loaded into half buffer 1, while the processing elements read the contents of half buffer 2. The half-buffers repeatedly swap roles until the entire stream has been read. Figure 6-9 shows the converse double buffering cycle used to write a stream.

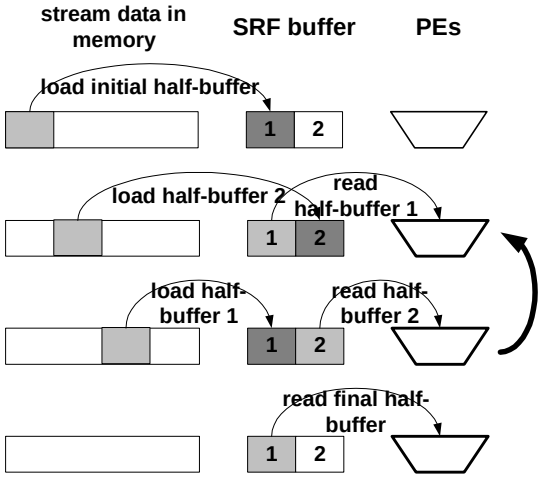



**FIGURE 6-8. Double-buffered stream read**

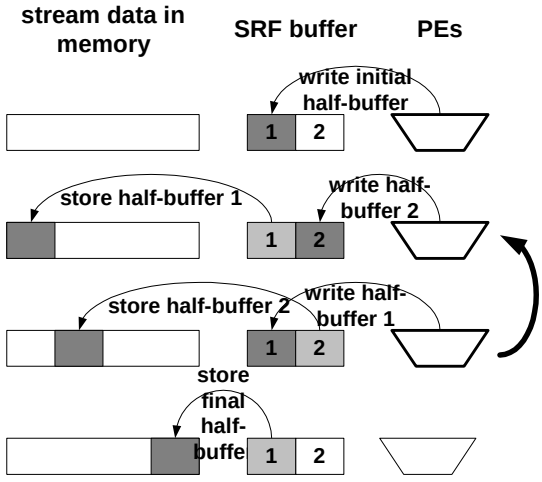



**FIGURE 6-9. Double buffered stream write**

Stream scheduling tries to assign stream accesses to the same data made by different operations to the same buffer and preserve that buffer between stream accesses in order to reduce memory accesses. The simplest, but worst method for performing two stream operations is to load all inputs from memory, perform the first operation, store all outputs back to memory, then do the same for the second operation as shown in Figure 6-10. If the two operations access the same data, assigning both stream accesses to the same buffer and ensuring that no intervening access is assigned to an overlapping buffer reduces needed memory accesses. In the best case, one stream operation writes data to a buffer in the SRF, and a later operation reads that data from the same buffer as shown in Figure 6-11. Even if one stream operation reads data that is loaded from memory, later operations can still read that data from the same buffer, eliminating the need for further loads.

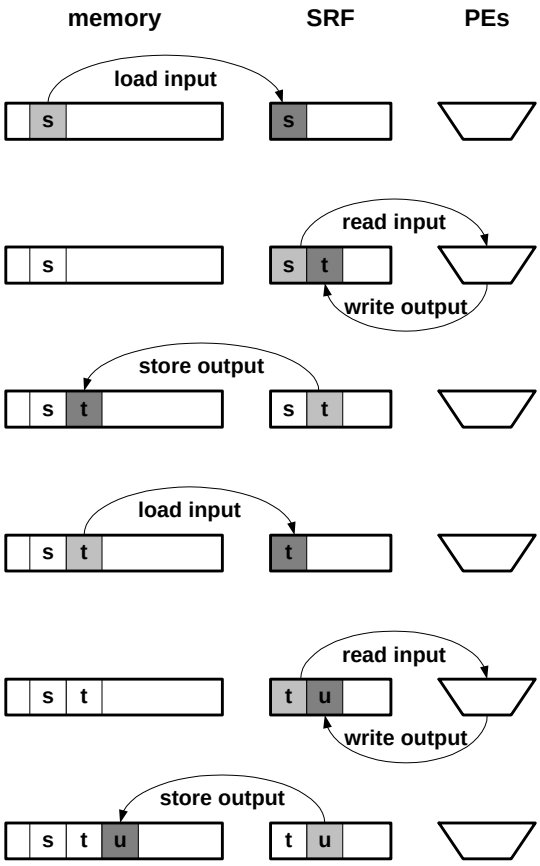



**FIGURE 6-10. Stream operations with accesses assigned to different buffers**

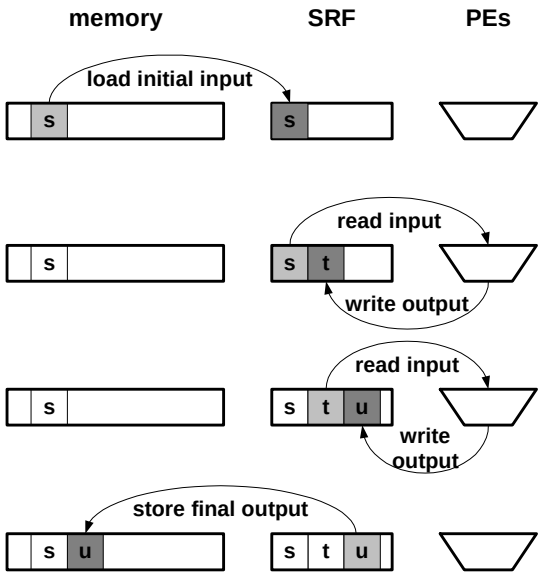



**FIGURE 6-11. Stream operations with accesses assigned to the same buffer**

If a stream access requires a memory access, stream scheduling tries to assign it and the stream accesses for an adjacent operation to disjoint buffers to allow the memory accesses to occur in parallel with execution as shown by Figure 6-12. If a stream read for one operation requires a load, stream scheduling tries to assign it to a buffer that does not overlap with a buffer assigned to an access made by the previous operation as with the load of *u* in Figure 6-12. Similarly, if a stream write for one operation requires a store, stream scheduling tries to assign it to a buffer that does not overlap with an access made by the next operation, as with the store of *t* in Figure 6-12.

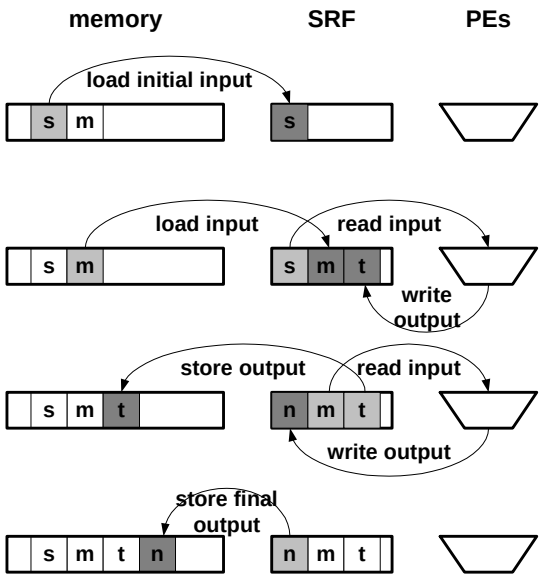



**FIGURE 6-12. Buffers arranged to allow parallel execution and memory accesses**

In order to reduce memory accesses and allow required memory accesses to occur in parallel with execution, stream scheduling allocates buffers in two dimensions: space and time. Each word in the SRF defines a unit of space. Each stream operation defines a unit of time. A stream access has a fixed height and width and a fixed location in time, but can be assigned to any location in space. Stream scheduling allocates a buffer as a rectangle in this two dimensional space that encloses the stream accesses assigned to it. By allocating a buffer shared by multiple accesses over time as well as space, stream-scheduling ensures that intervening stream accesses do not overwrite data contained in the buffer. Stream

scheduling extends the buffer in time slightly before the first access if it requires a load, and slightly after the last access if it requires a store. These extensions, called *load shadows* and *store shadows*, prevent overlapping buffers from being allocated for adjacent operations allowing the memory accesses to occur in parallel with execution.

Figure 6-13 shows the buffers for the example program in two dimensions. Two buffers are highlighted. The leftmost highlighted buffer shows how all stream accesses to *c* are assigned to the same buffer and no intervening stream access is assigned to an overlapping buffer, eliminating memory accesses. The rightmost highlighted buffer shows how the load shadow for the required load of the stream *a* prevents an overlapping buffer from being assigned to a stream access for the previous operation, allowing the load to occur in parallel with execution of Kernel5.

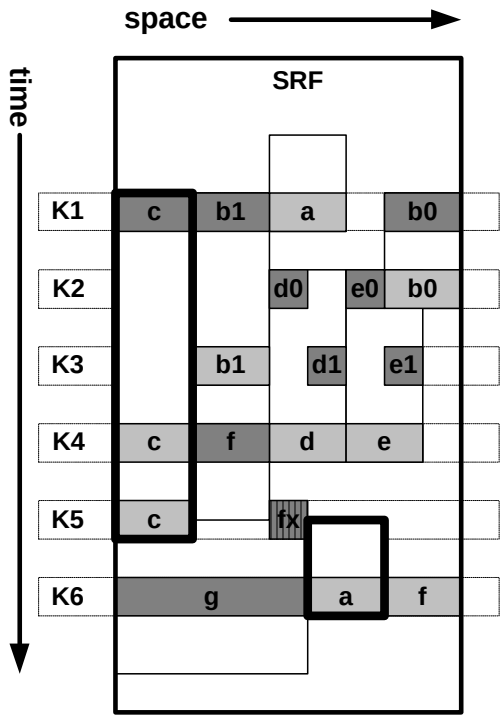



**FIGURE 6-13. Buffers in two-dimensions for the example program**

## 6.3 Algorithm

In essence, stream scheduling initially assigns each group of compatible stream accesses to a buffer, then repeatedly divides or shrinks buffers until all buffers fit in the SRF. More precisely, stream scheduling performs the following steps, each of which is described in detail later in this section:

1. Determine which stream accesses are double-buffered
2. Assign each group of compatible accesses to the same buffer
3. Mark stream accesses that require memory accesses
4. Repeatedly divide each buffer in time and space if it is possible to do so without requiring additional memory accesses
5. Extend buffers with load or store shadows
6. Position buffers in the SRF
7. If the buffers do not fit in the SRF, reduce a buffer and repeat steps 3-7

### Step 1. Determine which stream accesses are double-buffered

First, stream scheduling determines which stream accesses for each operation need to be double-buffered in order to ensure that all streams accessed by that operation can fit in the SRF simultaneously. It initializes the *double-buffer size* for each stream access to the size of the accessed stream, indicating that the stream access does not need to be double-buffered. If the total double-buffer size of the stream accesses for an operation exceeds the size of the SRF then stream scheduling reduces the double-buffer size of the stream access to the largest stream, forcing it to be double-buffered at run time. It sets the double-buffer size for that stream access equal to the size of the SRF minus the total of all other double-buffer sizes or a minimum double-buffer size, which ever is greater. If the total double-buffer size still exceeds the size of the SRF, it repeats the process with the stream access to the second largest stream, then with the stream access to the third largest stream, and so on until the total double-buffer size does not exceed the size of the SRF.

This policy is dictated by the fact that the cost of double-buffering one stream in a smaller buffer and reloading it later is usually less than the cost of double buffering two streams in larger buffers and reloading both streams later. The minimum double-buffer size is the

double-buffer size for which these costs are equal under reasonable assumptions. The overhead of double-buffering a stream is proportional to the number of half-buffer swaps, equal to the size of the stream divided by the size of a half-buffer. The cost of reloading a stream is equal to the size of the stream divided by the memory system bandwidth. A simple mathematical analysis shows that the minimum double-buffer size is equal to twice the overhead of a half-buffer swap divided by the memory system bandwidth, assuming two streams with sizes equal to the SRF size. For Imagine, the overhead of a half-buffer swap is approximately 300 cycles and memory system bandwidth is approximately 1 cycle per word so the absolute minimum buffer size is roughly 600 words.

Figure 6-14 shows the stream accesses in the example program with double cross-hatching used to indicate double-buffering. In the example, Kernel6 reads two 2 kiloword input streams, *a* and *f*, and writes a 10 kiloword output stream, *g*. Initially, the total double-buffer size for these stream accesses is 14 kilowords which is greater than the size of the 9 kiloword SRF. Stream scheduling reduces the double-buffer size of the stream access to the largest stream, *g*, highlighted below, to the total SRF size minus the double-buffer size of the other two stream accesses, or 9 - (2 + 2) = 5 kilowords.

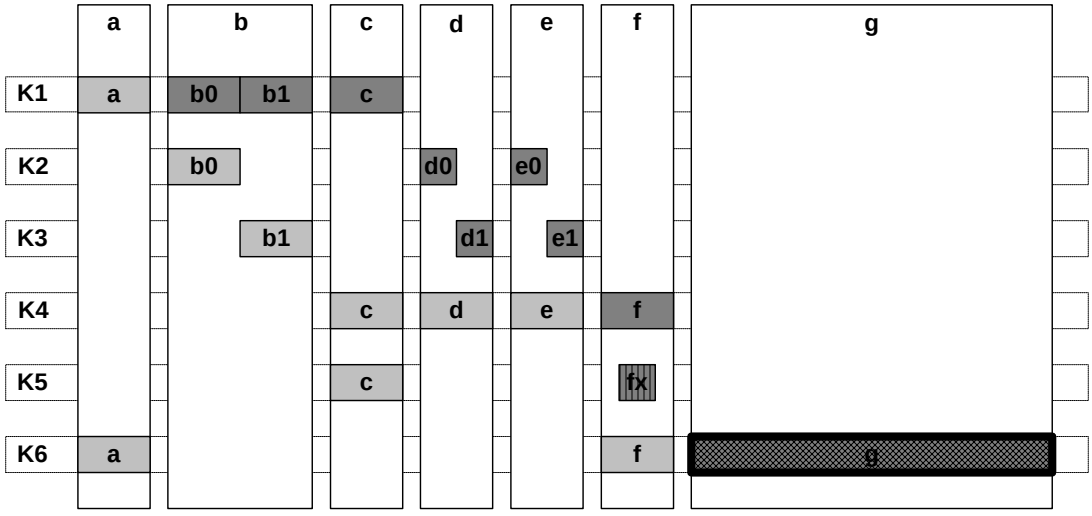



**FIGURE 6-14. Example program with double-buffering**

### Step 2. Assign accesses to buffers

Stream scheduling initially tries to assign as many *compatible* stream accesses to each buffer as possible. Two stream accesses are compatible if they access subsequences of a common supersequence of records. For example, a stream access to the first ten records in a basic stream is compatible with a stream access to the last ten records, but not with a stream access to all the odd records. A double-buffered stream access is inherently incompatible with all other stream accesses because it cycles data through a buffer, overwriting the contents.

SRF design limitations also restrict which stream accesses are compatible. If the SRF requires that all stream accesses start at an SRF addresses divisible by a fixed factor, then compatible streams must be portions of the same sequence of records that are offset by an amount divisible by that factor. For Imagine, the SRF requires that all accesses start with an SRF address divisible by 32. Assuming records are one word in size, an access to records 0 to 31 is compatible with an access to records 64 to 95, but not with an access to records 65 to 96.

Stream scheduling sets the initial size and time span of each buffer based on the stream accesses it encloses. The width of the buffer is equal to the size of the smallest common supersequence of records for all stream accesses assigned to the buffer, or the minimum double-buffer size of the double-buffered stream access assigned to the buffer. For example, a buffer that includes a stream access to records 30 to 39 and a stream access to records 50 to 59 has an initial width of 30 since the smallest common supersequence is records 30 to 59. Stream accesses are offset into the buffer by an amount equal to their offset into this smallest common supersequence. In the previous example, the access to records 50 to 59 is offset into the buffer by the size of 20 records. The initial time span of a buffer is from the operation that makes the first stream access assigned to it to the operation that makes the last stream access assigned to it.

Figure 6-15 shows the initial buffers for the example program. Two buffers are highlighted. The leftmost highlighted buffer contains three compatible stream accesses: to the

first half, second half, and all records of *d*. The buffer is sized to enclose all the accesses. The access to *d1*, the second half of *d*, is offset into the buffer by half the size of *d*. The rightmost highlighted buffer contains an access to *fx* which is not compatible with the stream accesses to *f* because it uses a different stride.

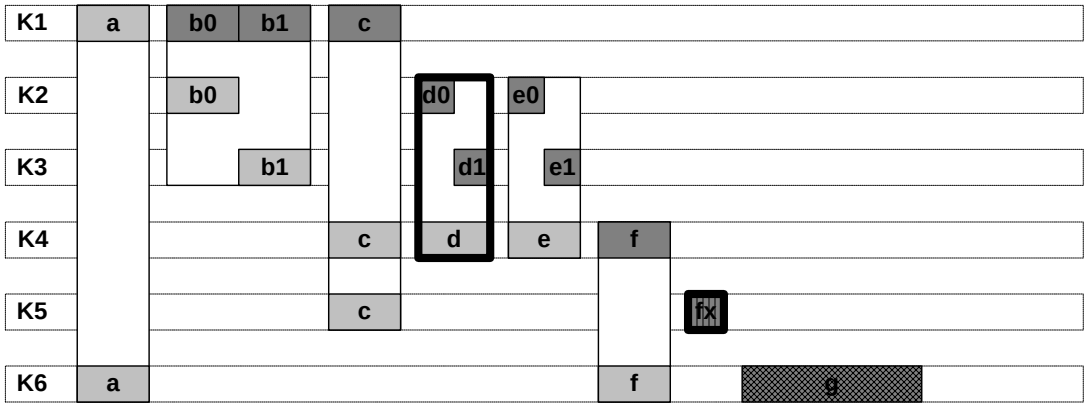



**FIGURE 6-15. Example program with initial buffers**

## Step 3. Mark stream accesses that require memory accesses

Stream scheduling marks all stream accesses that require memory accesses to synchronize data between buffers. Stream accesses assigned to different buffers can access the same data. If a stream write modifies data used by a stream read assigned to a different buffer then the stream write needs to store to memory and the stream read needs to load from memory to propagate the changes between buffers. If a stream write modifies data used by a stream read assigned to the same buffer that requires a load from memory then that stream write also needs to store to memory to incorporate the changes into the loaded data.

Stream scheduling first determines which stream writes reach which stream reads using data flow analysis. A stream write reaches a stream read if any of the records that it modifies could be used by that stream read. To make this determination, stream scheduling traces all paths of execution backward from each stream read, noting stream writes it encounters that could modify the records used by the stream read. It also accumulates a set of intervening accesses: all stream writes encountered and all stream reads encountered

that require loads to the same buffer as the current stream read. It terminates the current path when the set of intervening accesses *covers* the stream read. A set of stream accesses covers a specific stream access if the union of all records accessed by the set of stream accesses is a superset of all records accessed by the specific stream access. For instance, a set of two stream accesses, one to records 0 to 9 and one to records 10 to 19, covers a stream access to records 0 to 14, but not a stream access to records 0 to 29. To make sure the analysis terminates, stream scheduling records the set of intervening accesses when it enters a basic block and terminates a path when attempting to enter a basic block if it has already entered that basic block with the same set or a smaller subset of intervening accesses.

When it has determined which stream writes reach which stream reads, stream scheduling marks stream reads and then streams writes that require memory accesses. A stream read requires a memory access if it is reached by a stream write in a different buffer. A stream write requires a memory access if it reaches a stream read in a different buffer or a stream read which requires a memory load. Figure 6-16 shows a case in which a stream write in one buffer does not reach a stream read in another buffer, because intervening accesses cover the stream read. Figure 6-17 shows a case in which a stream write in one buffer reaches a stream read in another buffer, so both require memory accesses.

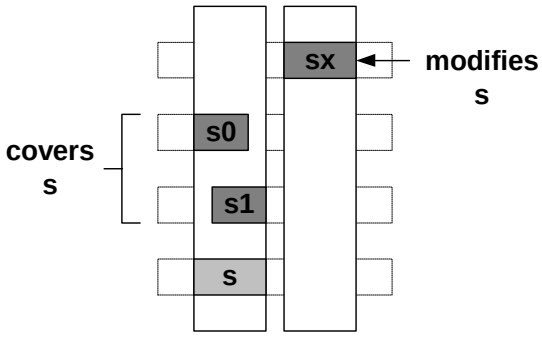



**FIGURE 6-16. Write to *sx* does not reach read of *s***

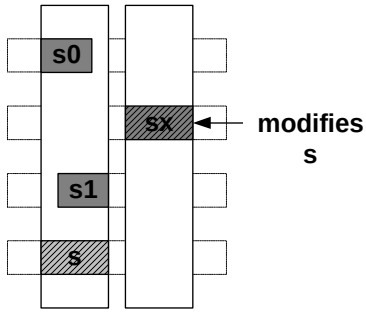



**FIGURE 6-17. Write to *sx* reaches read of *s***

In the example program, the stream accesses to *f* and *fx* involve the same data but are assigned to different buffers as shown in Figure 6-18. Stream scheduling analyzes the stream read of *f* as described above and finds that the stream writes to *f* and *fx* reach the stream read. Since the stream write to *fx* reaches the stream read and is assigned to a different buffer, the stream read requires a load. Both stream writes also require stores since they reach a stream read that requires a load.

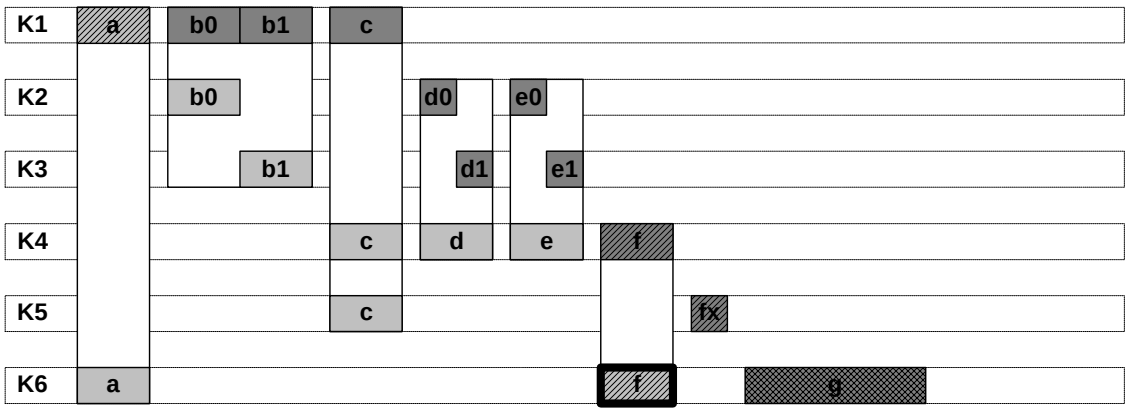
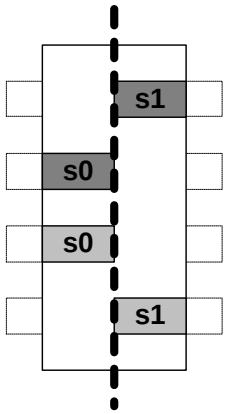



**FIGURE 6-18. Example program with required memory loads**

## Step 4. Repeatedly divide each buffer in space and time

Stream scheduling repeatedly divides each buffer in space or in time if it is possible to do so without inducing additional memory accesses. It is possible to divide a buffer in space if the stream accesses assigned to the buffer can be divided into two groups that access disjoint ranges of records. Stream scheduling divides the buffer at the separation between ranges as shown in Figure 6-19. It is possible to divide a buffer in time if the stream accesses assigned to the buffer can be divided into two groups such that no stream write in one group reaches a stream read in the other group that does not require a load and the span from the earliest access to the last access in each group is disjoint in time. Stream scheduling divides the two buffers as shown in Figure 6-20.

The example program contains two buffers that can be divided, highlighted in Figure 6-21. The buffer that contains accesses to *b0* and *b1*, the first and second halves of *b*, can be



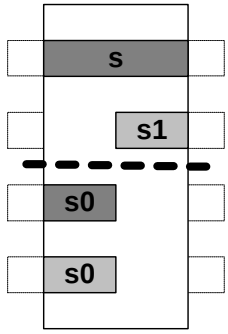




**FIGURE 6-19. Buffer divided in space** **FIGURE 6-20. Buffer divided in time**

divided in space into one buffer for each half. The buffer that contains accesses to *f* can be divided in time because the stream write to *f* only reaches a stream read that requires a load. Figure 6-22 shows the divided buffers.

