CONDITIONAL TECHNIQUES FOR STREAM PROCESSING KERNELS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Ujval J. Kapasi
February 2003

Abstract

The stream programming model casts applications as a set of sequential data streams that are operated on by data-parallel computation kernels. Previous work has shown that this model is a powerful representation for media processing applications, such as image- and signal-processing, because it captures the locality and concurrency inherent to an application. This careful handling of important application properties results in kernels that are compute-intensive—i.e., kernels that perform a large number of arithmetic operations per unit of inter-kernel communication bandwidth. Furthermore, the stream model can be implemented with efficient VLSI designs, such as the Imagine Programmable Stream Processor. The Imagine chip supports 48 ALUs on a single die, operating at over 200MHz. This large number of ALUs provides a high peak performance, but makes efficiently executing kernels with conditional code a challenge.

We will introduce two techniques for efficiently supporting kernel conditionals, such as if-statements, case-statements, and while-loops: conditional routing and conditional streams. Conditional routing is a code transformation that exploits the trade-off of increasing inter-kernel communication in order to increase the performance of kernel inner-loops containing conditional code. The second technique we will discuss is the use of a conditional stream, which is a mechanism to reduce the amount of load-imbalance that arises between parallel processing clusters on a single stream processor chip. Load-imbalance results when different conditional paths are taken on different processing clusters, and causes one or more of the clusters to wait idle for the others to complete a kernel. We will also present a case study of the impact of these techniques on a programmable polygon rendering pipeline that contains many unpredictable conditionals. We show that our techniques improve the performance of this application by $1.9\times$.

Acknowledgements

Contents

List of Tables

List of Figures

List of Kernels

Chapter 1

Introduction

Recent research has shown that programmable stream processors efficiently support large numbers of ALUs on a single chip to achieve a high sustained performance on media processing applications, such as 2-D and 3-D graphics, image processing, and signal processing [Kapasi *et al.*, 2003; Khailany *et al.*, 2001]. Furthermore, stream processors dissipate considerably less power per operation than traditional programmable processors. For example, the *IMAGINE* streaming media processor supports 48 ALUs on a single die in a 0.15μm process. At the most power-efficient operating voltage of 1.2V, the processor sustains between 1.4 and 2.2 GFLOPS at 1.9W on representative media processing applications.

It turns out that many media processing applications are very regular: they repetitively perform the same set of operations on a large set of similar data elements. This can be modeled as a sequence of computation *kernels* that operate on data *streams*, where each kernel applies the same function to all the records in a stream. As we will see in the next chapter, stream processors are optimized to exploit this regularity, and this is in part why they are so efficient from a hardware perspective.

1.1 Conditionals are Challenging

However, the flip-side is that it is a challenge to efficiently execute conditionals, such as if-then-else statements and while-loops, since they seem to break this regularity assumption (Section 2.3 will discuss this in more detail). The end result is that the presence of data-dependent conditionals in an application reduces the fraction of the peak processor performance achieved. To address this, we will introduce new solutions that increase the performance of applications with data-dependent conditionals on stream processors. For example, the conditional techniques we introduce are able to speed up an OpenGL polygon rendering pipeline 1.9x over an implementation that only uses other existing techniques, such as speculation and predication. Furthermore, on *IMAGINE*, the additional hardware dedicated to our new conditional techniques occupies only about 1% of the chip area.

We are concerned with media processing applications in this thesis, and in particular, we will focus on a 3-D OpenGL polygon rendering pipeline. However, as stream processing technology matures, we envision that the contributions in this thesis will apply to other application domains as well. In fact, the techniques discussed here may become even more important as we try to reap the benefits of efficient stream processing hardware on increasingly difficult (i.e., irregular) applications.

1.2 Research Contributions

The primary contributions of this thesis to the fields of computer architecture and stream processing are:

1. A kernel-level code transformation, *conditional routing*, that improves the performance of kernels that contain conditional constructs, such as if- and case-statements and while-loops. Conditional routing improves kernel schedules on VLIW clusters by replacing conditional branches with conditional communication. Removing the conditional branches also enables efficient execution on stream processors that are built using a single-instruction multiple-data (SIMD) array of processing clusters.

2. A kernel-level code transformation, *state unrolling*, that improves the VLIW schedules of while-loops inside kernel main loops. State-unrolling is especially important when there is a long loop-carried dependency in the while-loop, rendering traditional loop-unrolling or software-pipelining ineffective.

3. A new mechanism, called *conditional streams*, which improves the load-balance between parallel processing clusters during kernel execution.

4. A micro-architecture for accelerating the performance of conditional stream operations.

5. An evaluation of the impact of the newly introduced conditional techniques on the performance of a 3-D polygon rendering application, which contains many conditional statements in its kernels.

1.3 Thesis Roadmap

In Chapter 2 we will motivate why stream processors are an interesting class of machines for the media processing domain, how they are built, and how they operate. In particular, we will show that stream processors can efficiently support a large number of ALUs by using a SIMD array of processing clusters, where each processing cluster is a statically scheduled VLIW engine. While stream processors are great at executing very regular codes, we will show that efficiently executing code with conditionals is a challenge on these stream processors. We will discuss three specific challenges introduced by the presence of conditionals: improving VLIW schedules, enabling efficient performance on the restrictive SIMD execution model, and reducing the load-imbalance between multiple processing clusters. Each of the next three chapters is devoted to addressing each of these challenges in turn.

Chapter 3 focuses on how one of our new techniques, *conditional routing*, addresses the first challenge. We will show that compared to existing techniques, conditional routing enables efficient VLIW schedules by converting conditional branches to conditional communication. Conditional routing can be applied to both if-statements and while-loops. We will also introduce another new technique, state unrolling, and show that it should be used in conjunction with conditional routing in order to improve the performance of while-loops even further.

The second challenge is addressed in Chapter 4. In this chapter we show that conditional routing enables efficient performance on stream processors with an array of SIMD processing clusters. The SIMD execution model is attractive because it enables simplified hardware, but in return places restrictions on control-flow. We will show how the performance of kernels implemented using conditional routing, unlike other techniques, is not reduced by these control-flow restrictions.

We address the third and final challenge in Chapter 5. This chapter shows how *conditional streams* reduce the load-imbalance between multiple clusters in a stream processor. Since conditional streams require hardware acceleration for good performance, we will present a micro-architecture to support them in this chapter.

Finally, we will present our conclusions in Chapter 6, and indicate fruitful areas for further research based on these conclusions.

Chapter 2

Stream Processing

The complexity of modern media processing, including 3D graphics, image compression, and signal processing, requires tens to hundreds of billions of computations per second. To achieve these computation rates, current media processors use special-purpose architectures tailored to one specific application. Such processors require significant design effort and are thus difficult to change as media-processing applications and algorithms evolve.

The demand for flexibility in media processing motivates the use of programmable processors. However, very large-scale integration constraints limit the performance of traditional programmable architectures. In modern VLSI technology, computation is relatively cheap—thousands of arithmetic logic units (ALUs) that operate at multi-gigahertz rates can fit on a modestly sized 1-cm^2 die. The problem is that delivering instructions and data to those ALUs is prohibitively expensive. For example, only 6.5 percent of the Itanium 2 die is devoted to the 12 integer and two floating-point ALUs and their register files [Naffziger *et al.*, 2002]; communication, control, and storage overhead consume the remaining die area. In contrast, the more efficient communication and control structures of a special-purpose graphics chip, such as the Nvidia GeForce4 [Montrym and Moreton, 2002], enable the use of many hundreds of floating-point and integer ALUs to render 3D images.

In part, such special-purpose media processors are successful because media applications have abundant parallelism—enabling thousands of computations to occur in parallel—and require minimal global communication and storage—enabling data to pass directly from one ALU to the next. A stream architecture exploits this locality and concurrency by partitioning the communication and storage structures to support many ALUs efficiently:

- operands for arithmetic operations reside in local register files (LRFs) near the ALUs, in much the same way that special-purpose architectures store and communicate data locally;
- streams of data capture coarse-grained locality and are stored in a stream register file (SRF), which can efficiently transfer data to and from the LRFs between major computations; and
- global data is stored off-chip only when necessary.

These three explicit levels of storage form a data bandwidth hierarchy with the LRFs providing an order of magnitude more bandwidth than the SRF and the SRF providing an order of magnitude more bandwidth than off-chip storage. This bandwidth hierarchy is well matched to the characteristics of modern VLSI technology, as each level provides successively more storage and less bandwidth. By exploiting the locality inherent in media-processing applications, this hierarchy stores the data at the appropriate level, enabling hundreds of ALUs to operate at close to their peak rate. Moreover, a stream architecture can support such a large number of ALUs in an area- and power-efficient manner. Modern high-performance microprocessors and digital signal processors continue to rely on global storage and communication structures to deliver data to the ALUs; these structures use more area and consume more power per ALU than a stream processor.

An example of a stream processor is the *IMAGINE* Processor (Appendix A), which targets the media processing application domain [Khailany *et al.*, 2001; Rixner *et al.*, 1998]. While retaining complete programmability, *IMAGINE* delivers roughly four times the peak performance of TI's high-performance floating-point DSP, the TMS320C6713 [TI, 2001], and does so with roughly two-thirds the energy per operation, despite being implemented in a CMOS process technology that is one generation older. Likewise, if compared to TI's fixed-point DSP, the TI C64x [Agarwala *et al.*, 2002], and normalized to the same process technology and operating voltage, *IMAGINE* sustains roughly $2.5\times$ the peak performance at roughly the same power-efficiency [Khailany, 2003], despite having the extra overhead of floating-point arithmetic support. The next section describes the key characteristics of the stream programming model and the following section describes stream processors in more detail.

2.1 Stream Programming Model

The central idea behind stream processing is to organize an application into streams and kernels to expose the inherent locality and concurrency in media-processing applications. In most cases, not only do streams and kernels expose desirable properties of media applications, but they are also a natural way of expressing the application. This leads to an intuitive programming model that maps directly to stream architectures with tens to hundreds of ALUs.

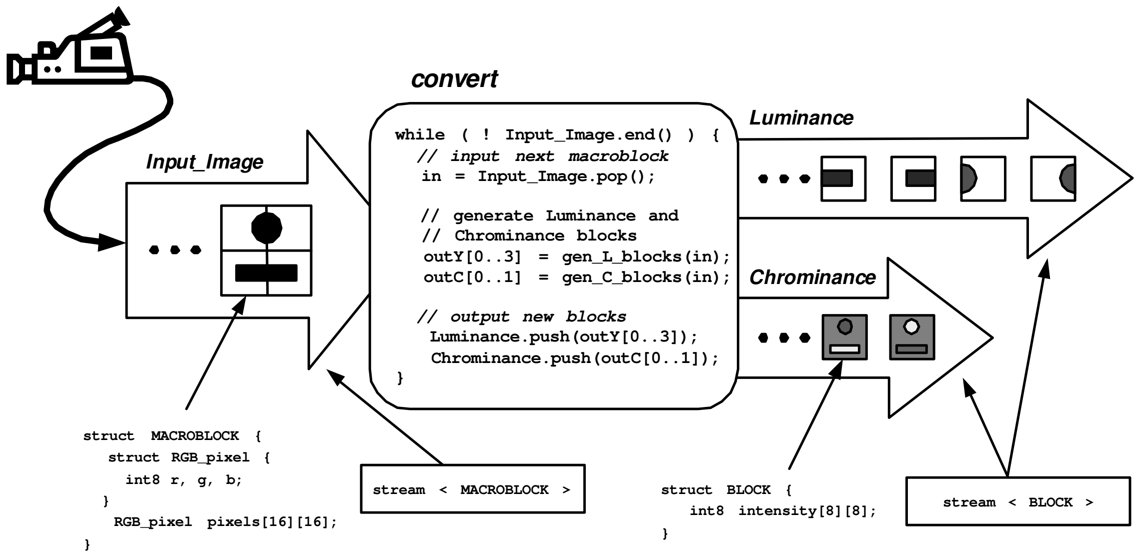
2.1.1 Example Application

Figure 2.1 illustrates input and output streams and a kernel taken from a MPEG-2 video encoder application. Figure 2.1a shows how a kernel operates on streams graphically, while Figure 2.1b shows this process in a simplified form of StreamC, a stream programming language.

`Input_Image` is a stream that consists of image data from a camera. Elements of `Input_Image` are 16×16 pixel regions, or macroblocks, on which the *convert* kernel operates. The kernel applies the same computation to the macroblocks in `Input_Image`, decomposing each one into six 8×8 blocks—four luminance blocks and two 4:1 sub-sampled chrominance blocks—-and appends them to the `Luminance` and `Chrominance` output streams, respectively.

Streams

As Figure 2.1a shows, streams contain a sequence of elements of the same type. Stream elements can be simple, such as a single number, or complex, such as the coordinates of a triangle in 3D space. Streams need not be the same length—for example, the `Luminance` stream has four times as many elements as the input stream. Further, `Input_Image` could contain all of the macroblocks in an entire video frame, only a row of macroblocks from



```
while ( ! Input_Image.end() ) {
  // input next macroblock
  in = Input_Image.pop();

  // generate Luminance and
  // Chrominance blocks
  outY[0..3] = gen_L_blocks(in);
  outC[0..1] = gen_C_blocks(in);

  // output new blocks
  Luminance.push(outY[0..3]);
  Chrominance.push(outC[0..1]);
}
```

```
struct MACROBLOCK {
  struct RGB_pixel {
    int8 r, g, b;
  }
  RGB_pixel pixels[16][16];
}
```

```
stream < MACROBLOCK >
```

```
struct BLOCK {
  int8 intensity[8][8];
}
```

```
stream < BLOCK >
```

(a)

```
stream < MACROBLOCK > Input_Image(NUM_MB);
stream < BLOCK > Luminance(NUM_MB*4), Chrominance(NUM_MB*2);

Input_Image = Video_Feed.get_macroblocks(currpos, NUM_MB);
currpos += NUM_MB;
convert(Input_Image, Luminance, Chrominance);
```

(b)

Figure 2.1: Streams and a kernel from an MPEG-2 video encoder application. (a) The *convert* kernel translates a stream of macroblocks containing RGB pixels into streams of blocks containing luminance and chrominance pixels. (b) Textual expression of the flow of streams through kernels. The syntax is similar to that of the StreamC language. The datatype of each stream is specified within the angle brackets (< >). Kernel invocation syntax is similar to that of a regular function call in “C”. The `get_macroblocks` method is assumed to be defined earlier; it simply returns a consecutive number of macroblocks starting at the specified position in the stream.

the frame, or even a subset of a single row. In the stream code in Figure 2.1b, the value of `NUM_MB` controls the length of the input stream.

Kernels

The *convert* kernel consists of a loop that processes each element from the input stream. The body of the loop first pops an element from its input stream, performs some computation on that element, and then pushes the results onto the two output streams. Kernels can have one or more input and output streams and perform complex calculations ranging from a few to thousands of operations per input element—one implementation of *convert* requires 6,464 operations per input macroblock to produce the six output blocks. The only external data that a kernel can access are its input and output streams. For example, *convert* cannot directly access the data from the video feed; instead, the data must first be organized into a stream.

Full application

A full application, such as the MPEG-2 encoder, is composed of multiple streams and kernels. This application inputs a sequence of video images and compresses it into a single bitstream consisting of three types of frames: intra-coded, predicted, and bidirectional. The encoder compresses I-frames using only information contained in the current frame, and it compresses P- and B-frames using information from the current frame as well as additional reference frames. For example, Figure 2.2 shows one possible mapping of the portion of the MPEG-2 encoder application that encodes only I-frames into the stream processing model. Solid arrows represent data streams, and ovals represent computation kernels.

The encoder receives a stream of macroblocks (`Input_Image`) from the video feed as input, and the first kernel (*convert*) processes this input. The discrete cosine transform (DCT) kernels then operate on the output streams produced by *convert*. Upon execution of all the computation kernels, the application either transmits the compressed bitstream over a network or saves it for later use. The application uses the two reference streams to compress future frames.

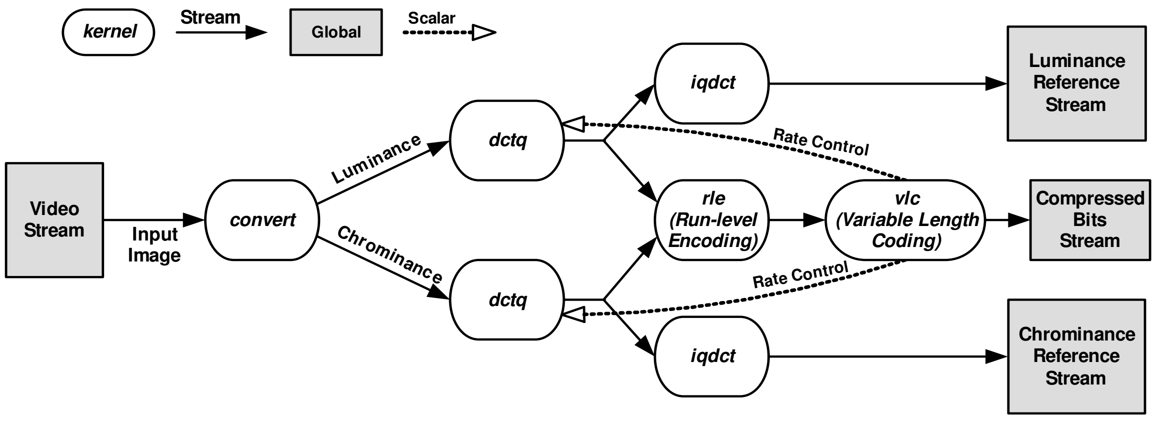


Figure 2.2: MPEG-2 I-frame encoder mapped to streams and kernels. The encoder receives a stream of macroblocks from a frame in the video feed as input, and the first kernel (*convert*) processes these. The discrete cosine transform (DCT) kernels then operate on the output streams produced by *convert*. Q=quantization; IQ=inverse quantization.

2.1.2 Locality and Concurrency

By making communication between computation kernels explicit, the stream-processing model exposes both the locality and concurrency inherent in media-processing applications. The model exposes locality by organizing communication into three distinct levels:

- *Local*. Temporary results that only need to be transferred between scalar operations within a kernel use local communication mechanisms. For example, temporary values in `gen_L_blocks` are only referenced within *convert*. This type of communication cannot be used to transfer data between two different kernels.

- *Stream*. Data are communicated between computation kernels explicitly as data streams. In the MPEG-2 encoder, for example, the `Luminance` and `Chrominance` streams use this type of communication.

- *Global*. This level of communication is only for truly global data. This is necessary for communicating data to and from I/O devices, as well as for data that must persist throughout the application. For example, the MPEG-2 Iframe encoder uses this level of communication for the original input data from a video feed and for the reference frames, which must persist throughout the processing of multiple video frames in the off-chip dynamic RAM (DRAM).

By requiring programmers to explicitly use the appropriate type of communication for each data element, the stream model expresses the applications inherent locality. For example, the model only uses streams to transfer data between kernels and does not burden them with temporary values generated within kernels. Likewise, the model does not use global communication for temporary streams.

The stream model also exposes concurrency in media-processing applications at multiple levels:

- *Instruction-level parallelism (ILP).* As in traditional processing models, the stream model can exploit parallelism between the scalar operations in a kernel function. For example, the operations in `gen_L_blocks` and `gen_C_blocks` can occur in parallel.

- *Data parallelism.* Because kernels apply the same computation to each element of an input stream, the stream model can exploit data parallelism by operating on several stream elements at the same time. For example, the model parallelizes the main loop in *convert* so that multiple computation elements can each decompose a different macroblock.

- *Task parallelism.* Multiple computation tasks, including kernel execution and stream data transfers, can execute concurrently as long as they obey dependencies in the stream graph. For example, in the MPEG-2 I-frame application, the two *dctq* kernels could run in parallel.

2.1.3 High-Level Stream Languages

Stream languages are used to textually code applications in the stream programming model. A language such as StreamC is used to code the flow of streams through kernels in an application, while a language such as KernelC is used to code the individual kernels. Both of these languages are specifically covered by [Mattson, 2001]. These languages are compiled using software tools designed specifically to both process stream languages and target stream architectures [Kapasi *et al.*, 2002b; Mattson *et al.*, 2000; Mattson, 2001].

Examples of how several different applications map to the stream programming model can be found in other references: Khailany et al. discuss the mapping of a stereo depth extractor [Khailany *et al.*, 2001; Kanade *et al.*, 1996]; Owens et al. discuss the mapping of a polygon rendering pipeline [Owens *et al.*, 2000] and a REYES graphics pipeline [Owens *et al.*, 2002]; Rajagopal et al. discuss the mapping of a software defined radio [Rajagopal *et al.*, 2002]; Rajagopal also discusses the mapping of several other wireless receiver algorithms [Rajagopal, 2004]; the mapping of the FFT algorithm is discussed in [Kapasi *et al.*, 2002a]; and, finally, the mapping of a network packet processing IPv4 stack is described in [Rai, 2003].

2.2 Programmable Stream Processors

A programmable stream processor [Kapasi *et al.*, 2003; Rixner, 2001] is designed to directly execute the stream programming model we have just described. A stream processor can be implemented very efficiently by taking advantage of the locality and concurrency exposed by the stream model. Stream and kernel programs are compiled directly to such a stream processor. Figure 2.3 shows the architecture of a baseline programmable stream processor, which consists of an application processor, a stream register file, and stream clients. The SRF serves as a communication link by buffering data streams between clients, as long as the data does not exceed its storage capacity. The two stream clients in the baseline stream processor—a programmable kernel execution unit (KEU) and an off-chip DRAM interface—either consume data streams from the SRF or produce data streams to the SRF. The KEU executes kernels and provides local communication for operations within the kernels, while the off-chip DRAM interface provides access to global data storage. We will first present an ISA for this type of stream architecture, and then we will discuss the details of how we can build one efficiently in VLSI technology.

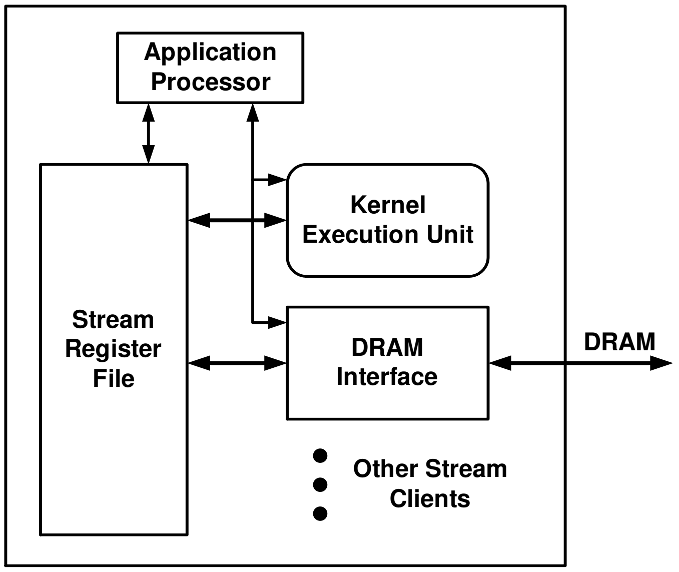


Figure 2.3: A programmable stream processor. The SRF streams data between clients—in this case the KEU, which is responsible for executing kernels and providing local communication for operations within the kernels, and the off-chip DRAM interface, which provides access to storage for global data.

2.2.1 Instruction Set Architecture (ISA)

The ISA of a stream processor is really composed of two unique ISAs: that of the application processor and that of the kernel execution unit. We shall describe both in turn, and then describe how they work together to execute a stream program.

Application Processor ISA

The application processor executes application code like that in Figure 2.1b. The application processor's RISC-like instruction set is augmented with stream-level instructions to control the flow of data streams through the system. The application processor sequences these instructions and issues them to the stream clients, including the DRAM interface and the KEU. The DRAM interface supports two stream-level instructions—LOAD_STREAM and STORE_STREAM—that transfer an entire stream between off-chip DRAM and the SRF. Additional DRAM interface arguments also can specify non-contiguous access patterns, such as non-unit strides and indirect access. The programmable KEU supports two stream-level instructions as well. The LOAD_KERNEL instruction loads a compiled kernel function into local instruction storage within the KEU. This typically occurs only the first time a kernel is executed; on subsequent kernel invocations, the code is already available in the KEU. The RUN_KERNEL instruction causes the KEU to start executing instructions that are encoded in its own instruction-set architecture, which is distinct from the application processor ISA.

Kernel Execution Unit ISA

Instructions in the KEU ISA control the functional units and register storage within the KEU, similar to typical RISC instructions. However, unlike a RISC ISA,

- KEU instructions do not have access to arbitrary memory locations, as all external data must be read from or written to streams, and
- special communication instructions explicitly handle data-dependencies between the computations of different output elements.

The first constraint preserves locality, and the programmer ensures the constraint is met when they structure the application as a set of kernels and streams. The communication instructions make it easier to exploit concurrency on data-parallel processors, without the need to reorganize data through the memory system. These two constraints of the kernel ISA maximize stream architecture performance.

Stream ISA

The ISA of a stream processor, or stream ISA, encapsulates both the application processor ISA and the KEU ISA. Accordingly, software tools translate high-level stream languages such as StreamC and KernelC directly to the ISA of a stream processor. For example, the *convert* kernel code shown in Figure 2.1a is compiled to the KEU ISA offline. Another compiler then compiles the application-level code shown in Figure 2.1b to the application processor ISA. The application processor executes this code and moves the precompiled kernel code to the KEU instruction memory during execution.