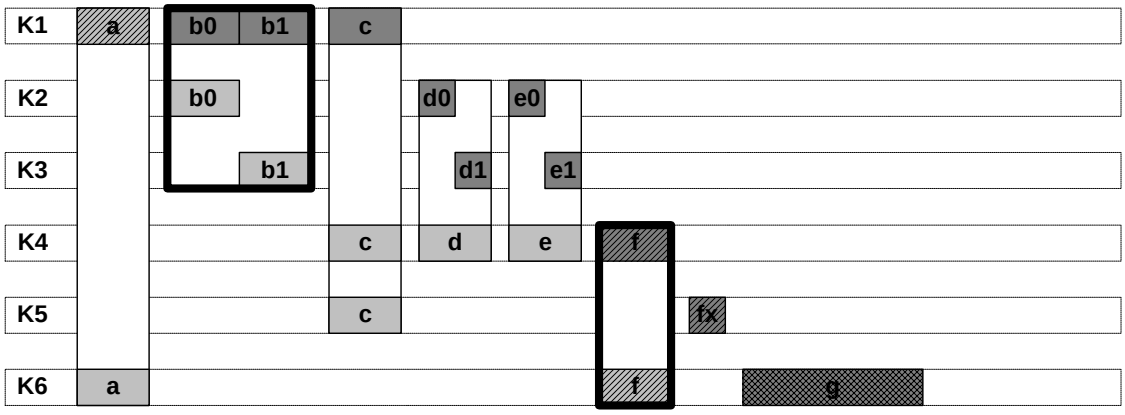



**FIGURE 6-21. Buffers that can be divided in the example program**

## Step 5. Extend buffers with load and store shadows

Stream scheduling extends buffers in time to allow memory accesses to occur in parallel with execution. By extending a buffer into an adjacent stream operation's unit of time, stream scheduling prevents buffers containing the adjacent operation's stream accesses from being allocated the same space in the SRF as the extended buffer. Thus, the memory

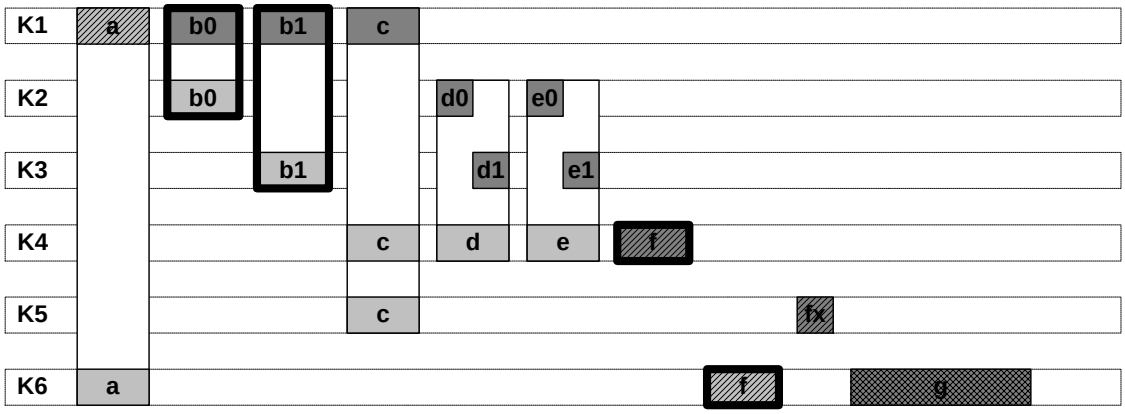



**FIGURE 6-22. Example program with divided buffers**

system can load or store the contents of the extended buffer in parallel with execution of the adjacent operation. However, if the adjacent operation writes the data that is loaded or reads the data that is stored, there is no point in extending the buffer because there is no possibility of parallelism.

Stream scheduling extends buffers based on the earliest and latest stream access assigned to each buffer. If the earliest stream access assigned to a buffer is a stream read, stream scheduling extends the buffer into the previous operation with a *load shadow* unless the previous operation writes the data that is being loaded. Similarly, if the latest stream access assigned to a buffer is a stream write, stream scheduling extends the buffer into the next operation with a *store shadow* unless the next operation reads the data that is being stored.

The example program contains three buffers that are extended with load and store shadows, highlighted in Figure 6-23. The buffer for the stream write to *fx* is extended with a store shadow because the next operation, Kernel6, does not read the data that is stored. Conversely, the buffer for the stream read to *f* is not extended with a load shadow because the previous operation, Kernel5, writes the data that is being loaded

## Step 6. Position buffers in the SRF

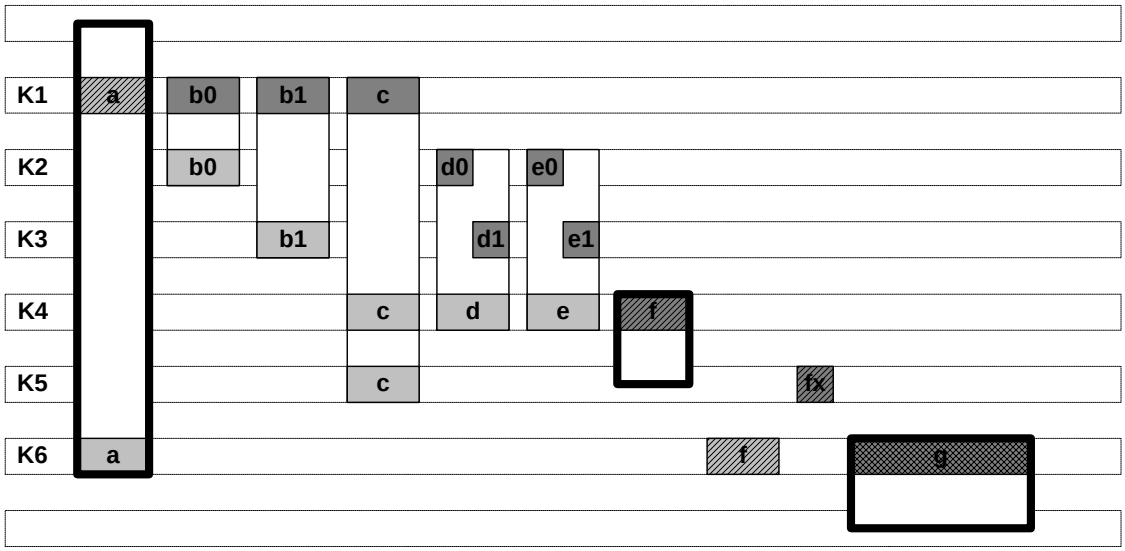



**FIGURE 6-23. Example program with load and store shadows**

Once stream scheduling has constructed the buffers, it attempts to position them in the SRF. Positioning the buffers in the SRF requires packing the rectangular buffers into a two-dimensional space with a fixed width equal to the size of the SRF. Each buffer has a fixed vertical position, but can have any horizontal position inside this space. This packing problem is the NP-hard “dynamic storage allocation” problem examined in [13][16]. Visualizing the buffers as rectangular wooden blocks on a rectangular grid, if there is an arrangement of buffers such that all of the buffers fit on the grid, it is possible to remove the buffers one at a time by sliding the rightmost buffer off to the right. Therefore, it is possible to position all of the buffers in the space by sliding each buffer as far left as possible in the reverse order of removal.

Stream scheduling selects buffers one at a time using a heuristic and positions each buffer at the leftmost possible position. The heuristic is based on the intuitive notion of trying to form long vertical strips of densely packed buffers. It tries to complete the current strip before starting another, and positions the largest buffers first so that smaller buffers can fill in the cracks. Stream scheduling selects a buffer based on three comparisons of successively less importance. It selects the buffer that can be positioned at the leftmost possible position among all buffers. If there is more than one such buffer, it selects the largest

buffer, based on its area. If there is more than one such buffer, it selects the buffer that contains the earliest stream access among those buffers.

Though stream scheduling nominally treats each buffer as a rectangle, buffers can be packed more densely by treating each buffer as a convex shell that “shrink-wraps” the stream accesses assigned to it. For this purpose, load and store shadows are treated as extending the earliest and latest accesses assigned to the buffer, respectively.

The buffers in the example program are positioned in the SRF in the order shown in Figure 6-24 to produce the arrangement shown in Figure 6-25. The first three buffers to be positioned are numbered. Initially, all buffers can be positioned at the leftmost margin so stream scheduling selects the largest buffer, buffer 1. All remaining buffers can be positioned at the right edge of buffer 1, so stream scheduling selects the largest remaining buffer, buffer 2. Next, only buffer 3 or the buffer containing the stream read of *f* by Kernel6 can still be positioned at the right side of buffer 1. Stream scheduling selects the largest of the two buffers, buffer 3. Unfortunately, this arrangement of buffers does not fit in the SRF (for this set of buffers there is no valid arrangement).

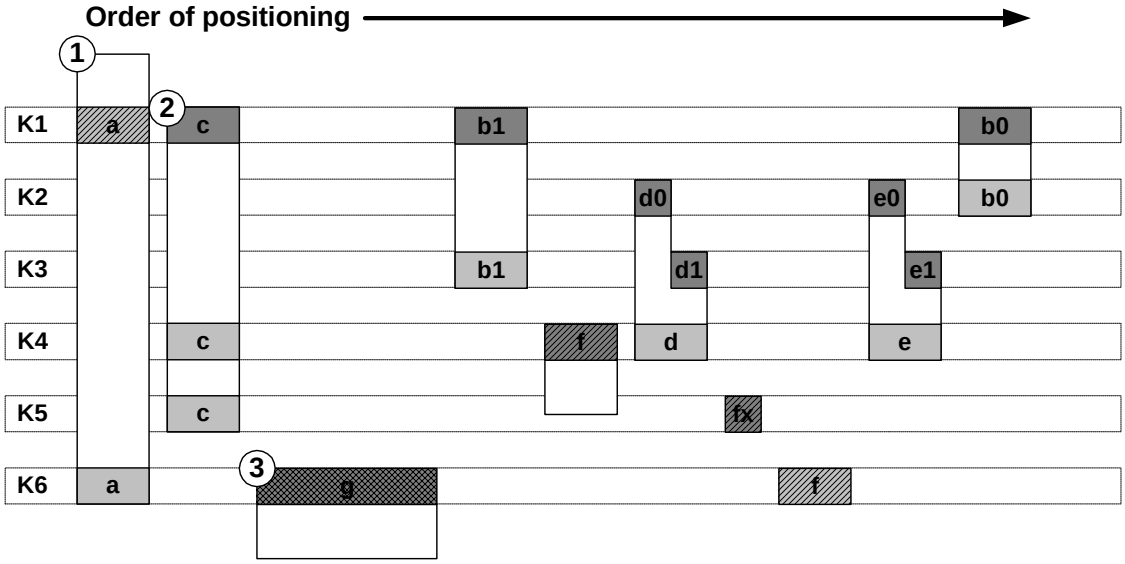



**FIGURE 6-24. Order of positioning for buffers in the example program**

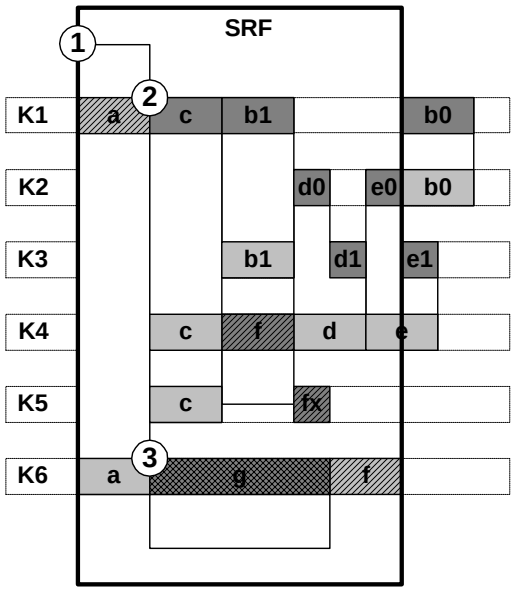



**FIGURE 6-25. First arrangement of buffers for the example program**

## Step 7. If the buffers do not fit, reduce a buffer and repeat Steps 3-7

If the arrangement of buffers produced in Step 6 does not fit in the SRF then stream scheduling reduces a buffer. Reducing a buffer involves one of two courses of action: shrinking a buffer that contains a double-buffered access or dividing a buffer in time.

To decide which buffer to reduce, stream scheduling first decides on which time step to reduce a buffer. If the total size of all the buffers on any time step is greater than the size of the SRF then there is no possible arrangement of buffers that will fit in the SRF. If such a time step exists, stream scheduling selects that time step. If no such time step exists, stream scheduling selects the time step with the buffer containing the stream access with the rightmost edge. If multiple time steps meet one of these criteria, stream scheduling selects the first such time step.

Next, stream scheduling decides which buffer on the chosen time step to reduce. As discussed in the description of Step 1, double-buffering overhead is less costly than reloading a stream, so stream scheduling shrinks a buffer that contains a double-buffered access whenever possible. If the chosen time step contains a double-buffered access that is larger

than the minimum double buffer size, stream scheduling shrinks that buffer. The logic of Step 1 ensures that there is at most one such buffer. If there is no such buffer, stream scheduling divides the buffer that makes the worst use of SRF space. In principle, the buffer with the most time between accesses makes the worst use of SRF space. However, dividing a large buffer that makes poor use of SRF space rather than a small buffer that makes good use of SRF space is bad trade-off if only a small amount of additional SRF space is needed. Hence, stream scheduling reduces the buffer with the greatest weight calculated by Equation 6.

$$\text{weight} = \frac{\text{time between accesses}}{\max(\frac{\text{buffer size}}{\text{needed size}}, 1.0)} \quad (6)$$

The time between accesses is the number of time steps between the latest access before the chosen time step and the earliest access after the chosen time step. Buffers with an access on the chosen time step have a time between accesses of 0. Buffers with load or store shadows on the chosen time step are treated as having a time between accesses of 2. The needed size is estimated by subtracting the size of the SRF from the total size of all buffers on the time step if the time step was chosen using the first criteria, or from the position of the edge of the rightmost buffer on that time step if the time step was chosen using the second criteria.

Stream scheduling then reduces the chosen buffer and tries to fit the buffers in the SRF again. It shrinks a buffer containing a double-buffered access by an amount equal to the needed size up to the minimum double-buffer size. It divides a buffer in time at the chosen time step. If only the load shadow or store shadow of the buffer crosses the chosen time step, it eliminates that shadow. After reducing the buffer, stream scheduling repeats Steps 3 through 7.

For the example program, stream scheduling chooses the time step on which Kernel4 occurs because the total size of the buffers on that time step exceeds the size of the SRF.

There are no double-buffered stream accesses on that time step, so it divides the buffer that makes the worst use of SRF space. Stream scheduling divides the buffer containing both reads from *a* as shown in Figure 6-26 because it has four time steps between accesses and a size equal to the needed space.

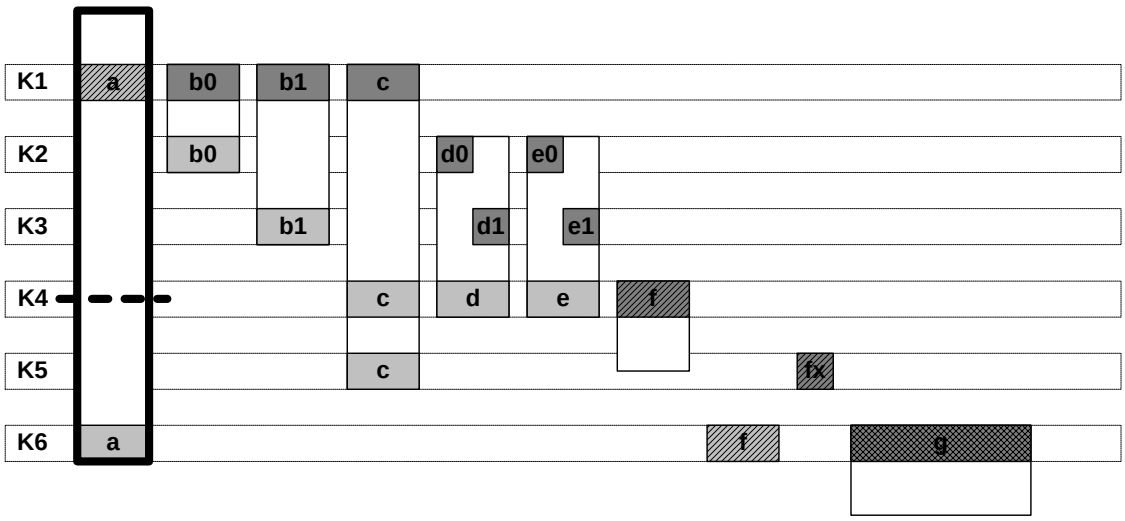



**FIGURE 6-26. Example program showing the buffer that is divided in time**

Stream scheduling then repeats Steps 3-7. Step 6 positions the buffers in the order shown in Figure 6-27 to produce the final arrangement of buffers shown in Figure 6-28.

### 6.3.1 Completion

Stream scheduling completes once all buffers fit in the SRF. Stream scheduling always completes. In the worst case, each buffer is repeatedly reduced until it contains only a single access and all load and store shadows are eliminated. Since Step 1 ensures that all accesses for each operation fit it the SRF, these single-access, shadowless buffers will always fit.

## 6.4 Special Cases

This section details several special cases that deviate from the general stream scheduling algorithm presented in the previous section. It describes the special handling and restric-

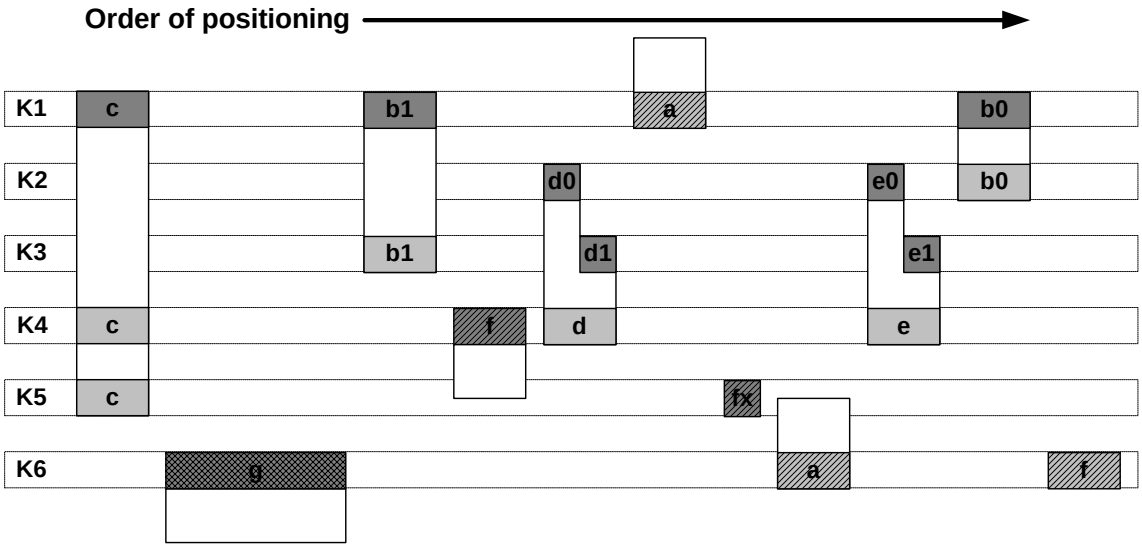



**FIGURE 6-27. Revised order of positioning for buffers in the example program**

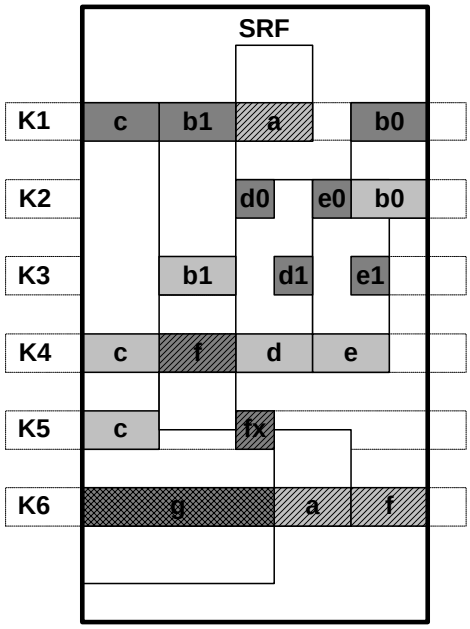



**FIGURE 6-28. Final arrangement of buffers for the example program**

tions for streams with variable lengths or bounds, streams with indexed access patterns, and stream operations other than kernels.

### 6.4.1 Variable length and variable bounds streams

Stream scheduling handles the unpredictability of variable length and variable bounds stream accesses by assuming the worst case. A variable length stream contains a number of records determined at run time. Stream scheduling assumes that a variable length stream access has maximum length when determining which streams it reaches or is reached by during data-flow analysis in Step 3, as shown in Figure 6-29. However, stream scheduling assumes that it has length zero when determining which stream accesses it covers, except for accesses to the same variable length stream, as shown in Figure 6-30. Variable bounds streams have start and end addresses that are determined at run time. An access to a variable bounds stream is not compatible with an access to any other stream since the two accesses could violate hardware restrictions regarding their start addresses. Since a variable bounds stream could access all or none of the records in the basic stream it is derived from it is treated like a variable length stream in Step 3.

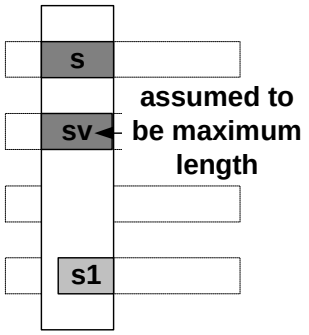



**FIGURE 6-29. write to *sv* reaches read of *s1***

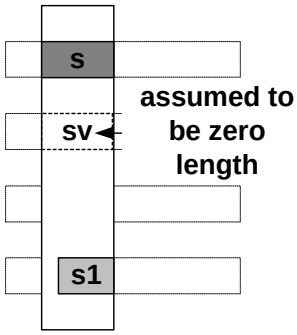



**FIGURE 6-30. *sv* doesn't cover *s1* so write to *s* also reaches read of *s1***

### 6.4.2 Indexed Streams

Stream scheduling treats any operation that uses a stream with an indexed access pattern as having the index stream as an additional input. The index stream contains the indexes of the records in the indexed stream within the basic stream that it is derived from. The index stream is only needed to load or store the indexed stream. Stream scheduling conservatively assumes that all uses of an indexed stream require the index stream, because division of buffers could force any stream access to load or store the stream. A more complex

approach would be to add and remove index stream accesses in Step 3 based on which indexed stream accesses require memory accesses. Since an indexed stream accesses an unpredictable set of records, an indexed stream access is handled like a variable length stream in the data-flow analysis of Step 3.

### 6.4.3 Other Stream Operations

Stream operations can be grouped into kernels, transfer operations, and copy operations. A kernel reads some input streams from the SRF and writes some output streams to the SRF. A transfer operation transfers a stream between the SRF and an explicit unit such as the host processor, network, or microcode store. A transfer operation is treated as a kernel that only reads one input stream or writes one output stream.

A copy operation requires special handling because it is performed by the implicit memory system. A copy operation makes two stream accesses: a stream read to the source stream and a stream write to the destination stream. However, it does not need to have a buffer in the SRF for both streams. If the source stream is written closer to the copy than the destination stream is read, the copy saves the source stream as the destination stream as shown in Figure 6-31. Otherwise, it loads the source stream as the destination stream as shown in Figure 6-32. In either case, the stream in the SRF is called the primary stream and the stream in memory is called the secondary stream. To handle a copy operation during SRF allocation, stream scheduling assigns the secondary stream access to a special buffer in Step 2. This buffer in never positioned in the SRF. Stream scheduling handles the primary stream access normally. Since the secondary stream access is the only stream access assigned to the special buffer, it always requires a memory access. In a postpass, stream scheduling maps the secondary steam access to the same location in the SRF as the primary stream access.

## 6.5 Summary

This chapter presented stream scheduling, a compiler extension for allocating the stream register file that replaces the cache in a stream processor. Stream scheduling allocates the SRF more efficiently than a run-time approach like stream caching because it uses a pro-

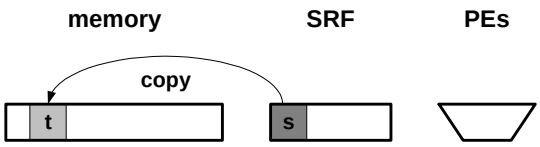



**FIGURE 6-31. if s is produced closer, streamCopy(s, t) saves s as t**

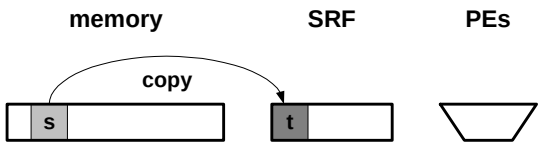



**FIGURE 6-32. if t is used closer, streamCopy(s, t) loads s as t**

file of the program that enables it to consider all stream accesses, not just past ones. Stream scheduling allocates the SRF in two-dimensions: space and in time. Essentially, it assigns all stream accesses in the profile to two-dimensional buffers, then tries to position all of the buffers in the SRF over time. If the buffers do not fit in the SRF, stream scheduling shrinks a buffer in space or divides a buffer in time and then tries again until it succeeds.

Stream scheduling uses the structure of a stream program to combine the performance advantages of programmer controlled memory access and the implementation efficiency of implicit memory access. Programmer controlled memory access usually yields better performance for media processing applications. However, hand-coding memory accesses is time consuming and architecture specific. Stream scheduling maps a stream program to any stream processor without programmer involvement.

# Chapter 7

# StreamC Compiler and Dispatcher

This chapter presents the *StreamC compiler* and *run-time dispatcher* used to compile stream programs and execute them on the Imagine media processor system, respectively, as shown in Figure 7-1. A stream program consists of a conventional program that contains kernel calls and stream copies and transfers as described in Chapter 3, hereafter collectively called *stream operations*. The StreamC compiler converts each stream operation in a stream program into a series of primitive operations executable by the Imagine processor, hereafter called *Imagine operations.* When the host processor is executing the stream program and encounters a stream operation, it invokes the run-time dispatcher, a software component that handles interaction with Imagine. The run-time dispatcher sends the corresponding Imagine operations produced by the StreamC compiler to the Imagine processor's *issue buffer*, a small on-chip buffer from which the Imagine processor issues operations. For example, the StreamC compiler might convert a kernel call into a series of four Imagine operations: writing two control registers, loading a stream, and executing the kernel. At run time, when the host processor reaches the kernel call, it invokes the run-time dispatcher which sends the four imagine operations to the issue buffer on Imagine.