Assuming that the portion of the video feed the processor is encoding currently resides in DRAM memory, the three lines of application code in Figure 2.1b would result in the following sequence of operations:

```
(ii = Input_Image, lum = Luminance, chrom = Chrominance)

load_stream   ii_SRF_loc, video_DRAM_loc, LEN
add           video_DRAM_loc, video_DRAM_loc, LEN
run_kernel    convert_KEU_loc, ii_SRF_loc, LEN,
              lum_SRF_loc, chrom_SRF_loc
```

The first instruction loads a portion of the raw video feed into the SRF. The second instruction is a regular scalar operation that only updates local state in the application processor. The final instruction causes the KEU to start executing the operations for the *convert* kernel. Similar instructions are required to complete the rest of the MPEG-2 I-frame application pipeline. Interestingly, a traditional compiler can only perform manipulations on simple scalar operations that tend to benefit local performance only. A stream compiler, on the other hand, can manipulate instructions that operate on entire streams, potentially leading to greater performance benefits.

To be a good compiler target for high-level stream languages, the stream processor's ISA must express the original program's locality and concurrency. To this end, three distinct address spaces in the stream ISA support the three types of communication that the stream model uses. The ISA maps local communication within kernels to the address space of the local registers within the KEU, stream communication to the SRF address space, and global communication to the off-chip DRAM address space. To preserve the stream models locality and concurrency, the processors ISA also exploits the three types of parallelism. To exploit ILP, the processor provides multiple functional units within the KEU. Because a kernel applies the same function to all stream elements, the compiler can exploit data parallelism using loop unrolling and software pipelining on kernel code. The KEU can exploit additional data parallelism using parallel hardware. To exploit task parallelism, the processor supports multiple stream clients connected to the SRF.

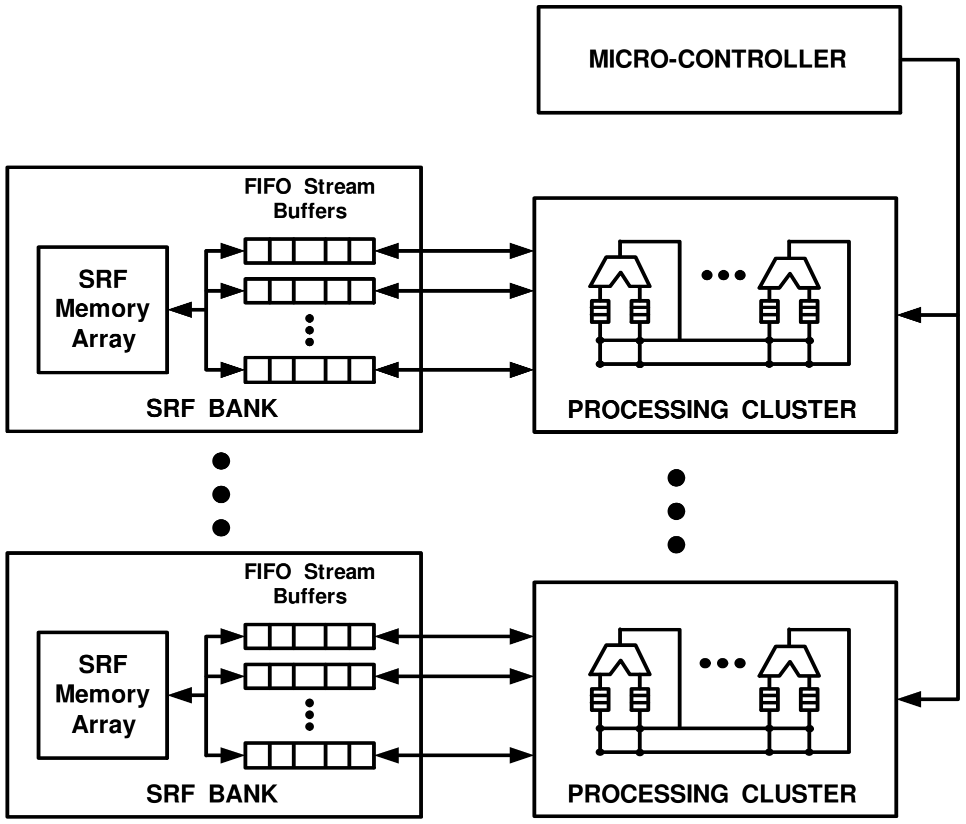


Figure 2.4: The architecture of an example kernel execution unit. There are several clusters organized in a SIMD fashion. Each cluster consists of five symmetric ALUs—i.e., they can all perform the exact same arithmetic operations. Also, there is a dedicated SRF bank for each processing cluster.

2.2.2 Micro-Architecture

Since we are focusing on kernel performance in this dissertation, it is worthwhile discussing the architecture of the KEU and SRF in slightly more detail. In particular, we will discuss several architectural optimizations we can apply that result in little to no performance loss since they exploit properties exposed by the stream model. Figure 2.4 shows the architecture of an example KEU.

Local Register Files (LRFs)

Each processing cluster is a VLIW engine that is built to take advantage of the ILP within kernel functions. All the ALUs are the same, and each ALU supports every floating point and fixed point arithmetic operation in the kernel ISA. Notice that our first optimization is to split the register files into an SRF and cluster LRFs. This relieves the LRFs from supporting the large capacity necessary for hiding memory latency. This maps well to the stream model, which explicitly captures the locality of all the intermediate values in a kernel. Also, each ALU has a register file dedicated to each of its inputs. The outputs of the ALUs are connected to the inputs of the register files by a fully connected switch. This *distributed* register file structure improves the VLSI implementation of a processing cluster by providing small two-ported register files, instead of a single large register file with ports for all the ALUs. [Rixner *et al.*, 2000]. Not shown are additional specialized functional units, that provide interfaces to modules such as a scratch-pad register file and the inter-cluster communication switch.

Multiple SIMD clusters

As shown in Figure 2.4, we have split the ALUs on a stream processor into multiple clusters, where each cluster operates on a different data element. When necessary, clusters can communicate with each other via an inter-cluster switch (not shown in figure). This might be required, for example, in order to obtain the data from neighboring clusters for a FIR filter. This cluster partitioning is done largely because a monolithic VLIW processing structure starts to become inefficient as we scale to higher numbers of ALUs [Khailany *et al.*, 2003]. Fortunately, the cluster partitioning doesn't significantly impact performance since data-parallelism is exposed by the stream model and is readily available in media processing applications. For example, Rixner previously reported that over a range of media processing kernels, an average speedup of $7.4\times$ was achieved on eight parallel processing clusters compared to a single cluster [Rixner *et al.*, 1998].

There are several options for how the clusters on a stream processor are controlled. The two extremes in the space are multiple-instruction multiple-data (MIMD) execution and single-instruction multiple-data (SIMD) execution. The KEU organizations for these

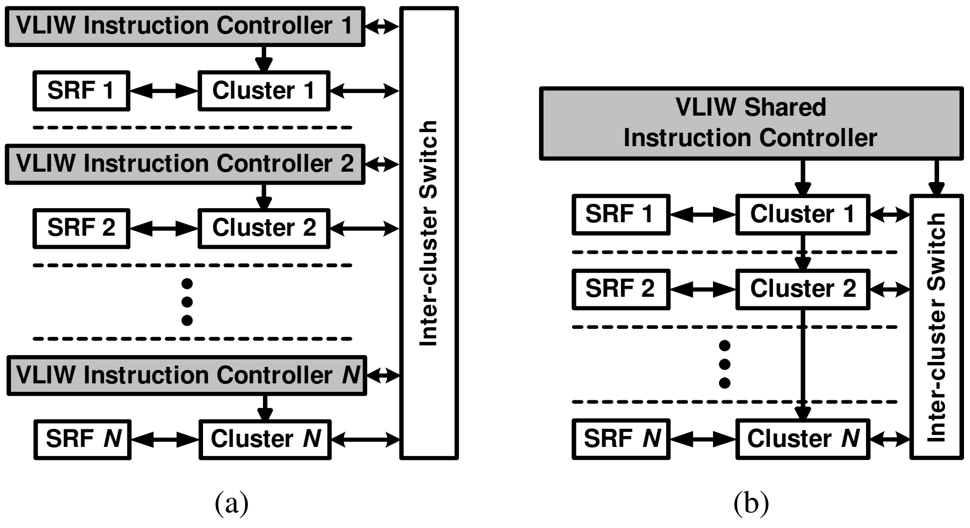


Figure 2.5: This figure shows two ways to organize the KEU in a stream architecture. The organization for the MIMD execution model is shown in (a), while the organization for the SIMD execution model is shown in (b). Other modules such as the application processor and memory system are not shown. In both organizations, each processing cluster operates on a different batch of stream elements in parallel. Notice that we are assuming that the SRF can be independently addressed by each cluster, even in the SIMD model.

two execution models are depicted graphically in Figure 2.5. In both models, every cluster executes a kernel on its own local batch of data. The differentiating feature is the independence of the VLIW instruction streams for each cluster. In the MIMD model every cluster can be executing a different instruction and a different kernel at any given time. In the SIMD model a single instruction controller issues exactly the same instructions to every processing cluster. With only a single controller, there is only one instruction stream and program counter (PC), and thus one control-flow. Therefore, the MIMD model is more flexible when each cluster wants to execute a different control-flow, a situation that arises due to kernel conditionals.

As shown in the figure, however, the MIMD model requires a separate VLIW instruction controller in return for this added flexibility. Furthermore, since large memory structures can only efficiently support a small set of independent ports, each kernel instruction memory can only serve a small set of clusters (most likely only one cluster). The impact of this extra hardware on the area of the KEU and SRF was evaluated using Khailany's analytical models [Khailany *et al.*, 2003]. For an eight cluster machine with a layout similar to the *IMAGINE* processor (Appendix A), the models indicate that the area of the KEU and SRF would be $1.6\times$ larger for a MIMD organization compared to a SIMD one. Not only does the MIMD option require more area, but it also increases the complexity in the design of the processor. For example, to support the MIMD model we would have to provide either multiple application processors (one per instruction controller), or burden a single application processor with both issuing operations to all the instruction controllers and synchronizing them.

Due to this area and complexity penalty, we would prefer to use a SIMD organization. Fortunately, this choice would not degrade the performance of any kernels that do not use conditionals. However, as we show in the next chapter, efficiently executing kernels with conditionals on SIMD organizations has been a challenge historically. Our new solutions will address this problem later in the thesis, thus allowing us to use the more area-efficient SIMD model without much performance loss across a large variety of kernels.

SRF Banking

As explained in [Rixner *et al.*, 2000], the architecture of the SRF incorporates two important optimizations: banking and stream buffer FIFOs. Banking ensures that each processing cluster connects only to its own local SRF bank in order to read and write stream elements. This is done in order to reduce the number of access ports necessary for the SRF memory array, which in turn reduces the number of internal communication paths in the SRF. This partitioning matches the data-parallelism in stream programming model well, since each cluster will operate on a different stream element. Therefore, during the execution of a kernel each cluster processes the set of elements that are in its own local SRF. Any sharing of stream elements between clusters, though, requires communication over the inter-cluster switch.

Stream Buffers

The second optimization, stream buffers, are added to further reduce the number of ports into the SRF memory arrays. The stream buffers are FIFOs that aggregate multiple accesses into one wide SRF memory array access. For example, consider an input stream, which transfers data from the SRF to the clusters. First the SRF will transfer a *wide* word, say 128 bits, from the SRF into the stream buffer FIFO. Then the processing cluster can read out each 32-bit word one by one from the stream buffer without requiring any more SRF memory array accesses. In this way, many stream buffers, each of which corresponds to a different logical stream in the kernel program, can time-multiplex a single port into the memory array. Like SRF banking, this also reduces the number of necessary internal communication paths in the SRF. This optimization also matches the stream programming model well, since kernels can only read or write streams in order, thereby providing sufficient spatial locality for the stream buffers to exploit.

In summary, the architectural optimizations made to the KEU and SRF enable more efficient hardware that occupies less silicon real-estate and operates with less power dissipation. At the same time, there is little degradation in kernel performance due to these hardware optimizations, since they match the properties of the stream programming model. For example, data-parallelism exposed by the stream model is exploited by banking the SRF and organizing the ALUs into many smaller VLIW engines (clusters), all controlled in SIMD. The stream model also explicitly captures fine-grained (kernel) and coarse-grained (stream) locality; the former allows us to use local register files in the KEU and the latter lets us use a software-managed SRF instead of a cache. Furthermore, by structuring external data accesses in kernels into sequential streams, the SRF implementation can be optimized via stream buffers. Further details on the VLSI-efficiency of stream processor architectures can be found in [Khailany *et al.*, 2001] and [Khailany, 2003].

2.3 Kernel Conditionals

As demonstrated in our discussion above, stream processors incorporate the main ideas from other explicitly parallel architectures, such as SIMD, vector, and VLIW machines.

Unfortunately, even though all of these architectures have traditionally excelled on regular data-parallel applications, the presence of conditionals can significantly degrade their performance. Since stream processors combine ideas from all these architectures, the problem is potentially even more serious for stream processors.

2.3.1 Motivating Example: The *geom_rast* Kernel

In order to introduce a real example that requires conditionals, we will use a kernel from a polygon rendering application. The kernel, *geom_rast*, implements polygon mesh assembly, vertex geometry and shading, triangle rasterization, and fragment shading. As seen in Kernel 2.1, *geom_rast* is typical in that it consists of an main-loop that reads an input from a stream on every iteration. Unlike other more regular kernels, however, there are several data-dependent conditionals embedded in the main-loop of *geom_rast*. By conditional statements, we are referring to if-statements, case-statements, and while-loops. In other words, we are referring to high-level conditional constructs, and not arbitrary conditional jumps.

The input stream is a sequence of points on a mesh. Each mesh starts with two points, and then every additional point is combined with two previous points to form a triangle that is on the surface of our model geometry. Since some points may be the first or second point in a new mesh, not every loop iteration will generate a valid triangle. The first if-statement in the kernel main-loop checks this condition. Further processing is only required if a valid triangle has been generated, otherwise the loop is restarted and a new mesh point is read. During the processing of a valid triangle, there are two more similar if-statements. One checks if the triangle is completely off the viewable screen, and one checks if the triangle is forward facing (and, hence, viewable). If a triangle meets these criteria, then a while-loop is executed that generates a *fragment* for every pixel on the screen that is covered by the current triangle. This operation is called rasterization. Note that the number of iterations of the while loop that are executed for each triangle will vary with the size of the triangle. The particular rasterization algorithm we used however does not necessarily produce a valid fragment on each loop-iteration, and therefore there is a final if-statement within the while-loop. All valid fragments for a triangle are written to the output stream.

Kernel 2.1 Pseudo-code for the *geom_rast* kernel from the RENDER application. This kernel contains five different conditional statements: four if-statements and one while-loop. Note that all code outside the main-loop of the kernel is not shown. The horizontal lines represent basic block boundaries, and the annotations represent the name of each basic block.

```
loop over $vtx\_stream$ begin           ------------------------------------------------
  $v \overset{\text{pop}}{\longleftarrow} vtx\_stream$;          // input: 8 words
  $v \leftarrow vertex\_shader(v)$;
  $tri, mesh\_state, valid\_tri \leftarrow assemble\_mesh(v, mesh\_state)$;           mesh
  if $valid\_tri$ then                  ------------------------------------------------
    $clipped \leftarrow check\_clipping(tri)$;                                        clip
    if $\neg clipped$ then              ------------------------------------------------
      $tri \leftarrow viewport\_transform(tri)$;
      $interpolants \leftarrow generate\_interp(tri)$;
      $culled \leftarrow check\_backfacing(tri)$;                                 viewport
      if $\neg culled$ then             ------------------------------------------------
        $rast\_tri \leftarrow raster\_prep(tri)$;
        $frags\_left \leftarrow$ true;                                          rasterprep
        while $frags\_left$ do          ------------------------------------------------
          $xyfrag, rast\_tri, frags\_left \leftarrow next\_pixel(rast\_tri)$;
          $valid\_frag \leftarrow inside\_tri(rast\_tri, xyfrag)$;                updatexy
          if $valid\_frag$ then         ------------------------------------------------
            $bary\_coords \leftarrow baryprep(xyfrag)$;
            $new\_frag \leftarrow apply\_interps(xyfrag, interpolants, bary\_coords)$;
            $out\_frag \leftarrow fragment\_shader(new\_frag)$;
            $fragment\_stream \overset{\text{push}}{\longleftarrow} out\_frag$;     // output: 6 words
                                                                               genfrag
          fi                            ------------------------------------------------
                                                                      check_frags_left
        od                              ------------------------------------------------
      fi
    fi
  fi                                    ------------------------------------------------
                                                                     check_stream_empty
end                                     ------------------------------------------------
```

2.3.2 Challenges

Fundamentally, conditionals in the main-loop of a kernel do not break the data-parallel nature of the programming model, as demonstrated by the *geom_rast* kernel. So, if stream processors are built to exploit this data-parallelism very efficiently, why would conditionals reduce performance so much? The reason is that, even though elements can be operated on in parallel, the kernel will execute a different set of operations, and hence follow a different control-flow path, for each element. This creates three challenges, which we discuss below.

1. *Conditionals result in poor VLIW schedules (i.e., low achieved ILP).*

   Data-dependent branches represent barriers to code motion when statically scheduling a kernel. For this reason, fewer operations can be combined together at compile time in order to obtain higher ILP. For example, in *geom_rast* the operations in the `CLIP` basic block cannot be combined with those in the `MESH` block since it is not guaranteed that `CLIP` will execute every time `MESH` executes. Worse, usually the VLIW schedules are efficiently packed using standard optimizations that convert data-parallelism to ILP, like software-pipelining and loop-unrolling. Because of the branches, however, we will see that these optimizations don't work well. The result is low achieved ILP, which in turn results in performance that is a small fraction of the peak. This problem is especially critical since stream processing clusters are built to support a peak of 10–20 operations in flight concurrently (2–4 pipelined operations in each of five parallel ALUs, for example).

2. *Execution on SIMD hardware will require* all *clusters to execute the* union *of the control-flow paths required by each cluster.*

   Consider the example of a while-loop within the main loop of the *geom_rast* kernel. Assume that the current triangle in one cluster requires 10 iterations of the while-loop, whereas the triangles in the other clusters only require one or two loop iterations. Unfortunately, all of the clusters will have to execute the loop 10 times in this case because of the SIMD execution model. This directly reduces the efficiency of the architecture since several clusters are wasting valuable ALU issue opportunities while waiting for another cluster to finish processing its triangle.

3. *Conditionals can result in load-imbalance between multiple processing clusters.*

   Consider what happens when the code that writes data to an output stream in a kernel is inside of an if-statement (as is the case with `fragment_stream` in *geom_rast*). Upon completion of this kernel, each cluster may have output a different number of elements to their local SRF bank. The next kernel in the application pipeline that processes this output stream will suffer from load-imbalance because each cluster will have a different amount of input data to process. This causes the cluster with few elements to process to waste otherwise useful ALU issue slots while waiting for the cluster with the most work to finish.

   Notice that even if we solved the SIMD problem (the second problem described above), the *geom_rast* kernel itself can still suffer from load-imbalance. For example, even though each cluster starts with the same number of inputs in their SRF banks, since each cluster will execute a different set of operations for each element, some clusters can finish earlier than other clusters, causing the early clusters to idle. This reduces the efficiency of the architecture as well.

Historically, highly parallel machines have not fared well on irregular applications containing many conditional statements. At first glance, the same holds true for stream processors, and in fact the problem might be even more pressing since stream processors aggregate so many parallel ALUs on a single chip. To guide our exploration of the problem and its solutions, we illustrated three sources of inefficiencies that arise when we broaden our application scope and start executing irregular applications on data-parallel stream processors. We did make the observation, however, that conditionals do not break the data-parallel nature of kernels, they just cause each cluster to execute different sets of operations. So we must look into ways that make the computation more regular between the clusters. The remainder of the thesis will present and evaluate new solutions for improving the performance of conditionals on stream processors.

Chapter 3

Improving VLIW Schedules with Conditional Routing

The first major challenge we identified for kernels with conditionals is to achieve a good VLIW schedule, by making sure that all ALUs are busy on each cycle. Let's assume that we have a processing cluster with five ALUs, where each ALU is the same and can perform all arithmetic operations. Furthermore, assume that the ALUs are pipelined such that it takes four cycles to execute a floating point addition or multiply, and that a new operation can be started on each cycle. Then, there can be up to 20 operations executing in parallel on such a cluster. The ILP in a kernel must be exploited in order to ensure that there are 20 useful operations in flight at any given time in the final VLIW schedule. However, this is beyond the ILP inherent in most media processing kernels. Therefore, generally, software-pipelining and/or loop unrolling are used to convert available data-parallelism into available ILP. This allows most media processing kernels to come within 80%–90% of the peak possible performance of such a VLIW processing cluster, *if there are no branches in the kernel.*

3.1 Motivation

As soon as we introduce conditionals into the main loop of a kernel, basic software-pipelining and loop-unrolling breaks down. These techniques rely on the assumption that each stream element (and hence each iteration of the main-loop of a kernel) will execute the exact same sequence of operations. This assumption is broken when you add a conditional into the main-loop, and thus we cannot apply these techniques in the normal manner any longer. This results in pretty poor overall performance. To illustrate the extent of the problem, we will take the example of the *geom_rast* kernel, introduced in Section 2.3.1. Consider compiling this kernel to a simple processing cluster with only one unpipelined ALU. Since this cluster can only support a single operation in flight at any time, the available ILP in the *geom_rast* kernel will certainly ensure that almost all of the ALU issue slots are occupied. Now, consider the speedup of the kernel when going from this simple cluster to the one with five pipelined ALUs. For *geom_rast*, without any further optimizations on the beefier cluster the speedup is only $2.9\times$, compared to a peak possible speedup of $20\times$. Clearly, there is a dire need for improving the performance on the larger cluster since the base kernel schedule only utilizes less than 15% of the available ALU resources, whereas up to 90% is typical of kernels without conditionals.

3.1.1 Existing Techniques: If-statements

There are existing methods for improving this performance, but they have limitations. Let's consider if-statements now, and we will discuss while-loops below. There are two types of techniques for if-statements. First, there are those that extend the data-parallel techniques, such as software-pipelining and loop-unrolling, to be compatible with if-statements. Second, there are *speculative* techniques that improve ILP by combining blocks based on guesses at compile time about which control-flow paths are the most common.

Data-parallel techniques

Let's first discuss the techniques that take advantage of data-parallelism and convert it to instruction-level parallelism. Loop-unrolling normally works by duplicating the body of the main-loop in a kernel. If there is data-parallelism, the operations for one iteration (or stream element) will not be data-dependent on the operations from the other iteration, and hence the available ILP will increase. However, all the operations in the body of the main-loop cannot be lumped together when conditionals are present, since each iteration may execute a different subset of the operations. Only the basic blocks that deterministically execute every iteration can be “unrolled” into a larger basic block. This is obviously not very fruitful if most of the operations are within conditional statements. Incidentally, this is the case for the *geom_rast* kernel of Kernel 2.1, since only the `MESH` basic block is deterministically executed every iteration.

Software pipelining also takes advantage of data-parallelism, by overlapping different portions of the same loop body with each other in order to increase the available ILP. Each such overlapping portion is called a *stage* of the pipeline. The concept can be extended to handle conditionals as well. When a conditional block is present, say an if-then-else statement, we have to generate two versions of the loop, one that overlaps the then-clause with the rest of the stages in the pipeline and one that overlaps the else-clause with the rest of the stages in the pipeline. This technique will work well for small conditionals, and is the approach taken by several previous researchers [Ebcioglu and Nakatani, 1990; Lam, 1988; Warter *et al.*, 1992; Stoodley and Lee, 1996]. However, the main problem with this technique is that the number of different versions of the loop that are generated can be large. This problem is referred to as *code explosion*. To see how the problem arises, consider what happens when the number of operations in one or both clauses of the if-statement gets large. In that case the operations from one of these clauses might span two or more software-pipeline stages. However, for each extra stage spanned, the number of stream elements that are concurrently executing the if-statement increases. Thus, the number of versions of the loop that have to be scheduled multiplies for each extra stage spanned, because we have to account for every possible simultaneous combination of path choices for the total number of stream elements in the if-statement at any given time. For similar reasons, code explosion also becomes a problem as the nesting of conditionals increases. Code explosion can be problematic for stream architectures, since they have a limited amount of instruction storage in the micro-controller, and for conventional architectures that rely on a fixed-size cache for providing sustained instruction bandwidth.

In order to get around the issues with loop-unrolling and software-pipelining when conditionals are present, we will instead turn to speculative techniques in order to increase ILP.

Speculative techniques: COMMON

The basic idea is to improve ALU utilization by *speculatively* executing operations. Speculation potentially increases the amount of ILP in a basic block by adding operations from future basic blocks into the current basic block, assuming at compile time that the branches delimiting the basic blocks will evaluate in a certain way. If some of the operations in the future basic block are not dependent on results from the current basic block, they can be scheduled in parallel with the existing operations and the ILP will increase.

We will classify the two types of techniques that use speculation as COMMON and PREDICATION. They differ in how they decide which basic blocks to combine, and in how they are implemented. COMMON (often referred to simply as *speculation* in the literature) attempts to find the most commonly taken control-flow path through the code. All basic blocks in this path are combined into a single block at compile time, potentially increasing the available ILP in that block. Of course, at execution time, we wouldn't know until the end of the speculative block whether or not the current stream element actually wanted to take the common path we optimized for. If the current stream element wanted to take a different control-flow path, then certain results will have to be discarded, and perhaps the computation may have to be restarted at an earlier point. The compiler must insert extra code, commonly called *fix-up code*, for this task.

A good example of an existing compiler technology that employs speculative techniques similar to COMMON is *trace scheduling*. Fisher introduced trace scheduling [Fisher, 1981], and later Ellis implemented it in a compiler [Ellis, 1986]. This technique develops traces that span multiple branches and basic blocks, and optimizes each trace. Traces are chosen for paths that are most commonly executed, and fix-up code is inserted into the other paths to account for mispredictions. A more recent example of a compiler algorithm that uses speculation is *superblock scheduling* [Hwu *et al.*, 1993]. Two things reduce the utility of speculation when elements start to stray from the common path. First, the speculative code was unnecessarily executed, and took up valuable ALU issue slots. Second, executing the fix-up code can be quite expensive, since the compiler generally focuses all effort on optimizing the common path and not the mispredict path.

Speculative techniques: PREDICATION

The PREDICATION technique also executes basic blocks speculatively, however it uses a different selection criteria to choose which blocks to speculate. First, PREDICATION speculatively executes basic blocks that are not necessarily on the most frequently executed control-flow path.. This choice is beneficial, for instance, if the block only contains a small number of operations, since the cost of possibly unnecessarily executing the small basic block is usually minimal. The second difference between COMMON and PREDICATION is that PREDICATION will speculatively execute multiple paths of control in the same block, and then only select results from a single control-flow path at the end of the block.

In order to implement predication, special hardware ensures that only the results from the correct control-flow path are retained. This can be implemented, for instance, by providing a boolean parameter to each operation that indicates whether or not the results from that operation should be stored. The boolean conditions are calculated using the same tests as the if-statements would have used. However, instead of using them as inputs to a conditional branch instruction they are used as predicates for the operations that would have been within the body of the if-statements. An example of a VLIW architecture that supports predication in order to improve achieved ILP is the IA-64 architecture [Dulong, 1998]. Also, previous research has attacked the problem of automatically converting the if-statement conditions into predicates, and has resulted in a compiler algorithm called *if-conversion* [Towle, 1976; Allen *et al.*, 1983]. A more recent compiler algorithm that uses predication is Mahlke *et al.*'s hyperblock scheduling [Mahlke *et al.*, 1992]. The formation of a hyperblock is similar to that of a normal trace, except it also incorporates predication in order to reduce the number of paths and exception cases. The researchers have found that judicious use of this technique yields good speedups on architectures that support predication when many small conditional clauses are present in the main-loop.

However, while PREDICATION avoids the performance penalty due to fix-up code, it still incurs the cost of executing code from extraneous control-flow paths. As a result, conditional clauses that have many operations can be problematic. Furthermore, as the number of control-flow paths in the kernel increase (due to nested conditionals, for example), the utility of predication starts to decrease.

So, the same problem remains for both COMMON and PREDICATION—at some point we have to execute unnecessary operations. This might be due to mispeculation or because we used predication to simultaneously follow multiple control-flow paths.

Combined techniques

In any case, even when you can very accurately predict the common control-flow path, these speculative techniques only help marginally on their own. They can certainly improve the ILP in a kernel main-loop, but can't guarantee to continue increasing the ILP until all ALU slots are busy. Instead, the most powerful existing techniques are those that combine the data-parallel techniques with speculative ones. For example, speculation can be combined with software-pipelining to generate a well pipelined schedule for the most common control-flow path through the loop, and when less frequent paths are required, control can jump out of the loop and handle the fix-up code. This tactic eliminates code-explosion and packs the ALUs efficiently for the most common path, and is typified in the Multiple-II Modulo Scheduling algorithm [Warter-Perez and Partamnian, 1995]. Another possible approach is to use predication in order to remove all the conditionals in a loop, after which the whole loop can be easily software-pipelined. This approach was first used by the Cydra 5 compiler [Rau *et al.*, 1989; Dehnert and Bratt, 1989]. Yet another option is to combine speculation and loop-unrolling, by assuming that successive loop-iterations will follow the same path. Instead of continuing to list all the possible combinations here, we will leave further discussion to Section 3.2, where the performance of these combined techniques will be compared to our new technique, conditional routing. We will see that even though these combined techniques use data-parallelism to improve the achieved ILP, they do not decrease the amount of unnecessary operations executed for extraneous control-flow paths. In contrast, we will show that conditional routing is able to use data-parallelism without having to execute operations from unnecessary control-flow paths.

3.1.2 Existing Techniques: While-loops

We will shift our focus now to the other type of conditional structure: a while-loop. It turns out that for a while-loop, an effective way to improve the achieved ILP is to use loop-unrolling to pack the operations in the deterministic blocks and then to use software-pipelining to optimize the *inner* while-loop. Unfortunately, this doesn't always help. Take the *geom_rast* kernel of Kernel 2.1 as an example again. First of all, only a small fraction of the operations are in basic blocks that always execute every iteration, rendering loop-unrolling ineffective. There are three if-statements outside the while-loop, which means more advanced techniques will have to optimize those basic blocks. In any case, even if we can optimize all the operations outside the while-loop well, software-pipelining will not help the while-loop itself much because of two reasons. First, the resulting schedule is constrained by a long loop-carried dependency. Second, the inner while-loop itself contains an if-statement!

Another possibility for handling this loop is to increase ILP by executing the while-loop for two stream elements at the same time. Since the two elements will not necessarily execute the same number of loop iterations, there will be times that we execute more operations than necessary. As before, this is not ideal, and we would like to come up with a technique that packs the ALUs, but only does so with useful operations.

Let's look at *geom_rast* once more in order to summarize the main point of this section, which is that the existing conditional techniques for increasing ILP are insufficient. The best speedup we can achieve for *geom_rast* after applying existing techniques for both if-statements and while-loops is $1.8\times$.[1] This brings us from 15% of peak ALU utilization to only 26%. Clearly we still have much room for improvement. The rest of the chapter will focus on a new technique, conditional routing, that can provide higher speedups.

3.2 Applying Conditional Routing to If-Statements

Conditional routing is a transformation that reorders the computation in a kernel containing an if-statement by splitting it into multiple kernels. The new kernels contain no conditional branches, which results in both good VLIW schedules and efficient execution on SIMD organizations. This section focuses on schedule improvements on a VLIW cluster for kernels

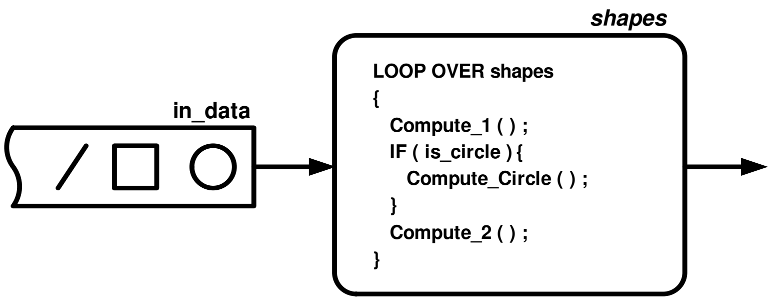
[1]These results were obtained by applying the existing techniques manually. Certainly, we might achieve a slight improvement by using a compiler to apply these techniques more optimally. However, the same improvement from using a complier can also be achieved for the new techniques we will introduce.

with if-statements, and the next section, Section 3.3, will do the same for while-loops. We will see, however, that the new kernels generated by the conditional routing optimization buffer intermediate results in the SRF. Thus, we must consider the impact of this increased SRF communication on overall performance, and whether reordering the work in the kernels also reorders the data elements in the intermediate and final streams.

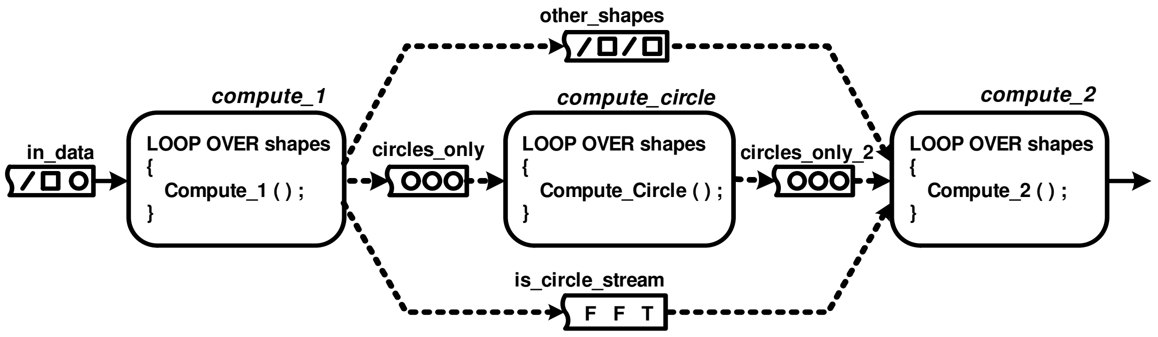
While stream processors are normally built with a set of SIMD processing clusters, we are only considering the performance of one VLIW cluster throughout this chapter. We do this in order to isolate the ILP benefits offered by conditional routing. Later, in Chapter 4, we will add the extra dimension of multiple clusters controlled in SIMD. Essentially, we are optimizing the individual unit of a cluster now, and in later chapters we will optimize how several clusters operate together.

Figure 3.1(a) shows the original version of an example kernel, *shapes*, that contains a conditional if-statement. This kernel processes a stream of shapes (lines, squares, and circles) and executes the pseudo-code shown for each stream element, completing the processing for one stream element before moving on to the next. In particular, for each stream element: the operations in Compute_1( ) will be executed; then, only if the stream element is a circle, the operations in Compute_Circle( ) will be executed; and, finally, the operations in Compute_2( ) will be executed, regardless of the shape of the stream element. This series of steps is repeated for each stream element.

The result of applying the conditional routing transformation to this example is shown in part (b) of Figure 3.1. The first thing to notice is that the code within the conditional statement has been separated out into its own kernel. Doing this results in three new kernels: *compute_1*, *compute_circle*, and *compute_2*. Their names suggest which set of operations from the original kernel are in each new kernel. With this new set of kernels, the operations in *compute_1* are applied to *every* element in the input stream, before the remaining kernels are applied to any stream elements. Note that this is different from the order which the original kernel executed its operations. This reordering of work is only valid if the dependencies from element to element in the original kernel satisfy certain constraints. In particular, there should be no loop-carried dependencies in the original kernel from Compute_Circle() to Compute_1( ), or from Compute_2( ) to either of the other two functions.



(a)


other_shapes
compute_1
compute_circle
compute_2
in_data
LOOP OVER shapes { Compute_1 ( ) ; }
circles_only
LOOP OVER shapes { Compute_Circle ( ) ; }
circles_only_2
LOOP OVER shapes { Compute_2 ( ) ; }
is_circle_stream
F F T

(b)

Figure 3.1: Applying conditional routing to a kernel containing an if-statement. The original kernel is shown in (a), while (b) shows the result of applying conditional routing.

```
/* Original version */
shapes(in_data, out_data);
```

(a)

```
/* With conditional routing */
compute_1(in_data, other_shapes, circles_only, is_circle_stream);
compute_circle(circles_only, circles_only_2);
compute_2(other_shapes, circles_only_2, is_circle_stream, out_data);
```

(b)

Figure 3.2: Application-level code for (a) the original kernel and for (b) the new set of kernels that use conditional routing to implement the if-statement.

The output streams of *compute_1* are then passed to the remaining kernels. In order to implement the semantics of the original conditional statement, the outputs of *compute_1* are split into two streams of shapes: one contains only the circles and the other contains only the other shapes. Furthermore, there is a third output stream, `is_circle_stream` that keeps track of which of the two new streams each element of the original stream went to. This third stream will be useful for combining the two split streams into one later on. At this point, the *compute_circle* kernel operates on the elements of the stream with only the circles. Finally, the *compute_2* kernel operates on both the stream containing the non-circles that was generated by the first kernel, as well as on the circles output by *compute_circle*. All stream elements are output by *compute_2* into a single stream. `is_circle_stream` is used by *compute_2* to recreate the original ordering of the stream elements. The case values dictate which input stream the *compute_2* kernel should read from to get the next input element. With the use of this case stream, the *compute_2* kernel can ensure that it inserts results into `out_data` in the same order as the original kernel in part (a) of Figure 3.1 would have. The application-level pseudo-code in Figure 3.2 illustrates the sequence of kernel calls needed for conditional routing, and also shows the kernels' inputs and outputs.

Kernel 3.1 Pseudo-code for the *compute_1* kernel showing the usage of the conditional routing output primitive ("pushif") for stream writes. Note that the stream names used inside the kernel coincide with the names of the stream arguments in the kernel call in Figure 3.2 only for clarity. These two sets of names can be completely different, and in actual code the arguments and the variables in the kernel will be matched via their order in the kernel header and their order in the kernel call, just as with a function call in the C language.

```
loop over $in\_data$ begin
    $In\_Shape \xleftarrow{\text{pop}} in\_data;$
    $Shape, is\_circle \leftarrow Compute\_1(In\_Shape);$

    /* The data structure Shape is only written to   */
    /* circles_only if is_circle is TRUE;            */
    /* otherwise circles_only is unchanged           */
    $circles\_only(is\_circle) \xleftarrow{\text{pushif}} Shape;$
    $other\_shapes(\neg is\_circle) \xleftarrow{\text{pushif}} Shape;$
    $is\_circle\_stream \xleftarrow{\text{push}} is\_circle;$
end
```

In order to implement the output routing for *compute_1* and the input routing for *compute_2*, we employ two *conditional routing primitives*. They each allow conditional access to a stream based on a boolean case value. To illustrate how they are used, see Kernel 3.1 for the pseudo-code for *compute_1*. In this kernel, the "pushif" operation is a conditional routing primitive. Essentially, it will only append the `Shape` record to the stream if `is_circle` is true. This is implemented very easily in hardware as one atomic operation, and hence does not require any conditional control-flow (i.e., branches) to implement. It is scheduled like any other stream access, and thus has minimal impact on performance.[2] The idea can easily be extended for the *compute_2* kernel (Kernel 3.2), which requires the second conditional routing primitive: "popif," which only removes the head element from a stream and only updates the target record variable if the specified boolean condition is true. No conditional routing primitives were required by the *compute_circle* kernel.

We have replaced data-dependent control-flow with data-dependent communication.

[2]Performance impact is minimal unless, as we show in Section 3.2.3, the record size of `Shape` is so large that scheduling the SRF accesses, and not the arithmetic operations, limits the performance of the kernel.

Kernel 3.2 Pseudo-code for the *compute_2* kernel showing the usage of the conditional routing input primitive ("popif") for stream reads.

```
**loop over** $\mathit{is\_circle\_stream}$ **begin**
  $\mathit{is\_circle} \xleftarrow{\text{pop}} \mathit{is\_circle\_stream}$;

  /* The data structure In_Shape is          */
  /* only updated if the case condition      */
  /* is TRUE; otherwise it is unchanged.     */
  $\mathit{In\_Shape} \xleftarrow{\text{popif}} \mathit{circles\_only\_2}(\mathit{is\_circle})$;
  $\mathit{In\_Shape} \xleftarrow{\text{popif}} \mathit{other\_shapes}(\neg \mathit{is\_circle})$;

  $\mathit{Shape} \leftarrow \mathit{Compute\_2}(\mathit{In\_Shape})$;

  $\mathit{out\_data} \xleftarrow{\text{push}} \mathit{Shape}$;
**end**
```

Furthermore, since the new communication is implemented with conditional routing primitives, and not with branches, there is no conditional control-flow in the kernel main-loops anymore. Thus we can efficiently apply software-pipelining and loop-unrolling to increase the ILP of each kernel. However, the important point about conditional routing is that it only executes as many operations as is strictly required. On the other hand, predication wastes valuable ALU slots because it executes multiple alternate paths anmd only keeps the result from one path. Likewise, speculation wastes ALU slots whenever a stream element strays from the common control-flow path. As we will see in the next section, this difference will determine the relative performance of the techniques.

3.2.1 Performance

The previous example demonstrated the conditional routing technique and qualitatively discussed its impact on performance. We will now quantitatively analyze the performance of conditional routing. However, before we present the data, we will briefly discuss the evaluation methodology and the software tools we used to obtain the data in this and future chapters.

Stream Processing Tools and Evaluation Methodology

In order to evaluate the performance of our proposed techniques on the stream processing architecture we have been discussing so far, we will use the compilation and simulation tools originally developed for the *IMAGINE* stream processor (Appendix A). All of the *IMAGINE* tools are configurable via a common machine description (MD) file. This MD file allows a developer to adjust the following properties (among others): the number of SIMD processing clusters, the number of ALUs within a cluster, the mix of instructions supported by each ALU, the size of the SRF, and the bandwidth between the SRF and processing clusters. Therefore, even though the specific cluster and KEU architectures we will evaluate in this thesis are slightly different from the *IMAGINE* architecture, the same tools will seamlessly support our experiments through the use of the MD file.

Since we are focussing on kernel performance in this thesis, the results are obtained by simulators that report performance based only on the lengths of the basic block schedules output by the VLIW kernel scheduler. This is a reasonable approach since the performance of kernels on a stream processor is very close to the predicted static schedule, since there are very few variable latency operations. In particular, unpredictable DRAM access times are avoided since kernels cannot access DRAM directly. There are two sources of inaccuracy however, with this approach.

First, SRF accesses can stall since a stream buffer may be full or empty, but these types of stalls do not occur often. For example, one study reports that across a set of media applications, kernel stalls accounted for less than 5% of application execution time [Ahn *et al.*, 2003]. The second inaccuracy is that using kernel loop performance doesn't necessarily translate to application performance. In general, memory and application processor bottlenecks may impact performance. We will largely ignore these in this dissertation, however, since our goal is to make kernel execution faster, regardless of how that kernel fits into the greater application.

Finally since we are using the schedule lengths generated by the VLIW kernel scheduler in order to evaluate the performance of conditionals in this thesis, we should mention what the capabilities of the scheduler are in this regard. First of all, our scheduler can only handle one conditional structure: a while-loop. All other conditional structures are built out of this. For example, an if-statement is a while-loop that may execute only zero or one times. This is slightly less efficient that supporting an if-statement directly, but does not affect the final schedule very much, especially for compute-intensive kernels. Furthermore, the current kernel scheduler does not automatically apply the optimizations we discuss in this thesis. The only exceptions are that the scheduler can automatically software-pipeline or unroll a loop—only if there are no conditionals nested in the loop. One of the goals of this thesis, however, is to show that the techniques we introduce have enough utility that they warrant further work to incorporate their application into a compiler.

If-Statement Synthetic Benchmark

We have used the methodology and tools we just described in order to do a more quantitative analysis of the performance of conditional routing. Since we are relying on manual conversion of code in order to implement the various conditional techniques, we have run most of our experiments using a synthetic benchmark, *synthetic_if*, shown in Figure 3.3. This benchmark, and the others through the thesis, attempt to extract the basic building blocks of conditional-statements. This allows us to run various experiments without changing large and complicated kernels by hand. The benchmark is similar in structure to the example we studied above, and the pseudo-code is shown in Kernel 3.3. There is one if-statement in the kernel, which splits the main loop of the kernel into three basic blocks: `INPUT` (includes the operations in TEST( )), `BODY`, and `OUTPUT`. The code within each of these basic blocks is synthetically generated, and is guaranteed to satisfy certain properties. In particular, there are no loop-carried dependencies from a basic block to any basic block above it (i.e., from `BODY` to `INPUT` or from `OUTPUT` to either of the other two blocks). Also, the length of the VLIW schedule for each basic block is limited by the critical path formed by dependencies between the operations, and not by the number of ALUs in the VLIW cluster (which is five for our target architecture).

The parameters of the benchmark that we will vary at first are the number of operations in the basic block `BODY` (W_{body}), the fraction of stream elements for which `BODY` is executed (p), and the amount of live state that needs to be passed to and from each basic block (S). We will assume that branch decisions are independent from one stream element to the next. Also, unless otherwise stated, the results assume infinitely long streams and

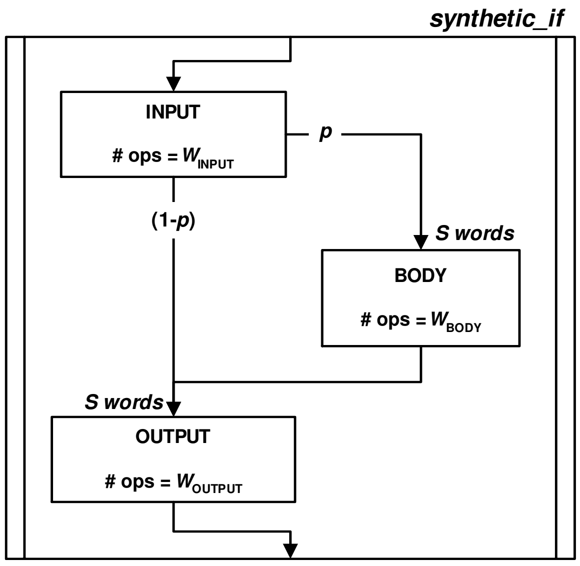


Figure 3.3: Basic block diagram of the *synthetic_if* benchmark. When illustrating basic block diagrams for kernels in this dissertation, only the flow for the main loop will be shown. Furthermore, the backward branch for the main loop is implicit and will not be indicated on the diagram. Finally, boxes with two vertical lines on either side represent the main loops of kernels, and boxes with single lines represent basic blocks. The parameters of the benchmark that we will vary throughout this chapter are illustrated: p is the probability that the if-statement evaluates to TRUE; W_{XYZ} is the number of arithmetic operations in the basic block XYZ; and S is the number of words of internal state that must be passed from one basic block to the next.

Kernel 3.3 Pseudo-code for the *synthetic_if* benchmark. The simple kernel has a single conditional block. The variables whose first letter is capitalized are multi-word structures. Furthermore, all functions are in-lined into the kernel, and do not update their input parameters. The horizontal lines delineate basic block boundaries.

```
loop over in_stream begin                    --------------------------------
    $In\_state \overset{\text{pop}}{\longleftarrow} in\_stream$;
    $State \leftarrow INPUT(In\_state)$;
    $cond \leftarrow TEST(State)$;
                                                                      INPUT
    if cond then                             --------------------------------
        $State \leftarrow BODY(State)$;                                BODY
    fi                                       --------------------------------

    $Out\_state \leftarrow OUTPUT(State)$;
    $out\_stream \overset{\text{push}}{\longleftarrow} Out\_state$;          OUTPUT
end                                          --------------------------------
```

hence only depend on the schedule produced for code in the main loop of the kernel. We will relax this assumption later in the chapter to study the effect of non-main-loop code and software-pipelining setup and tear-down.

Conditional Routing Performance

Conditional routing is applied to the *synthetic_if* kernel in exactly the same way as the example in the previous section, yielding three kernels, one for each basic block. Figure 3.4 shows the impact of conditional routing on this benchmark. The graph shows the execution times for $W_{\text{body}} = 32$ operations, $W_{\text{input}} = W_{\text{output}} = 17$ operations, and $S = 2$ words.

The original kernel schedule is between 300% to 340% longer than the ideal schedule execution time over the range $p = \{0 \ldots 1\}$. The number of cycles to execute IDEAL is generated by simply dividing the number of arithmetic operations by the number of ALUs. ORIGINAL is so much slower than IDEAL because the schedules for the three basic blocks are limited by the critical path of each block, since there is not enough ILP to pack the VLIW schedules for the five-ALU cluster. Applying conditional routing improves the schedule because the three new kernels can all be software pipelined efficiently, since

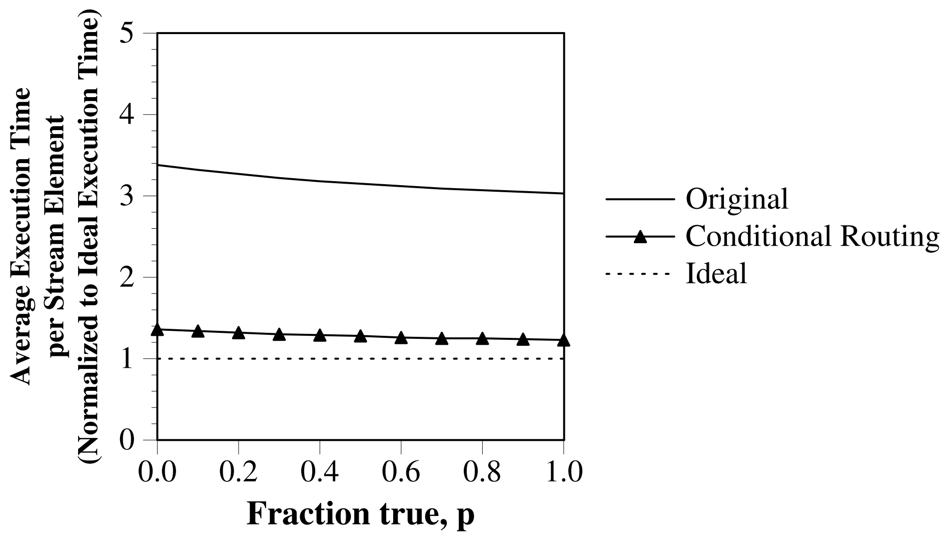


Figure 3.4: Impact of conditional routing on execution time for *synthetic_if*. The execution times are shown normalized to IDEAL. $W_{\text{body}} = 32$ operations; $W_{\text{input}} = W_{\text{output}} = 17$ operations; $S = 2$ words.

none of them contain any conditional control-flow. In fact, the effective execution time per stream element is within 16%–21% of the ideal. This relatively small overhead is due to the extra operations that are required to implement the routing of the stream elements between the kernels and to ensure that the order of the final output stream is correct.