This chapter describes how the StreamC compiler converts stream operations into Imagine operations using *profile compilation*. Profile compilation exploits the limited amount of data-dependent processing in a stream program by generating one or more near-static profiles of the program and compiling each profile. This technique makes the stream operations in the program more predictable, enabling the StreamC compiler to be more

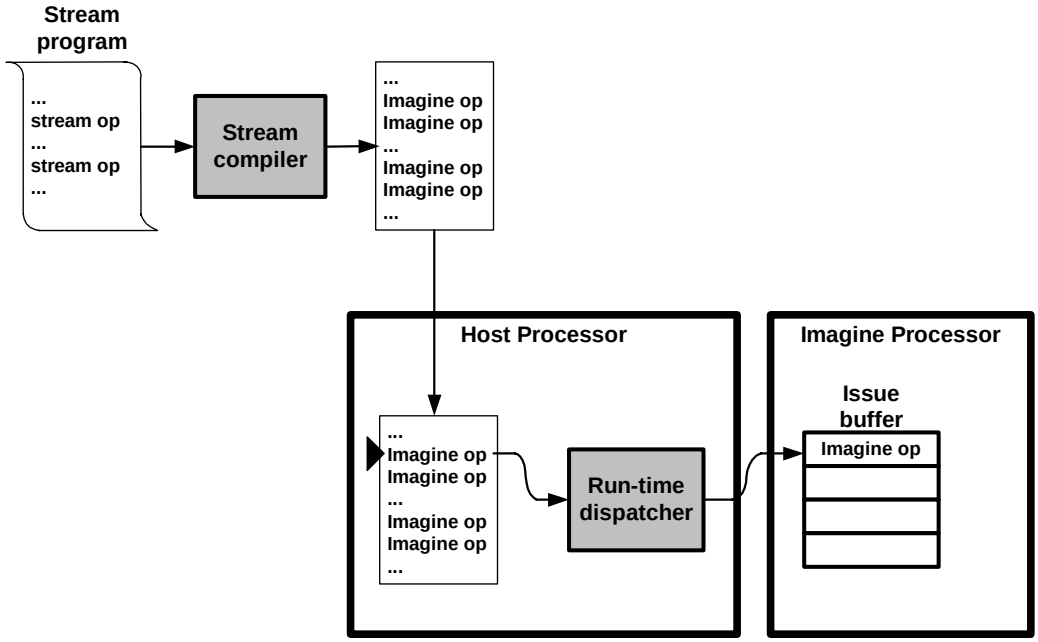



**FIGURE 7-1. Roles of the StreamC compiler and run-time dispatcher**

efficient. To compile a profile, the StreamC compiler converts each stream operation into one or more Imagine operations. First, the StreamC compiler allocates resources such as the stream register file. Next, it generates the Imagine operations. Lastly, it assigns each Imagine operation to an issue slot in the Imagine processor's issue buffer, and encodes dependencies on other Imagine operations that could be in the issue buffer.

The run-time dispatcher coordinates all communications between the host processor and the Imagine processor needed to execute a primitive operation. The primary role of the run-time dispatcher is to *dispatch* Imagine operations to the Imagine processor. It updates each Imagine operation to reflect any data-dependent streams, waits for the issue slot assigned to the Imagine operation to become empty, and places the Imagine operation in the issue slot. The run-time dispatcher also manages data transfers between the host processor and the Imagine processor and coordinates double-buffered stream accesses.

This chapter consists of three sections. Section 7.1 describes the StreamC compiler. Section 7.2 describes the run-time dispatcher. Section 7.3 discusses stream program optimizations that can be used to improve performance.

## 7.1 StreamC Compiler

This section presents the StreamC compiler. It discusses how the StreamC compiler uses profile compilation. It describes the three steps the StreamC compiler uses to convert each stream operation into Imagine operations: allocating resources, generating a sequence of Imagine operations, and assigning each Imagine operation to an issue slot and encoding its dependencies on other Imagine operations in the issue buffer.

### 7.1.1 Profile compilation

The StreamC compiler is optimized for stream programs, media processing applications written using the stream programming model that involve a predictable sequence of stream operations under a specific set of parameters. Media processing applications are characterized by a consistent transformation from an input data structure to an output data structure. The sequence of stream operations used to perform this transformation is dictated almost entirely by a small set of parameters, such as image size. For a fixed set of parameters, there are few data-dependent variations in the control flow between stream operations or the stream accesses made by those stream operations. Most data-dependent variations that do occur are simple iterative or conditional control-flow, or stream accesses with data-dependent length or bounds. StreamC requires the programmer to annotate these data-dependent variations as described in Chapter 3.

Figure 7-2 shows a stream program called EyeMatch that illustrates these properties. EyeMatch is the core of a simple optical identification system. It tries to find a match for a reference image of an eye in a live image using five steps, numbered 1-5 in Figure 7-2. First, it transfers the live image and the reference image into streams. Second, it computes a color histogram of the eye using a kernel called GenHist. Third, it compares the histogram of the eye to a histogram of each eye-size block of the image, recording the results as possible matches. Fourth, it sorts the possible matches and eliminates all possible matches

that do not meet a specific threshold. Lastly, it tests each remaining possible match using two comparison methods until a good match is found. For specific image sizes, there are few data-dependent variations in EyeMatch. Only the length of the stream of sorted possible matches, the position of each possible match tested, and the number of possible matches that are tested before a match is found are data-dependent. These variations are annotated as described in Section in Figure 7-2.

The StreamC compiler exploits the relative lack of data dependent variations by generating a profile of the stream program for a fixed set of parameters and compiling the profile. A profile is the actual sequence of stream operations used to execute a program. If the stream program contains any data-dependent control-flow, the program is divided into basic blocks based on the data-dependent control flow constructs. The sequence of stream operations used to execute each basic block is recorded only the first time the basic block is executed. For each stream operation, the profile contains the stream accesses that operation makes. For each stream access, the profile contains a unique identifier for the underlying basic stream, and the start address, end address, data dependence parameter, and access pattern (if applicable).

A profile can be generated dynamically based on actual execution, or statically using compiler analysis. Dynamic profile generation involves compiling the stream program using naive resource allocation, executing it, and recording the actual sequence of stream operations. Dynamic profile generation requires a test program that executes all stream operations in the stream program at least once to ensure a complete profile. However, it is simple to implement and provides typical length information for variable length streams. Static profile generation requires inlining all functions, unrolling all loops that are not data-dependent, applying very strong constant propagation, and extracting the stream operations. It is more robust than dynamic profile generation but more complex to implement and provides no information on variable length streams.

```
// tries to find a match for a reference image of an eye in another image
// returns true if successful
bool EyeMatch(int* imgPtr, int imgW, int imgH,  // live image to search
    int* eyePtr, int eyeW, int eyeH)            // reference image of eye
{
  stream<byte4> image(imgW * imgH);
  stream<byte4> eye(eyeW * eyeH);
  stream<byte4> eyeHist(256);
  int numBlocks = ((imgW / eyeW) * (imgH / eyeH));
  // struct PossibleMatch { int x; int y; float quality; };
  stream<PossibleMatch>  possibleMatches(numBlocks);
  stream<PossibleMatch>  sortedMatches(numBlocks, VARIABLE_LENGTH);
  PossibleMatch* possibleMatchesPtr = new PossibleMatch[numBlocks];

  // 1. load image and eye into streams
  StreamLoadBin(image, imgPtr, imgW * imgH);
  StreamLoadBin(eye, eyePtr, eyeW * eyeH);

  // 2. compute color histogram for the eye
  GenHist(eye, eyeHist);

  // 3. with each block of the image, compute color histogram,
  // compare to eye histogram, and add entry to possible matches
  int i = 0;
  for (int x = 0; x < imgW; x += eyeW) {
    for (int y = 0; y < imgH; y += eyeH, i++) {
      stream<byte4> block(image, y*imgW + x, (y+eyeH-1)*imgW + x + eyeW,
        FIXED, STRIDE, imgW, eyeW);
      stream<byte4> blockHist(256);
      GenHist(block, blockHist);
      CompHist(blockHist, eyeHist, possibleMatches(i, i + 1), x, y);
    }
  }

  // 4. sort the possible matches
  SortMatches(possibleMatches, sortedMatches);
  // save sorted matches from stream
  int numPossibleMatches = possibleMatches.readLength();
  StreamSaveBin(sortedMatches, possibleMatchesPtr);

  // 5. test possible matches until a match is found
  bool match = false;
  for_VARIABLE (int i = 0; i < numPossibleMatches && !match; i++) {
    int x = possibleMatchesPtr[i].x;
    int y = possibleMatchesPtr[i].y;
    stream<byte4> block(image, y*imgW + x, (y+eyeH-1)*imgW+ x + eyeW,
      VARIABLE_BOUNDS, STRIDE, imgW, eyeW);
    uc<int> matchUC1, matchUC2;
    CompImageA(block, eye, matchUC1);
    CompImageB(block, eye, matchUC2);
    match = matchUC1.readUC() || matchUC2.readUC();
  }

  delete possibleMatches;
  return(match);
}
```

**FIGURE 7-2. Stream program EyeMatch**

Figure 7-3 shows the profile of EyeMatch for a 200x200 live image and a 100x100 reference image. The profile lists each stream operation and all stream accesses made by each stream operation in EyeMatch.

```
// BASIC BLOCK 0
StreamLoad
  write image[0, 40000]

StreamLoad
  write eye[0, 10000]

GenHist
  read  eye[0, 10000]
  write eyeHist[0, 256]

GenHist
  read  image[0, 19900, FIXED, STRIDE, 200, 50]
  write blockHist0[0, 256]

CompHist
  read  blockHist00[0, 256]
  read  eyeHist[0, 256]
  write possibleMatches[0, 1]

...

GenHist
  read  image[20100, 40000, FIXED, STRIDE, 200, 50]
  write blockHist3[0, 256]

CompHist
  read  blockHist3[0, 256]
  read  eyeHist[0, 256]
  write possibleMatches[3, 4, VARIABLE_LENGTH]

SortMatches
  read  possibleMatches[0, 4]
  write sortedPossibleMatches[0, 4, VARIABLE_LENGTH]

StreamSave WAIT
  read  sortedPossibleMatches[0, 4, VARIABLE_LENGTH]

// BASIC BLOCK 1
for_VARIABLE_start

CompImageA
  read  image[0, 19900, VARIABLE_BOUNDS, STRIDE, 200, 50]
  read  eye[0, 10000]

CompImageB
  read  image[0, 19900, VARIABLE_BOUNDS, STRIDE, 200, 50]
  read  eye[0, 10000]

for_VARIABLE_end
```

**FIGURE 7-3. Profile of EyeMatch(--, 200, 200, --, 100, 100)**

Profile compilation allows the StreamC compiler to manage critical resources such as the stream register file much more efficiently, resulting in better performance. Without profile compilation, the StreamC compiler has to deal with more complex control flow and cannot determine the position and length of most streams. Consider allocating the stream register file under these conditions. The obvious policy involves a fixed number of buffers of constant size. This policy wastes SRF space by assigning small streams to buffers that are too large, and forces large streams to be double-buffered unnecessarily. Further, more complex control flow and inability to predict which stream accesses will be compatible would require frequent memory accesses to synchronize the contents of the SRF with memory. The SRF could be managed at run-time when more information about the current stream accesses is available, but inability to predict future stream accesses leads to significant inefficiencies as described in Section 6.1. Further, making the run-time dispatcher too complex can negatively impact performance if it takes more time to determine how to perform an operation than to perform the operation.

### 7.1.2 Resource allocation

The StreamC compiler allocates four types of resources for each stream operation in the profile: space in the stream register file (SRF), space in off-chip memory, space in the *microcode store*, and *control registers*. These resources are all explained in more detail in Section 2.3. The SRF is an on-chip memory that holds the working set of streams. The off-chip memory holds streams that cannot fit in the SRF. The microcode store is an on-chip memory used to store the compiled microcode for kernels that are executed by the processing elements. The control registers consist of stream descriptor registers (SDRs) that hold the position and length of a stream in the SRF and memory address registers (MARs) that hold the start address and access pattern of a stream in memory. Imagine operations do not specify this information, just refer to appropriate control registers.

The StreamC compiler allocates resources to each stream operation based on the type of operation and the stream accesses it makes. Later, it translates each stream operation into multiple Imagine operations that use those resources. The most important resource allocated by the StreamC compiler is the SRF, which is allocated using stream scheduling as

presented in Chapter 6. The StreamC compiler also uses variations on the stream scheduling algorithm to allocate the other resources. Stream scheduling is used to allocate a different kind of memory like the microcode store or the off-chip memory by redefining reads, writes, loads, and stores appropriately, removing any references to double-buffering, and making other minor changes specific to the resource. Stream scheduling is also used for ad hoc register allocation by treating each variable as a unit length "stream." This use of stream scheduling is primarily an implementation convenience. Conventional register allocation could also be used.

The StreamC compiler allocates the resources in a specific order because allocation of some resources influences allocation of other resources. It allocates the microcode store before the SRF because kernels that need to be loaded into the microcode store must pass through the SRF. It allocates the SRF before memory because streams that are spilled need be allocated space in memory. It allocates control registers last because they are used to refer to space in the SRF and memory.

First, the StreamC compiler allocates the microcode store. It uses a version of stream scheduling that only considers kernel calls. Each kernel call reads the microcode for that kernel. If a microcode read requires a load, that load indicates that the microcode needs to be loaded from memory into the SRF then read from the SRF into the microcode store. To handle these loads, the StreamC compiler inserts a special stream operation called a KernelLoad before each such kernel call after the microcode store has been allocated.

Figure 7-4 shows the microcode store allocation for EyeMatch, using the conventions introduced in Chapter 6. Only kernels are considered, and each kernel makes a single microcode read. Cross-hatched accesses indicate kernel calls that require loading the kernel from memory. The ellipsis indicates omitted kernel calls. The kernel GenHist, highlighted in Figure 7-4, is representative. It is loaded at the start of the allocation and remains in the microcode store until the last call to the kernel.

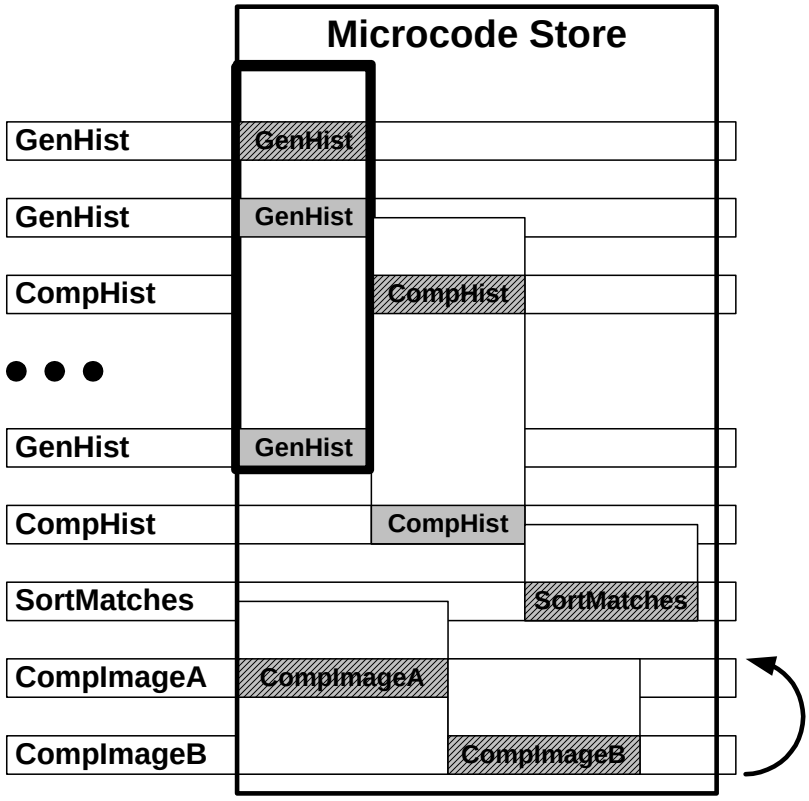



**FIGURE 7-4. Microcode store allocation for EyeMatch**

Second, the StreamC compiler allocates the SRF using the stream scheduling algorithm as presented in Chapter 6. Stream scheduling assigns each stream access to a location in the SRF, and determines which accesses require memory loads and/or stores. Stream scheduling also determines which stream accesses are double-buffered.

Figure 7-5 shows the SRF allocation for EyeMatch. This application contains a relatively high amount of memory traffic. However, all memory traffic shown is required for this application given the size of the SRF, not induced by poor SRF allocation.

Third, the StreamC compiler allocates off-chip memory. Again, it uses a variation of the stream scheduling algorithm. It considers only stream reads that require loads and stream writes that require stores. Since memory does not have many of limitations of the SRF, it

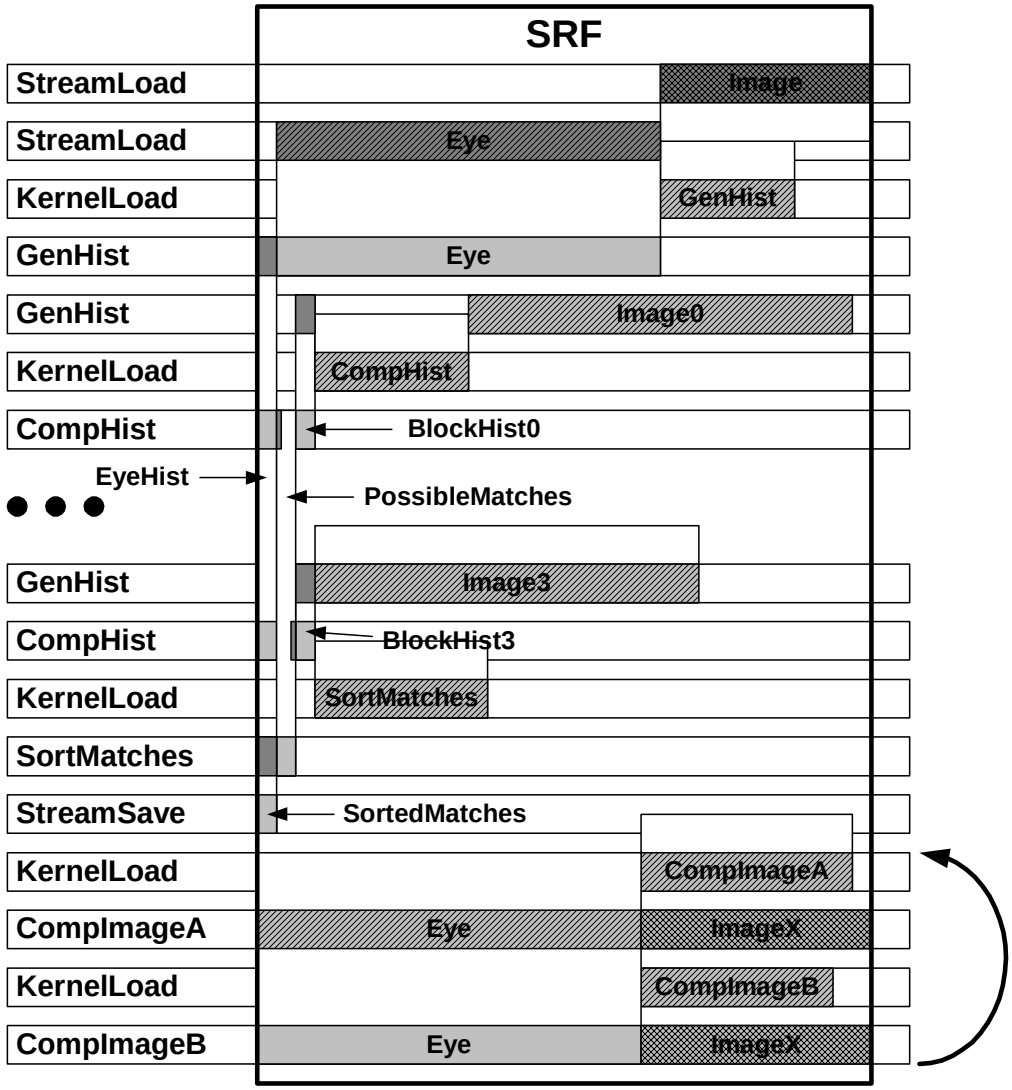



**FIGURE 7-5. SRF allocation for EyeMatch**

simply assigns all stream accesses to the same underlying basic stream to a buffer and packs those buffers into the SRF. If it fails, it terminates with an out of memory error.

Figure 7-6 shows the memory allocation for EyeMatch. Only stream accesses that require memory accesses are considered in memory allocation. Visually, only accesses that are cross-hatched in Figure 7-5 appear in Figure 7-6. The memory allocation for the basic stream *image* is highlighted in Figure 7-6. The stream program starts with a StreamLoad-Bin that stores the entire basic stream. Next, the loop that computes a histogram for each

block in the image loads a series of derived streams that each refer parts of the basic stream *image*. Lastly, the final loop that evaluates possible matches loads one or more derived streams that refer to a data-dependent parts of the basic stream *Image*, indicated with dotted rectangles.

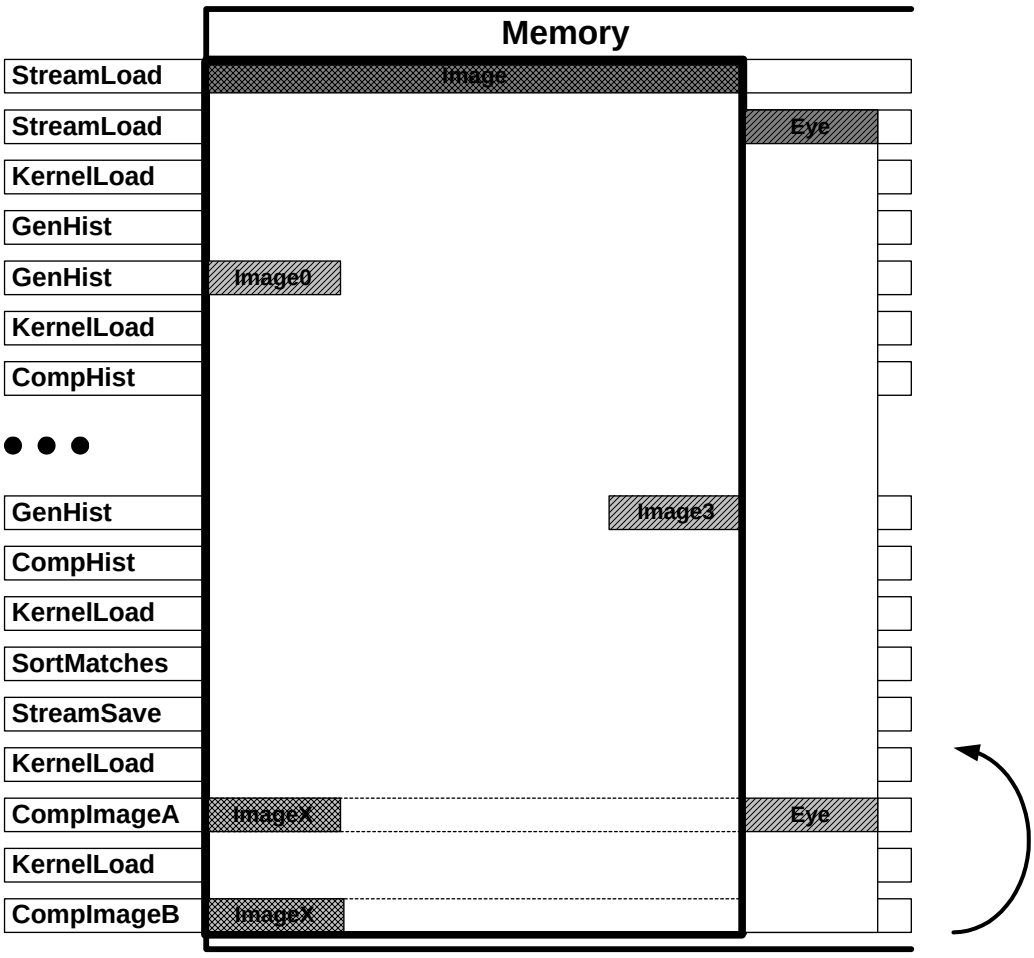



**FIGURE 7-6. Memory allocation for EyeMatch**

Fourth, the StreamC compiler allocates control registers for each stream access. The StreamC compiler allocates control registers for each stream access a stream operation makes. The StreamC compiler allocates one SDR for each stream access to describe the location in the SRF that is accessed (or two SDRs if the stream access is double-buffered,

one to describe each half-buffer in the SRF). If the stream access requires a memory access, it allocates one MAR to describe the location in memory that is accessed.

Control register allocation is motivated by the unique way in which control registers are used. All control register values are either fixed or computed at run-time for a data-dependent stream. Imagine operations only read control registers. (There is one exception. If a kernel writes to a variable length stream it updates the SDR that contains the length of that stream, but the new value is always read by the host.) Hence, control register allocation consists of assigning each read to a register and deciding when the host needs to write those registers. Additional writes by the host may be needed to reinitialize registers if more registers are needed than are available. The StreamC compiler allocates control registers using another variation of the stream scheduling algorithm which treats register values as unit length streams. Control register reads to fixed values are compatible if they are equal regardless of what the register is used for. Control register reads to values that are determined at run time for a data dependent stream are only compatible with other reads to the same value for that stream. Control register reads that are marked as “requiring loads” need writes by the host.

Figure 7-7 shows the SDR and MAR allocation for EyeMatch. Cross-hatching indicates control register uses that require register writes by the host. Dotted-fill is for clarity only. The abbreviation DB indicates control registers for a double-buffered access. Visually, each SDR read corresponds to a stream access in Figure 7-5, and each MAR read corresponds to a memory access in Figure 7-6. The highlighted SDR value is representative. It describes the space in the SRF used to hold each block of the live image for the loop that computes the histogram of each block. The SDR value is written by the host when it is first read for a stream access. Since the value is fixed, it is reused for later stream accesses even though those accesses are to different streams.