3.2.2 Comparison to Other Techniques

While the previous section shows that conditional routing demonstrates good absolute performance over a range of values of p, we must still compare its performance to existing techniques. As we discussed in Section 3.1.1, the best existing techniques for handling if-statements combine speculation and predication with data-parallel techniques, such as loop-unrolling and software-pipelining. This section will demonstrate that conditional routing is the only technique that works well over the whole range of values of p. Furthermore, we will show that the magnitude of the advantage of conditional routing can be even more dramatic when we consider more complicated benchmarks (*synthetic_if_else* and *synthetic_case*, Section 3.2.2).

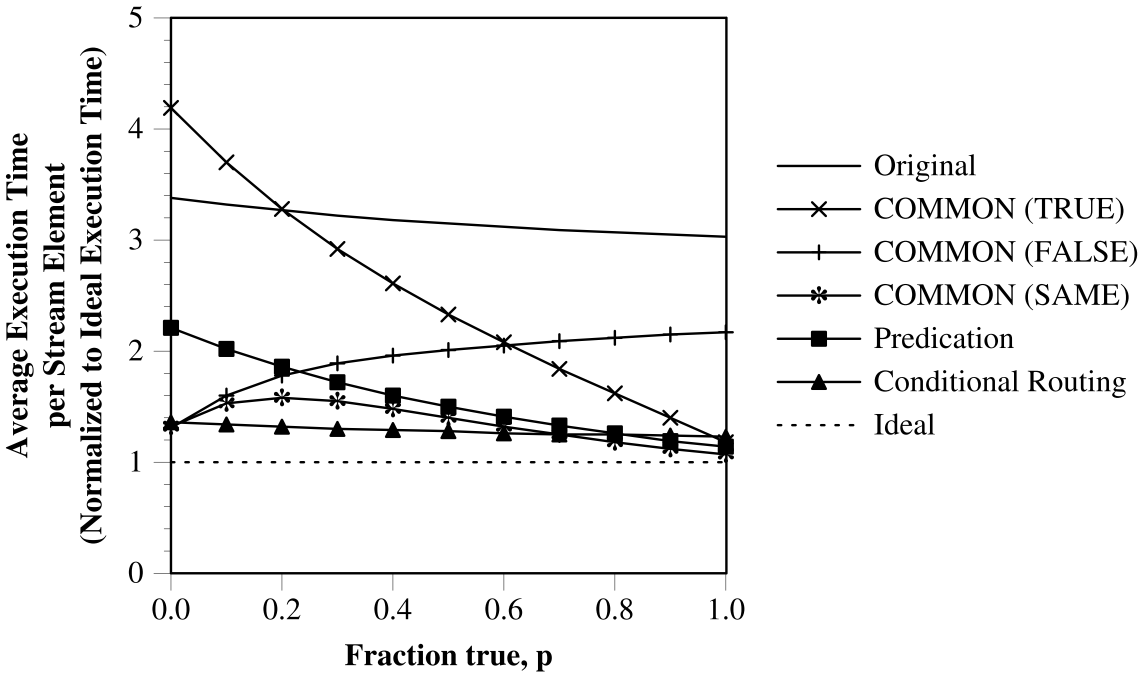


Figure 3.5: Comparison of execution times for existing techniques and conditional routing for the *synthetic_if* kernel. The execution times are shown normalized to IDEAL. $W_{\text{body}} = 32$ operations; $W_{\text{input}} = W_{\text{output}} = 17$ operations; $S = 2$ words.

Performance Comparison for Different Values of p

The particular techniques we will compare against below are: COMMON (TRUE), COMMON (FALSE), COMMON (SAME), and PREDICATION. Results for all techniques on *synthetic_if* are shown in Figure 3.5. The superior results for conditional routing illustrate that it does not need to execute unnecesary operations, unlike speculation and predication.

Speculation generates a more efficient schedule for the common control-flow path, at the expense of the less frequent paths. However, to generate enough available ILP to fully utilize five pipelined ALUS, speculation must be combined with loop-unrolling. While this makes the common case more efficient, it also tends to make the performance penalty increase for mispeculation. This sensitivity to mispeculation is the main problem with these techniques, and only make them useful when we can really accurately predict the control-flow statically. We demonstrate this with three performance curves, each one optimized for a different common case. COMMON (TRUE) is optimized for the case when most elements execute the if-statement (`cond` is TRUE). COMMON (FALSE) is optimized for the case when most elements do not enter the body of the if-statement (`cond` is FALSE). Finally, COMMON (SAME) is optmized for the case when the evaluation of the if-statement doesn't necessarily tend to lean one way or the other on average, but when successive stream elements tend to follow the same path (i.e., the value of `cond` is highly correlated from one stream element to the next).

To illustrate how to generate schedules for this technique, we have implemented COMMON (TRUE) as a source code transformation and shown the pseudo-code in Kernel 3.4. The advantage of this code is that when the if-statement conditions evaluate to TRUE, performance is not degraded by a poor schedule for `BODY`. The operations for `BODY` are combined with operations in other basic blocks via speculation, and with operations in `BODY` from other iterations via loop-unrolling. Notice that operations were chosen for speculative execution without any regard to exceptions that could be raised. This is because we are assuming that our stream processor architecture does not support the trapping of exceptions in hardware—instead exceptions must be checked manually and the code must be inserted by the programmer. For this benchmark we assume that exceptions are not important and do not need to be checked in software.[3] So, the only restriction on executing operations speculatively, is that the operation does not modify any persistent state; and if it does, the modification needs to be undone in the code that handles mispredictions.

[3]This is actually the case for most kernels we have implemented to date, so it is a reasonable assumption to make.

Kernel 3.4 Pseudo-code for the *synthetic_if* benchmark with speculative code inserted that optimizes the resulting schedule for the case where the if-statement condition evaluates to TRUE. The code is also loop-unrolled so that two iterations are executed every trip through the loop.

```
loop over $in\_stream$ begin
    $In\_state \xleftarrow{\text{pop}} in\_stream;$
    $StateA1 \leftarrow INPUT(In\_state);$
    $cond1 \leftarrow TEST(StateA1);$

    $In\_state \xleftarrow{\text{pop}} in\_stream;$
    $StateA2 \leftarrow INPUT(In\_state);$
    $cond2 \leftarrow TEST(StateA2);$

    $StateB1 \leftarrow BODY(StateA1)$                    // Speculatively execute BODY( )
    $StateB2 \leftarrow BODY(StateA2)$

    if $\neg cond1 \lor \neg cond2$ then                            // Fix-up code
        if $\neg cond1$ then $StateB1 \leftarrow StateA1$ fi
        if $\neg cond2$ then $StateB2 \leftarrow StateA2$ fi
    fi

    $Out\_state \leftarrow OUTPUT(StateB1);$
    $out\_stream \xleftarrow{\text{push}} Out\_state;$
    $Out\_state \leftarrow OUTPUT(StateB2);$
    $out\_stream \xleftarrow{\text{push}} Out\_state;$
end
```

The COMMON (TRUE) line in Figure 3.5 is only 18% slower than IDEAL when $p = 1.0$, which is the case it is optimized for. However, as p decreases, the mispeculation penalty increases. In particular, executing BODY when it is not necessary and executing the less efficient fix-up code causes the performance when $p = 0.0$ to be even worse than the plain unoptimized schedule (ORIGINAL). Even by $p = 0.9$ the performance of COMMON (TRUE) is worse than CONDITIONAL ROUTING. Thus, this particular version of the schedule is only useful if we know p is very close to 1. The COMMON (FALSE) curve shows similar properties to COMMON (TRUE). It performs well in the case it is optimized for ($p \approx 0.0$), but pays a mispeculation penalty as p deviates from this value. COMMON (SAME) actually performs well when $p = 0.0$ *and* when $p = 1.0$ because there is a high correlation between successive stream elements at those points. However, the mispeculation penalty even for COMMON (SAME) makes it perform worse than CONDITIONAL ROUTING for most values of p.

Besides mispeculation penalties, another problem with this technique is that loop-unrolling can lead to an increase in the size of the scheduled kernel code. For *synthetic_if*, the COMMON (TRUE), COMMON (FALSE), and COMMON (SAME) schedules were 205, 258, and 171 VLIW instructions long, whereas the three kernels for CONDITIONAL ROUTING totaled to 70 instructions and PREDICATION was only 19 instructions long. If the instruction storage in the micro-controller is insufficient to hold the working set of kernels in an application, then kernels have to be swapped in and out dynamically. This can increase the demands on DRAM bandwidth and SRF capacity.

Predication makes a different trade-off. Instead of only executing the common path and executing expensive fix-up code when another path is required, predication executes multiple paths and then selects the correct results. The important effect of predication is to remove conditional branches from the schedule. The result is that software-pipelining and loop-unrolling are very effective in increasing the achieved ILP. So, this technique is useful when the increased performance due to the improved schedule quality outweighs the overhead of executing the operations from multiple control-flow paths. This can happen, for instance, when the most commonly taken control-flow path contains many more operations than the other control-flow paths.

On our target architecture, instead of augmenting every supported operation type in the

Kernel 3.5 Pseudo-code for the *synthetic_if* kernel after optimizing it using predication. Instead of using a full set of predicated operations, the only predicated operation used by this version is a hardware SELECT operation.

```
loop over in_stream begin
    $In\_state \xleftarrow{\text{pop}} in\_stream;$
    $State \leftarrow INPUT(In\_state);$

    $cond \leftarrow TEST(State);$

    /* Execute BODY() whether or not we actually need to */
    $State\_tmp \leftarrow BODY(State);$

    /* Choose the correct version of State based on cond */
    $State \leftarrow \textbf{select}(cond, State\_tmp, State);$

    $Out\_state \leftarrow OUTPUT(State);$
    $out\_stream \xleftarrow{\text{push}} Out\_state;$
end
```

kernel ISA with the ability to be predicated, we will only provide one predicated operation, namely the SELECT operation. This operation essentially mimics the C “?:” operator, and is similar to the one supported in the Multiflow compiler and hardware [Lowney *et al.*, 1993]. Kernel 3.5 illustrates how to implement the *synthetic_if* kernel using predication and how the SELECT operation is used.

The absolute execution time of the predication technique does not depend on how often each control-flow path is taken. This means that the execution time normalized to IDEAL will deteriorate with decreasing p since IDEAL gets faster. However, notice that while the performance at $p = 0.0$ is much worse than CONDITIONAL ROUTING, it is not as bad as COMMON (TRUE). This is because predication handles choosing the right results with SELECT operations, and it does not need to execute inefficient fix-up code to handle the `cond` = TRUE case.

The conclusion from this analysis is that the performance of conditional routing is the least affected by the value of p, since it does not execute any potentially unnecessary operations from speculative control-flow paths.

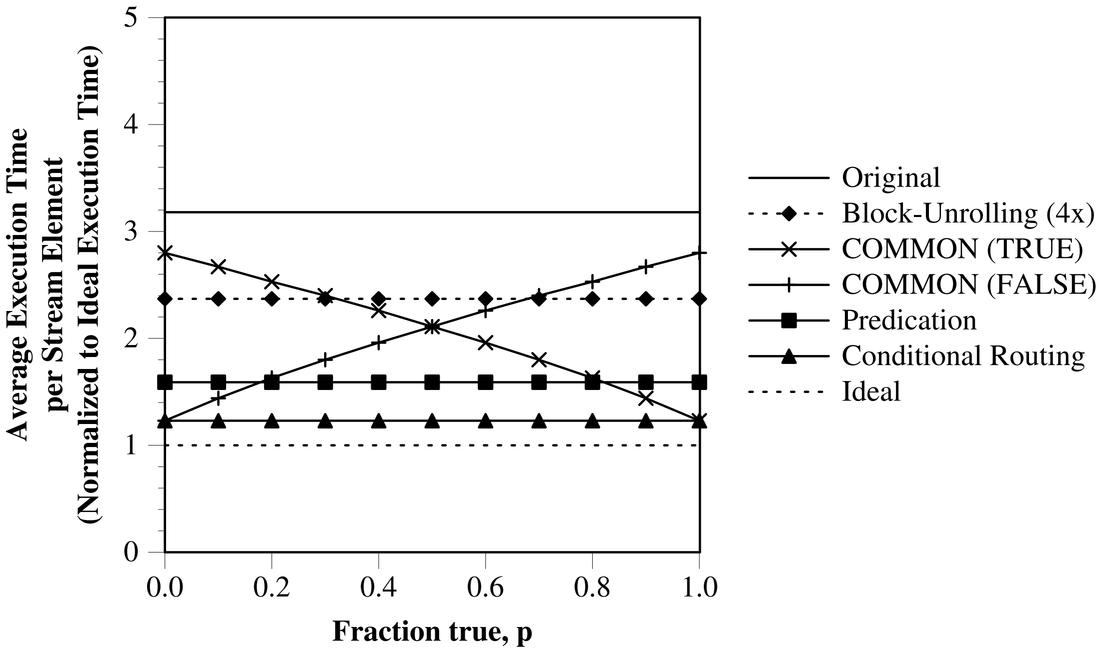


Figure 3.6: This graph shows the results for various conditional techniques on the *synthetic_if_else* benchmark. The execution times are normalized to IDEAL. $W_{\text{if}} = W_{\text{else}} = 32$ operations, $W_{\text{input}} = W_{\text{output}} = 17$ operations, $S = 2$ words.

Performance Comparison for Generalized If- and Case-Statements

It turns out that the benefits of conditional routing translate from our simple *if-then* benchmark to more complicated conditional statements. In fact, the benefits of conditional routing are even more pronounced for these cases. If-statements with an else-clause and perhaps many elseif-clauses can appear in kernel inner loops. A similar construct, the case-statement, can also appear. Conditional routing handles these types of statements by allocating a separate intermediate stream and a new kernel to each clause of the if-statement, or case of the case-statement. Figure 3.6 shows results for *synthetic_if_else*, a micro-benchmark very similar to *synthetic_if*, except that it also contains an else-clause. As before, the value of p indicates how often the then-clause is taken. Also, Figure 3.7 shows results for *synthetic_case*, which contains an eight-way case statement. The value of p in this case indicates how often the first case is taken; the other seven are taken with equal probability.

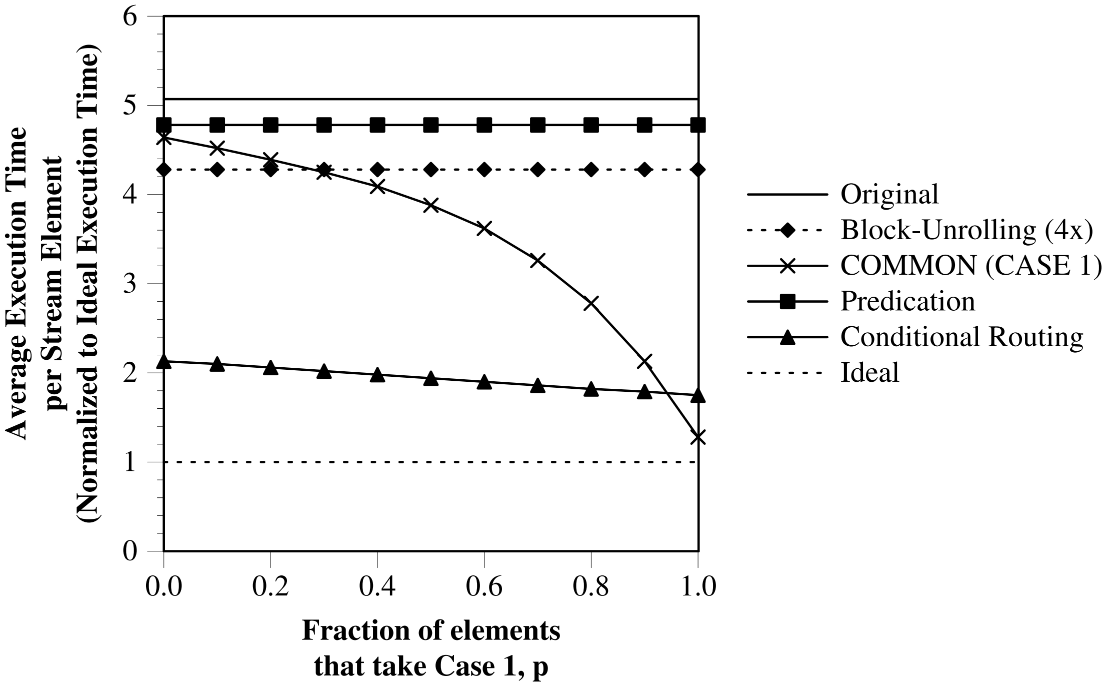


Figure 3.7: This graph shows the results for various conditional techniques on the *synthetic_case* benchmark. The execution times are normalized to IDEAL. The value of p in this case indicates how often the first case is taken; the other seven are taken an equal amount of times. $W_{\text{case}} = 32$ operations, $W_{\text{input}} = W_{\text{output}} = 17$ operations, $S = 2$ words.

For both of these kernels, it is clear that conditional routing is superior for almost all values of p. This is despite the fact that, for *synthetic_case*, there is still almost a 2x slowdown compared to IDEAL. This slowdown occurs because as the number of clauses increases, the amount of overhead for doing the conditional input and output routing increases. For example, since the architecture we used only supports a total of eight streams for both the input and output, we had to insert extra kernels to do the input and output routing in two stages for *synthetic_case*. Also notice that PREDICATION is never better than CONDITIONAL ROUTING for *synthetic_if_else* and is never better than either CONDITIONAL ROUTING or COMMON for *synthetic_case*. This is because the advantages of PREDICATION start to diminish as the number of possible control-flow paths increase, since this increases the number of unnecessary basic blocks that are executed for each stream element.

3.2.3 Technique Selection

So far we have only looked at a particular version of the *synthetic_if* benchmark, and concluded that conditional routing offered advantages for this kernel. However, kernels come in all different sizes and shapes, and conditional routing may not be the best choice for every kernel. Thus we must understand how to choose which which is best in each situation, a problem which we will refer to as *technique selection*. In this section we will show that the important parameters that affect technique selection are basic block size and the amount of live state in the kernel. Furthermore, we will generalize our knowledge of how to optimize simple conditionals in order to optimize more complicated conditionals, such as nested if-then-else statements. As a side note, the results and discussion in this section is information that is critical for implementing automatic translation of control-flow constructs in a compiler.

Block-Size

Figure 3.8 shows what happens when we vary the number operations in the basic block BODY. The first thing to notice is that the execution times increase relative to IDEAL, as W_{body} increases, for the following: COMMON (TRUE) when $p = 0.1$ and for COMMON (FALSE) when $p = 0.9$. This is because the mispeculation penalty and fix-up code get increasingly expensive as W_{body} increases. For a similar reason, the exceution time of PREDICATION increases as W_{body} increases for all three graphs (but at a faster rate for lower p). However, the curves for CONDITIONAL ROUTING are all fairly constant as W_{body} increases for all values of p, again because CONDITIONAL ROUTING never speculatively executes BODY.

The above information is interesting because it highlights the magnitude of the advantage CONDITIONAL ROUTING provides at larger W_{body}. However, we would like to look at the information in a slightly different format in order to learn how to pick the best performing technique for a particular size of BODY.

For the case of the *synthetic_if* kernel, as we can see from Figure 3.8, either PREDICATION or CONDITIONAL ROUTING is generally the best of the four techniques shown.[4] So, for the *synthetic_if* kernel, given any number of operations in BODY, we can choose the optimal technique by selecting one of these two. Clearly, from what we have seen, this selection function should require p and W_{body} as inputs. Figure 3.9 shows what this selection function looks like as a function of these two inputs. Essentially, PREDICATION performs better when p is high and W_{body} is small. In fact, when W_{body} is less than roughly eight operations, PREDICATION is always the better choice. Once the block size increases, however, CONDITIONAL ROUTING is the optimal technique unless $p \approx 1$.

While the value of W_{body} is known at the time of compilation, often the value of p is not known. The selection function in Figure 3.9 can still be used to guide the technique selection process even in this situation. Let us look at two different scenarios, with each one subjecting the selection decision to different optimization constraints. Assuming that the value of p is completely random, with a uniform probability over the range 0.0 to 1.0, here is how one would attack each scenario.

1. *Real-time scenario: minimize the worst-case execution time.* In this case we want to choose the technique that minimizes the maximum possible execution time. In this case, the maximum for both techniques we are considering occurs when $p = 1.0$. Furthermore, the maximum is always less for PREDICATION, so this is the technique one would choose in a real-time scenario where the worst-case execution time is

[4]The slight ripple in the curve is due to noise that stems from the fact that the kernel scheduler does not always find the optimal schedule, but is generally anywhere within 0%–10% of the optimal.

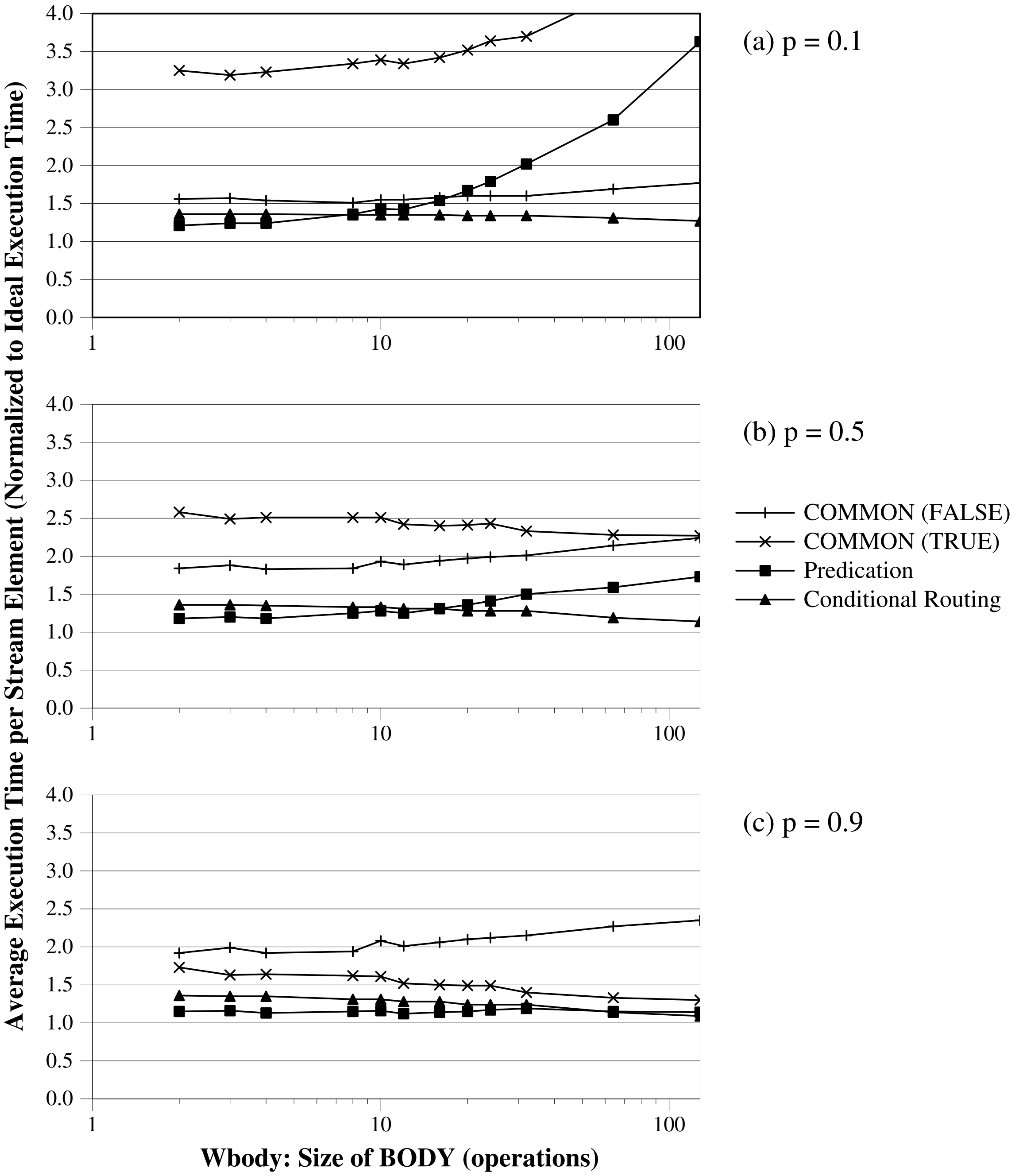


Figure 3.8: These graphs show the impact of varying the number of operations in the basic block `BODY` on the execution time for the various if-statement techniques on the *synthetic_if* kernel. $W_{\text{input}} = W_{\text{output}} = 17$ operations, $S = 2$ words.

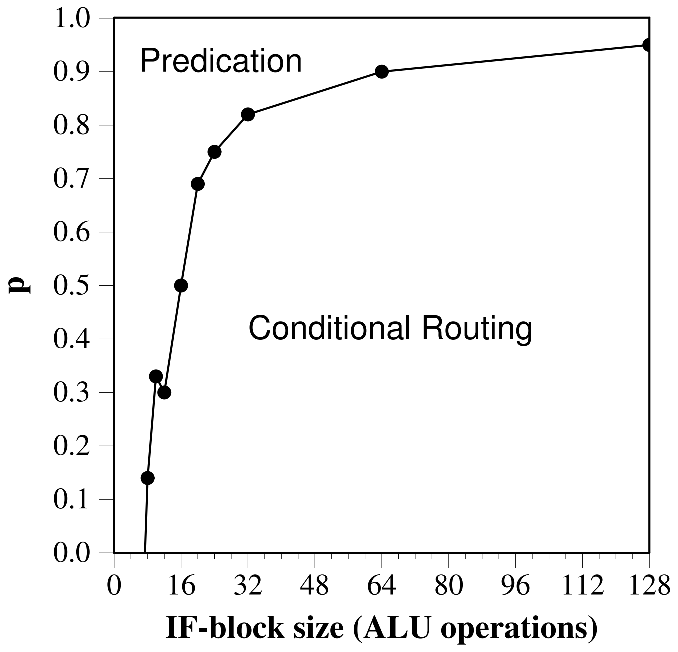


Figure 3.9: This graph illustrates the technique selection function for *synthetic_if*, based on two input parameters: p and W_{body}. The graph can be interpreted as a phase diagram, where each region corresponds to a technique. The extent of the region indicates the range of parameter values for which the corresponding technique is optimal. The slight ripple in the curve is not due to anything systemic, and the magnitude of the anomaly is within the noise. The noise stems from the fact that the kernel scheduler does not always find the optimal schedule, but is generally anywhere within 0%–10% of the optimal. $W_{\text{input}} = W_{\text{output}} = 17$ operations; $S = 2$ words.

important.

2. *No external constraints*. A reasonable thing to do in this situation would be to try to reduce the selection error over the range of p, where the selection error for a value of p is measured as the difference between the execution time of the technique selected and the execution time of the optimal technique for that value of p. Because the execution times of both PREDICATION and CONDITIONAL ROUTING are linear with respect to p, it is easy to show that the correct thing to do is to make a selection assuming that $p = 0.5$. Thus for $W_{\text{body}} < 16$ operations, PREDICATION is the better choice in this situation, while CONDITIONAL ROUTING makes more sense when $W_{\text{body}} > 16$ operations.

We've only looked at the size of the `BODY` block so far, and not of `INPUT` and `OUTPUT`. It turns out that W_{input} and W_{output} only affect the difference in performance between CONDITIONAL ROUTING and PREDICATION and not the crossover point at which we would select one over the other. As W_{input} and W_{output} decrease, we would expect the difference between PREDICATION and CONDITIONAL ROUTING to increase. This is because the ratio of $W_{\text{body}}/(W_{\text{input}} + W_{\text{output}})$ determines how much the unnecessary operations impact the performance of PREDICATION. Therefore, as the ratio increases, so does the potential performance benefit that CONDITIONAL ROUTING offers at low values of p. However, theoretically, W_{input} and W_{output} shouldn't affect the crossover point between PREDICATION and CONDITIONAL ROUTING at a given W_{body}, except when the block sizes get smaller than a few operations.[5] This is because the crossover point only depends on the number of unnecessary operations executed by PREDICATION, and hence only on W_{body}.

Impact of SRF Communication

The other parameter that affects technique selection is the number of words of state that are live at the input and output of the `BODY` block. This is because the amount of live state determines the amount of extra SRF communication that is necessary with the CONDITIONAL ROUTING technique. It turns out that, CONDITIONAL ROUTING is affected more adversely

[5]When the block sizes are too small, it is not very efficient to schedule them as separate kernels. This would cause PREDICATION to be more attractive.

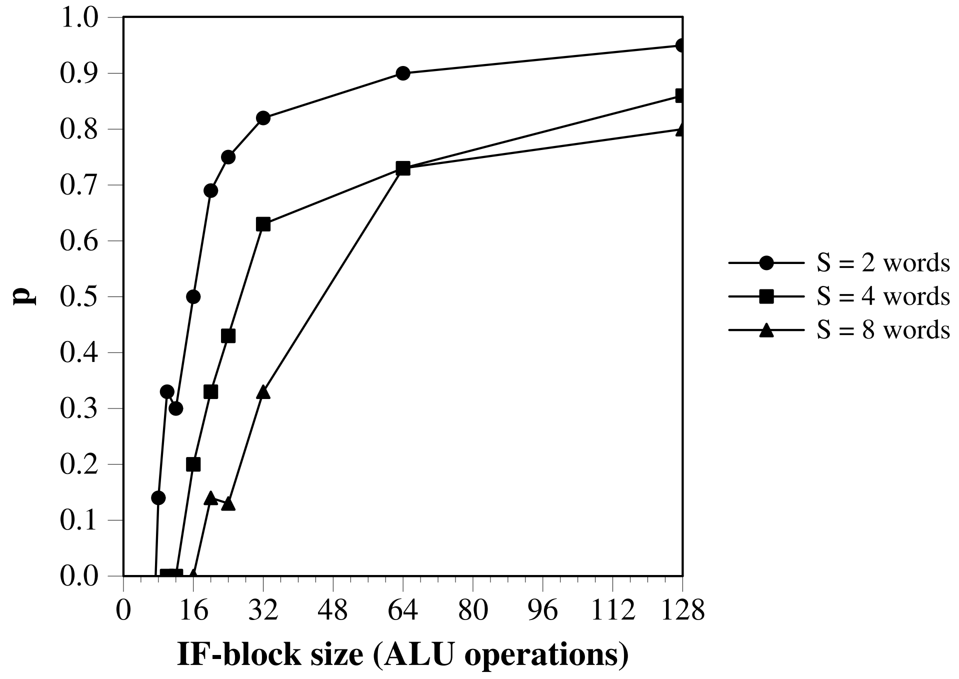


Figure 3.10: This graph shows how the crossover point between where CONDITIONAL ROUTING is better and where PREDICATION is better changes with the amount of live state in the kernel. (PREDICATION is better to the left of the line.) This happens because the performance of CONDITIONAL ROUTING deteriorates faster than the performance of PREDICATION as S increases. For reference, Figure 3.3 shows how value of S affects the kernel structure. This graph uses the same values of kernel parameters as Figure 3.9. $W_{\text{body}} = 32$ operations, $W_{\text{input}} = W_{\text{output}} = 17$ operations.

by increasing S than either COMMON or PREDICATION. Figure 3.10 shows clearly the impact of this on technique selection, since PREDICATION becomes the better technique for a larger fraction of kernels compared to CONDITIONAL ROUTING as S increases. So, in order to complete our study of technique selection, we must delve deeper into the impact of S on performance. To this end, Figure 3.11 shows the results of increasing the amount of SRF communication required on the normalized execution time of the various techniques on the *synthetic_if* kernel. Notice that we increased the number of operations in each basic block for this graph in order to vary S over a wide range of values.

The normalized execution time of each method increases as the amount of live state increases in the benchmark (the ideal execution time is unaffected by the size of the live

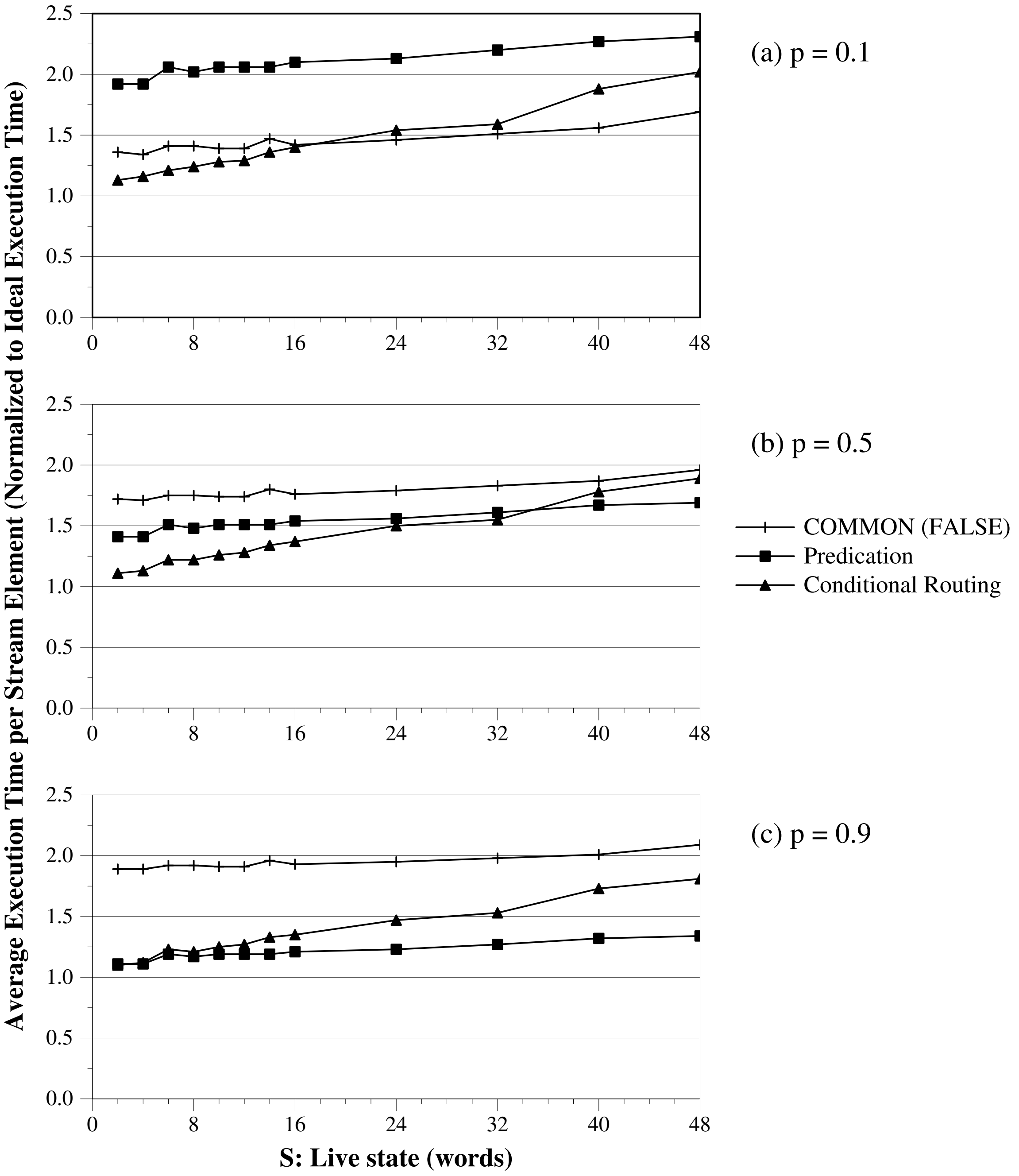


Figure 3.11: These graphs show the impact of increasing the amount of live state, S, in the *synthetic_if* kernel on the execution time for the various if-statement techniques. $W_{\text{body}} =$ 128 operations, $W_{\text{input}} = W_{\text{output}} = 65$ operations.

state in the kernel). The execution time of COMMON (FALSE) increases slightly because of a property of the synthetic benchmark that causes the length of the dependency chain to slightly increase when S is increased. The normalized execution time of PREDICATION increases because of the extra SELECT operations that are added into the schedule. The normalized execution time of CONDITIONAL ROUTING, however, increases at a faster rate than the other methods with increasing S. This is because there is an additive effect of the increase in execution time of all three kernels.

- The schedule lengths for the *input* and *body* kernels increase because the scheduler we used inserts pass-through instructions to communicate the result of an ALU operation to the SRF output stream. These are necessary because the ALU operations that produce the different fields of the output record don't always get scheduled in the same order as the data needs to be output to the stream (fields within a record are output to streams in a strict order that all kernels obey in order to exchange data from one to another). Since there is no buffering specifically for the output streams, the record fields are stored in the ALU register files as they are produced. Eventually, each pass-through operation will require an ALU instruction slot in order to communicate the record field from its register file to the output stream buffer.
- The schedule length for the *output* kernel increases with increasing live state because it needs to add SELECT operations, which are needed to implement the "popif" operator, used in the *compute_2* kernel (Kernel 3.2). These SELECT operations are essentially used to control whether the target variable is updated or not with the result of the "popif" operator.

These cumulative effect of the increase in all three kernels causes the performance of CONDITIONAL ROUTING to deteriorate faster with larger live state sizes than the other techniques. For this kernel, though, CONDITIONAL ROUTING is still the best technique for $p = 0.5$ up to a record size of 32 words. However, as block sizes get smaller, especially for the `INPUT` and `OUTPUT` basic blocks, the effect of the SRF communication will be even larger and CONDITIONAL ROUTING will be more adversely affected.

Complex Conditional Structures

Unlike the simple if-statements we have been considering up until now, more complex conditional statements need not necessarily be implemented entirely using only one technique. The observation we will make is that each clause of a complex if-statement or case-statement can be written as an independent simple if-statement of the type in the *synthetic_if* kernel. We can then independently apply the best algorithm for each new simple if-statement. As we have come to conclude, there are three main factors in deciding what type of optimization to apply to a simple if-statement: p, W_{body}, and S. Each clause of the more complex conditional statement may have a different characteristic triplet of values, and hence may require a different technique for best results.

We can even extend this idea to nested if-statements. Nested if-statements can be flattened so that each clause of every level of every if-statement can be re-written as a simple top-level if-statement, whose entry condition is a union of all the conditions for all the if-statements it was originally embedded within, and whose execution frequency (p) is the product of the execution frequencies of all the if-statements it was embedded within. These new if-statements can be optimized as we just discussed above.

However, while reducing the more complex situations to a series of simple if-statements seems like a viable solution that could improve performance, it seems like we might be throwing away some information about the structure of the control-flow. More sophisticated algorithms might not consider each clause absolutely independently, and might capitalize on global information to perhaps optimize the method selection further. For example, if the same subset of live state is needed by two different clauses, it might make sense to group them together regardless of their individual values of p and W_{body}. We leave it as an open compiler problem to develop the best set of heuristics for automatically compiling a set of complex if-statements with arbitrary nesting.

Technique Selection Summary

Speculative techniques (COMMON and PREDICATION) improve VLIW schedules by filling ALU slots with operations that are potentially unecessary. Therefore, larger conditional block sizes affect these techniques detrimentally. However, on the other hand CONDITIONAL ROUTING uses data communication in order to organize data such that we can process them in a more regular fashion. Hence, CONDITIONAL ROUTING was affected detrimentally by larger amounts of live state within a kernel, which corresponds to a larger amount of necessary SRF communication.

We also discussed options for implementing more complicated conditional structures, and made the observation that different techniques may have to be used in conjunction for optimal results. We presented a simple technique, which flattens all nested conditionals, and handles all resulting top-level conditional clauses independently. We have left the investigation of more sophisticated technique selection algorithms for future work.

3.2.4 SRF Allocation

So far, our discussions have only only focused on kernel-level issues. One application-level issue that is important is the increased demand on SRF capacity required by conditional routing, due to the intermediate streams it generates when splitting a kernel. The impact of this extra demand manifests itself in several ways. First, it may require a smaller overall batch size, which in turn can increase the overhead due to short stream effects, as discussed earlier. Second, in more extreme cases, the stream scheduler may have to insert stream spills to off-chip DRAM and stream restores in order to make room for the intermediate streams. The impact of these spills and restores on performance depends on how memory limited the overall application is. Third, these streams can place extra burden on other processor resources, such as the stream descriptor register file, which can generate extra stream instructions to make sure the necessary information is cached at any given time. These extra instructions can burden the application processor with extra work as well. The performance impact of these effects will not be studied further in this thesis, and must be evaluated on a case by case basis.

There have been a couple of proposed ideas to reduce the capacity necessary for intermediate streams by alleviating fragmentation in the SRF. In the worst case, the intermediate stream for each path will require its own storage for up to the number of elements in the input stream. Thus, in the worst case an if-then-else statement will require $2N$ words of intermediate storage if the input stream contains N words, which is a fragmentation overhead of 2x. One possible optimization would be to allow a method for storing more than one stream in a contiguous area of the SRF, if the maximum of the sum of the stream lengths is known. One way to implement this would be to store one stream starting at one end of the SRF buffer, and to store the other stream starting at the opposite end of the buffer and advancing the storage pointer in reverse order. For more than two streams, one could interleave streams at the granularity of a SRF block, assuming the necessary hardware support was available for keeping track of the next block in any particular logical stream. For n streams that sum to a total of N words, this would require $N + n(B/C - 1)$ words instead of nN words, where B is the block size of the SRF and C is the number of clusters. With a more flexible SRF access method, such as an in-lane indexable-SRF [Jayasena *et al.*, 2004], we could forgo the extra hardware for keeping track of the next-block and instead manage the necessary free list in software.

3.2.5 Conditional Routing Summary for If-Statements

Existing methods for achieving close to the peak efficiency of a VLIW cluster for kernels with if-statements rely mainly on speculatively executing operations. These techniques work well when the number of operations that need to be speculatively executed are small, or when the predictions of the control-flow path are accurate. Conditional routing helps fill the void by enabling a stream processing VLIW cluster to reach close to the peak ILP even when these conditions don't hold. Conditional routing accomplishes this by displaying good levels of efficiency over a wide range of values of p and W_{body}. However, we did see that the performance of conditional routing is sensitive to the number of words of live state that are transferred into and out of the conditional statements.

Furthermore, blindly applying conditional routing isn't always the best choice. We studied how technique selection is affected by different parameters. Large basic-block sizes deteriorate the performance of speculative techniques. Large amounts of SRF communication for intermediate streams deteriorate conditional routing performance. Hence, kernels that stand to gain the most from CONDITIONAL ROUTING are those with conditional blocks that require a lot of processing, and at the same time have relatively small record sizes.

Table 3.1 Summary of parameter sensitivities for if-statement conditional techniques. The first three apply to simple if-statement clauses, while the last applies to more complex statements, such as if-then-else and case-statements.

	CONDITIONAL ROUTING	PREDICATION	COMMON
p	Good all round, even when unpredictable	Good when high	Good when predictable
W_{body}	Better when large	Good when small	Not affected very much, if p is predicted correctly
S	Bad when large	Slightly worse when large	Slightly worse when large
Number of elsif-clauses (or cases)	Small degradation when high, due to extra routing	Large degradation when high, due to increased number of unnecessary control-flow paths	Large degradation when high, due to potentially greater unpredictability

3.3 Applying Conditional Routing to While-Loops

This section will demonstrate how conditional routing can be applied to while-loops, in addition to the if-statements we discussed above. There are two methods of applying conditional routing to while-loops: expanded conditional routing and flattened conditional routing. These techniques solve the main problem, which is increasing ILP even when the while-loop contains a long loop-carried dependency. In addition, conditional routing also works well even when each while-loop is only executed for a small number of iterations, or when the number of iterations is highly variable from one stream element to the next.

A simple example of a kernel with a while-loop is shown in Figure 3.12. This type of kernel might be necessary, for instance, if a numerical algorithm required iterating until a certain accuracy or precision was achieved, and the number of iterations required to do so was potentially different for each stream element. In this example, the while loop executes at least once per stream element, but as we will show later, this is not a necessary condition. Also, just as we did for the original example kernel for if-statements, we will assume that there are no loop-carried dependencies from COMPUTE_ITERATIVE( ) to COMPUTE_1( ) or from COMPUTE_2( ) to either COMPUTE_1( ) or COMPUTE_ITERATIVE( ), although often there *is* a loop-carried dependency within the processing of a stream element from one iteration to the next of the inner while-loop.

The first problem with scheduling kernels like the one in Figure 3.12, is that often the inner while-loop schedule is limited by a long critical-path that updates the loop-carried state in the while-loop, preventing effective software-pipelining or loop-unrolling. Furthermore, since a while-loop may execute many times per stream element, a higher fraction of the execution time can potentially be devoted to the while-loop basic block, making its poor schedule impact performance even more than it did for an if-statement. Second, even if the while-loop software-pipelines well, if it only executes a small number of iterations per stream element, the overhead of priming the software pipeline will impact overall performance.

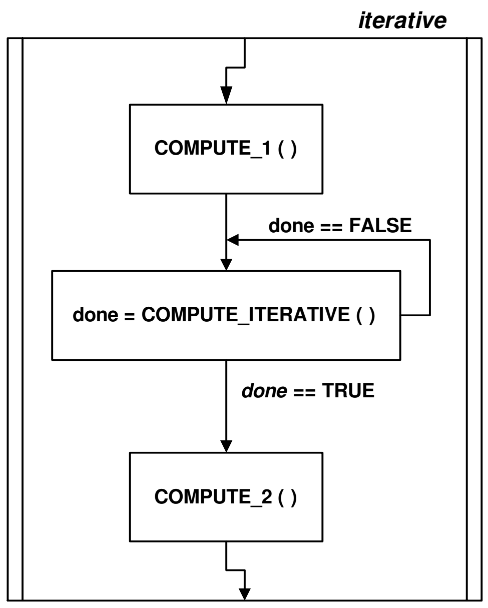


Figure 3.12: An example of a kernel, *iterative*, that contains a while-loop.

3.3.1 Expanded Conditional Routing

As we did with if-statements, conditional routing can be used to reorder the work in a kernel containing a while-loop. The goal is to schedule the while-loop basic block as a separate kernel which can be scheduled efficiently. However, unlike the case with an if-statement, the schedule for a new kernel that is formed by simply transplanting the while-loop will still be limited by the loop-carried dependency in the while-loop. In order to overcome this problem, we can think of *expanding* the while-loop as a series of if-statements and then apply conditional routing to each of these if-statements. Figure 3.13 illustrates this concept.

If we converted the control-flow diagram shown in the figure using conditional routing, each of the basic blocks (i.e., each box in the figure) would become a separate kernel call, and each kernel would finish processing all stream elements before the execution of the next kernel began. This is shown explicitly in Figure 3.14, which lists the application-level pseudo-code for implementing expanded conditional routing on this example. At the outset, the *compute_1* kernel would process all the input stream elements, generating a stream of intermediate results. Then, the *compute_iterative* kernel executes the first iteration of the

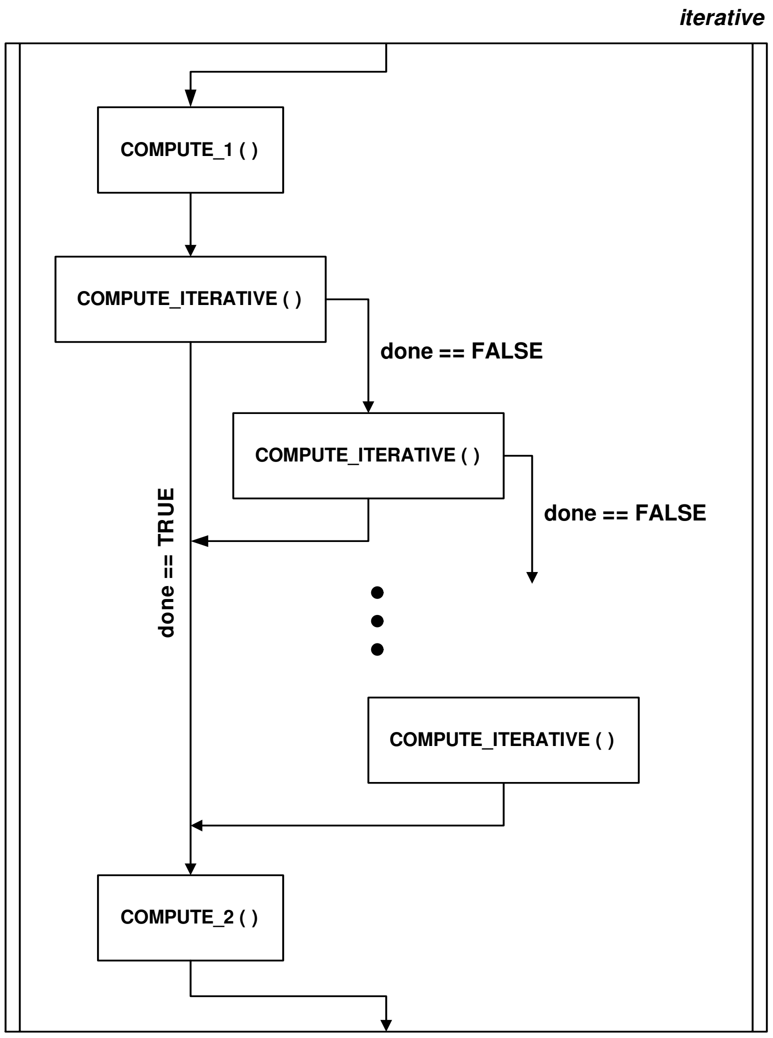


Figure 3.13: Expanding the control-flow diagram for a kernel with a while-loop using a series of if-statements.

```
/* Original:  iterative(in_data, out_data); */
compute_1(in_data, not_done);
done_data.clear();
while not_done.length() > 0 do
   compute_iterative(not_done, tmp_not_done, new_done_data);
   done_data.append(new_done_data);
   not_done = tmp_not_done;
od
compute_2(done_data, out_data);
```

Figure 3.14: Application-level pseudo-code for expanded conditional routing.

while-loop on this intermediate stream, generating two new streams: one with elements that needed to execute only one iteration of the while-loop (the *new_done_data* stream), and another with the remaining stream elements. The same *compute_iterative* kernel is invoked with the latter stream as input, this time executing the second iteration of the while-loop. It generates two streams itself. The first is appended to *done_data*, and the other is left as input for the next kernel invocation. This process is continued until the stream element(s) that require the maximum number of iterations have been processed, and all the computed outputs are in the *done_data* stream.