Once the StreamC compiler completes resource allocation, it has assigned each kernel to a location in the microcode store, each stream access to a location in the SRF, each memory access to a location in memory, and each control register read to a register. It has also

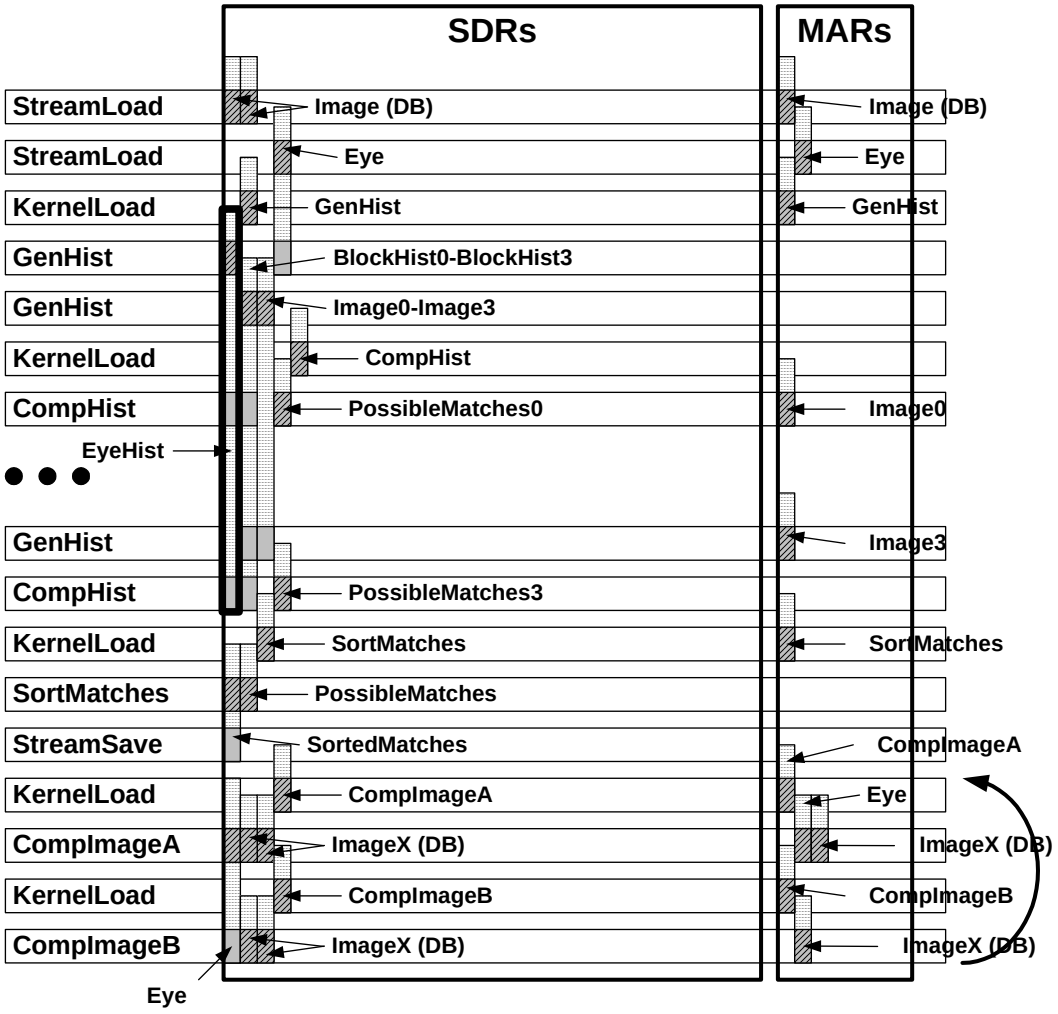



**FIGURE 7-7. Control register allocation for EyeMatch**

determined which kernels require kernel loads, which stream accesses require memory accesses, and which control register reads require writes by the host.

### 7.1.3 Operation translation

The StreamC compiler translates each stream operation into one or more Imagine operations. In general, an Imagine operation writes or (rarely) reads a control register, transfers a stream between the SRF and another destination, or starts or restarts a kernel. Figure 7-8 lists the Imagine operations by purpose.

| Imagine operation | Description |
|---|---|
| ***General purpose*** | |
| SDR Write | writes a stream descriptor register |
| MAR Write | writes a memory access register |
| Memory Load | loads a stream from memory to the SRF |
| Memory Store | stores a stream from the SRF to memory |
| ***Kernels*** | |
| UCR Write | writes a microcontroller register, used to pass initial microcontroller variable values |
| UCR Read | reads a microcontroller register, used to return final microcontroller variable values |
| SDR Length Read | reads the final length of a variable length stream output |
| Microcode Load | loads kernel microcode from the SRF into the microcode store |
| Kernel Start | starts a kernel |
| Kernel Restart | continues a kernel with a new stream replacing an exhausted stream |
| ***Host transfers*** | |
| Host Send | sends a stream from the SRF to the host |
| Host Receive | receives a stream from the host to the SRF |

**FIGURE 7-8. Imagine operations by purpose**

Essentially, the StreamC compiler translates a stream operation into a sequence of Imagine operations that write control registers as needed, load input streams from memory as needed, perform one or more core operations specific to the type of stream operation, and store output streams to memory as needed. More specifically, it translates each stream operation into the following sequence of Imagine operations, with stream operation specific portions given by Figure 7-9:

1. SDR Write(s) for each stream access
2. *Other register writes (see Figure 7-9)*
3. MAR Write for each stream access that requires a memory access
4. Memory Load for each stream read that requires a memory access
5. *Core operations (see Figure 7-9)*
6. Memory Store for each stream write that requires a memory access
7. *Register reads (see Figure 7-9)*

| **Stream operation** | **Other register writes** | **Core operation(s)** | **Register reads** |
|---|---|---|---|
| <Kernel Name> | UCR Write for each UC argument | Kernel Start<br><br>Kernel Restart for each double-buffered stream access | UCR Read for each UC argument<br><br>SDR Length Read for each variable length stream write |
| KernelLoad | - | Microcode Load | - |
| StreamLoadBin | - | Host Receive | - |
| StreamSaveBin | - | Host Send | - |
| StreamCopy | - | - | - |

**FIGURE 7-9. Stream operation specific Imagine operations**

The StreamC compiler translates the first stream operation in EyeMatch, the StreamLoadBin of *image*, into five Imagine operations as shown in Figure 7-10. The StreamLoadBin makes a single stream access, a stream write to *image*. That stream access reads two SDRs since it is double-buffered and one MAR since it requires a memory write. All three of these reads require writes by the host, as shown in Figure 7-7. The core operation of a StreamLoadBin is a Host Receive. Lastly, the stream write requires a Memory Store as shown in Figure 7-6. The MAR Write, Host Receive, and Memory Store are used for double-buffering, indicated in Figure 7-8 with the abbreviation DB.

Figure 7-10 also shows how the StreamC compiler translates the second through fourth stream operation in EyeMatch. The second stream operation is another StreamLoadBin, which is translated much like the first StreamLoadBin except that is not double-buffered

so it only requires one SDR Write. The third operation is a Kernel Load which makes a single stream read that requires a memory load. It is translated into an SDR Write, an MAR Write, a Memory Load, and a Microcode Load (the core operation of a Kernel-Load). The fourth operation is a kernel call. It reads one stream and writes another, but only requires an SDR Write for the later since the SDR value used for the stream read was already written by the second StreamLoadBin. The only other Imagine operation it requires is a Kernel Start, the core operation of a kernel call.

| Stream operation | Imagine operations |
|---|---|
| StreamLoadBin<br>  write image[0, 40000] | SDR Write (SDR 0, pos 10256, len 3064)<br>SDR Write (SDR 1, pos 13320, len 3064)<br>MAR Write (MAR 0, pos 0) DB<br>Host Receive (SDR0) DB<br>Memory Save (SDR0, MAR0) DB |
| StreamLoadBin<br>  write eye[0, 10000] | SDR Write (SDR 2, pos 256, len 10000)<br>MAR Write (MAR 1, pos 0)<br>Host Receive (SDR 2)<br>Memory Save (SDR 2, MAR 1) |
| KernelLoad<br>  read  GenHist[0, 3500] | SDR Write (SDR 1, pos 10256, len 3500)<br>MAR Write (MAR 0, pos 1045076)<br>Memory Load (SDR 1, MAR 0)<br>Microcode Load (SDR 1, instr 0) |
| GenHist<br>  read  eye[0, 10000]<br>  write eyeHist[0, 256] | SDR Write (SDR 0, pos 0, len 256)<br>Kernel Start (instr 0, SDR 2, SDR 0) |

**FIGURE 7-10. Imagine operations for first four stream operations in EyeMatch**

### 7.1.4 Issue slot assignment and dependency analysis

The StreamC compiler assigns each Imagine operation to an issue slot in the Imagine processor's issue buffer at compile time. When the run-time dispatcher tries to send an operation to Imagine, it waits until the issue slot assigned to the operation becomes available. The StreamC compiler tries to assign operations that could occur near one another to different issue slots. The StreamC compiler assigns issues slots to Imagine operations in the order they appear in the stream program, using a modified round robin policy. Essentially, it assigns the first operation to issue slot 0, then assigns the next operation to issue slot 1,

and so on until it reaches the last issue slot. Then it starts over and assigns the next operation to issue slot 0.

The StreamC compiler deviates from this simple progression slightly to ensure that Imagine operations that could occur close to one another depending on control flow are not assigned the same issue slot. Using the simple progression, it could assign the Imagine operation before the body of an if statement to issue slot 3, assign the Imagine operations in the body to other issue slots, then, having come full-circle, also assign the operation after the body to issue slot 3. If the if-statement is not taken, the run-time dispatcher would have to wait before it could send the second Imagine operation assigned to issue slot 3. To avoid this problem, it skips occasional issue slots in basic blocks that contain more than half as many Imagine operations as there are issue slots. It skips a number of issue slots so that the last Imagine operation in the basic block is assigned to the same issue slot as the Imagine operation immediately before the basic block. Returning to the example involving overuse of issue slot 3, skipping one slot inside the basic block would result in the operation immediately afterward being assigned to issue slot 4.

The StreamC compiler implements this policy by skipping a number of issue slots after the Imagine operations that compose each stream operation as given by the following equation:

$$\text{issue slot increment} = \frac{\text{Imagine ops in block MOD number of issue slots}}{\text{stream operations in block}} \qquad (7)$$

The first basic block in EyeMatch consists of 16 stream operations that translate into 48 Imagine operations. The issue slot increment assuming 32 issue slots is thus (48 MOD 32) / 16 = 1.0. Thus, the five Imagine operations that compose the first stream operation are assigned to issue slots 0 through 4, one issue slot is skipped, and the four Imagine operations that compose the second stream operation are assigned to issues slots 6 through 9, and so on. These skipped issue slots ensure that the last operation in the basic block is assigned to slot 31, the same as the operation before the basic block (if there was one).

The StreamC compiler then analyzes the dependencies of each Imagine operation upon other operations that could be in the issue buffer when it dispatched. The current Imagine operation is dependent on a previous Imagine operation if both operations access the same SRF, memory, or microcode store space or use the same control register and either or both Imagine operations modify the contents. The dependency is classified as read-after-write (RAW) if only the previous Imagine operation modifies the contents, write-after-read (WAR) if only the current Imagine operation modifies the contents, or write-after-write (WAW) if both Imagine operations modify the contents. A kernel restart operation is also considered WAR dependent upon the kernel operations it restarts.

For each Imagine operation, the StreamC compiler determines if it is dependent on any operation that could be in the issue buffer when it is dispatched. The StreamC compiler considers the latest previous Imagine operation in the current basic block that is assigned to each issue slot. If no previous Imagine operation in the current basic block is assigned to a particular issue slot then the StreamC compiler also considers the latest Imagine operation assigned to that issue slot in each basic block that could immediately precede the current basic block, and so on.

The StreamC compiler encodes the dependencies of each Imagine operation in three bit-masks called the dependency masks. Each dependency mask contains a bit corresponding to each issue slot in the issue buffer. If the current operation is dependent on any operation that could be in an issue slot as determined above, then the bit corresponding to that issue slot is set in the appropriate mask(s). Two of these dependency masks, the RAW mask and the WAR mask, are used by the Imagine processor to determine when to issue the operation. The RAW mask encodes all RAW and WAW dependencies. The WAR mask encodes all WAR dependencies. The other dependency mask, the DifferentDB mask, is used by the run-time dispatcher for determining when the Imagine operation can be dispatched. The DifferentDB mask encodes all dependencies upon Imagine operations that are repeatedly dispatched for the purpose of double-buffering a different stream access.

The StreamC compiler also constructs two special dependency masks for any Imagine operation that is repeatedly dispatched for a double-buffered access. These masks, called the RAW DB mask and the WAR DB mask, are constructed like the normal RAW and WAR masks except that the StreamC compiler considers only operations used for the same stream access, but considers all such operations both before and after the current operation.

Figure 7-11 shows the RAW masks for the first fifteen Imagine operations in EyeMatch. Each row contains an Imagine operation, the issue slot it is assigned to, and the part of its dependency mask corresponding to the issue slots shown. The last Imagine operation is a the kernel GenHist, shown in bold in Figure 7-11. It is RAW dependent on three SDR Write operations in issue slots 0, 6, and 16 that write SDRs it uses, the Host Receive in issue slot 8 that writes a stream that it reads, and the Microcode Load in issue slot 14 that writes the GenHist microcode. The RAW dependency mask for the GenHist kernel has the bits corresponding to issue slots of these operations set: 0, 6, 8, 14, and 16.

## 7.2 Run-time Dispatcher

This section presents the run-time dispatcher. It describes how the run-time dispatcher sends Imagine operations to the issue buffer. It also discusses how the run-time dispatcher coordinates data transfers between the host and Imagine and manages double-buffering.

### 7.2.1 Imagine operation dispatching

The primary role of the run-time dispatcher is to dispatch Imagine operations from the host processor to the Imagine processor’s issue buffer. When the host processor encounters a stream operation, it calls a method of the run-time dispatcher to send the corresponding Imagine operations to the Imagine processor. The run time dispatcher updates the Imagine operation as required, waits until the Imagine operation can be dispatched, and sends it to the issue buffer.

First, the run-time dispatcher updates the Imagine operation to reflect any data-dependent streams that affect it. A data-dependent stream has a variable length or bounds. These val-

| Imagine operation | Slot | RAW dependency mask... | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| SDRWrite (SDR 0, pos 10256, len 3064) | 0 | | | | | | | | | | | | | | | | | |
| SDRWrite (SDR 1, pos 13320, len 3064) | 1 | | | | | | | | | | | | | | | | | |
| MARWrite (MAR 0, pos 0) DB | 2 | | | | | | | | | | | | | | | | | |
| HostReceive (SDR0/SDR1) DB | 3 | 1 | 1 | | | | | | | | | | | | | | | |
| MemoryStore (SDR0/SDR1, MAR0) DB | 4 | 1 | 1 | 1 | 1 | | | | | | | | | | | | | |
| (skipped by issue slot assignment) | 5 | | | | | | | | | | | | | | | | | |
| SDRWrite (SDR 2, pos 256, len 10000) | 6 | | | | | | | | | | | | | | | | | |
| MARWrite (MAR 1, pos 0) | 7 | | | | | | | | | | | | | | | | | |
| HostReceive (SDR 2) | 8 | | | | | | | 1 | | | | | | | | | | |
| MemoryStore (SDR 2, MAR 1) | 9 | | | | | | | 1 | 1 | 1 | | | | | | | | |
| (skipped by issue slot assignment) | 10 | | | | | | | | | | | | | | | | | |
| SDRWrite (SDR 1, pos 10256, len 3500) | 11 | | 1 | | | | | | | | | | | | | | | |
| MARWrite (MAR 0, pos 1045076) | 12 | | | 1 | | | | | | | | | | | | | | |
| MemoryLoad (SDR 1, MAR 0) | 13 | | 1 | 1 | | | | | | | | 1 | 1 | | | | | |
| MicrocodeLoad (SDR 1, start 0) | 14 | | 1 | | | | | | | | | 1 | | 1 | | | | |
| (skipped by issue slot assignment) | 15 | | | | | | | | | | | | | | | | | |
| SDRWrite (SDR 0, pos 0, len 256) | 16 | 1 | | | | | | | | | | | | | | | | |
| **GenHist (SDR 2, SDR 0)** | **17** | **1** | | | | | | **1** | | **1** | | | | | **1** | | **1** | |

**FIGURE 7-11. RAW masks for first fifteen Imagine operations in EyeMatch**

ues are reflected in the location of the stream in the SRF and in memory, which are encoded in the SDR and MAR used to access the stream. The run time dispatcher modifies the SDR Write(s) and MAR Write for stream accesses to data-dependent streams to reflect the actual length, start, and end of the streams.

For example, the final loop of EyeMatch evaluates a data-dependent series of blocks of the live image looking for a match. The current block of the live image is accessed by the derived stream *block*, which has variable bounds since it could refer to any part of image. Before dispatching the MAR Write for a stream access to *block*, the dispatcher updates it to reflect the actual start and end of the stream. For instance, if the image is located at

memory address 0, the second 50x50 block of the image starts at memory address 50. If the final loop were to try and match the second block, the dispatcher would modify the MAR Write for the operation write a start address of 50.

Second, the run-time dispatcher waits until the Imagine operation can be dispatched. The run-time dispatcher determines if an Imagine operation can be dispatched based on two criteria. The first criteria is that the issue slot assigned to the Imagine operation must be empty. Before sending any Imagine operations, the run-time dispatcher queries the Imagine processor to determine which slots are empty. It sends as many Imagine operations as it can before it encounters an Imagine operation that is assigned to an occupied issue slot. The run-time dispatcher then queries the Imagine processor again to determine which issue slots have become empty since it last checked. When an issue slot becomes empty, the run-time dispatcher also knows that whatever operation it last sent to that slot has completed.

The second criteria is that, if the Imagine operation depends on any other Imagine operation that is repeatedly dispatched as part of a different double-buffered access, that other Imagine operation must be dispatched for the last time. The run-time dispatcher uses a bitmask called the CurrentDB mask to track the status of double-buffering operations so that it can enforce this requirement. The CurrentDB mask contains a bit corresponding to each issue slot. When a double-buffering operation is dispatched for the first time, the bit corresponding to the issue slot that operation is assigned to is set. When the repeated operation is dispatched for the last time, that bit cleared. In order to dispatch an operation, the CurrentDB mask ANDed with the DifferentDB mask of that operation must be empty.

For example, the first StreamLoad in EyeMatch requires a double-buffered stream write to store the live image to memory. The StreamC compiler translates the StreamLoad into several Imagine operations as shown in Figure 7-10, one of which is a MAR Write that the run-time dispatcher repeatedly dispatches for the double-buffered stream write. The first time it dispatches that operation, it sets a bit in the CurrentDB mask. The run-time dispatcher waits to dispatch any Imagine operation that depends on that MAR Write but is for

a different stream access until after the last time it dispatches the MAR Write. To make this determination, it ANDs the Imagine operation's DifferentDB mask with the CurrentDB mask and only sends it if the result is empty.

Finally, the run-time dispatcher sends the operation to the Imagine processor along with the RAW and WAR masks for that Imagine operation. The Imagine processor then issues the operations in the issue buffer out of order, using the dependency masks to preserve consistency.

### 7.2.2 Host/Imagine data transfers

The run-time dispatcher coordinates two kinds of data transfers between the Imagine processor and the host: registers reads and stream transfers. There are two kinds of register reads. SDR Read Length reads the length component of an SDR and updates the length of a variable length stream. UCR Read reads a UCR and updates a microcontroller variable. There are also two kinds of stream transfers. Host Receive sends a stream from the host to Imagine and Host Send sends a stream from Imagine to the host.

The run-time dispatcher handles all register reads without stopping execution of the stream program on the host, unless the variable that is updated by the register read is needed. The run-time dispatcher sends the register read operation to Imagine like any other Imagine operation. When the Imagine processor completes the register read, it places the value that was read in a special register corresponding to the issue slot assigned to the register read operation. When the run-time dispatcher checks the status of the available issue slots and finds a completed read, it retrieves the value from that special register and updates the appropriate variable. If the stream program attempts to read that variable before a pending register read completes, it invokes the run-time dispatcher to continually poll the status of the available issue slots until the variable is updated.

In the final loop of EyeMatch, two kernel calls are used to determine if a possible match between a block of the live image and the reference image is actually a match. Each kernel call writes its result to a microcontroller variable argument. Since both kernels are called

before either microcontroller variable is read in the stream program, the run-time dispatcher dispatches the Imagine operations for both kernel calls, including register reads to update the microcontroller variables with the result of each kernel. When the stream program tries to read one of the microcontroller variables, it invokes the run-time dispatcher to wait for the corresponding register read to complete.

The dispatcher handles host transfers by stopping the execution of the stream program on the host until the transfer completes. It sends all of the Imagine operations for the stream operation that includes the transfer, waits for the transfer to begin, transfers the data between the host processor and Imagine, then continues with the stream program.

For example, the first stream operation in EyeMatch contains a StreamLoadBin used to send the live image to the Imagine processor. It translates into five Imagine operations as shown in Figure 7-10. The run-time dispatcher sends these operations to the host, waits for the Host Receive to begin, transfers the data, then continues with the stream program.

### 7.2.3 Double-buffering

A double-buffered stream access involves cycling portions of a large stream through two halves of a smaller buffer in the SRF. One cycle of a double-buffered stream read consists of three Imagine operations: a Write MAR to increment the address of the current portion of the stream, a Memory Load to load that portion into a half-buffer, and a core operation to read the contents of that half-buffer. One cycle of double-buffered stream write consists of a similar sequence of three operations: a Write MAR to increment the address of the current portion of the stream, a core operation to write that portion into a half-buffer, and a Memory Store to store the contents of that half-buffer. To implement double-buffering, the run-time dispatcher repeatedly dispatches the three double-buffering operations that compose a cycle, in order. A complete double-buffered access involves a number of cycles equal to the length of the stream divided by the size of a half-buffer, rounded up.

Each time the run-time dispatcher redispatches an operation in the cycle, it updates it to reflect the current portion of the stream and/or half-buffer. The run-time dispatcher

replaces the operation's RAW and WAR masks with the operation's RAW DB and WAR DB masks. The run-time dispatcher increments the memory address of the MAR Write. It toggles between the two SDRs that describe the two half-buffers in the case of the Memory Load, Memory Store, or core operation. Since the length of a stream is not always divisible by the size of a half-buffer, the run-time dispatcher performs a special SDR Write before dispatching the MAR write for the last time to update one of the SDRs to reflect the length of the remainder of the stream.

The first StreamLoad in EyeMatch requires a double-buffered stream write to store the 40000-word live image to memory through two 3064-word half buffers. The StreamC compiler translates that StreamLoad into the Imagine operations shown in Figure 7-10. The run-time dispatcher repeatedly dispatches the operations marked with DB in Figure 7-10, resulting in the actual sequence of Imagine operations shown in Figure 7-12.

```
// init
SDR Write (SDR 0, pos 10256, len 3064)
SDR Write (SDR 1, pos 13320, len 3064)
// cycle 1
MAR Write (MAR 0, pos 0)
Host Receive (SDR0)
Memory Save (SDR0, MAR0)
// cycle 2
MAR Write (MAR 0, pos 3064)
Host Receive (SDR1)
Memory Save (SDR1, MAR0)
...
// cycle 13
MAR Write (MAR 0, pos 36768)
Host Receive (SDR0)
Memory Save (SDR0, MAR0)
// cycle 14
SDR Write (SDR 1, pos 13320, len 168)
MAR Write (MAR 0, pos 39832)
Host Receive (SDR1)
Memory Save (SDR1, MAR0)
```

**FIGURE 7-12. Imagine operations for double-buffered stream write to *image***

## 7.3 Optimizations

This section presents two optimizations used to improve the performance of stream programs. Strip-mining involves processing a large input stream in small batches so that intermediate streams will fit in the SRF. Software-pipelining divides a loop into stages and overlaps the stages to increase parallelism. The StreamC compiler performs both of these optimizations in a semi-automated manner: it suggests the strip-mining batch size and processes a specially tagged source file to produce a source file containing a software pipelined loop.

### 7.3.1 Strip-mining

Strip-mining [49] involves processing a large input stream in smaller batches so that the intermediate streams produced while processing a batch will all fit in the SRF. Since most stream programs operate on inputs that are larger than the SRF by themselves, this optimization is essential for good performance. A typical stream program consists of a series of stream operations that process an initial input to produce a final output. Each stream operation in the series writes one or more outputs that are read as inputs by the next stream operation. If the output of a stream operation is larger than the SRF, that stream operation writes it to memory using double-buffering, and the next stream operation reads it from memory using double-buffering. This sequential double-buffering limits the throughput of those operations to the available memory bandwidth. To eliminate this bottleneck, strip-mining applies the series of stream operations to a small portion of the initial input to produce a small portion of the final output, such that the output of every stream operation fits in the SRF. It then applies the series to another small portion of the initial input to produce another small portion of the final output, and so on until all of the initial input has been processed. The size of the largest portion of initial input such that the output of every stream operation fits in the SRF is termed the *strip size*. Figure 7-13 illustrates basic and strip-mined dataflow for a simple stream program containing three kernels.