There are two performance advantages with this new set of kernels over the original. First, since the *compute_iterative* kernel only executes one iteration of the while-loop on each stream element, the loop-carried dependencies in the original while-loop will not limit the kernel from being software-pipelined. And second, for cases where most stream elements require only a small number of iterations, the overhead to prime the software-pipelines will not be incurred for each element.

On the other hand, there are potential drawbacks as well. For example, one detail we glossed over in the above explanation is that all the loop-carried state in the while-loop must be output for each stream element, after each iteration. As we saw in our discussion of if-statements, if this is a large amount of state, it could increase the SRF communication to a level where it significantly decreases the performance advantages of conditional routing. The extra overhead from this state could also reduce the strip-size in the SRF, increasing short-stream effects in the overall application. Another drawback depends on the distribution of the number of loop iterations required for the stream elements. Consider the case when there is only a small minority of stream elements that require a large number of iterations, while most stream elements only require a much smaller number of iterations. Then short-stream effects arise from calling the *compute_iterative* kernel many times on only a small number of elements. Finally, notice that the ordering of the final `out_data` stream is not necessarily the same as the ordering of the output stream from the original kernel. This could be handled by performing a sort on the `out_data` stream, but this will eat away at the performance benefits offered by the method.

In summary, the expanded conditional routing technique is an extreme point in terms of SRF capacity and bandwidth requirements: loop state is written to the SRF on every iteration for every stream element. Furthermore, expanded conditional routing places demands at the application-level, which may degrade overall performance. For example, this technique can significantly increase the number of stream-level instructions. Also, a while-loop is now introduced into the application-level code, making scheduling of stream-level instructions more challenging. Note, however, that the results presented in this chapter were obtained using kernel performance only, and hence will not include these application-level effects. In any case, the next section will discuss a different algorithm, flattened conditional routing, that makes more moderate demands on the SRF and application processor.

3.3.2 Flattened Conditional Routing

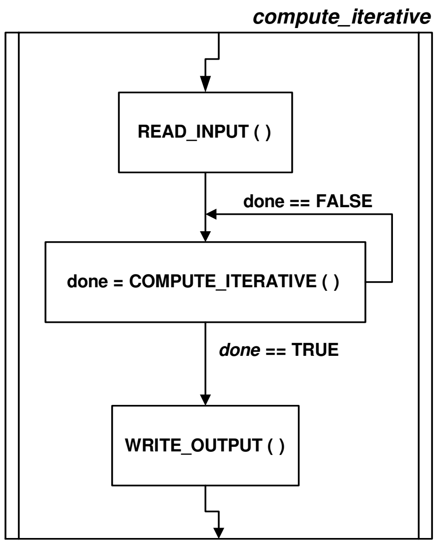
To recap, the two main problems with while-loops are that loop-carried dependencies within the loop limit its schedule quality, and small numbers of iterations per stream element increase the overhead due to software-pipelining the loop even if the loop-carried dependency is insignificant. Another way to address the problem, while avoiding the extreme SRF requirements of expanded conditional routing, is to use a combination of conditional routing and *loop-flattening* [von Hanxleden and Kennedy, 1992].

We begin by splitting the original kernel into three separate ones. As usual, the intermediate results produced by the *compute_1* kernel will be stored to the SRF, and the intermediate results after executing all the while-loops will be stored to the SRF as well. However, unlike expanded conditional routing, we will not buffer intermediate loop state between while-loop iterations in the SRF—instead we will process the entire while-loop for each stream element until completion before starting to process the next element.

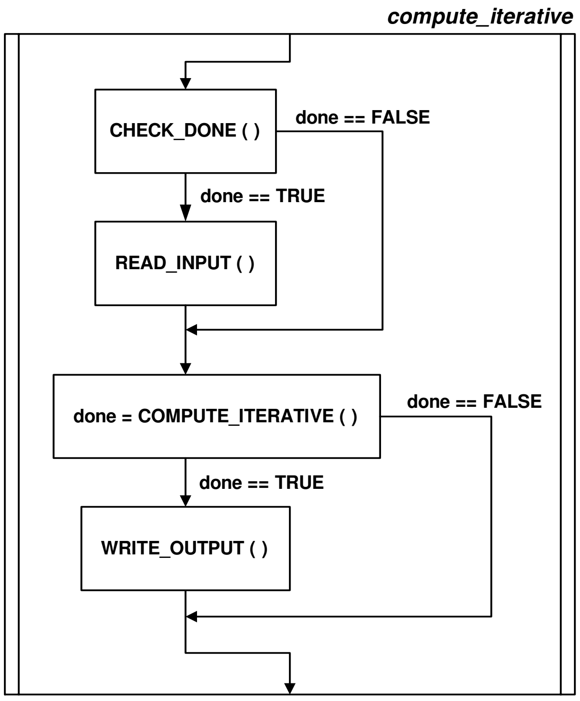
The structure of the main-loop in the new *compute_iterative* kernel will look like the original kernel, except the code before and after the while-loop will only contain stream input and output operations (see Figure 3.15a). Applying loop-flattening to this kernel hoists the code in the while-loop, i.e., the code in COMPUTE_ITERATIVE( ), into the main-loop. The result of loop-flattening is shown in Figure 3.15b. Essentially, the code in COMPUTE_ITERATIVE( ) is run every iteration of the main loop. However, the code that reads in new data and writes out the results are embedded in if-statements and only execute if the current stream element has completed the requisite number of while-loop iterations. Assuming *done* is initialized correctly in the preamble to the kernel loop (the preamble is not shown in the figure), and that the loop condition is modified to wait for the last stream element and the last while-loop iteration for that element, then this new singly-nested kernel loop is functionally exactly the same as the original doubly-nested loop.

Of course, we eliminated one control-flow construct, namely the while-loop, and introduced two others, namely the if-statements. Fortunately, the only code in the if-statements are stream accesses, so we can easily replace the entire if-statements with conditional routing primitives. This results in a kernel with a main-loop with only one basic-block, and is illustrated in Kernel 3.6. Since the first iteration of a new stream element will follow just as seamlessly as another iteration for the same stream element, there is no need to pay a penalty every time we start processing a new stream element. Thus, even if stream elements only execute a small number of iterations, the performance of flattened conditional routing will not significantly degrade. Finally, notice, that unlike expanded conditional routing, the output of the final kernel, `out_data`, will be in the same order as the original kernel.

Now, we can try to software-pipeline or loop-unroll the kernel normally. Unfortunately, however, in general the software-pipelining and loop-unrolling optimization will not always work well on the new kernel. This is because any loop-carried dependencies between each iteration that might limit the ILP are still problematic. Not only are these loop-carried dependencies from the original while-loop still present, but we have introduced a new one in the form of the *done* variable. For this reason, we will introduce state-unrolling to increase the achieved ILP in spite of any loop-carried dependencies.



(a)



(b)

Figure 3.15: Control-flow diagram for the *compute_iterative* kernel (a) before and (b) after applying loop-flattening.

Kernel 3.6 Pseudo-code for the *compute_iterative* kernel after applying loop-flattening. In general, the data structure that is input from the SRF will not be the same as the data structure that is written out to the SRF. They are the same here for simplicity.

```
$done \leftarrow$ **true**;
**loop while** $\neg in\_stream.empty() \vee \neg done$ **begin**
    $State \xleftarrow{\text{popif}} in\_stream(done)$;
    $State, done \leftarrow COMPUTE\_ITERATIVE(State)$;
    $out\_stream(done) \xleftarrow{\text{pushif}} State$;
**end**
```

3.3.3 State-Unrolling

While software-pipelining and loop-unrolling try to execute multiple *consecutive* iterations of the while-loop for the *same* stream element in parallel, applying the state-unrolling transformation to a flattened loop causes multiple loop iterations for *different* stream elements to be scheduled together. A good way to think about this is that there are two virtual processors, each with their own private loop-state. When one of the virtual processors, say VP1, finishes the last iteration for its stream element, it sets its private *done* variable. Then, only the results for VP1 will be written to the SRF and new inputs read in. The situation is illustrated in Figure 3.16. In contrast to expanded conditional routing, which had to keep alive the intermediate loop state for *every* stream element, state-unrolling only needs to keep track of the live state for a small fixed number of elements. So, while expanded conditional routing needed to buffer the loop state in the SRF, state-unrolling can keep the live state at any point in time in the LRFs.

In order to implement this scenario on a single cluster, we duplicate the loop code *and* loop state for each virtual processor. Since we are assuming there are no dependencies from one basic block to a previous basic block in the original kernel, then each new copy of the loop code will be independent and can be scheduled together to increase the available instruction parallelism. The final version of the code for the *compute_iterative* kernel after applying loop-flattening and state-unrolling is shown in Kernel 3.7. Notice that some extra logic is necessary for handling the slightly more complex control. In particular, the

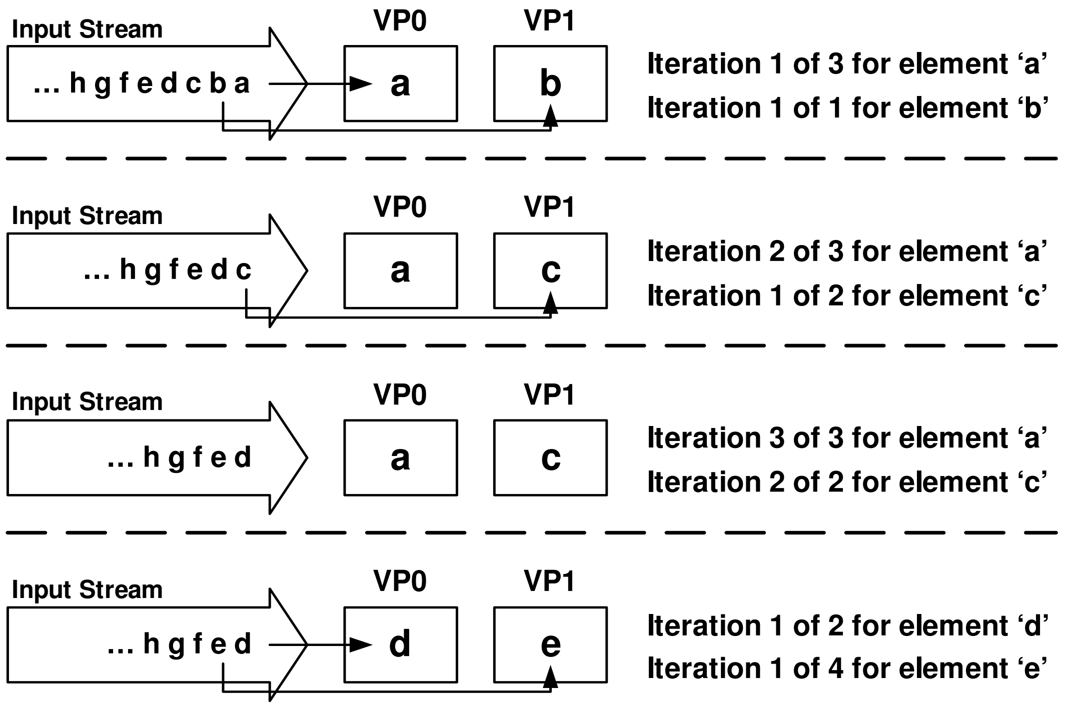

Input Stream
... h g f e d c b a
VP0
VP1
a
b
Iteration 1 of 3 for element ‘a’
Iteration 1 of 1 for element ‘b’
Input Stream
... h g f e d c
VP0
VP1
a
c
Iteration 2 of 3 for element ‘a’
Iteration 1 of 2 for element ‘c’
Input Stream
... h g f e d
VP0
VP1
a
c
Iteration 3 of 3 for element ‘a’
Iteration 2 of 2 for element ‘c’
Input Stream
... h g f e d
VP0
VP1
d
e
Iteration 1 of 2 for element ‘d’
Iteration 1 of 4 for element ‘e’

Figure 3.16: Two virtual processors executing the same while-loop, but on different stream elements. This scenario is implemented on a single cluster using loop-flattening and state-unrolling, where the latter duplicates the loop code *and* loop state for each virtual processor.

Kernel 3.7 Pseudo-code for the *compute_iterative* kernel after applying loop-flattening and state-unrolling (for a total of two sets of loop states).

```
$done\_1 \leftarrow$ true;
$done\_2 \leftarrow$ true;
loop while $\neg in\_stream.empty() \lor \neg done\_1 \lor \neg done\_2$ begin
  /* Virtual Processor 1 */
  $waiting\_for\_VP2 \leftarrow done\_1 \land in\_stream.empty()$;
  $State\_1 \xleftarrow{\text{popif}} in\_stream(done\_1 \land \neg waiting\_for\_VP2)$;
  $State\_1, done\_1 \leftarrow COMPUTE\_ITERATIVE(State\_1)$;
  $done\_1 \leftarrow$ select($waiting\_for\_VP2$, true, $done\_1$);
  $out\_stream(done\_1 \land \neg waiting\_for\_VP2) \xleftarrow{\text{pushif}} State\_1$;

  /* Virtual Processor 2 */
  $waiting\_for\_VP1 \leftarrow done\_2 \land in\_stream.empty()$;
  $State\_2 \xleftarrow{\text{popif}} in\_stream(done\_2 \land \neg waiting\_for\_VP1)$;
  $State\_2, done\_2 \leftarrow COMPUTE\_ITERATIVE(State\_2)$;
  $done\_2 \leftarrow$ select($waiting\_for\_VP1$, true, $done\_2$);
  $out\_stream(done\_2 \land \neg waiting\_for\_VP1) \xleftarrow{\text{pushif}} State\_2$;
end
```

logic for the *waiting_for_VP*n variables is new. Notice that we could have implemented some of this new control logic using if-statements. For example, the subset of operations within COMPUTE_ITERATIVE( ) to calculate *done_1* do not need to be executed if *waiting_for_VP2* is true. However, since this conditional only arises rarely, namely for the last few iterations of the kernel, and since we cannot apply conditional routing anyway because of dependencies within the main loop, we chose to implement the logic using predication.

Notice that using flattened conditional routing with state unrolling will produce a final output that is ordered differently than the output of the original kernel. We will discuss how to handle this in a future section.

Kernel 3.8 Pseudo-code for the *synthetic_while* benchmark. The kernel contains a while-loop whose body will execute zero or more times for each stream element. Also, the kernel will generate an output every iteration of the while-loop to a second output stream.

```
loop over $in\_stream$ begin
    $In\_state \xleftarrow{pop} in\_stream$;
    $State, done \leftarrow INPUT(In\_state)$;
                                                  INPUT: 17 arithmetic operations
    while $\neg done$ do
        $State, done \leftarrow BODY(State)$;
        $out\_stream2 \xleftarrow{push} State$    BODY: 68 arithmetic operations
    od

    $Out\_state \leftarrow OUTPUT(State)$;
    $out\_stream \xleftarrow{push} Out\_state$;   OUTPUT: 17 arithmetic operations
end
```

3.3.4 Performance

As we did with our optimizations for if-statements, we will use a synthetic benchmark to evaluate the performance of the techniques we have introduced. Kernel 3.8 lists the pseudo-code for the *synthetic_while* kernel. It is a little more general in its structure as compared to the example kernel, *iterative*, that we have been studying so far. In particular, the while-loop in *synthetic_while* may be executed zero times for a stream element, which will introduce extra control logic into the kernel. Additionally, there is a stream output generated every iteration of the while-loop as well as at the completion of the while-loop. Finally, we have made the kernel configurable so that we can change the length of the critical path in the basic block `BODY` without changing the number of operations in that basic block.

The speedups of the conditional routing techniques on the *synthetic_while* benchmark are shown in Figure 3.17. The speedups are compared to the first existing technique we will consider: the SWP technique. To apply this technique, we first unrolled the main-loop four times in order to efficiently schedule the operations in `INPUT` and `OUTPUT`. Then we software-pipelined each of the four inner while-loops. For all graphs, the IDEAL results show what the speedup would be if there were no software-pipelining priming overheads, and if the operations could be perfectly scheduled (i.e., ignoring any loop-carried dependencies).

Impact of the Length of Loop-Carried Dependencies

The difference between the top and bottom set of graphs is the length of the longest loop-carried dependency in the while-loop. The version of the loop that produced the top set of graphs has a relatively short loop-carried dependency chain, enabling software-pipelining to achieve near optimal schedules. The other version of the kernel, however, has a relatively long loop-carried dependency that renders software-pipelining and loop-unrolling optimizations fairly ineffective. As a comparison, simply software-pipelining the loop with the short dependency chain produced a schedule that was 15 cycles long, whereas the loop with the longer loop-carried dependency was scheduled in 33 cycles despite having the exact same number of arithmetic operations. Thus, for each method, the increase in speedup, going from a graph on top to the graph directly below it, is due to the ability of that method to increase the achieved ILP in spite of a long dependency. Expanded conditional routing (EXCR) and flattened conditional routing with state unrolling (FLCRSU) are particularly good at this. Without state-unrolling however, simple flattened conditional routing (FLCR) doesn't do anything special to circumvent the long dependency—in fact, it adds a small number of cycles to the dependency chain in order to calculate the loop condition, `done`. The benefit of using state-unrolling in addition to flattened conditional routing, therefore, can be clearly seen by the increased speedup of FLCRSU over FLCR on the bottom set of graphs.

Impact of the Number of While-Loop Iterations Executed

The difference between the left set of graphs and the right set of graphs is the number of iterations of the while-loop each stream element executes. For SWP, the primary overhead is the cost of priming the software-pipeline for each stream element. This effect however, becomes insignificant as the number of iterations increase, as is demonstrated by the reduction in speedup of IDEAL from roughly 3.5$\times$ for 2 iterations per stream element, to roughly

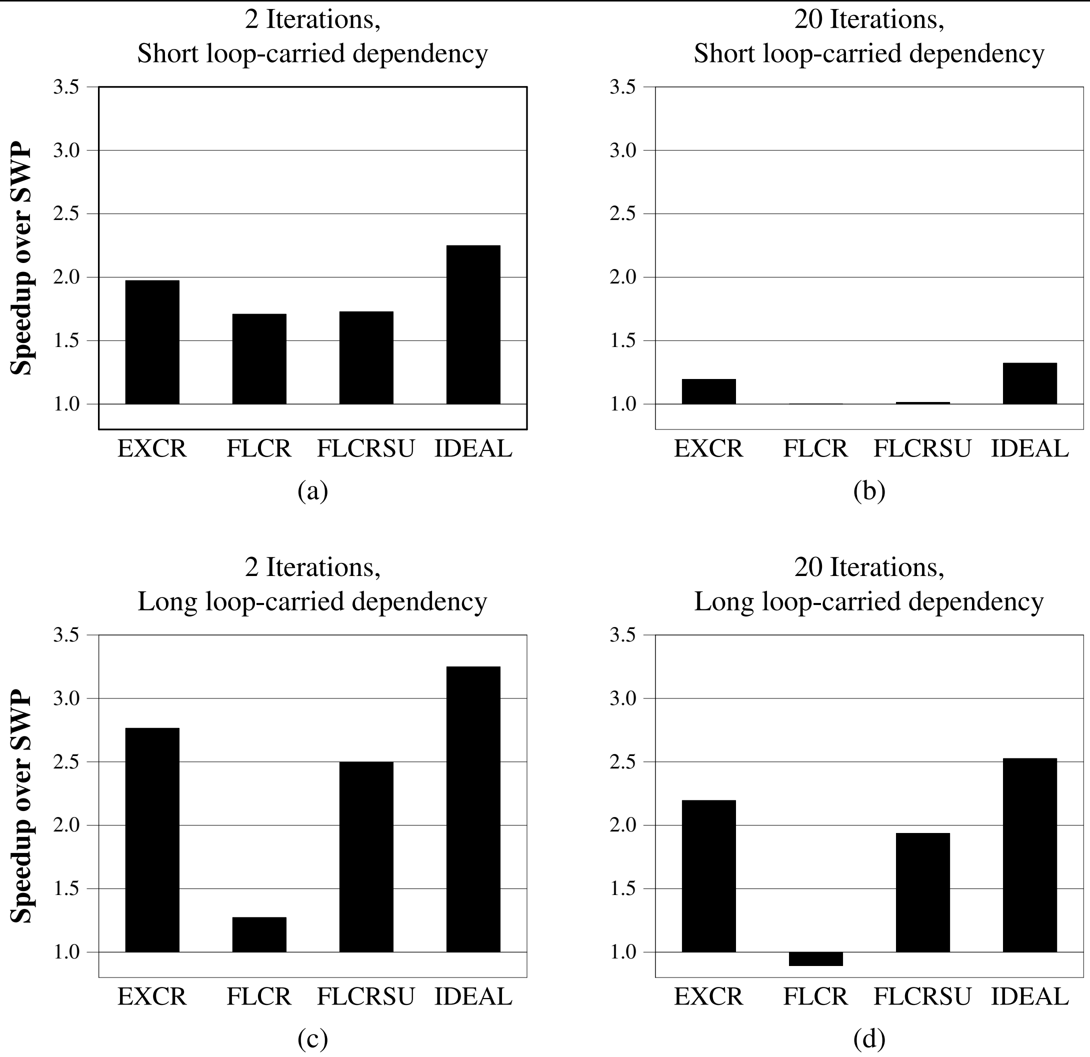


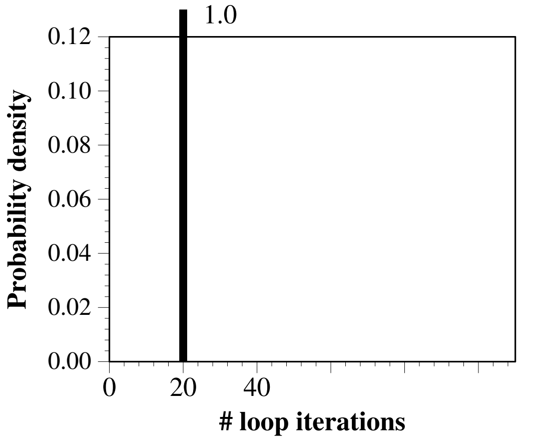
Figure 3.17: These four graphs show the speedup of the conditional routing techniques on the *synthetic_while* benchmark compared to the SWP technique, which simply software-pipelines the while-loop. The speedups are shown for the cases where there are an average of 2 (a,c) and 20 (b,d) iterations of the while-loop per stream element (none of the elements required zero iterations). The graphs also show results for two different while-loops. In the top two graphs (a,b), the software-pipelined schedule for the while-loop is resource limited (i.e., it has enough ILP to keep all the ALUs busy) because the loop has only relatively short loop-carried dependency chains. In the bottom two (c,d) graphs the software-pipelined schedule is limited by a long loop-carried dependency chain. EXCR = Expanded Conditional Routing; FLCR = Flattened Conditional Routing; FLCRSU = Flattened Conditional Routing with State Unrolling (4 virtual processors).

2.5× for 20 iterations per stream element. Notice that the speedup of all three conditional routing methods increase when going from a graph on the right to the left, indicating that they are all effective at reducing the impact of this overhead.

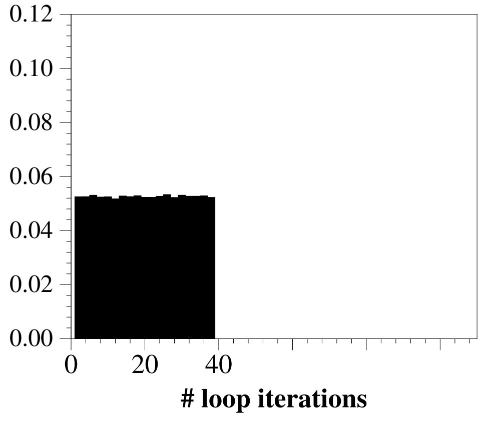
Impact of the Distribution of Loop-Iterations Over Stream Elements

Another important factor that affects the performance of these while-loop methods is the distribution of loop-iterations over the stream elements. For example, Figure 3.18 shows four different cases, all with the same *average* number of iterations per stream element. The previous graphs shown in Figure 3.17 used an input set with no variance, i.e., an input set whose distribution was similar to Figure 3.18(a). The graph in Figure 3.19, however, shows the impact of changing the distribution of iterations on the performance of the techniques we have been discussing (using the same version of the kernel as in Figure 3.17(d)). For reference, we used a batch size of 128 for the results shown in the graph. The speedups, as before, are compared to the SWP technique. Note that the performance of SWP, FLCR, and IDEAL will be the same for each dataset since they are unaffected by variations in the number of iterations per stream element.

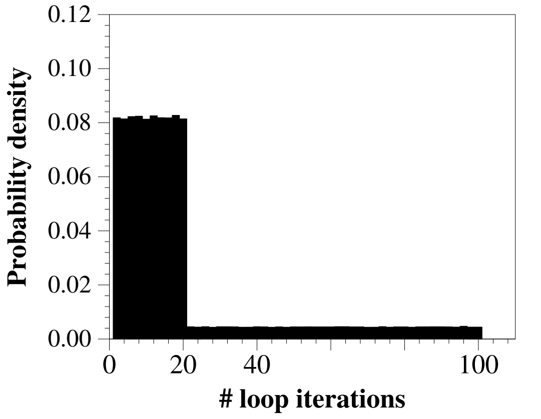
The first set of speedups is shown for the other existing while-loop technique we mentioned in Section 3.1.2. We will refer to the technique as *simple* state-unrolling (SSU), in order to highlight the similarity to our state-unrolling technique. SSU applies state-unrolling without flattening the loop first. This technique simply makes each basic block, even the while-loop basic block, operate on four stream elements at once. This technique will achieve good ILP because of the parallelism between the four element being processed concurrently. Extra control code is added to the inner while-loop to make sure the loop runs as many times as is necessary to complete the maximum number of iterations that is required by the four elements the loop is currently processing. However, unlike FLCRSU, the loop must execute until all virtual processors have finished processing their element. Thus, variance in the number of iterations per stream element can cause performance to degrade sharply. Comparing SSU to FLCRSU is interesting because it shows the overhead of loop-flattening when there is no variance in the input data set, and when the variance increases, the comparison shows the advantage of using loop-flattening to avoid synchronizing at the end of every set of elements the virtual processors operate on.



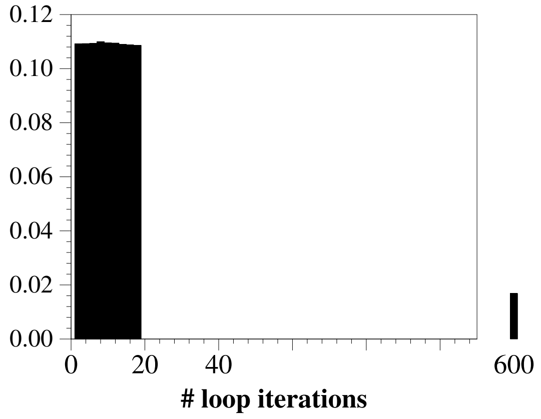
(a) Zero-Variance PDF
Standard Deviation = 0.0 iterations



(b) Uniform PDF
Standard Deviation = 5.5 iterations



(c) Spread PDF
Standard Deviation = 11.1 iterations



(d) Bimodal PDF
Standard Deviation = 38.1 iterations

Figure 3.18: These probability density functions correspond to the four sets of inputs used to study the impact of varying the number of iterations of the while-loop that each stream element executes. In each case, the average number of iterations per stream element is 20 iterations; the variability, however, is different for each case.

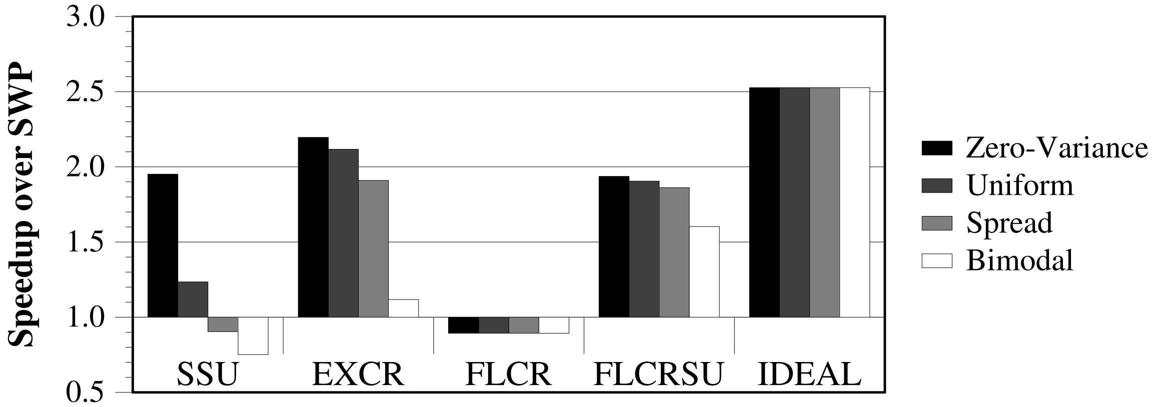


Figure 3.19: This graph shows the performance of various implementations of *synthetic_while* for four different distributions of iterations over the input stream elements (as shown in Figure 3.18). As before, the speedups are shown compared to ORIGINAL. Batch sizes of 128 stream elements were used as input to the various versions of the kernels. EXCR = Expanded Conditional Routing; FLCR = Flattened Conditional Routing; FLCRSU = Flattened Conditional Routing with State Unrolling (4 virtual processors); SSU = Simple State-Unrolling (i.e., applying state-unrolling without flattening the loop first).

The EXCR method has a significant degradation in performance for the "Bimodal" case. This is because it is affected by an overhead that increases with the maximum number of iterations required by any stream element in the input dataset. Since the kernel has to be called once for each iteration of the while-loop, the overhead of priming the software-pipeline for the main-loop of the kernel can be large if only a small number of elements require a large number iterations. The FLCRSU technique, on the other hand, suffers from fewer overheads with increasing variance in the distribution. The only overhead it incurs is due to the fact that all virtual processors may not finish processing their last element at exactly the same time. Note that the overheads for EXCR and FLCRSU that are due to variation in the input set will increase with decreasing batch size, and vice versa.

Performance Summary for Applying Conditional Routing to While-Loops

EXCR on average comes within 87% of the not-to-be-exceeded speedup of IDEAL, and FLCRSU comes within 76%, for the four cases shown in Figure 3.17. When we consider varying distributions of loop-iterations over different stream elements, then we see that

FLCRSU is more impervious to the variance. The performance of SSU is the most susceptible on the other hand, and the performance of EXCR is also quite susceptible to variance. For this benchmark, FLCRSU seems to have the best all-round performance.

For the experiments in this section, four virtual processors were used to perform the state-unrolling. We empirically determined that four was the least we could use while still providing enough ILP to pack the ALU slots efficiently. In general, when selecting the proper number of unrollings to use there is a balance between better packing of the VLIW schedule and increased susceptibility to variance in the dataset. If specific information on the run-time distribution of loop-iterations is known at compile-time, it may be profitable to limit the amount of state-unrolling in favor of more consistent performance across a variety of dataset distributions. Furthermore, the amount of state-unrolling can affect the final code-size of the the kernels, and for that reason we may want to limit the amount of state-unrolling as well. For *synthetic_while*, the code-sizes for the different techniques were: SWP, 184 VLIW instructions; SSU, 118 VLIW instructions; EXCR, 50 VLIW instructions; FLCR, 76 VLIW instructions; FLCRSU, 108 VLIW instructions. The methods with state-unrolling (SSU and FLCRSU) had a high number of instructions. SWP also had a high number of instructions because the `INPUT` and `OUTPUT` basic blocks which come before and after the while-loop, needed to be unrolled. Only EXCR didn't require any unrolling, and as a result it was compiled to less than half the number of VLIW instructions as any of the other methods.

3.3.5 Ordering

As we noted earlier in the chapter, both the EXCR and FLCRSU techniques reorder the final outputs of the kernel. Therefore, for applications that require that the elements in the stream must be kept in a strict order between kernels, we must consider the added overhead of reordering elements in addition to the raw kernel performance. Thus, EXCR and FLCRSU are only useful when either the order that the output stream is produced is not dictated by application requirements, or when the cost of restoring the order of the output stream elements is small enough that it does not negate the performance benefits of the optimizations. As an example, the graphics pipeline application that the *geom_rast* kernel is part of, does not require that the output elements generated by *geom_rast* be kept in any strict order. On the other hand, compression and decompression algorithms require strict ordering of elements at certain points in the application pipeline—hence if a kernel used these techniques we might have to reorder the output elements at some point after the kernel.

This section will explore the different options for reordering the output elements such that the stream is in the same order as if we ran the kernel with no optimizations. We will account for sorting the elements in both `out_stream` and `out_stream2` from the *synthetic_while* kernel. We will first analyze an implemention that executes a full sort on the resulting output streams. We will show that in most cases this reduces the overall performance of the conditional routing techniques considerably. Therefore, in order to combat this performance degradation, we will consider some alternate methods for restoring order.

Preliminaries

Before we delve into the performance of the different sorting techniques, we need to handle a couple of preliminaries. Firstly, there are two output streams in the *synthetic_while* kernel, each one producing a different number of elements. One of the streams, `out_stream`, will contain a single output per input stream element. The second stream, `out_stream2`, will contain an output per iteration of the while-loop, and hence multiple ouputs per input-stream element. We shall call the former a *per-element* output stream, and the latter a *per-iteration* output stream.

Secondly, in order to sort the elements, one field of each data record must be able to act as a key in the sort; if not, then an extra field has to be added by the kernel with the while-loop. One simple way to do that is to keep a counter that is incremented and inserted into each output record. Since we may not know the total number of per-iteration outputs a stream element will generate *a priori*, we may need to sort per-iteration output streams using two keys: the position of the original stream element within the input stream, and the position of this per-iteration output relative to all the outputs generated only by the original stream element.

Full sort technique

The simplest way to maintain ordering is to run the output stream through another kernel, or series of kernels, that will perform a full sort on the output data. Two factors determine how much impact the cost of the sort will have on performance: 1) the number of while-loop iterations per stream element; and, 2) whether we have to sort the per-element output stream and/or the per-iteration output stream. The effect of these two factors on the overall performance of *synthetic_while* is shown in Figure 3.20. We used a batch size of 128 to generate the graph. In order to sort the output streams, we used a merge-sort algorithm. We should note that the cost of the sort varies as $O(N \log N)$, and hence will get relatively more expensive as the batch size increases.

When each stream element executes an average of only two iterations of the while-loop, the cost of sorting `out_stream` reduces the speedup of the conditional routing techniques to roughly 1.4x–1.5x (Figure 3.20(a)). However, with 20 iterations per stream element (Figure 3.20(b)), the cost of the sort is amortized over a larger number of while-loop iterations, and the speedups increase for EXCR and FLCRSU to close to the speedups achieved without the sort. If we have to sort `out_stream2` as well, EXCR and FLCRSU are slower than even SWP, regardless of the number of iterations per stream element (Figure 3.20(c,d)). FLCR however, is unaffected by the ordering constraint since it does not reorder the contents of either output stream.

Split-merge technique

It turns out that if we employ state-unrolling, however, we can avoid the prohibitive cost of a full sort by taking advantage of our knowledge of the specific pattern of reordering that takes place. A possible technique for restoring order without a full sort is to provide separate output streams for each virtual processor, and then merge these streams after the kernel ends. This technique, which we shall call *split-merge*, works well because each virtual processor produces an output stream which will be in order. Thus, recreating the original ordering simply requires only $\lg P_v$ merge passes through the streams, where P_v is the number of virtual processors that were used. This works for both the per-element

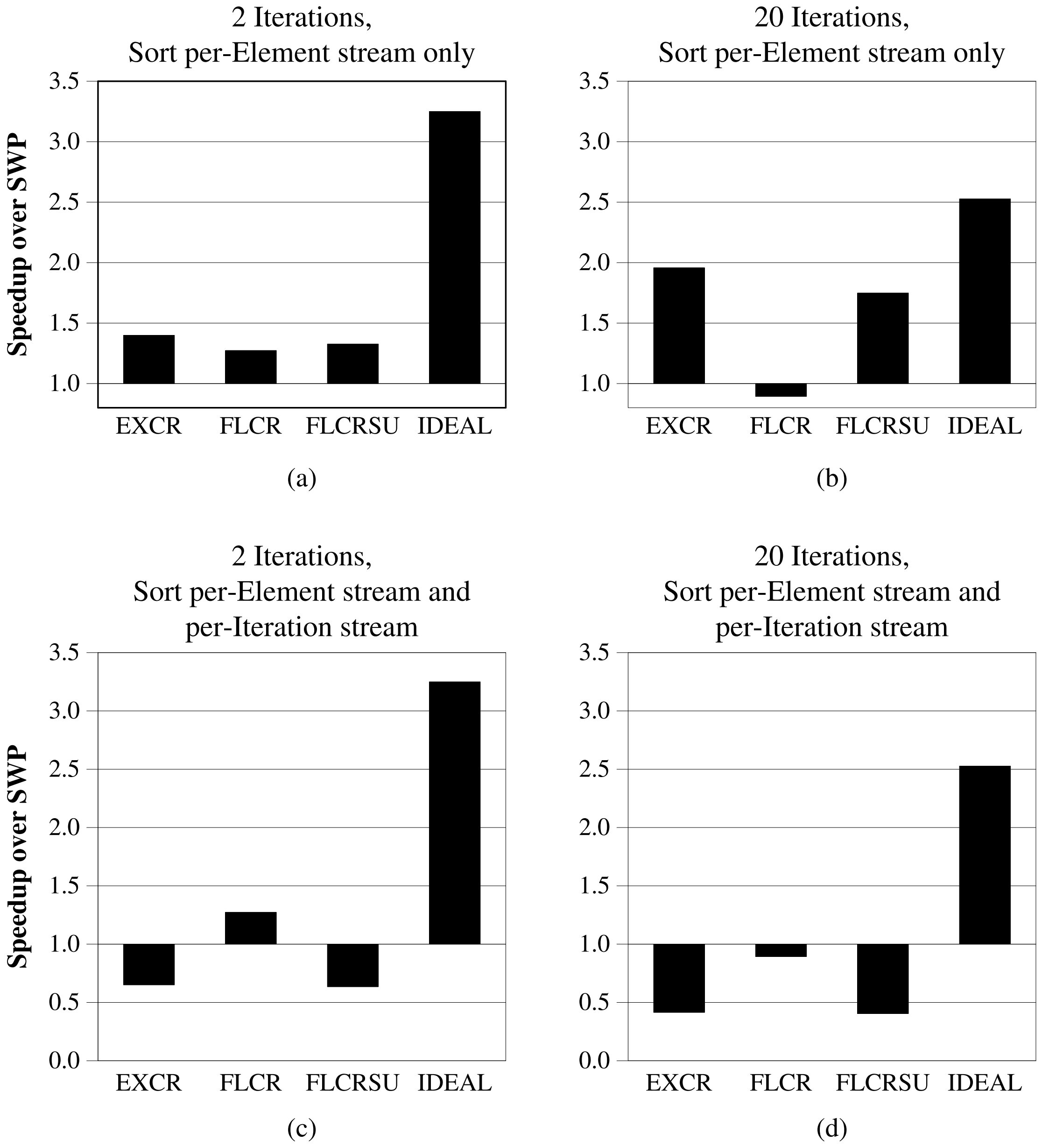


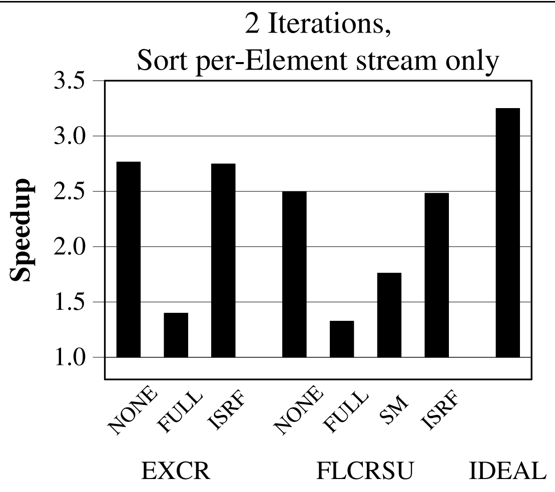
Figure 3.20: These graphs show the impact of having to perform a brute-force sort on the output stream(s) of the *synthetic_while* kernel in order to maintain ordering. We used a batch size of 128 elements, and a merge-sort algorithm to implement the sort. EXCR = Expanded Conditional Routing; FLCR = Flattened Conditional Routing; FLCRSU = Flattened Conditional Routing with State Unrolling (4 virtual processors).

and per-iteration output streams. The performance of the split-merge technique for state-unrolled kernels is shown in Figure 3.21 for FLCRSU. The graphs show that the split-merge technique improves performance over the full sort, but still leaves a lot of room for improvement compared to IDEAL. Additionally, it still causes FLCRSU to have a slowdown compared to the SWP technique for the 20-iteration case with per-iteration stream sorting. Remember, our kernel was state-unrolled four times. Less unrollings will require less overhead for the split-merge technique, and more unrollings will require more overhead.

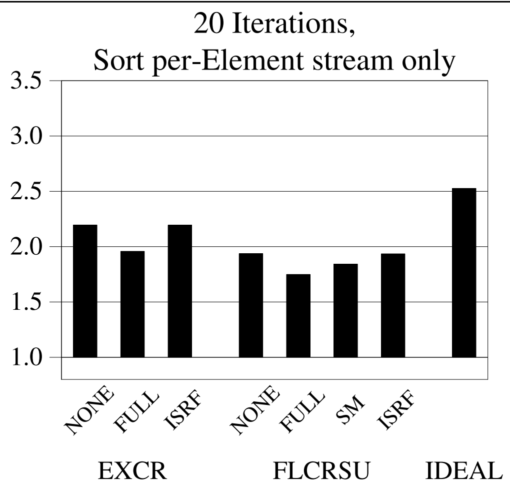
Indexable-SRF (ISRF) technique

By providing more flexible addressing in the SRF, we can improve the performance of FLCRSU even further, and unlike split-merge, also improve the performance of EXCR. Instead of the strictly sequential stream access provided by our baseline SRF, an *in-lane indexable-SRF* [Jayasena *et al.*, 2004] allows random access into streams, at the cost of potentially lower access bandwidth. In other words, with an indexable-SRF we can access streams using an arbitrary pointer into the stream, instead of being limited to the *push* or *pop* accessors. Implementing ordering for a per-element stream is simple with an indexable-SRF. Since we can know exactly the position of every output within a per-element output stream, we can simply write each output element to its correct location once we have finished processing it. This works for both EXCR and FLCRSU.

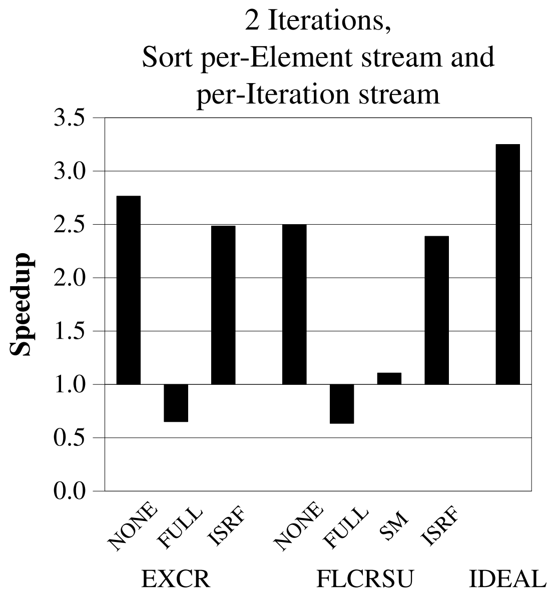
The indexable-SRF can be used in the same way to store the elements of a per-iteration output stream directly to their sorted location, as well. However, this only works if the number of iterations each stream element requires is known at the start of the loop. This may not be the case for many interesting kernels, however. We can still deal with this harder case, but it will take a couple of extra passes and keeping track of some extra state. In particular, instead of storing the outputs directly to their correct location, a stream is generated that records the number of per-iteration elements produced by each input stream element once it completes the while-loop. We will refer to this stream as `count_stream`. We use the indexable capability so that `count_stream` is in order at the end of the kernel, just as we would do with the per-element outputs. After completing the original kernel and generating `count_stream`, a running sum is calculated by a second kernel and stored to `sum_stream`. This new stream gives us the location in the final per-iteration output



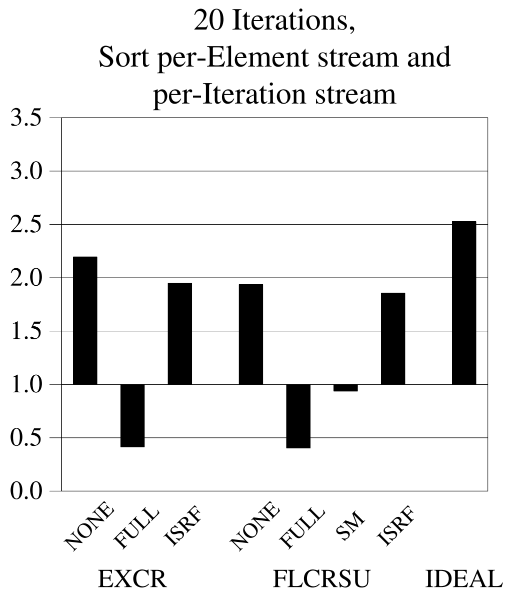
(a)



(b)



(c)



(d)

Figure 3.21: These graphs show the results of having to maintain ordering for *synthetic_while*, but using more efficient techniques than a brute-force sort. EXCR = Expanded Conditional Routing; FLCR = Flattened Conditional Routing; FLCRSU = Flattened Conditional Routing with State Unrolling (4 virtual processors); NONE = no sorting of any output streams; FULL = output stream(s) are sorted using a brute-force full sort; SM = uses split-merge technique for sorting; ISRF = uses an indexable-SRF for sorting. Note that the ISRF results for (c,d) are only valid if we can determine the number of iterations required for each stream element before we start the while-loop.

stream of the first output of each input stream element. Then sorting simply requires a pass over all the per-iteration outputs, using the first key as a lookup into `sum_stream`. A third kernel then adds the value of the second key to the value read from `sum_stream` in order to obtain the final position of the element in the output stream.

According to the graphs in Figure 3.21, if we have an indexable-SRF available on the architecture, then the speedups achieved are quite close to the speedups without sorts for both EXCR and FLCRSU across the board. Of course, the main disadvantage is the cost of the extra hardware. Jayasena, *et al.* reported that this type of SRF indexing increased the area of the SRF by 11%, and the area of an entire eight-cluster processor by 1.5%. Their study, however, found that most of the additional area was not due to the extra control logic, but instead for the extra addressing logic required for accessing individual words instead of entire four-word lines from each SRF bank. So, if we wanted to reduce the area overhead of an indexable SRF, we could instead read a whole line even when only a single word is requested. This would eliminate almost all the extra addressing logic, but would reduce the achieved random access bandwidth of the SRF significantly. We used this lower bandwidth version for our results.

Sorting via the memory system

A final possibility is to sort streams through the memory system. The performance impact of doing this varies from application to application, depending on how burdened the memory system is to begin with, and depending on how well the application can tolerate the latency of sending the stream to memory and having to potentially load it again for the next kernel. We do not analyze the performance of this sorting technique here.

3.3.6 Conditional Routing Summary for While-Loops

SWP is an efficient way to implement while-loops within the kernel main loop only if both of these conditions hold true: the number of iterations per stream element is not small, and the loop schedule is not limited by a long loop-carried dependency. Thus, if either or both of these conditions do not hold, then the techniques we have introduced in this chapter are important in order to improve the ILP of the kernel schedules. In particular, we introduced expanded conditional routing (EXCR), flattened conditional routing (FLCR), and state-unrolling (FLCRSU). We found that the best performing options were EXCR and FLCRSU. However, both expanded conditional routing and state-unrolling produce an output stream that can be in a different order than the original version of the loop. We showed that if ordering is a necessary constraint imposed by the application, then the addition of an indexable-SRF can eliminate much of the overhead of restoring the proper order of the output stream.

A summary of all the techniques studied in this chapter is provided in Table 3.2.

Table 3.2 Summary of while-loop conditional techniques. Each column is dedicated to a particular property of the kernel or application, and each row indicates how a technique handles each property.

	Small number of iterations per stream element	Long loop-carried dependency in while-loop	High variance in number of iterations per stream element	Ordering of the output stream must be preserved
SWP	Overhead for priming software-pipeline	Decreases performance	Does not affect performance	No additional processing is necessary
SSU	No extra overhead	Does not significantly affect performance	Poor performance on "Bimodal" distribution	Can use full sort, split-merge, or indexable-SRF
EXCR	No extra overhead	Does not significantly affect performance	Poor performance on "Bimodal" distribution	Can use full sort, or indexable-SRF
FLCR	No extra overhead	Decreases performance	Does not affect performance	No additional processing is necessary
FLCRSU	No extra overhead	Does not significantly affect performance	Performance is fairly resistant to variance	Can use full sort, split-merge, or indexable-SRF

3.4 RENDER Performance

In order to study the effects of the new techniques in this and future chapters on a real kernel and application, instead of just on synthetic benchmarks, we will use the RENDER application. RENDER generates the first frame of the SPECviewperf 6.1.1 advanced visualizer benchmark with lighting and blending disabled and all textures point-sampled from a 512×512 texture map. The image was rendered into a 720×720 framebuffer window. In particular, we will focus on the first kernel in this application, *geom_rast*, which we introduced earlier in Section 2.3.1. Figure 3.22 shows a stream and kernel diagram of the entire RENDER application, and illustrates how the *geom_rast* kernel fits into the application. We will give performance results for the entire application, but will not discuss the details here; instead we will only go into depth for the *geom_rast* kernel in order to keep the discussion focused. More details on the entire RENDER application can be found in other references [Owens *et al.*, 2000; 2002; Owens, 2002]. Also, while the RENDER application does contain conditionals in kernels other than *geom_rast* [Kapasi *et al.*, 2000], we will not be experimenting with them in this thesis.

In order to zoom in on the *geom_rast* kernel, a graphical view of its control-flow is shown in Figure 3.23. This basic block diagram also indicates the values of some of the parameters of the kernel that we have identified are important in deciding which conditional technique is the most appropriate and efficient for each conditional statement. These include the average probability that each if-statement is taken or the average number of iterations that the while-loop executes, the number of operations in each basic block, and the number of words of live state that are transferred into and out of the basic blocks. Note, we have not indicated those pieces of live state, such as constants, that are invariant across all iterations of the main-loop. Also, the average branch percentages and loop-iteration numbers are only for the particular scene which we rendered; a different scene or different viewpoint of the same scene would have different values.