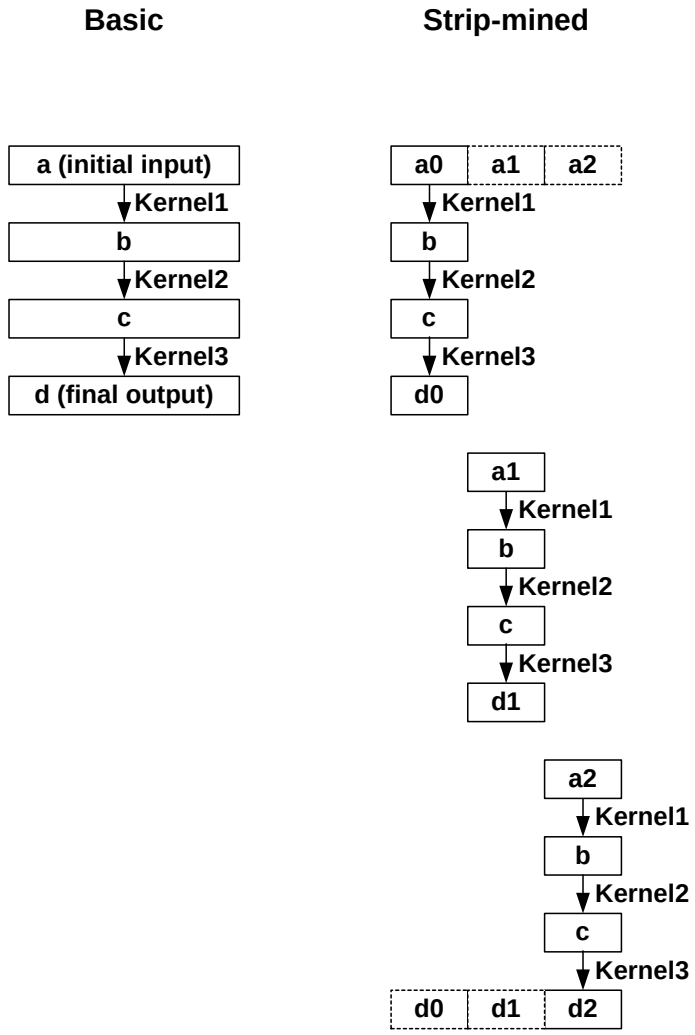



**FIGURE 7-13. Basic and strip-mined data flow**

The StreamC compiler estimates a strip size when it compiles a program and writes that result to a log file, but leaves the actual strip-mining of the stream program to the programmer. Most stream programs do not fit the ideal strip-mining model. Often, there are application-specific constraints on how the initial input or final output can be divided. Sometimes, kernels must be changed depending on the amount of data being processed. These constraints make automatic strip-mining difficult to implement in a general manner. Therefore, the StreamC compiler provides a mechanism for determining the approximate size of a portion of the initial input such that the output of every operation in the series will fit in the SRF. The mechanism assumes that the length of all streams is proportionate to

the strip size. It allocates the SRF as though it were infinite size to determine the amount of space needed to process the entire input. It then multiplies the length of the initial input stream by the actual size of the SRF divided by the size needed to process the entire input.

For example, suppose the loop in EyeMatch that computes the histogram for each block in the live image also applied a preprocessing kernel to each block. The input and output of this preprocessing kernel would be too large to fit in the SRF. The preprocessing kernel and histogram generation kernel could be strip-mined as shown in Figure 7-14. Rather than preprocess and compute a histogram for the entire block, a strip-mined loop processes a small number of rows each time. This change requires replacing the GenHist kernel with an AddHist kernel which takes the old histogram as an additional input and adds the new rows to produce a new histogram

```
// 3. with each block of the image, compute color histogram,
// compare to eye histogram, and add entry to possible matches if similar
for (int x = 0; x < imgW; x += eyeW) {
  for (int y = 0; y < imgH; y += eyeH) {
    stream<byte4> block(image, y*imgW + x, (y+eyeH-1)*imgW + x + eyeW,
      FIXED, STRIDE, imgW, eyeW);
    stream<byte4> blockHist(256);
    // add small number of rows of block to histogram each iteration
    for (int i = 0; i < eyeH; i += stripRows) {
      stream<byte4> blockRows = block(i * stripRows, (i + 1) * stripRows);
      Preprocess(blockRows, preProcBlockRows);
      Addhist(preProcBlockRows, blockHist, blockHist)
    }
    CompHist(blockHist, eyeHist, possibleMatches(i, i + 1), x, y);
  }
}
```

**FIGURE 7-14. Strip-mined loop**

### 7.3.2 Software-pipelining

Software-pipelining involves dividing a loop into stages and overlapping execution of one stage of one iteration with execution of another stage of another iteration. Software-pipelining can be used to hide the memory access time of a *sequential memory access*, a memory access that must occur between a pair of sequential kernels. A sequential memory access occurs when the result of one kernel is stored to memory and then reloaded using a different access pattern as an input to the next kernel, or when the result of a kernel is used

as an index stream for an indexed stream loaded as an input to the next kernel. In either of these cases, the second kernel cannot start immediately after the first kernel, it must wait for the intervening memory load to complete as shown in Figure 7-15. Software pipelining can hide the latency for this memory access by overlapping execution of a kernel from another stage with the sequential memory access as shown in Figure 7-16.

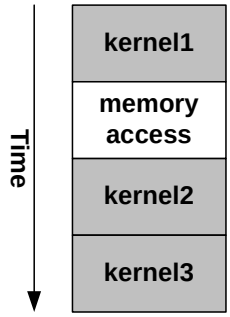



**FIGURE 7-15. Loop with sequential memory access**

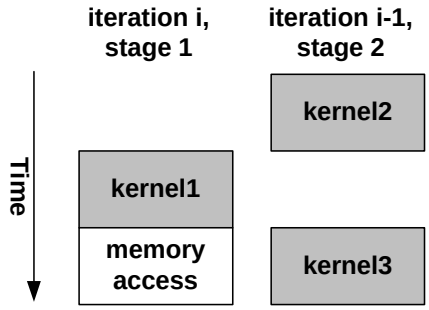



**FIGURE 7-16. Software-pipelined loop**

The StreamC compiler implements a software-pipelining algorithm that is designed to handle the varying latencies of kernels and memory accesses. It tries dividing the loop into two stages after each stream operation in the loop. For each pair of possible stages, it splits each stage into a series of alternating groups of kernels and sequential memory accesses. It splits each stage into the same number of groups, some of which may be empty, such that each memory access group in one stage has a corresponding kernel group in the other stage as shown in Figure 7-17. Initially, kernels and memory accesses are assigned to the first possible group, preserving ordering and dependencies between operations in different stages.

The StreamC compiler then repeatedly moves operations between groups. It can move the first or last operation in a particular group to the previous or next group of similar operations, respectively, as shown in Figure 7-18, provided that it preserves ordering and does not violate dependencies between operations in different stages. It moves an operation if

doing so reduces estimated run time. It also moves an operation if doing so does not increase estimated run time and reduces the highest ratio between the total memory access time of one group and the total kernel execution time in the corresponding group. This second criteria avoids covering one memory access with a minimum number of kernels while covering another with an excessive number of kernels. For these purposes, the StreamC compiler estimates memory access time and kernel execution time based on data gathered during the profiling run. The total run time is calculated by summing the higher of the total memory access time or the total kernel execution time for all pairs of corresponding groups. The StreamC compiler also moves operations counter to these criteria with a diminishing random chance to avoid becoming stuck in a local minima, similar to simulated annealing [26].

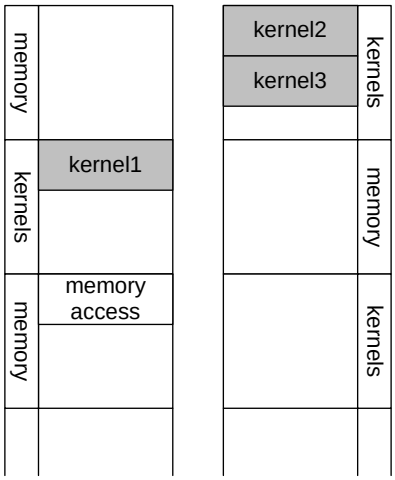



**FIGURE 7-17. Software-pipelining groups**

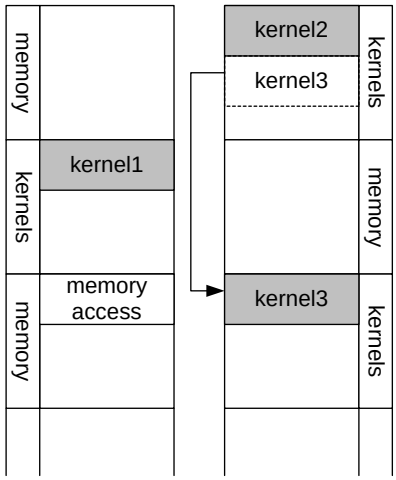



**FIGURE 7-18. Moving an operation between groups to reduce run time**

Once the StreamC compiler has divided the operations into groups for each possible pair of stages, it selects the pair of stages with the minimum run time or, given equal run times, the minimum highest ratio of memory access time to kernel execution time. It then orders the operations starting with the first group of memory accesses followed by the corresponding group of kernels, then the next group of memory accesses followed by the corresponding group of kernels, and so on. Based on this ordering, it transforms a source file

that contains a loop with specially tagged pieces of code as shown in Figure 7-19[1] and produces a source file with a software-pipelined loop as shown in Figure 7-20.

```
while(...) {
   kernel1(...);
   kernel2(...);
   kernel3(...);
}
```

**FIGURE 7-19. Input to StreamC compiler software-pipelining**

```
// stage 1 prologue
kernel1(...);

while (...) {
   // stage 2
   kernel2(...);
   // stage 1
   kernel1(...);
   // stage 2
   kernel3(...);
}

// stage 2 epilogue
kernel2(...);
kernel3(c, d);
```

**FIGURE 7-20. Output from StreamC compiler software-pipelining**

## 7.4 Summary

This chapter presented the StreamC compiler and the run-time dispatcher, the two components required to compile and execute a stream program on the Imagine media processor. First, it described how the StreamC compiler efficiently translates stream operations into Imagine operations. Second, it covered how the run-time dispatcher dispatches these Imagine operations to the issue buffer of the Imagine processor. Lastly, it discussed two optimizations used to improve performance of stream programs.

The StreamC compiler presented in this chapter is not integrated with the normal C++ compiler used to compile the rest of the stream program; integration of the two would allow additional optimizations. For example, an integrated compiler with access to all data flow information could reorder StreamC operations to improve performance.

1. To make parsing the source file easier, the actual implementation requires “tagging” the code containing each kernel or other stream operation. These tags are omitted for simplicity.

# Chapter 8

# Evaluation

This chapter presents a quantitative evaluation of the KernelC compiler and the StreamC compiler. For each compiler, it describes an evaluation methodology and a set of benchmarks, then presents and analyzes the results of applying the methodology to the benchmarks. The evaluation methodology used for both compilers compares the compiler to an alternative that is very expensive in terms of hardware cost and/or programmer effort, but delivers very good performance. In both cases, the compilers are shown to deliver equivalent or superior performance with significantly less hardware and/or programmer effort.

This chapter consists of two sections. Section 8.1 evaluates the KernelC compiler, with emphasis on communication scheduling. Section 8.2 evaluates the StreamC compiler, with emphasis on stream scheduling.

## 8.1 KernelC Compiler

### 8.1.1 Evaluation Architectures

The KernelC compiler was evaluated by compiling a set of benchmark kernels for two variations of the Imagine stream processor: one with a single register file and one with distributed register files (DRF). The single register file architecture provides a performance baseline. The distributed register file architecture tests the effectiveness of communication scheduling on an architecture that makes extensive use of shared interconnect and multiple register files.

In the single register file architecture shown in Figure 8-1, each functional unit input or output is connected to the same register file by dedicated interconnect. In the DRF architecture shown in Figure 8-2, each functional unit input is connected to a small two-ported register file. Each functional output can drive any one of ten global buses connected to all register files, though each register file can only be written by one of those buses on a cycle. This architecture is nearly identical to the architecture used to implement Imagine, though it has fewer buses in order to be a better test of communication scheduling. Figure 8-1 through Figure 8-2 also show the area, power consumption, and register file access delay estimated for each architecture using the methods in [44], normalized relative to estimates for the single register file architecture.

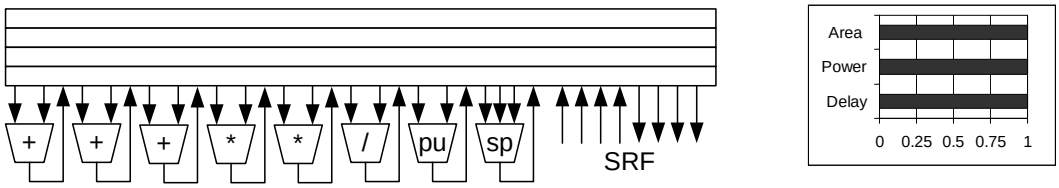



**FIGURE 8-1. Single register file architecture**

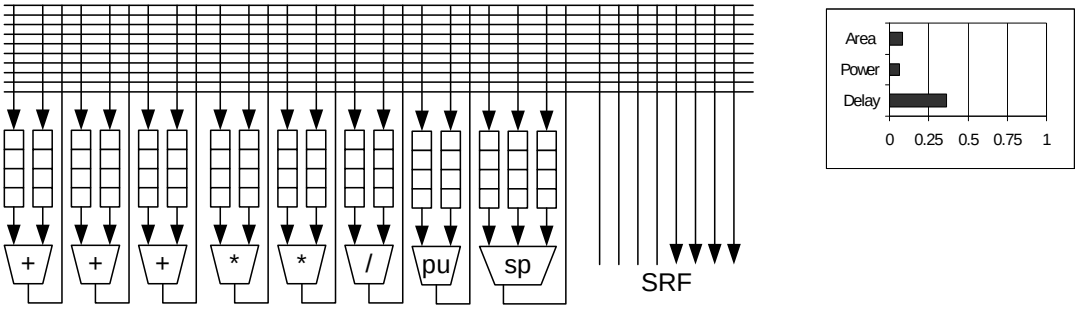



**FIGURE 8-2. Distributed register file architecture**

Both architectures include the same mix of eight functional units: three adders, two multipliers, a divider, a permutation unit (pu), and a scratchpad (sp). The permutation unit permutes values between Imagine's eight SIMD processing elements. The scratchpad is a small word-addressable memory used for local arrays. All functional units except the

scratchpad unit implement the copy operation. The functional units have identical latencies in both architectures, with representative latencies shown in Figure 8-3.

| Operation | Lat. | Operation | Lat. |
|---|---|---|---|
| **Adder** | | **Divider** | |
| Logical operation | 1 | Integer divide | 22 |
| Integer add/subtract | 2 | Floating-point divide | 16 |
| Floating-point add/subtract | 4 | **Permutation Unit** | |
| **Multiplier** | | Permute | 1 |
| Integer multiply | 4 | **Scratch Pad** | |
| Floating-point multiply | 4 | Scratch pad read/write | 1 |
| **All** | | | |
| Copy | 1 | | |

**FIGURE 8-3. Representative latencies**

### 8.1.2 Evaluation Benchmarks

Figure 8-4 shows the benchmarks used to evaluate the KernelC compiler. These benchmarks include all of the kernels in the StreamC compiler benchmarks (except for programmable polygon rendering, since the kernels are similar to span-based polygon rendering), and several other image-processing, signal-processing, and sorting benchmarks. In general, each benchmark kernel consists of single loop that iterates over the records of an input stream, along with a short prologue and epilogue. Most of these loops are software pipelined using modulo software pipelining [28]. The loops dominate kernel execution time, so the performance of a given kernel is inversely proportional to the schedule length of the loop. Figure 8-4 also show the number of operations in the inner loop of each benchmark.

### 8.1.3 Results

This section presents the schedule length, operation count, and register demand results for the DRF architecture relative to the single RF architecture. Each is presented as a distribu-

| Kernel | Description | Ops |
|---|---|---|
| **Blockwarp** | | |
| blockwarp | blockwarp for image-based renderer | 105 |
| **Depth Extraction** | | |
| blockfill | fills a stream with a constant value | 11 |
| blocksad | sum of absolute difference over a sliding window | 109 |
| byte2word | unpacks 8-bit pixels into 16-bit pixels | 12 |
| convfx3x3 | convolves a row with a 3x3 filter | 56 |
| convfx7x7 | convolves a row with a 7x7 filter | 174 |
| exdepth | extracts depth from SAD values | 16 |
| extemp3 | handles final rows for 3x3 convolution | 11 |
| extemp7 | handles final rows for 7x7 convolution | 23 |
| **FFT** | | |
| fft1024 | 1024-point fast fourier transform | 340 |
| **MPEG2 Encoding** | | |
| blocksearch | searches reference image to determine motion vector | 409 |
| corr | correlates current macroblock with reference macroblock | 262 |
| dct | discrete cosine transform | 207 |
| diff | computes the difference between two macroblocks | 208 |
| icolor | color space conversion from RGB to YCrCb | 70 |
| idct | inverse discrete cosine transform | 249 |
| idxgen | generate addresses to access a macroblock | 87 |
| mv2idx | converts a motion vector into addresses | 151 |
| pcolor | color space conversion from RGB to YCrCb | 705 |
| rle | run-length encodes a macroblock | 25 |
| **Span-based Polygon Rendering** | | |
| compact_recycle | compacts conflicting fragments | 46 |
| glshader | computes shading and lighting values | 295 |
| hash | hashes conflicting fragments | 107 |

| | | |
|---|---|---|
| mergefrag | merges two streams of sorted fragments | 148 |
| project | perspective projection | 164 |
| sort32frag | sorts groups of 32 fragments | 826 |
| spanrast | converts a span into fragments | 154 |
| spansgen | converts a triangle into spans | 355 |
| spansprep | prepares a triangle for span generation | 540 |
| xform | transforms a triangle from object space to screen space | 50 |
| zcompare | compares z values of two streams of fragments | 22 |
| **Q-R Matrix Decomposition** | | |
| backsub1 | performs forward elimination before back substitution | 16 |
| backsub2 | performs first part of back substitution | 20 |
| backsub3 | performs second part of back substitution | 34 |
| house2 | Householder transformation | 159 |
| sumsqr | computes the sum of squares | 73 |
| update1 | computes matrix transformation for row update | 101 |
| update2 | applies matrix transformation for row update | 104 |
| **Sorting** | | |
| bisort | bitonic sorts a stream of integers | 74 |
| merge | merges two streams of sorted integers | 68 |
| sort32 | sorts groups of 32 integers | 231 |

**FIGURE 8-4. KernelC benchmarks**

tion for the set of benchmark kernels. Specific results for each benchmark kernel can be found in Appendix A.

Figure 8-5 through Figure 8-7 present the performance results for the benchmarks. Figure 8-5 and Figure 8-6 show the distribution of the relative and actual differences between the schedule length for the DRF architecture and that for the single register file architecture.

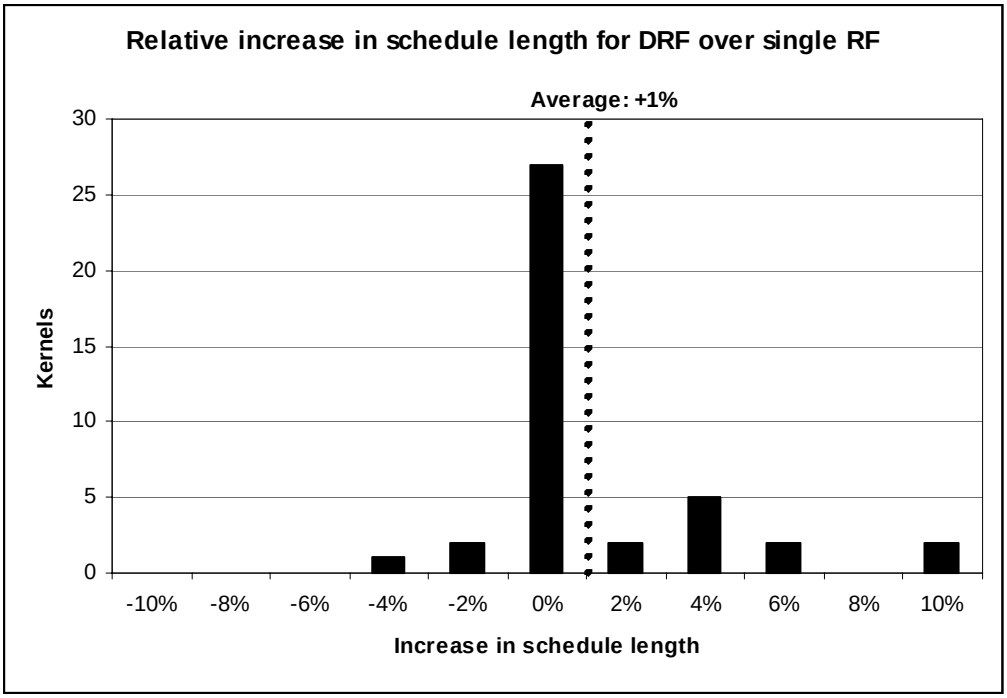



**FIGURE 8-5. Relative increase in schedule length for DRF over single RF**

Figure 8-7 shows the distribution of the relative differences between the DRF schedule length and a computable lower bound (CLB) on schedule length. The CLB is equal to the maximum of two component bounds, one dictated by dependencies between operations and the other by available resources. For a non-software pipelined loop, the first such bound is the critical path. The critical path is the maximum sum of operation latencies along a path from an operation at the top of the dependency graph to an operation at the bottom of the dependency graph. For a software-pipelined loop the first bound is the reoc-currence minimum iteration interval (RMII). The RMII is the maximum sum of operation latencies along a path between any two operations with a write-after-read dependency (such operations can be at most one iteration interval, the length of an iteration of the soft-ware-pipelined loop, apart). The second bound is the resource limit, equal to the maximum number of operations that can only be scheduled on one kind of functional unit divided by the total number of such functional units available. Figure 8-7 also shows which of these limits dictates the bound for each kernel.

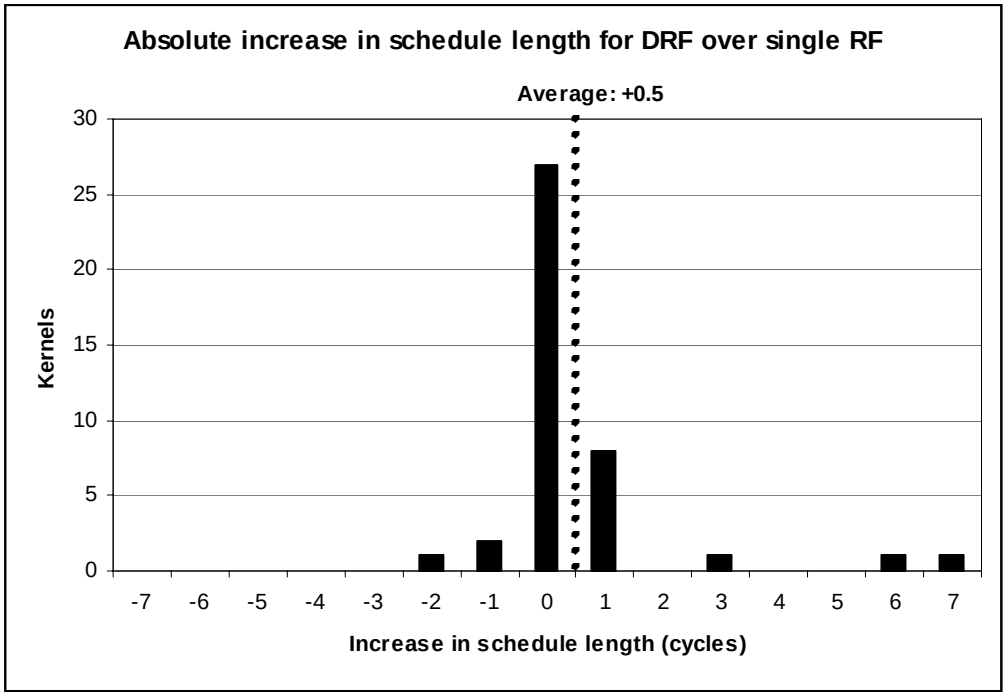



**FIGURE 8-6. Increase in schedule length for DRF over single RF in cycles**

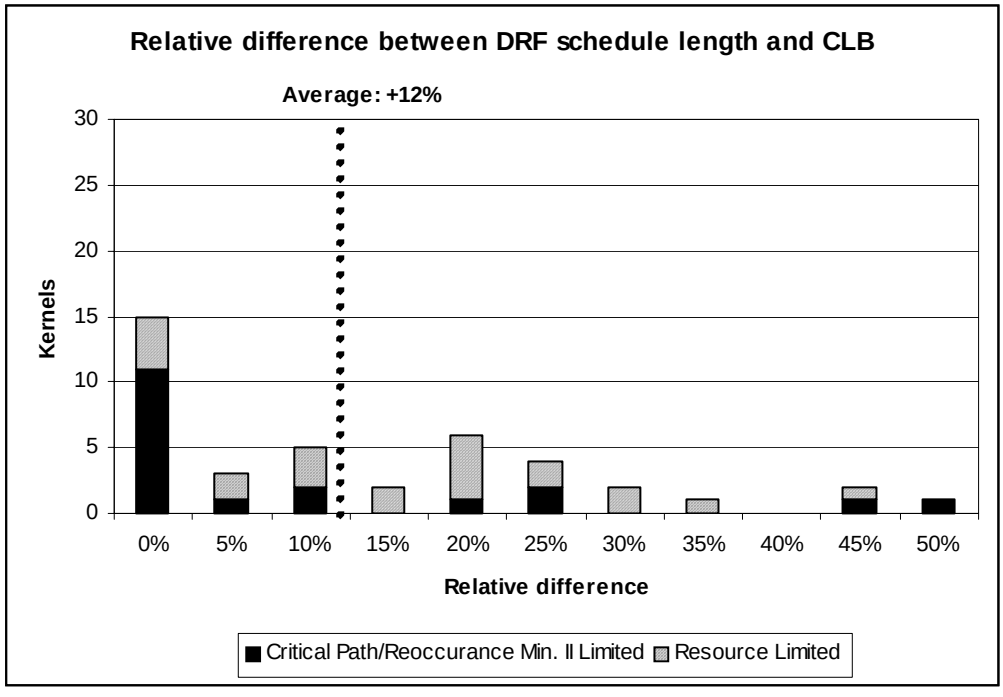



**FIGURE 8-7. Relative difference between DRF schedule length and CLB**

Figure 8-8 shows the distribution of the increase in the number of operations in the schedules for the DRF architecture over those for the SRF architecture due to the addition of copy operations to move data between register files.

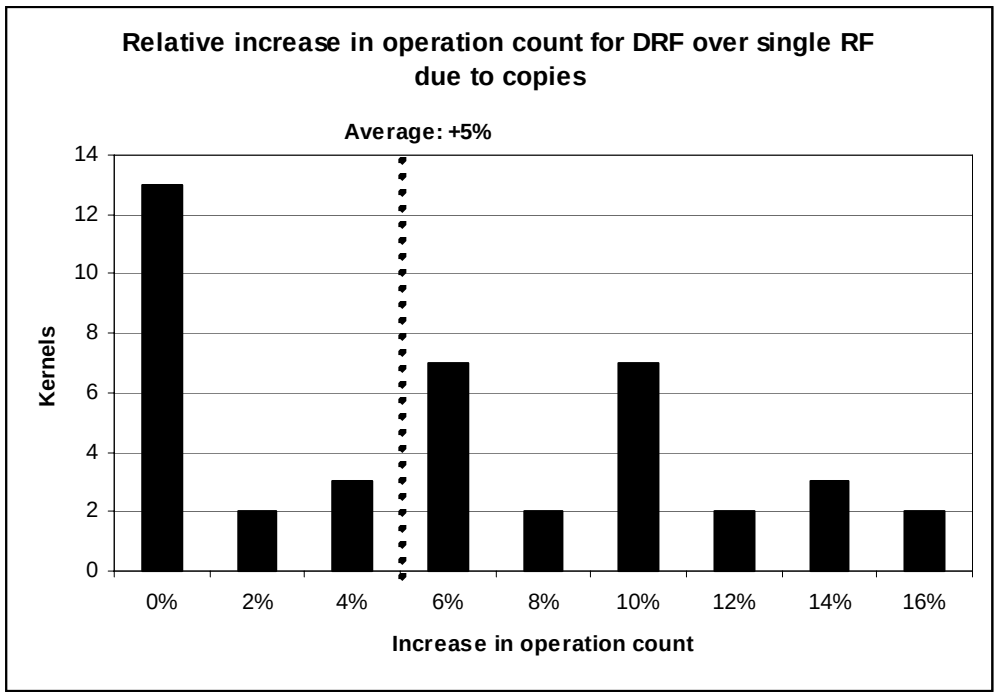



**FIGURE 8-8. Relative increase in operation count for DRF over single RF due to copies**

Figure 8-9 through Figure 8-11 show the increase in register demand for the DRF architecture over the SRF architecture. Figure 8-9 shows the distribution of the increase in register demand due to duplication among multiple registers files, calculated by comparing the total number of registers used in all register files. Figure 8-10 shows the increase in register demand due to load imbalance among register files. It is calculated by comparing the maximum number of registers used in any one register file multiplied by the number of register files and divided by the total number of registers used. Lastly, Figure 8-11 shows the increase in register demand, the product of these two factors.

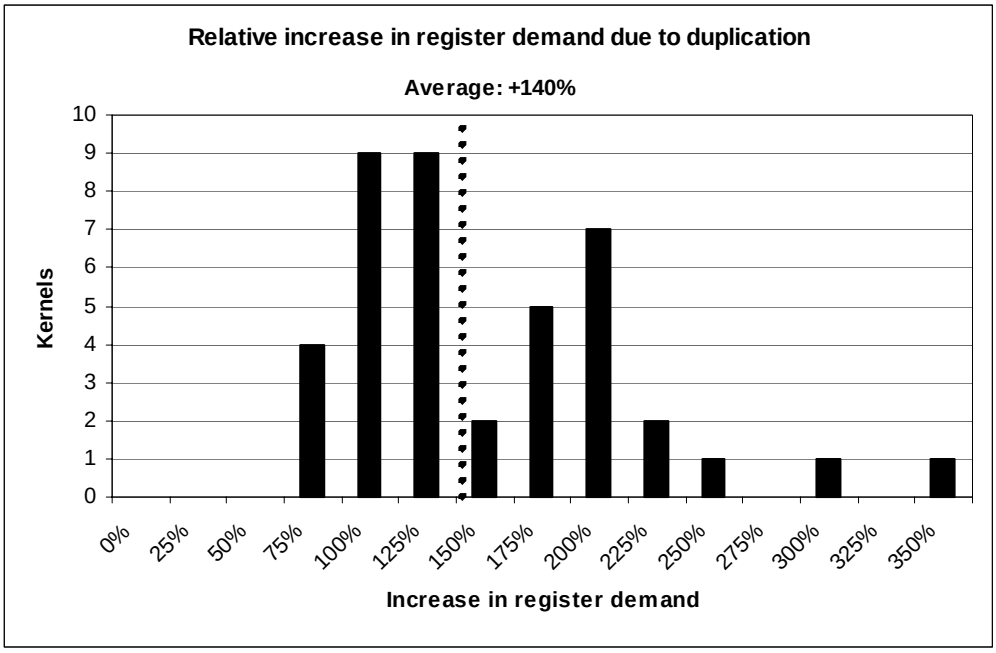



**FIGURE 8-9. Relative increase in register demand due to duplication**

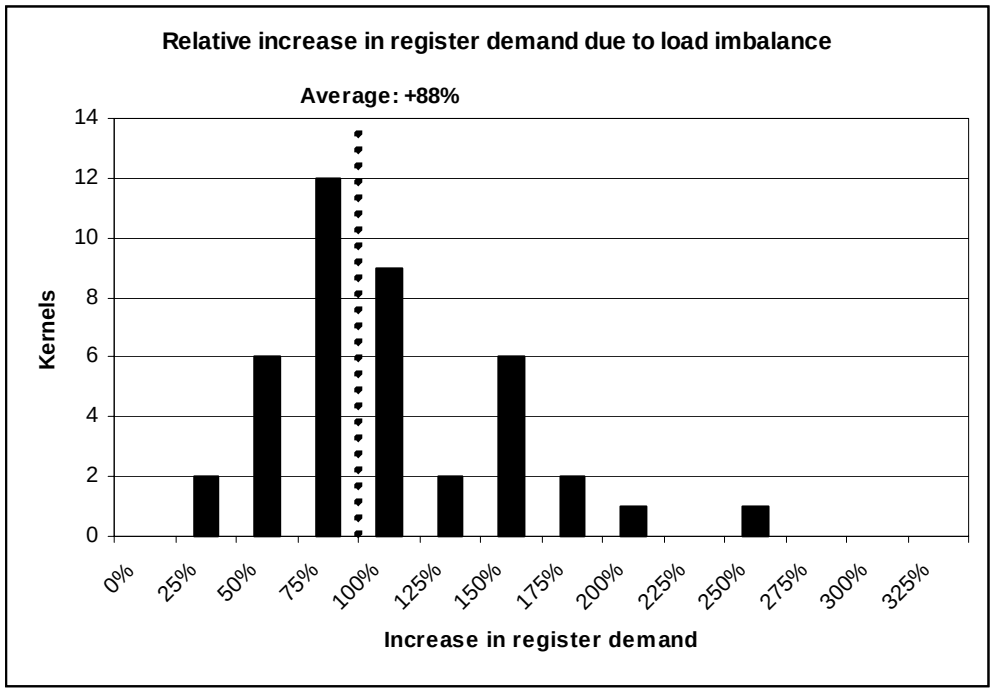



**FIGURE 8-10. Relative increase in register demand due to load imbalance**

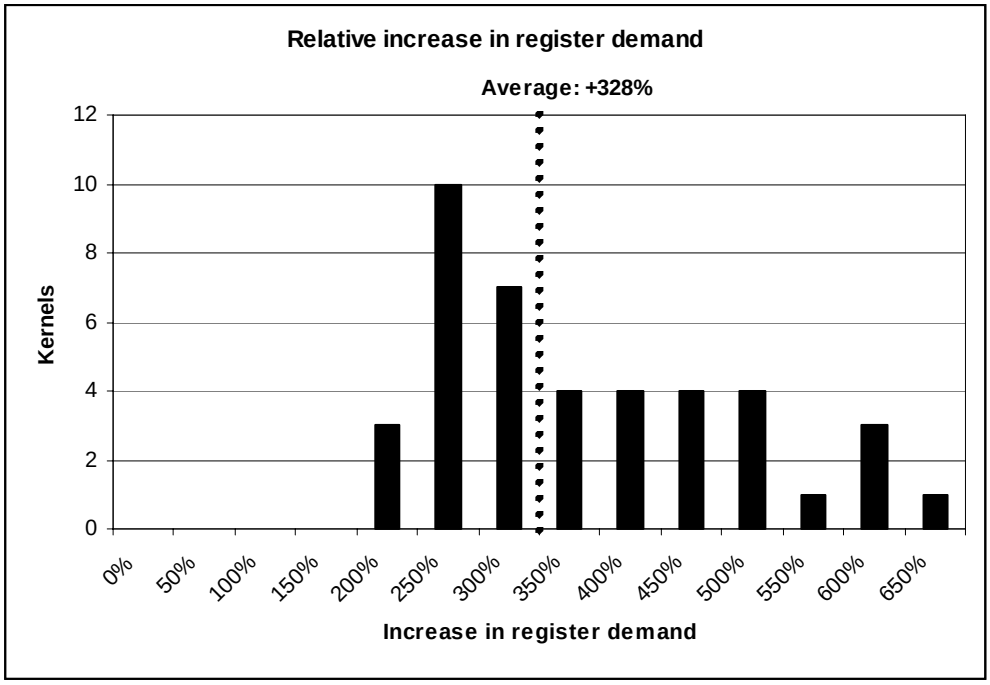



**FIGURE 8-11. Relative increase in register demand**

## 8.1.4 General analysis

These results demonstrate that the KernelC compiler, using communication scheduling with supporting optimizations, effectively schedules media processing kernels for an architecture with shared interconnect and multiple register files. The schedule lengths for the DRF architecture are on average within 1% of those for the single register file architecture, and within 12% of a computable lower bound on schedule length. The increase in operation count due to copy operations is low, averaging only 5%. The KernelC compiler does not consider register pressure during scheduling, and duplication and load imbalance among multiple register files result in average increase in register demand of 348%. Since the size of a register in the evaluation DRF architecture is less than 2.5% that of a register in the single RF architecture [44], even the increase in register pressure without any attempt at minimization is more than countered by the lower hardware cost of registers.

The KernelC compiler's ability to manage shared interconnect and multiple register files is validated primarily by comparing the schedule lengths for the DRF architecture to those

for the single register file architecture, which are approximate lower bounds under the assumptions of this evaluation. This evaluation assumes that both architectures have the same number of functional units and the same operation latencies. In reality, a DRF architecture could support many more functional units with the same total area and power. Further, due to lower register file access time, a DRF architecture would offer lower operation latencies than a single register file architecture. The results given in this section ignore these advantages. With the same number of functional units and the same operation latencies, the length of the optimal schedule for any architecture with shared interconnect and multiple register files is strictly equal to or worse than that for a single register file architecture, since copy operations may be required and operations may be delayed due to interconnect resource conflicts.

The average increase in schedule length for the DRF architecture over the single RF architecture, as shown in Figure 8-5, is less than 1%. This result shows that communication scheduling with supporting optimizations avoids almost all performance degradation due to shared interconnect and multiple register files, even when ignoring their advantages. In general, the differences between schedule lengths for the two architectures fall within the random variation of the heuristic VLIW scheduling algorithm, about plus or minus one cycle (due to these random variations, a small number of DRF schedules are slightly shorter than the corresponding single register file schedule). Only one kernel, spansprep, has a schedule length increase of more than 5%. It is discussed in more detail below.

The performance of the KernelC compiler is further validated by comparing the schedule lengths to a computable lower bound (CLB), as shown in Figure 8-7. Most schedule lengths for both architectures approach the CLB. The CLB is the same for both architectures since it only considers functional units as resources. The average difference between schedule lengths and the corresponding CLB for the DRF architecture is 12%. The CLB is a strict lower bound, but it is often not achievable because resource constraints are more complex than reflected by the resource limit. For instance, the pcolor kernel contains a large number of one- and two-cycle latency operations that are scheduled on the adders. Since operations with different latencies cannot be perfectly pipelined, this significantly

increases schedule length resulting in a very dense schedule that is still 40% higher than the CLB. As shown in Figure 8-7, the majority of kernels with schedule lengths greater than the CLB have a CLB equal to the resource limit, indicating at least one type of heavily utilized functional unit.

Beyond schedule length, a scheduler for multiple register file architectures has two important performance objectives: minimizing copy operations and minimizing register demand. Copy operations used to move values between register files occupy hardware resources and increase power consumption. Increased register demand either requires larger register files or more spilling.

The Kernel scheduler reduces the number and performance impact of copy operations by using communication scheduling to manage data movement and the communication cost heuristic to efficiently assign operations to functional units. The schedule length results discussed previously support the effectiveness of these techniques in reducing the performance impact of copy operations. The results presented in Figure 8-8 further show that the total number of copy operations is also relatively low. On average, the additional copy operations in the schedules produced for the DRF architecture increase the total number of operations by only 5% over the schedules for the single register file architecture.

In general, architectures with multiple register files require more total registers for two reasons [44]. First, the same value may need to be duplicated in multiple register files. Second, since the registers are statically divided between register files, registers need to be over provisioned to meet the varying load distribution among register files. These demands can be measured for a specific kernel by determining the total number of registers used in all register files to measure increased demand due to replication, and by multiplying the highest number of registers used in any one register file by the number of register files and dividing by the total number of registers used to measure increased demand due to load imbalance among register files.

The KernelC compiler, which is optimized for maximum performance, does not consider register demands during scheduling. Often, maximizing performance and minimizing register usage dictate contrary choices during scheduling. For instance, from a performance point of view it is often desirable to store a value in multiple register files to allow operations that use that value to occur on multiple functional units, but doing so increases register usage. As shown by the results in Figure 8-11, the kernel scheduler produces schedules for a DRF architecture that demand on average of 348% more registers. However, the size of a register in the DRF architecture is approximately 2.5% of that required for a register in the single RF architecture. The DRF architecture has an estimated area that is only 13% of that of the single RF architecture, including this increase in registers. Even with the KernelC compiler's emphasis on performance, the increase in register demand is more than balanced by the decreased hardware cost of registers in a DRF architecture. Reducing register demand in architectures with multiple register files is outside of the scope of this thesis, but is an important area of future work.

### 8.1.5 Benchmark analysis

This section analyzes two interesting kernels to provide specific examples of the effects described in the previous section: convfx7x7, which convolves an image with a 7x7 filter, and spansprep, which prepares a span, or line of pixels, for rasterization.

#### Convfx7x7

Convfx7x7 demonstrates the ability of communication scheduling to handle shared interconnect allocation for a dense schedule resulting from a software pipelined kernel with excess instruction level parallelism. Denser schedules have more competition for resources, and are more rigorous tests for communication scheduling. Convfx7x7 uses a software-pipelined loop that contains a large number of multiplications, additions, and operations to pack and unpack data values. Figure 8-12 shows the 29-cycle schedule produced for a single register file architecture. It displays the functional units on the horizontal axis and the cycles on the vertical access. Operations are shown as rectangles with height indicating latency. Copy operations are shown with a chevron, and look like enve-

lopes. A small number of copy operations appear in single register file schedules to propagate replicated state between software-pipeline stages.

Figure 8-13 shows the 30-cycle schedule produced for a DRF architecture, which is virtually identical. For this particular kernel, the DRF schedule is one cycle longer than the single register file schedule, but this variation is within the random variation of the heuristic VLIW scheduling algorithm. Overlaid on the schedule for the DRF architecture are the communications between operations, shown as lines connecting the communicating operations. This figure illustrates the general density of communication in this application. Communication scheduling manages the shared interconnect for these communications without significant performance degradation.

## Spansprep

Spansprep is a near worst-case kernel for a DRF architecture because it contains operations that almost fully occupy all functional units, leaving very little room for copy operations. However, even for this kernel, the KernelC compiler produces a schedule for the DRF architecture that is only 9% longer that for the single RF architecture. In the single register file schedule for spansprep depicted in Figure 8-14, all seven functional units capable of performing copy operations (adders, multipliers, divider, permutation unit) are almost fully occupied. The DRF schedule shown in Figure 8-15 contains an additional 43 copy operations which result in a schedule length increase of 6 cycles.

The KernelC compiler adds copy operations to the schedule for the DRF architecture primarily to deal with competition for the single write port of each register file. In theory, competition for register file write ports should never be an issue for the DRF architecture because all register files can only be read by one functional unit input. Hence, at most one word can be read per cycle, which can be supplied by a single write port. In reality, operations that need to write to a particular register file are not evenly distributed. When two operations need to write to the same register file on a single cycle, the output of one operation is written to a different register file then later copied to the destination register file. Very high levels of instruction level parallelism make this occurrence more likely, and also

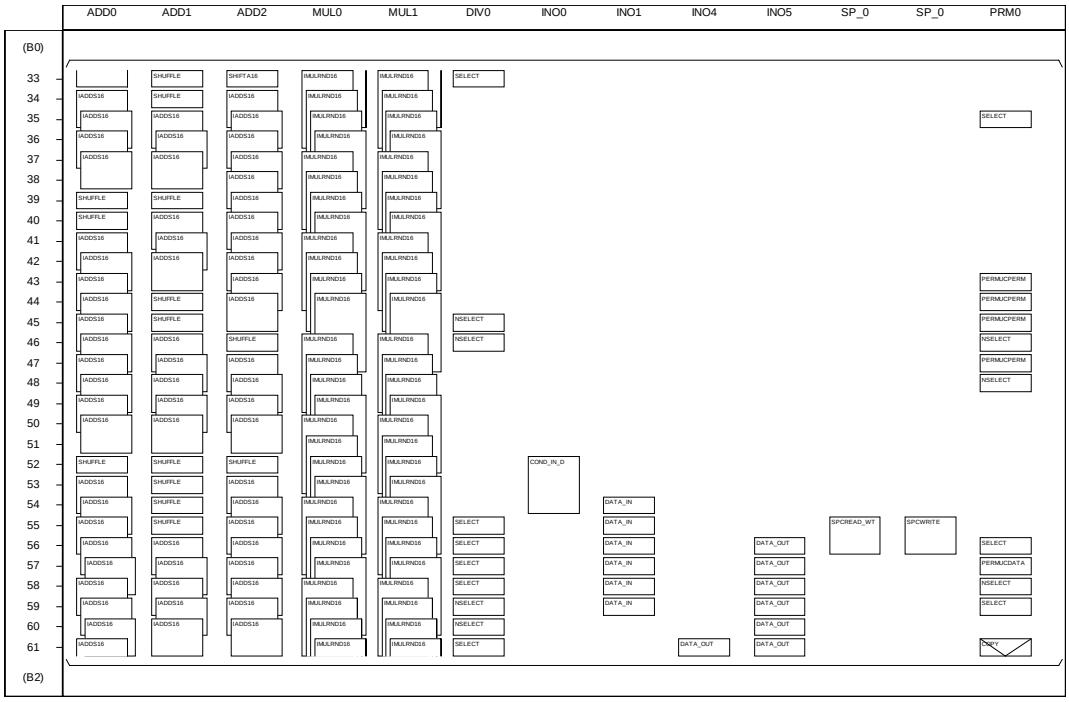



**FIGURE 8-12. Single register file schedule for convfx7x7 (29 cycles)**

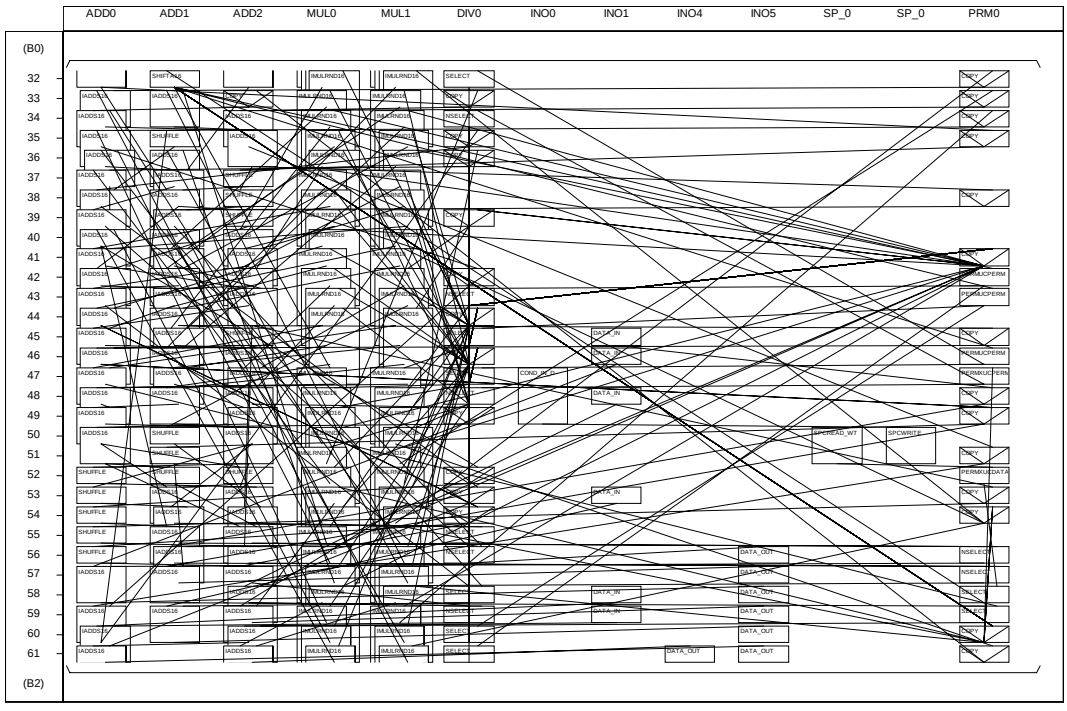



**FIGURE 8-13. Distributed register file schedule for convfx7x7 (30 cycles)**

make it more difficult to schedule the resulting copy operation without increasing schedule length.

Figure 8-15 highlights one example of this effect. Two operations, an addition and a select, need to write to multiplier 0's left register file on cycle 12. One operation, the addition, writes directly to that register file. The other operation, the select, instead writes to one of the permutation unit's register files. A copy on cycle 14 moves the value from that register file to the multiplier's register file.

## 8.2 StreamC Compiler

### 8.2.1 Methodology

The StreamC compiler was evaluated by executing a set of benchmarks on a cycle accurate simulator of the Imagine stream processor, using three different SRF allocation methods: stream scheduling, stream caching, and by hand. All runs used identical kernels. Runtime was measured from an initial state with all kernels and the input in off-chip memory to a final state with the output in off-chip memory.

All of the StreamC compiler benchmarks except Q-R decomposition and programmable rendering were implemented in two different languages: StreamC, and *macrocode*. Macrocode, which is essentially an assembly language consisting of Imagine operations, requires the programmer to allocate the SRF and specify all loading and storing of streams by hand. Figure 8-16 shows a StreamC kernel call and the equivalent macrocode (with minor syntax simplifications). Compared to StreamC, macrocode has the same disadvantages as any assembly language compared to a high-level language. It has a steeper learning curve and requires low-level understanding of the hardware. It is more time consuming to write, debug, and optimize and is not portable. However, properly optimized macrocode delivers very high performance. The macrocode versions of the benchmarks were extensively optimized by programmers who were very familiar with the Imagine

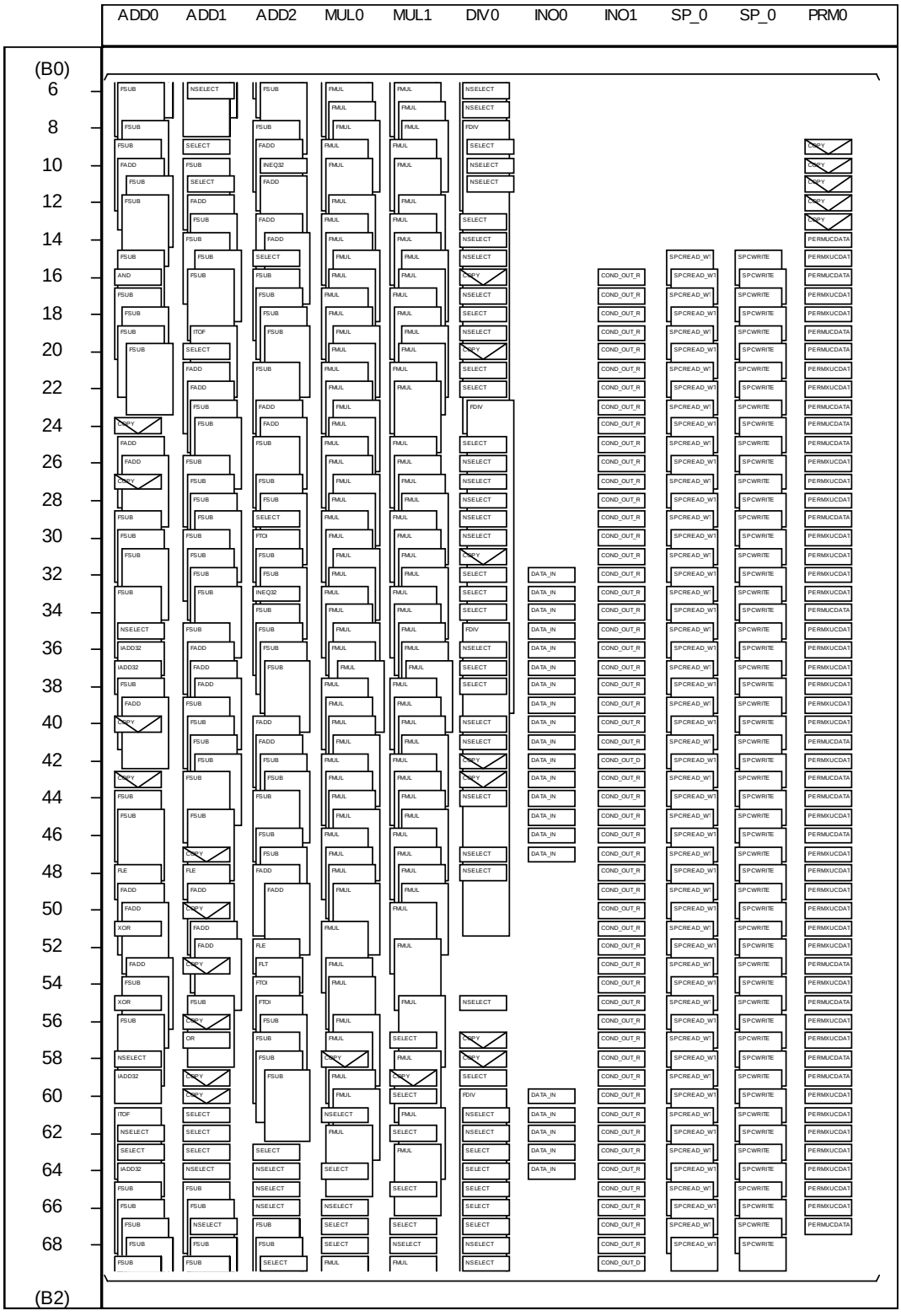



**FIGURE 8-14. Single register file schedule for spansprep (64 cycles)**

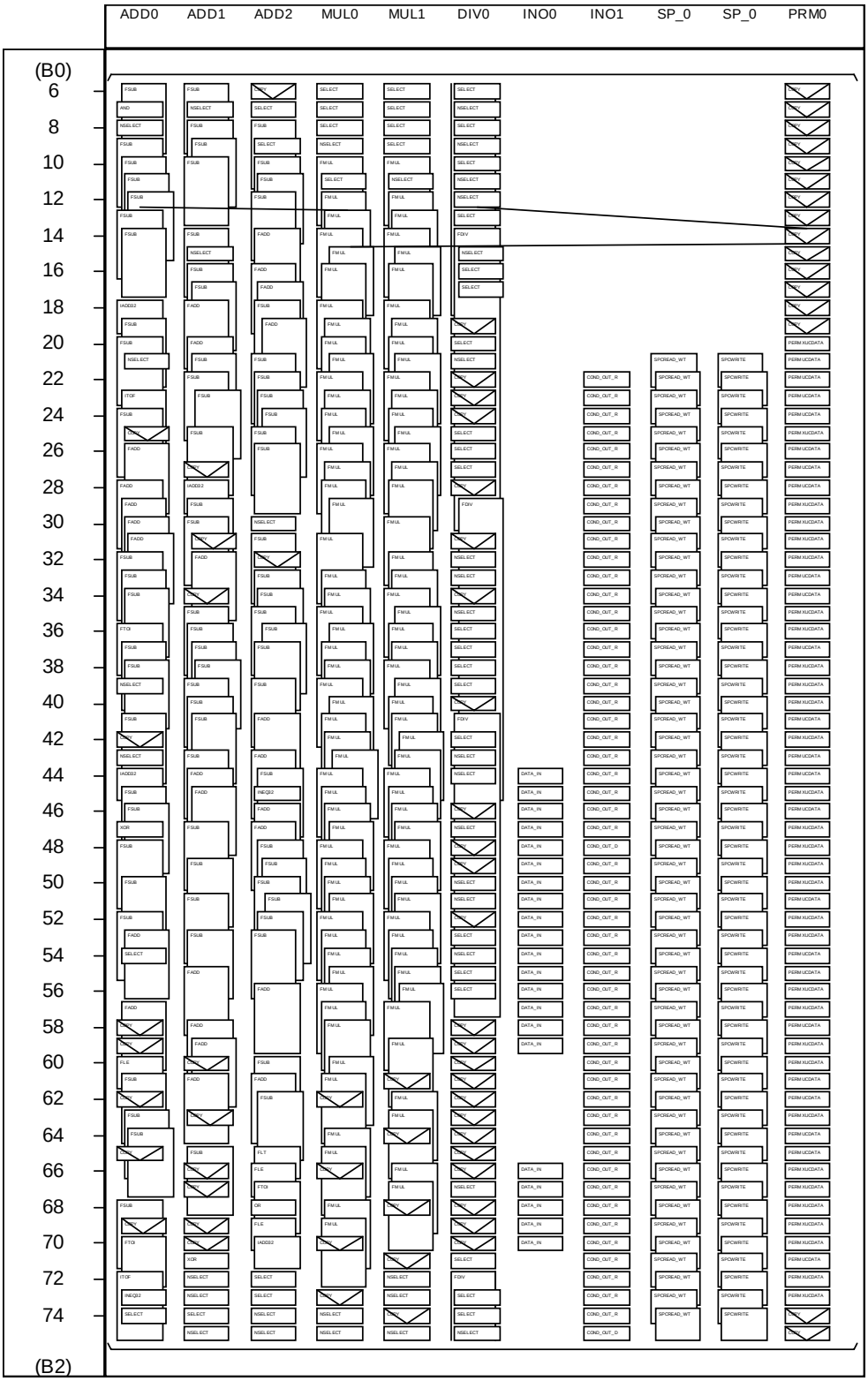



**FIGURE 8-15. DRF schedule for spansprep (70 cycles)**

architecture. The StreamC versions were optimized using the StreamC compiler as described in Section 7.3.

**StreamC**

```
foo (a, b, c);
```

**Macrocode**

```
// write MARs (memory address,
// access mode, stride)
a_MAR.write(0x100100,
  mode_stride, 1);
b_MAR.write(0x100200,
  mode_stride, 1);
c_MAR.write(0x100300,
  mode_stride, 1);

// write SDRs (SRF address,
// length, record size)
a_SDR.write(0x100, 256, 1);
b_SDR.write(0x200, 256, 1);
c_SDR.write(0x300, 256, 1);

// load inputs
memory_load(a_MAR, a_SDR);
memory_load(b_MAR, b_SDR);

// call kernel
kernel_start(foo, 2, 1,
  a_SDR, b_SDR, c_SDR);

// store output
memory_store(c_MAR, c_SDR);
```

**FIGURE 8-16. Equivalent StreamC and Macrocode**

The StreamC version of each benchmark was executed twice: once with the SRF allocated at compile time using stream scheduling, and once with the SRF allocated at run time using stream caching. Stream caching is a simpler alternative to stream scheduling that manages the SRF as a "cache of streams" using a least-recent-use replacement policy. It is described in more detail in Section 6.1.

### 8.2.2 Benchmarks

The StreamC compiler was evaluated using five benchmarks: depth extraction, MPEG2 encoding, span-based polygon rendering, Q-R decomposition, and programmable polygon rendering. Each benchmark is described[1] in detail below:

## Depth extraction

The depth extraction benchmark computes depth information from two 320x240 8-bit grayscale stereo images. It is based on Kanade's algorithm [22], though two images are used instead of the multiple cameras used in the video-rate stereo machine.

The depth extraction process is separated into two main steps: filtering the two images and extracting the depth information from the filtered images. Each input image is filtered by processing each row of the image using three kernels: one kernel unpacks the input row from 8-bits per pixel to 16-bits per pixel, and two kernels that convolve the unpacked row with a 7x7 filter followed by a 3x3 filter. Streams of partial sums are kept between rows to support the 2D convolution in the 1D streaming model. Next, the depth information is extracted using a kernel that computes a sliding window sum of absolute differences between two rows of the two filtered images. The kernel is applied repeatedly with varying disparities, and the disparity at which the sum of absolute differences about a point is smallest determines the depth of that point.

## MPEG2 encoding

The MPEG2 encoding benchmark encodes three frames of a 320x288 24-bit color image. The frames are encoded as an I-Frame followed by two P-Frames. The encoding is complete except for the final Huffman bit-coding, which is inherently serial and thus better left to a scalar processor.

The MPEG encoding process for a P-Frame consists of a series of eight kernels that operate on 16x16 macroblocks. The first kernel converts macroblocks from RGB space into luminance-chrominance space. The second and third kernels determine motion vectors to a macroblock in the reference image that is similar to each macroblock and convert the motion vectors into indices required to load that block, respectively. The fourth, fifth, and sixth kernels compute the differences between each macroblock and its reference macroblock, take the discrete cosine transform (DCT) of the differences, and run length encode

1. Some portions of the following benchmark descriptions were adapted from [46] and unpublished work by John Owens.

the results. The last two kernels take the inverse DCT of the results, and correlate them with the previous reference image for use as part of the next reference image. The MPEG encoding process for an I-frame is similar, except that it does not involve a reference image and so lacks the second, third, and fourth kernels.

## Span-based polygon rendering

The Span-Based Polygon Rendering benchmark renders a 512x512 24-bit color image of a sphere using a conventional graphics pipeline. The sphere is finely subdivided into 81,920 unmeshed triangles and generates 361,816 fragments. Backface culling is disabled so each drawn pixel has depth-complexity of 2. The sphere is Gouraud shaded and lit by three positional lights with diffuse and specular lighting components.

The span-based rendering pipeline has three conceptual stages: geometry, rasterization, and compositing, each of which is implemented as a series of kernels. The geometry stage transforms object space triangles into screen space triangles. It uses three kernels that transform the coordinates of the triangles from object space to screen space, perspective project the triangles, and apply shading and lighting to the triangles. The rasterization stage converts screen space triangles into fragments. It also uses three kernels: one that prepares the triangles, one that converts each triangle into spans, and one that rasterizes each span into fragments. The compositing stage composes the final image from the fragments. First, it uses a kernel to hash the fragments based on their coordinates in order to identify conflicting fragments. Second, it uses two kernels to merge-sort the conflicting fragments. Lastly it iteratively applies two kernels to compact conflicting fragments and composite fragments into the final image based on their z-values relative to previously rendered fragments.

## Q-R decomposition

The Q-R Decomposition benchmark decomposes an 192x96 matrix of floating-point numbers into an orthogonal Q matrix and an upper triangular R matrix such that their product

is equal to the input matrix using the compact Y-W representation [47] of the blocked-Householder transform.

The Q-R decomposition process first computes the matrix R and then computes the matrix Q using back substitution. To compute the matrix R, it divides the matrix into 8x8 blocks of elements. It successively transforms each block along the diagonal to upper triangular form and updates all blocks directly to the right of that block using the householder kernel. After transforming each block, all blocks directly below the block are conceptually (not physically) zeroed. All blocks below the current block and on or to the right of the diagonal are updated by successively applying the update1 and update2 kernels. Once the matrix R has been computed, the matrix Q is computed using forward elimination by the backsub1 kernel, and backward substitution by successive application of the backsub2 and backsub3 kernels.

## Programmable polygon rendering

The Programmable Polygon Rendering benchmarks are variations on a flexible polygon rendering pipeline. The benchmarks use different shading kernels to render six different scenes:

- **Sphere:** the same sphere rendered by the span-based rendering benchmark, with back-face culling enabled.
- **ADVS:** the first frame of the SPECviewperf 6.1.1 Advanced Visualizer benchmark with lighting and blending disabled and all textures point-sampled from a 512x512 texture map.
- **Earth:** a globe with protruding elevations rendered by perturbing the positions and normals of each vertex of a tessellated sphere using a single combined per-vertex displacement/bump map texture lookup. The scene is lit with a single positional light.
- **Verona:** a globe with protruding elevations reflecting the Cafe Verona. Verona begins with the same per-vertex position displacement as Earth, and adds an additional “environment-mapped-bump-map” calculation that involves a dependent texture read.
- **Pin:** a realistic bowling pin shaded with 5 successive textures as well as a procedurally specified light with diffuse and specular components.
- **Marble:** a marble bowling pin shaded using procedural turbulence involving 4 noise calculations per fragment and requiring no texture maps.

Like the span-based rendering pipeline, the programmable rendering pipeline has three stages: geometry, rasterization, and compositing. The compositing stage is identical to that of the span-based rendering pipeline, but the geometry and rasterization stages differ. The geometry stage takes a “mesh” of connected vertices as input and produces a stream of triangles as output. It consists of a series of kernels which compute and/or lookup texture information for each vertex, assemble vertices into triangles, divide any triangles which cross a viewport edge, and cull triangles which face away from the viewport or are outside of the viewport. The rasterization stage consists of a series of kernels that prepare each triangle for rasterization, rasterize each triangle directly into fragments, propagate texture information from the triangles to the fragments, and compute and/or lookup the final texture information for each fragment. Thought this conceptual order of kernels is the same for all scenes, the scenes use different shaders that have widely varying computation and memory access requirements and, consequently, varied software pipelining.

### 8.2.3 Results

Figure 8-17 through Figure 8-20 summarize the results for the benchmarks. All results are normalized to those for the stream scheduler. The results for programmable polygon rendering are the average of the six scenes. The raw results for each benchmark kernel can be found in Appendix A. Figure 8-17 presents the performance results for the benchmarks using each of the three stream register file allocation methods. All run times are measured in cycles from a state with all inputs stored in memory to a state with all outputs stored in memory. Figure 8-18 through Figure 8-20 show the strip size, memory traffic, and processing element occupancy for each of the benchmarks. Strip size is measured as the size of the input data processed by each iteration of the strip-mined loop. Memory traffic is measured as total number of words loaded and stored. Occupancy is measured by dividing the total time spent executing kernels by the total run time.

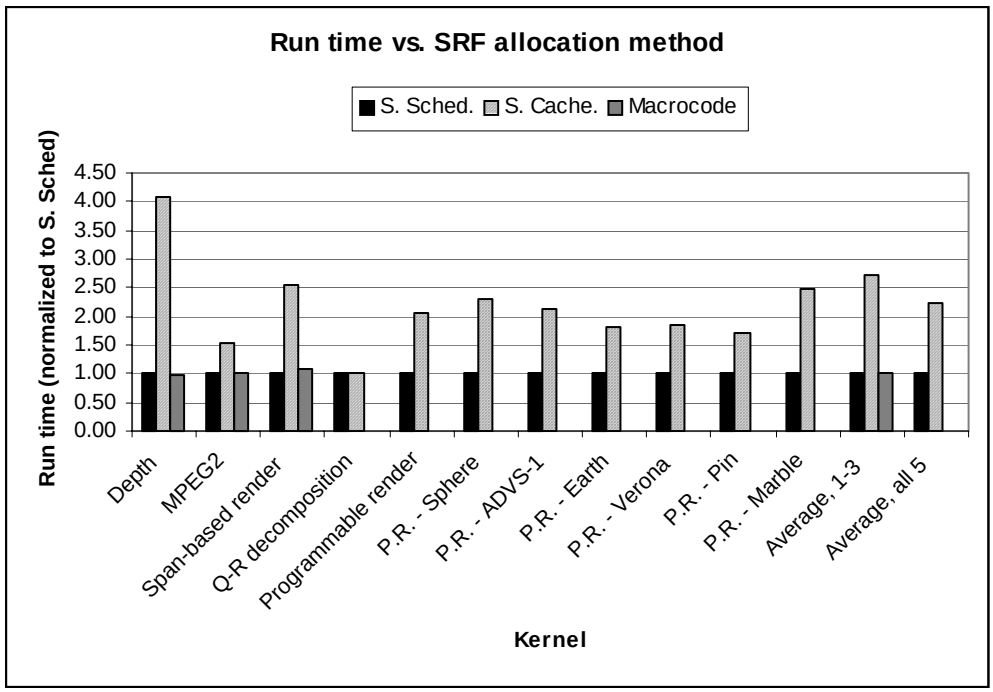



**FIGURE 8-17. StreamC benchmark run times**

### 8.2.4 General analysis

These results demonstrate that StreamC compiled using stream scheduling achieves performance that is comparable to, and sometime better than, hand-optimized macrocode. Stream caching significantly increases memory traffic, resulting in inferior performance. This section analyzes the underlying factors that dictate performance on Imagine and how these results derive from those factors.

Imagine application performance (assuming identical kernels) is dictated by three factors: strip size, memory traffic, and execution/memory parallelism. The strip size is the amount of data processed by each iteration of the strip-mined loop as described in Section 7.3. The larger the strip size, the fewer iterations of the strip-mined loop are required to process the input. Since each iteration adds overhead to start and stop each kernel (e.g. priming and draining a software pipelined loop), increasing the strip size reduces the total execution time of the kernels. The total memory traffic is the total number of words loaded and stored by the application. All applications require some memory traffic to load initial

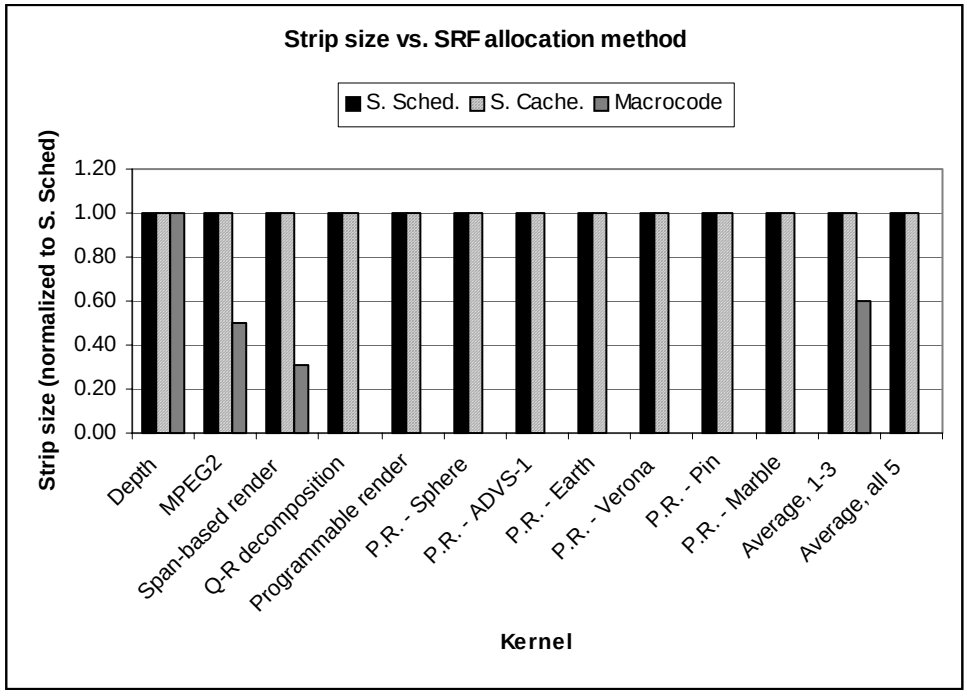



**FIGURE 8-18. Strip size**

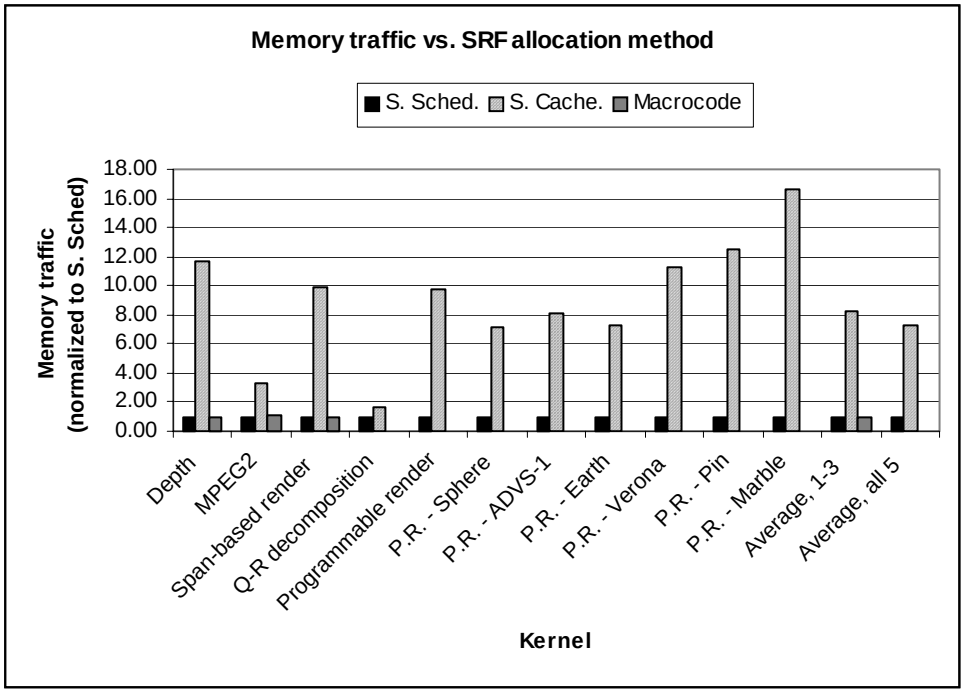



**FIGURE 8-19. Memory traffic**

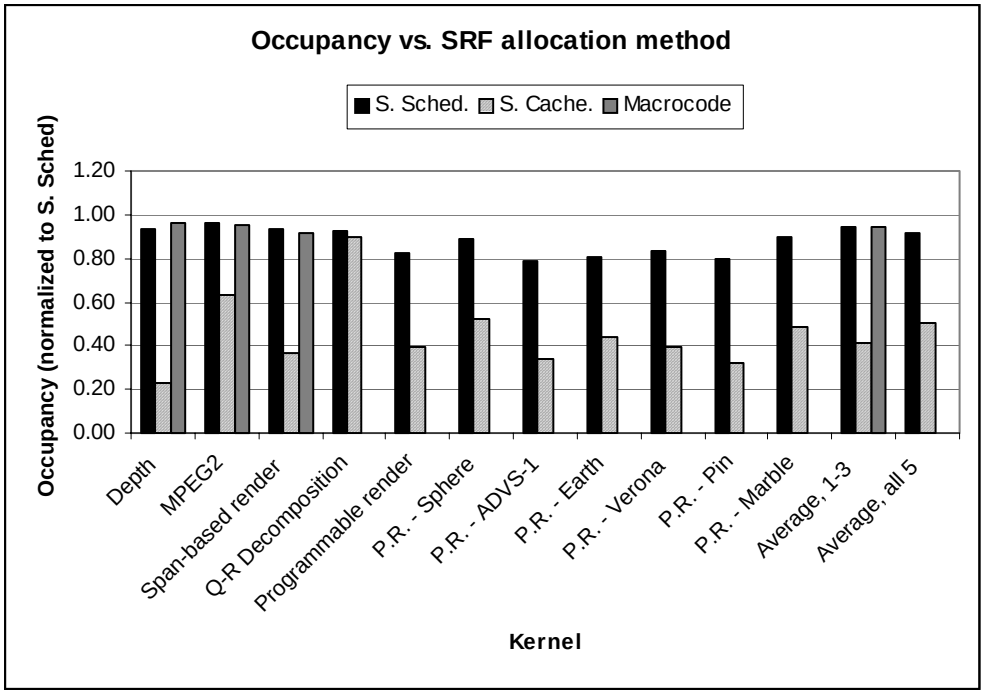



**FIGURE 8-20. Occupancy**

inputs and store final outputs. Additional memory traffic is generated when intermediate data is spilled from the SRF to memory, possibly increasing run time. The occupancy defines the fraction of run time spent doing useful work executing kernels. Occupancy reflects the amount of memory access and execution parallelism achieved in the application. Memory traffic that occurs in parallel with the execution of kernels does not increase run time, and therefore does not decrease occupancy. Some execution and memory traffic is inherently serial within the context of nearby kernels, for instance when a kernel produces a stream of addresses used as an index stream to load the input to the next kernel. However, this memory traffic can still be hidden by executing kernels from other parts of the program using a technique such as software pipelining. In practice, most non-parallel memory traffic is due to either to poor resource allocation (for instance, when a kernel waits to be executed because it needs to write to a location in the SRF occupied by a stream that is being stored) or to excessive memory traffic.

These three factors are often inversely related, and maximum performance involves finding the best combination of factors rather than maximizing any one factor. Increasing strip size can force some intermediate data to be spilled out of the SRF, or prevent some kernels from occurring in parallel with memory traffic because there is not enough space in the SRF for the data involved in both operations. Memory traffic can be decreased by keeping data that is reused between iterations in the SRF, but doing so often requires decreasing the strip size or reducing parallelism. Memory/execution parallelism can be increased by software pipelining the strip-mined loop so that kernels from one stage can be executed at the same time as serial memory accesses between kernels of the other stage. However, software pipelining requires intermediate data from different stages to fit in the SRF simultaneously, which usually requires decreasing strip size.

Stream scheduling often achieves a larger strip size than a macrocode programmer, improving performance. Macrocode programmers tend to allocate the SRF in a regular, conceptually straightforward manner. Stream scheduling produces a less regular, non-intuitive layout that enables it to fit more data in the SRF. Stream scheduling also handles loading initial inputs and/or storing final outputs in a more efficient manner. Macrocode programmers usually software pipeline the loop and place these loads and stores in the first stage and last stage, respectively. This approach ensures that the memory access can always occur in parallel with execution but requires allocating space to hold the initial inputs and final outputs for the entire duration of the loop. In contrast, stream scheduling uses shadows to ensure that the loads and stores can occur in parallel with one or more kernels as described in Section 6.3. Occasionally, using shadows causes it to allocate the SRF so that memory accesses cannot occur completely in parallel with execution. This loss of parallelism occurs when it reduces the duration of the shadows in order to fit all streams in the SRF and/or the duration of the shadows encompasses kernels with very short execution times. However, the loss is usually more than made up for by the increased strip size.

The StreamC executed with stream caching does not perform nearly as well as the StreamC compiled with stream scheduling. Stream caching increases memory traffic to

the point where all benchmarks except Q-R decomposition become memory bound, which severely reduces performance. Since stream caching cannot anticipate future accesses, all output streams must be stored back to memory. Further, streams are often arranged poorly in the SRF, resulting in more loads than would otherwise be required. Since identical StreamC was executed with stream scheduling and stream caching, the strip sizes are same. However, stream caching often failed to fit all of the intermediate data in the SRF resulting in even more memory traffic. These problems are discussed in more detail in Section 6.1. The performance of stream caching is similar to the performance of a conventional cache with perfect prefetching since streams are loaded in advance of execution. Thus, these results also demonstrate why a conventional cache is poorly suited to streaming applications.

### 8.2.5 Benchmark analysis

This section analyzes each benchmark to highlight specific examples of the tradeoffs described in the previous section.

#### Depth extraction

The depth extraction benchmark demonstrates the importance of minimizing memory traffic for applications with high throughput kernels. Depth extraction has only modest SRF requirements, but many of the kernels involved have short run times relative to the amount of data they produce. Stream scheduling and macrocode both make effective use of the SRF and require the minimum amount of memory traffic, resulting in almost identical run times (the StreamC run time is slightly higher due to a second-order effect involving the order of accesses made by the SRF clients). Stream caching, however, requires roughly twelve times as much memory traffic because it stores all intermediate results, most of which are never reused. Since there is proportionally little kernel execution time to hide the memory access time, these unnecessary stores quadruple run time.

#### MPEG2 encoding

Though the stream scheduled and macrocode versions of the MPEG benchmark have virtually the same run-time, the stream scheduled version actually does more work. The MPEG benchmark is strip-mined to process a batch of macroblocks each iteration. Stream scheduling allocates the SRF efficiently enough to process an entire row of macroblocks each iteration, primarily because it does not software pipeline loading the initial inputs and storing the final outputs. The macrocode only processes half a row of macroblocks each iteration. The stream scheduled version is able to search an entire row of reference macroblocks using the blocksearch kernel, increasing the total execution time of that kernel. Excluding the increase in the blocksearch kernel run time, the stream scheduled version is 3% faster.

## Span-based polygon rendering

The polygon rendering benchmark demonstrates the effect of different tradeoffs between strip size and parallelism. The primary difference between the stream scheduled version and the macrocode involves how loading the initial triangles and storing the final depth and color values are software pipelined. The macrocode version pipelines each of these memory accesses as a separate software pipeline stage. It devotes SRF space to hold the data for the entire loop. The stream scheduled version does not, enabling it to process 256 triangles per batch, while the macrocode can only process 80 triangles per batch.

Both the StreamC and macrocode versions are software pipelined in order to parallelize an inherently serial memory access. The polygon rendering benchmark consists of the series of kernels described in Section 8.2.2. The compact kernel produces a stream of addresses into the depth buffer that is used to load a stream of z-values for the next kernel, zcompare. This memory access is inherently serial. Both versions of polygon rendering are software pipelined so that kernels from the next iteration can be executed at the same time as this memory access. This reduces strip size, but the cost of the memory access is higher than the overhead of all the kernels in the loop.

## Q-R decomposition

Q-R decomposition is the only benchmark for which stream caching delivers performance near that of stream scheduling, due to a relatively high ratio of computation to memory traffic and a working set which eventually fits in the SRF. The majority of computation involves repeated passes by the householder kernel. Each pass generates output which is used as the input to the next pass. Both stream caching and stream scheduling keep this data in the SRF, but stream caching writes it back to memory unnecessarily. However, the time required to store the data is slightly less than the time required to produce it so this extra memory traffic is hidden. After transforming each block along the diagonal, the update kernels are used to update each of the remaining rows. Initially, all the rows do not fit in the SRF. The least-recently-used replacement policy employed by stream caching ejects all rows over the course of an update as a result. The additional time required to reload the first of these rows slows stream caching relative to stream scheduling, but reloading later rows is hidden by the update of earlier rows. Eventually, all of the remaining rows fit in the SRF and both SRF allocation methods keep them there. For this particular benchmark, the stream caching run-time approaches that of stream scheduling. However, stream caching still requires 71% more memory traffic due to the unnecessary writes. Higher computation speed relative to memory access time would rapidly degrade performance under stream caching.

## Programmable polygon rendering

The programmable polygon rendering benchmarks pose several significant challenges, resulting in slightly lower occupancy (average of 83%). First, they contain many sequential memory accesses due to texture look-ups. Second, the size of most streams varies widely, since a fixed number of vertices assembles into an unpredictable number of triangles, which rasterize into an unpredictable number of pixels, etc. Third, some of the streams are so small that the time to execute a kernel is less than the time to dispatch the required operations from the host processor to Imagine. Fourth, handling unpredictable stream lengths which can sometimes be zero requires data-dependent control flow.

The factors compound one another. Sequential memory accesses can be made to occur in parallel with execution using software pipelining, but varying output sizes make the time required for memory accesses and kernels unpredictable. The software pipelining algorithm described in Section 7.3.2 mitigates this unpredictability by estimating times based on average lengths and trying to cover all memory accesses with a constant proportion of execution rather than covering some memory accesses with the minimum amount of execution and others with more than enough. The time to dispatch operations can be hidden by dispatching them ahead of time, but operations can be only be dispatched up to the next data-dependent branch. The stream scheduler mitigates this cost somewhat by minimizing the number of operations that need to be dispatched by hoisting redundant operations out of loops. This cost to dispatch operation would be diminished significantly by integrating the host on the same chip as Imagine.

## 8.3 Summary

This chapter presented a quantitative evaluation of the compilers described in this thesis, with an emphasis on the communication scheduling portion of the KernelC compiler and the stream scheduling portion of the StreamC compiler. The results in this chapter demonstrate that by using communication scheduling the KernelC compiler can schedule kernels on a distributed register file architecture with schedule lengths comparable to an ideal single register file architecture. The results in this chapter also demonstrate that by using stream scheduling the StreamC compiler can deliver application performance that is comparable to, and in some cases better than, macrocode hand-written and optimized by expert imagine programmers. Stream caching, a simpler technique that manages the SRF at run-time, results in significantly worse performance due to increased memory traffic.

# Chapter 9

# Conclusion

## 9.1 Summary

The Imagine Media Processor introduces architectural innovations to meet the demands of media processing applications, but these innovations place additional burdens on the compiler. Media processing applications demand very high arithmetic rates and data bandwidth. To meet the arithmetic demands, Imagine connects functional units to multiple register files with shared interconnect instead of to a single register file with dedicated interconnect, which enables it to support many more functional units. To meet the data bandwidth demands, Imagine uses a stream register file instead of a cache, which requires an application to explicitly load and store long sequences of data called streams. These two innovations place additional burdens on the compiler: allocating the shared interconnect and managing the stream register file.

This thesis presents a programming system for the Imagine media processor that introduces an implementation of the *stream programming model* and two compiler techniques to support these architectural innovations. The stream programming model divides an application into two parts: *kernels,* computation intensive functions that operate on streams, and a *stream program* that defines the high-level control- and data-flow between kernels. The kernels are written using a language called KernelC. The KernelC compiler uses *communication scheduling* to allocate the shared interconnect and manage data movement between functional units and multiple register files. The stream program is

written using a language called StreamC. The StreamC compiler uses *stream scheduling* to manage the stream register file and determine when to load and store streams.

The programming system presented in this thesis enables efficient high-performance application development for Imagine. Multiple applications have been implemented for Imagine, including stereo depth extraction, MPEG2 encoding, Q-R decomposition, and polygon rendering. Experimental results presented in this thesis demonstrate that the KernelC compiler, scheduling kernels used in these applications, can produce schedules for an architecture with multiple register files with shared interconnect that are comparable to those for the same architecture with an ideal single register file. Stream scheduling delivers performance equal to or better than a programmer can achieve with a kind of assembly language called macrocode that requires allocating the SRF by hand, and significantly outperforms a run-time “stream caching” approach.

## 9.2 Future Work

This thesis focused on the essential portions of a programming system for Imagine; there are several capabilities that would be useful extensions to this system. These capabilities include extending communication scheduling to consider register pressure and extending the StreamC compiler to a multiprocessor system.

### 9.2.1 Communication scheduling with register pressure

Communication scheduling could be improved by considering register pressure when assigning communications to routes. As described in this thesis, communication scheduling only considers the availability of shared interconnect resources. Considering register pressure would result in trying to store values in as few register files as possible, adding copy operations to allow a value to be stored in a another register file until just before it is used, and preferentially scheduling operations on functional units that can access register files with relatively low register pressure.

### 9.2.2 Multiprocessor systems

Stream programs map to multiprocessor systems with relative ease because the stream programming model makes high-level data flow and parallelism explicit, but finding the best arrangement of kernels on processors is a challenging problem. The simplest arrangement is to run each kernel only on a specific processor. More complicated arrangements include stripmining the application and running a small number of strip-mined loop iterations simultaneously on different processors.

## 9.3 Epilogue

Programming models and compilers need to evolve to allow more efficient application development for media processing architectures. Media processing applications are becoming the dominant desktop workload and are already prevalent in embedded systems [11]. As the importance of these applications increases, processors optimized for media processing will become more and more common. At present, applications for these processors often are written at a very low level in order to achieve good performance. As this thesis demonstrates, combining the right programming model and compiler techniques can allow high-level development of these applications without sacrificing performance.

# Appendix A: Detailed Results

The following are the detailed results for the KernelC compiler:

| Kernel | Loop sched. len. | | Computable lower bound | | | | Operations | | Registers | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | SRF | DRF | CLB | SW pipe. | Crit. path/ RMII | Res. limit | SRF | DRF | SRF | DRF (dup) | DRF (imb) |
| **Block warper** | | | | | | | | | | | |
| blockwarp | 62 | 63 | 58 | n | 58 | 22 | 105 | 109 | 46 | 86 | 204 |
| **Depth Extraction** | | | | | | | | | | | |
| blockfill | 4 | 4 | 4 | n | 4 | 4 | 11 | 11 | 5 | 22 | 34 |
| blocksad | 14 | 14 | 12 | y | 12 | 12 | 117 | 128 | 44 | 82 | 136 |
| byte2word | 7 | 7 | 7 | y | 7 | 5 | 13 | 13 | 9 | 29 | 34 |
| convfx3x3 | 10 | 11 | 9 | y | 9 | 7 | 59 | 61 | 40 | 62 | 119 |
| convfx7x7 | 29 | 30 | 28 | y | 9 | 28 | 175 | 200 | 108 | 165 | 374 |
| exdepth | 11 | 11 | 10 | n | 10 | 5 | 16 | 16 | 9 | 30 | 34 |
| extemp3 | 4 | 4 | 4 | n | 4 | 4 | 11 | 11 | 6 | 19 | 34 |
| extemp7 | 6 | 6 | 6 | y | 6 | 6 | 23 | 23 | 6 | 23 | 34 |
| **FFT** | | | | | | | | | | | |
| fft8c1024 | 42 | 42 | 42 | y | 32 | 42 | 341 | 369 | 69 | 132 | 204 |
| **MPEG2 Encoding** | | | | | | | | | | | |
| blocksearch | 96 | 99 | 77 | n | 69 | 77 | 409 | 466 | 91 | 187 | 323 |
| corr | 38 | 37 | 32 | y | 22 | 32 | 262 | 276 | 51 | 109 | 187 |
| dct | 40 | 40 | 36 | y | 14 | 36 | 220 | 242 | 73 | 130 | 238 |
| diff | 34 | 34 | 32 | y | 16 | 32 | 208 | 218 | 36 | 75 | 153 |
| icolor | 15 | 15 | 12 | y | 10 | 12 | 77 | 87 | 38 | 73 | 136 |
| idct | 41 | 42 | 37 | y | 14 | 37 | 250 | 281 | 63 | 178 | 408 |
| idxgen | 35 | 35 | 35 | n | 35 | 32 | 87 | 87 | 28 | 68 | 187 |
| mv2idx | 44 | 45 | 36 | y | 8 | 36 | 151 | 165 | 36 | 100 | 238 |
| pcolor | 143 | 150 | 107 | n | 72 | 107 | 705 | 763 | 86 | 191 | 272 |
| rle | 7 | 7 | 7 | n | 7 | 5 | 25 | 25 | 47 | 130 | 221 |
| **Polygon Rendering** | | | | | | | | | | | |
| compact_recycle | 10 | 10 | 10 | y | 10 | 8 | 56 | 59 | 21 | 46 | 68 |
| glshader | 118 | 117 | 99 | y | 12 | 99 | 328 | 326 | 144 | 218 | 510 |
| hash | 18 | 18 | 17 | y | 10 | 17 | 124 | 129 | 66 | 107 | 204 |
| mergefrag | 45 | 45 | 34 | n | 31 | 34 | 149 | 154 | 39 | 99 | 136 |
| project | 27 | 27 | 27 | y | 27 | 27 | 197 | 205 | 78 | 141 | 238 |
| sort32frag | 247 | 245 | 208 | n | 130 | 208 | 826 | 905 | 50 | 132 | 204 |
| spanrast | 26 | 27 | 21 | y | 16 | 21 | 156 | 165 | 40 | 74 | 119 |
| spansgen | 61 | 61 | 53 | y | 30 | 53 | 360 | 398 | 60 | 117 | 204 |
| spansprep | 64 | 70 | 57 | y | 57 | 54 | 564 | 607 | 82 | 178 | 408 |
| xform | 12 | 12 | 12 | y | 12 | 9 | 50 | 50 | 27 | 54 | 153 |
| zcompare | 7 | 7 | 7 | y | 7 | 5 | 23 | 23 | 10 | 28 | 51 |
| **Q-R Matrix Decomposition** | | | | | | | | | | | |
| backsub1 | 14 | 14 | 14 | n | 14 | 5 | 16 | 16 | 17 | 46 | 68 |
| backsub2 | 19 | 19 | 16 | n | 16 | 5 | 20 | 20 | 20 | 56 | 102 |
| backsub3 | 32 | 32 | 31 | n | 31 | 10 | 34 | 34 | 16 | 43 | 85 |
| house2 | 23 | 24 | 23 | y | 10 | 23 | 159 | 184 | 101 | 212 | 340 |
| sumsqr | 10 | 10 | 10 | y | 10 | 9 | 74 | 78 | 21 | 50 | 102 |
| update1 | 37 | 37 | 26 | y | 26 | 16 | 101 | 102 | 39 | 85 | 289 |
| update2 | 16 | 16 | 16 | y | 8 | 16 | 110 | 119 | 37 | 79 | 204 |
| **Sorting** | | | | | | | | | | | |
| bisort | 10 | 10 | 10 | y | 10 | 8 | 75 | 83 | 20 | 59 | 85 |
| merge | 22 | 22 | 15 | y | 15 | 13 | 69 | 70 | 20 | 59 | 102 |
| sort32 | 52 | 53 | 52 | y | 10 | 52 | 233 | 250 | 32 | 87 | 153 |

The following are the detailed results for the StreamC compiler:

| | Run Time (cycles) | | | Kernel Exec. Time (cycles) | | |
|---|---|---|---|---|---|---|
| | **Stream Sched.** | **Stream Cache** | **Macro-code** | **Stream Sched.** | **Stream Cache** | **Macro-code** |
| **Depth** | 2128680 | 8719210 | 2067664 | 1988066 | 1988066 | 1987936 |
| **MPEG2** | 4304550 | 6533840 | 4309075 | 4121082 | 4121082 | 4099464 |
| **Span-based render** | 11242940 | 28732810 | 12345158 | 10503796 | 10503796 | 11360700 |
| **Q-R Decomposition** | 712770 | 732690 | | 657140 | 657140 | |
| **Programmable render** | | | | | | |
| P.R. - Sphere | 13084160 | 22378320 | | 11633104 | 11633104 | |
| P.R. - ADVS-1 | 3552850 | 8210430 | | 2804293 | 2804293 | |
| P.R. - Earth | 13007930 | 23988590 | | 10479312 | 10479312 | |
| P.R. - Verona | 23353710 | 49574070 | | 19372688 | 19372688 | |
| P.R. - Pin | 9975960 | 24794120 | | 7933138 | 7933138 | |
| P.R. - Marble | 12130460 | 21843240 | | 10866323 | 10693943 | |

| | Memory Traffic (words) | | | Strip Size (varies, see unit) | | | |
|---|---|---|---|---|---|---|---|
| | **Stream Sched.** | **Stream Cache** | **Macro-code** | **Stream Sched.** | **Stream Cache** | **Macro-code** | **unit** |
| **Depth** | 907176 | 10551976 | 915232 | 1 | 1 | 1 | rows |
| **MPEG2** | 938648 | 3126104 | 970904 | 20 | 20 | 10 | blocks |
| **Span-based render** | 2480496 | 24699984 | 2400728 | 256 | 256 | 80 | triangles |
| **Q-R Decomposition** | 286400 | 489616 | | 8 | 8 | | rows |
| **Programmable render** | | | | | | | |
| P.R. - Sphere | 2295600 | 16437864 | | 480 | 480 | | vertices |
| P.R. - ADVS-1 | 797680 | 6426944 | | 224 | 224 | | vertices |
| P.R. - Earth | 2373696 | 17319808 | | 64 | 64 | | vertices |
| P.R. - Verona | 3163736 | 35634312 | | 40 | 40 | | vertices |
| P.R. - Pin | 1716448 | 21406848 | | 88 | 88 | | vertices |
| P.R. - Marble | 976608 | 16211176 | | 64 | 64 | | vertices |

# Bibliography


[1] Aho, A., Sethi, R., and Ulman, J., *Compilers*. Addison-Wesly Publishing Company, 1998.

[2] Basoglu, C., et al., “The MAP-CA VLIW-based Media Processor From Equator Technologies Inc. and Hitachi Ltd.”, www.equator.com, Nov. 2002.

[3] Benitez, M., and Davidson, J., “Code generation for streaming: an access/execute mechanism.” *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp. 132-141.

[4] Breternitz, M., and Shen, J., “Implementation optimization techniques for architecture synthesis of application-specific processors.” *Proceedings of the 24th Annual International Symposium on Microarchitecture*, Nov. 1991, pp. 114-123.

[5] Capitanio, A., Dutt, N., and Nicolau, A., “Partitioned register files for VLIWs: a preliminary analysis of tradeoffs.” *Proceedings of the 25th Annual International Symposium on Microarchitecture*, Dec. 1992, pp. 292-300.

[6] Carter, J., et al., “Impulse: building a smarter memory controller.” *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, Jan. 1999, pp. 132-141.

[7] Catthoor, F., Dutt, N., and Kozyrakis, C., “How to solve the current memory access and data transfer bottlenecks: at the processor architecture or at the compiler level?” *Proceedings of Design, Automation, and Test in Europe Conference 2000*, Mar. 2000, pp. 426-433.

[8] Colwell, R., et al. “Architecture and implementation of a VLIW supercomputer.” *Proceedings in Supercomputing*, Nov. 1990, pp. 910-919.

[9] Dehnert, J., and Towle, R., “Compiling for the Cydra 5." *Journal of Supercomputing*, Jan. 1993, 182-227.

[10] Desoli, G. “Instruction assignment for clustered VLIW DSP compilers: A new approach.” Technical Report HPL-98-13, Hewlett-Packard Laboratories, Feb. 1998.

[11] Diefendorff, K. and Dubey, P. “How multimedia workloads will change processor design.” *Computer*, Sept. 1997, pp. 43-45.

[12] Ellis, J., *Bulldog: A compiler for VLIW architectures*. MIT Press, 1986.

[13] Fabri, J., “Automatic storage optimization,” *Proceedings of the ACM SIGPLAN 1979 Symposium on Compiler Construction*, 1979, pp. 83-91.

[14] Fernandes, M., Llosa, J., and Topham, N., “Distributed modulo scheduling.” *Proceedings of the 5th Annual International Conference on High Performance Computer Architecture*, Jan. 1999, pp. 130-134.

[15] Fisher, J., “Trace scheduling: a technique for global microcode compaction.” *IEEE Transactions on Computers*, July 1981, pp. 478-490.

[16] Gergov, J., “Algorithms for compile-time memory optimization.” *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1999, pp. 907-908.

[17] Grossman, J., and Dally, W., “Point sample rendering.” *Proceedings of the 9th Eurographics Workshop on Rendering*, June 1998, pp. 181-192.

[18] Hennessy, J., and Patterson, D., *Computer architecture: a quantitative approach.* Morgan Kaufmann Publishers, 1990.

[19] Hoogerbrugge, J., and Corporaal, H., “Transport-triggering vs. operation-triggering,” *International Conference on Compiler Construction*, April 1994.

[20] Huff, R., “Lifetime-sensitive modulo scheduling”, *Proceedings of the Conference on Programming Language Design and Implementation*, June 1993, pp. 258-267.

[21] Hwu, W., et al., “The superblock: an effective structure for VLIW and superscalar compilation,” *Journal of Supercomputing*, July 1993, pp. 229-248.

[22] Kanade, T., et al., “Development of a video-rate stereo machine.” *Proceedings of the International Robotics and Systems Conference*, 1995, pp. 95-100.

[23] Kapasi, U., et. al, “Efficient conditional operations for data-parallel architectures.” *Proceedings of the 33rd Annual Symposium on Microarchitecture*, Dec. 2000, pp. 159-170.

[24] Khailany, B., et al., “Imagine: signal and image processing using streams (ppt).” *Hotchips 12*, Aug. 2000.

[25] Khailany, B., et al., “Imagine: media processing with streams.” *IEEE Micro*, Mar./April 2001.

[26] Kirkpatrick, S., Gelatt, C., Vecchi, M., “Optimization by simulated annealing,” *Science*, May 1983.

[27] Kozyrakis, C., and Patterson, D., “A new direction in computer architecture research,” *IEEE Computer*, Nov. 1998, pp. 24-32.

[28] Lam, M., “Software pipelining: an effective scheduling technique for VLIW machines.” *Proceedings of the Conference on Programming Language Design and Implementation*, June 1988, pp. 318-328.

[29] Lavery, D., and Hwo, W., “Unrolling-based optimizations for modulo scheduling.” *Proceedings of the 28th Annual International Symposium on Microarchitecture*, 1995, pp. 327-337.

[30] Lee, C., and Smith, J., “A study of partitioned vector register files.” *Proceedings of the 1992 Conference on Supercomputing*, July 1992, pp. 94-103.

[31] Lee, W., et al., “Space-time scheduling of instruction-level parallelism on a raw machine,” *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

[32] Lowney, P., et al., “The Multiflow trace scheduling compiler.” *Journal of Supercomputing*, Jan. 1993, pp. 51-142.

[33] Mangione-Smith, W., Abraham, S., and Davidson, E. “Register requirements of pipelined processors.” *Proceedings of the International Conference on Supercomputing*, July 1992, pp. 260-271.

[34] Mattson, P., et al., “Communication scheduling,” *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000, pp. 82-92.

[35] McKee, A., et al., “Design and evaluation of dynamic access ordering hardware, *Proceedings of the 1996 International Conference on Supercomputing*, May 1996, pp. 125-132.

[36] McKee, A., et. al., “Smarter Memory: Improving Bandwidth for Streamed References,” *IEEE Computer*, July 1998, pp. 54-63.

[37] Muchnick, S., A*dvanced Compiler Design and Implementation*, Morgan Kaufmann, 1997.

[38] Nystrom, E., and Eichenberger, A., “Effective cluster assignment for modulo scheduling.” *Proceedings of the 31st Annual International Symposium on Microarchitecture*, Dec. 1998, pp. 103 - 114.

[39] Owens, J., Dally, et al., “Polygon rendering on a stream architecture.” *Proceedings of the 2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, Aug. 2000, pp. 23-32.

[40] Ozer, E., Banerjia, S., and Conte, T., “Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures.” *Proceedings of the 31st Annual International Symposium on Microarchitecture*, Dec. 1998, pp. 308-315.

[41] Ramakishnan, S., “Software-pipelining in PA-RISC compilers.” *Hewlett-Packard Journal*, June 1992.

[42] Rau, B., Glaeser, C., and Picard, R., “Efficient code generation for horizontal architectures: Compiler techniques and architectural support." *Proceedings of the International Symposium on Computer Architecture*, July 1982, pp. 131-139.

[43] Rixner, S., et al., “A bandwidth-efficient architecture for media processing,” *Proceedings of the 31st Annual International Symposium on Microarchitecture*, Dec., 1998, pp. 3-13.

[44] Rixner, S., et al., “Register organization for media processing.” *6th International Symposium on High-Performance Computer Architecture*, Jan. 2000, pp. 375-386.

[45] Rixner, S., et al., “Memory Access Scheduling”, *27th Annual International Symposium on Computer Architecture*, June 2000, pp. 128-138.

[46] Rixner, S., *A bandwidth efficient architecture for a streaming media processor.* Ph.D. thesis, Massachusetts Institute of Technology, 2000.

[47] Schreiber, R., and Van Loan, C., “A storage-efficient WY representation for products of Householder transformations,” *SIAM Journal of Scientific and Statistical Computing*, 10(1):53--57, 1989.

[48] Stotzer, E. and Leiss, E., “Modulo scheduling for the TMS320C6x VLIW DSP architecture,” *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, May 1999, pp. 28-34.

[49] Wolfe, M., “More iteration space tiling.” *Proceedings of the 1989 Conference on Supercomputing*, Nov. 1989, pp. 655-664.