In order to assess the impact of conditional routing on the VLIW schedule of *geom_rast*, in Table 3.3 we show the execution times of for ORIGINAL, COMMON, PREDICATION, and CONDITIONAL ROUTING. The COMMON implementation optimizes the kernel for

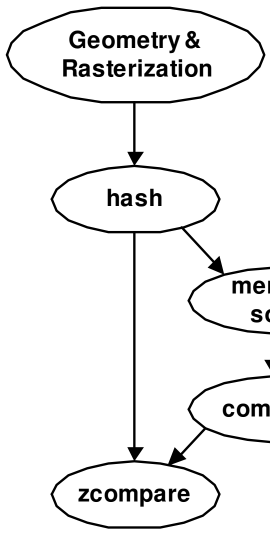


Figure 3.22: Polygon rendering application stream and kernel diagram. We will be applying the conditional techniques we have studied in this dissertation to the first kernel, which performs the geometry and rasterization work for the pipeline. We will refer to this kernel as *geom_rast*.

the branch statistics shown in Figure 3.23 using a combination of speculation and loop-unrolling, and improves the kernel execution time by $1.8\times$. We manually searched the space for the best basic blocks to speculatively execute, and the amount of unrolling to use. PREDICATION is a combination of predication and software-pipelining. It eliminates all the conditional branches from within the kernel schedule, by speculatively executing all control flow paths (for while-loops, this amounts to using loop-flattening). However, with so many nested conditionals, and especially with a while-loop nested so deeply, this technique is bound to be slow. This is because even though it can efficiently software-pipeline the resulting schedule, there are so many unnecessary operations that are executing every loop iteration. As a result, for this kernel, PREDICATION actually does worse than the original, unoptimized kernel. The CONDITIONAL ROUTING result uses a combination of predication, conditional routing, and software-pipelining to gain the best speedup of $2.3\times$. For CONDITIONAL ROUTING, we implemented conditional routing and predication to the various conditional blocks using the technique selection guidelines we outlined in the Section 3.2.3. The while-loop was implemented with FLCRSU.

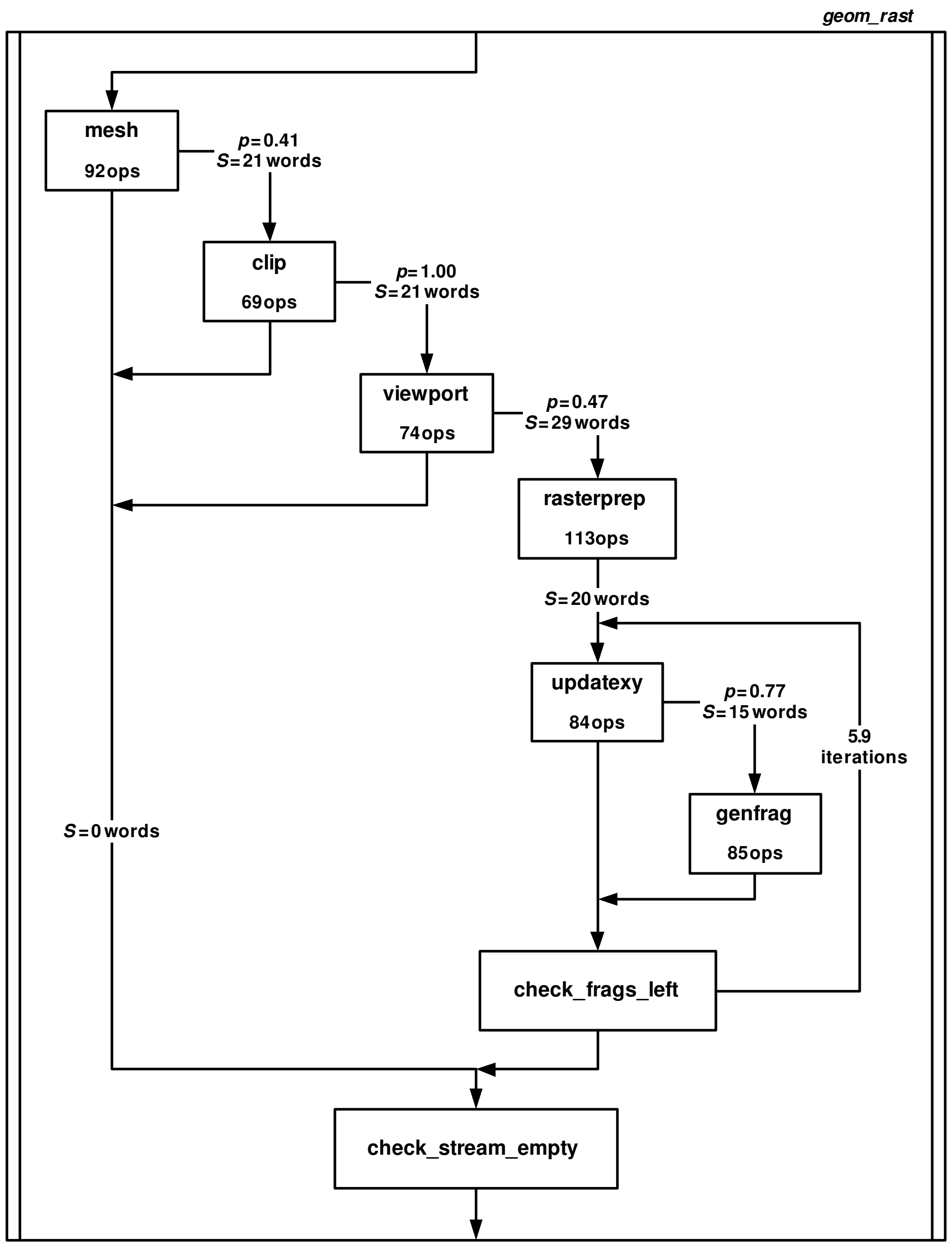


Figure 3.23: Basic block diagram for the *geom_rast* kernel. The basic block names correspond to those used in Kernel 2.1. Indicated on the diagram are the number of arithmetic operations in each basic block, the amount of live state that is transferred along each control flow path, as well as the probability that each control flow path is taken.

Table 3.3 Performance of conditional techniques for the *geom_rast* kernel and the entire RENDER application on a VLIW processing cluster.

	geom_rast		RENDER	
Method	Cycles	Speedup	Cycles	Speedup
ORIGINAL	14,707,868	1.0	19,781,539	1.0
COMMON	8,204,487	1.8	13,278,158	1.5
PREDICATION	19,654,392	0.75	24,728,063	0.8
CONDITIONAL ROUTING	6,499,555	2.3	11,601,738	1.7

The difference in relative execution times are less, however, when you take into account all of the kernels in the application, since there are not as many conditionals that stand to benefit from conditional routing. For the entire application, the speedup due to CONDITIONAL ROUTING is $1.7\times$. Interestingly, the input batch size we could handle with CONDITIONAL ROUTING didn't decrease, despite the additional intermediate streams that need to be allocated in the SRF. This is because the SRF allocation was almost entirely determined by the number of fragments, which are generated by *geom_rast* and used by all the following kernels in RENDER. Thus the smaller intermediate streams didn't impact the overall possible batch size. Also, code size for the different versions of the kernels matches what we would expect. ORIGINAL (320 instructions) and COMMON (371 instructions) are large, while PREDICATION (157 instructions) and CONDITIONAL ROUTING (221 instructions) are smaller. This is because ORIGINAL has very poor ILP which lengthens the schedule and resulting code size. COMMON increases ILP via speculation and more loop-unrolling. This fills in some of the empty ALU slots, but at the expense of larger code size because of extra fix-up code and extra iterations of unrolling. PREDICATION uses software-pipelining only and hence has the smallest code size. CONDITIONAL ROUTING uses software-pipelining as well as selective use of loop-unrolling, and as a result has a code size in the middle of the range.

In summary, CONDITIONAL ROUTING achieves the best schedule on our five-ALU VLIW cluster out of all the techniques. It is instructive to circle back to the result we presented as motivation at the start of the chapter. We showed that existing techniques were only able to achieve about 26% of peak utilization of the ALUs in our VLIW processing cluster. Conditional routing increased this utilization for *geom_rast* to 33%. While this represents a $1.25\times$ speedup, clearly there is still room for much improvement in order to reach the 80%–90% utilization that kernels without conditionals achieve. We will consider suggest possibilities for further improvements in Chapter 6.

3.5 Dynamic Techniques

It is worthwhile to contrast our treatment of static conditional techniques in this chapter with dynamic techniques. There have been a myriad of dynamic techniques proposed, and many are commonplace in general purpose processors today. The most important ones fall under the category of *branch prediction*. The idea behind branch prediction is essentially to predict which way a branch will evaluate to allow the instructions from the predicted control-flow path to be scheduled before the branch direction is known for sure. Good introductions to dynamic branch prediction are provided in [Smith, 1981] and [McFarling and Hennessy, 1986]. Some more recent schemes are described in [Yeh and Patt, 1991], [Eden and Mudge, 1998], and [Seznec *et al.*, 2002]. These techniques are useful only when combined with dynamic instruction issue techniques. However, the combination of dynamic issue logic (as opposed to static VLIW) and dynamic branch predictors can add a significant amount of area and power to a design. The reason that general purpose processors need to use these techniques is because they are targetting applications that do not have the data-parallel nature that media processing applications have. Our goal in this chapter has been to avoid these options in an effort to keep the stream processing cluster small and cool.

Chapter 4

Improving SIMD Performance with Conditional Routing

The previous chapter explored the benefits of conditional routing for for a *single* VLIW processing cluster. In this chapter we will extend our analysis to explore the performance of kernels with conditional statements running on stream processors with *multiple* clusters, under SIMD control. Earlier, in Section 2.2.2, we motivated the use of the SIMD execution model via three main points: 1) SIMD hardware is more efficient; 2) the SIMD model is simpler; and 3) many applications are regular enough that the restrictions made by the SIMD model do not affect their performance. However, we alluded to the fact that kernels with conditionals suffer large efficiency losses on SIMD machines. In this chapter we will study how to remedy this. In particular, we will show that the COMMON technique performs quite poorly on SIMD machines, but that PREDICATION and CONDITIONAL ROUTING both are unaffected by the restricted SIMD model of execution. However, when combined with the advantages in VLIW efficiency from last chapter, especially as compared to PREDICATION, CONDITIONAL ROUTING has a clear advantage over the other techniques on a SIMD machine.

Table 4.1 1 to 8 cluster speedups existing conditional techniques for the *geom_rast* kernel on a stream processor with SIMD clusters.

Method	1 to 8 Cluster Speedup
ORIGINAL	2.5
COMMON	3.3
PREDICATION	4.2

4.1 Motivation

The problem with executing conditionals on SIMD machines is that every cluster has to follow the union of the control-flow paths required by each cluster. This inefficiency limits the possible speedup of an eight-cluster SIMD machine over a single-cluster machine on kernels with conditionals. We would expect this speedup for a data-parallel kernel such as *geom_rast* to be close to $8\times$. However, the actual speedups, shown in Table 4.1, are much lower in reality. ORIGINAL only improves by $2.5\times$. COMMON improves by only $3.3\times$, which is less than half of the maximum possible. Statistical differences between the required processing for the element in each cluster causes the performance to be so poor for these two methods. Essentially, the amount of time to process each element is lengthened by the time spent executing unnecessary basic-blocks that other clusters need to execute.

On the other hand, PREDICATION completely removes all the branches in the kernel so that all clusters always follow the same control-flow path through the main loop. This is perfectly suited for SIMD execution. The performance of PREDICATION is still degraded though, unfortunately, because each element will require a different number of iterations after loop-flattening, and hence some clusters will finish their *entire batch* before others. However, there are more elements over which to average out these statistical differences in PREDICATION compared to the other techniques—i.e., over an entire batch instead over a single element. So, if we assume the required control-flow is independent from one element to the next, by the law of large numbers we would expect the impact on performance to be smaller for PREDICATION. While the independence assumption is not exactly accurate for the input dataset we used for the *geom_rast* kernel, the speedup of PREDICATION is still higher, at $4.2\times$. As a side-note, we will attack the problem of reducing the impact of the

Table 4.2 Comparison of existing conditional techniques for the *geom_rast* kernel on a stream processor with eight SIMD clusters.

Method	Speedup over ORIGINAL
COMMON	2.4
PREDICATION	1.3

load-imbalance across an entire batch of elements in the next chapter.

While the SIMD speedup is better for PREDICATION, if we remember back to the last chapter, we showed that the efficiency on each individual VLIW processing cluster for *geom_rast* is quite low for PREDICATION. In fact it was lower than that of ORIGINAL. Thus the absolute performance on an eight cluster machine is actually still almost two times slower for PREDICATION than for COMMON, as is seen in Table 4.2. In summary, we are in the unfortunate position that COMMON can produce reasonably well-packed VLIW schedules (within 25% of the best achieved by CONDITIONAL ROUTING for *geom_rast*), but does not offer good efficiency on multiple SIMD clusters. On the other hand, PREDICATION offers better global SIMD speedup when adding more clusters, but this is not very useful because the base performance on each individual VLIW cluster is poor. This motivates us to find a better a solution; we will show in the rest of the chapter that conditional routing is such a solution.

4.2 Performance

This section will analyze the performance of our if-statement and while-loop benchmarks (*synthetic_if* and *synthetic_while*) with a two-step approach. For each benchmark, we will first compare the speedup of each technique when going from a single-cluster machine to an eight-cluster SIMD machine. This will show that conditional routing is not adversely affected by the restricted (lock-step) execution model of a SIMD machine. Second, we will compare the overall performance of conditional routing on an eight-cluster SIMD machine. This comparison will roll-up the impact of efficient ALU packing on each VLIW cluster and the impact of good performance with SIMD control into one number.

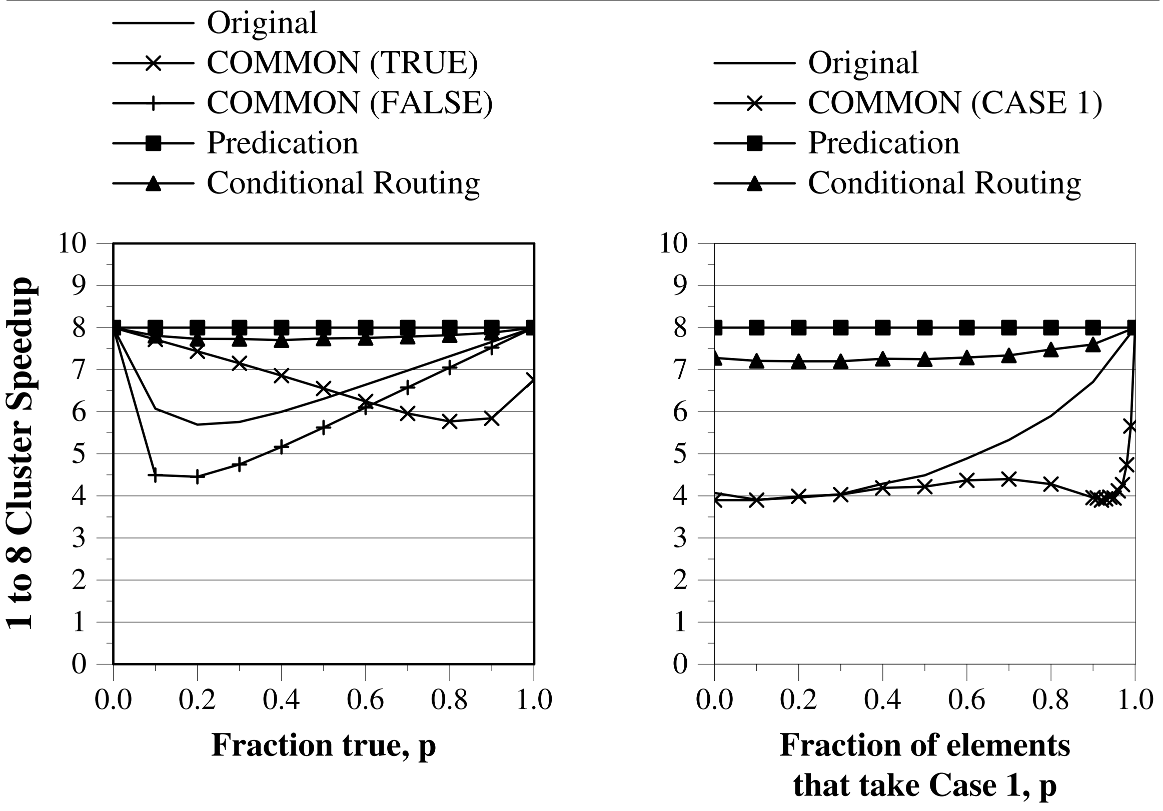
4.2.1 If-Statements

As we saw earlier with *geom_rast*, the performance of both the ORIGINAL and COMMON techniques on *synthetic_if* scales dismally from a single cluster to eight SIMD clusters (Figure 4.1). The scaling for the COMMON techniques is particularly bad because mis-peculation causing *every* cluster to execute the fix-up code, not just the original offending cluster. The penalty is bad for *synthetic_if*, but it gets even worse on *synthetic_case*. In fact, the COMMON technique only has a speedup of around $4\times$ for almost all values of p on *synthetic_case*.

Predication has been the time-honored method to counter the problem exhibited by ORIGINAL and COMMON. Its use in this regard dates back to the Solomon machine [Slotnick *et al.*, 1962]. Another notable machine with hardware predication support is the Cydra 5 supercomputer [Rau *et al.*, 1989], although they called it *directed-dataflow execution*. Also, almost all vector processors since the Cray-1 [Russell, 1978] support predication via masked operations for the same purpose. The superior performance of PREDICATION is indicated by an $8\times$ speedup across the board for both benchmarks.

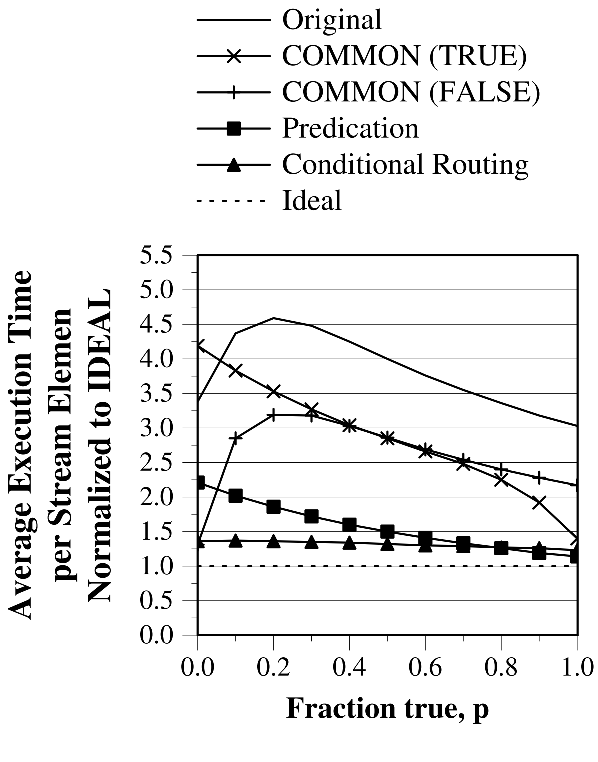
CONDITIONAL ROUTING however does almost as well. Small performance degradations are caused by load-imbalance across the entire dataset, since each cluster may not have the exact same number of elements that need to execute the body of the if-statement due to statistical variations. However, these slight performance degradations become insignificant when we compare overall performance in Figure 4.2.

As expected, the performance of ORIGINAL and the variants of COMMON are quite poor on the eight-cluster SIMD machine. We can see why PREDICATION has been the technique of choice on SIMD processors. For example, compare the graphs Figure 3.7 and Figure 4.2(b). PREDICATION is the worst technique for *synthetic_case* on a single-cluster machine. However, the performance of the other techniques has decreased sufficiently (as indicated in Figure 4.1) that PREDICATION is now the superior existing technique for a large fraction of the values of p on the same kernel. However, CONDITIONAL ROUTING is able to combine good VLIW schedules with good SIMD scaling, and as a result executes as fast or faster than PREDICATION for almost all values of p on both graphs. For these benchmarks, CONDITIONAL ROUTING is clearly a solution to the problem we introduced

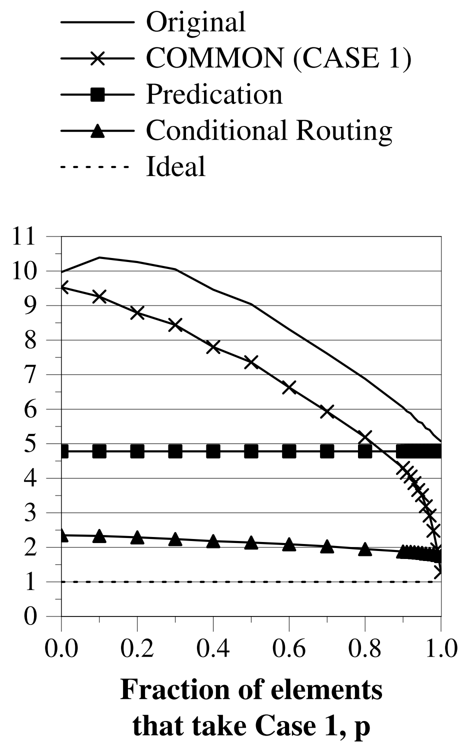


(a) synthetic_if (b) synthetic_case

Figure 4.1: Impact of SIMD execution model on if-statement techniques. The graph shows the speedup going from a single-cluster to an eight-cluster SIMD machine. The elements within an input batch were independently and randomly generated (while satisfying the constraint on the value of p). $W_{\text{body}} = W_{\text{case}} = 32$ operations; $W_{\text{input}} = W_{\text{output}} = 17$ operations; $S = 2$ words.



(a) synthetic_if



(b) synthetic_case

Figure 4.2: Comparison of if-statement techniques on an eight-cluster SIMD machine. The elements within a batch were independently and randomly chosen (while satisfying the constraint on the value of p). $W_{\text{body}} = W_{\text{case}} = 32$ operations; $W_{\text{input}} = W_{\text{output}} = 17$ operations; $S = 2$ words.

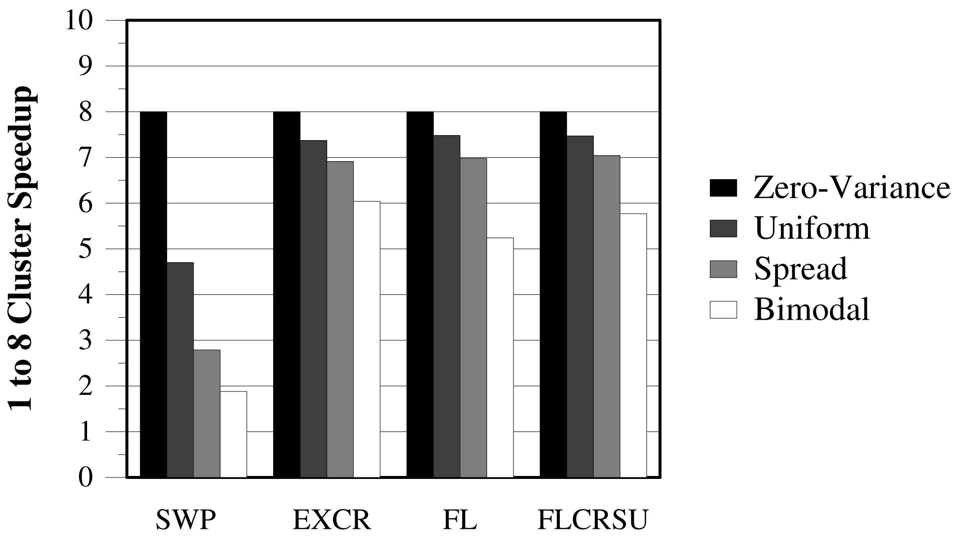


Figure 4.3: Impact of SIMD execution model on while-loop techniques. The graph shows the speedup going from a single-cluster to an eight-cluster SIMD machine. The version of the kernel used is the same as in Figure 3.17(d), while the four probability distributions used are the same as in Figure 3.18.

at the beginning of the chapter.

4.2.2 While-Loops

The SIMD execution model degrades the performance of while-loops because *all* clusters must execute a while-loop as long as *any* cluster still has not exited the loop. With the SWP technique, which we discussed in Section 3.3, every stream element will require early clusters to idle while the cluster with the largest element finishes. As shown in Figure 4.3, this limits the speedup of SWP to only about $2\times$ for the "Bimodal" distribution.

In order to get around this problem, previous researchers have employed the loop-flattening (FL) technique. As we discussed earlier, loop-flattening allows many SIMD processors to execute while-loops on their local set of data without requiring a synchronization between all the processors at the beginning and end of every element. Loop-flattening achieves its goal by transforming a nested loop into a single loop that allows every processor (or cluster) to execute the same code, regardless of whether they were starting the first iteration for a new element or whether they were starting another iteration for an old stream element. Statistical variations are then averaged over the whole batch, which reduces the amount of idling for clusters. The benefits of loop-flattening were first demonstrated in [Sherryl and Pappas, 1990]. They applied it to the calculation of the Mandlebrot Set on a MasPar MP-1 processor. Van Hanxleden, *et al.* generalized this technique, and also coined the term "loop-flattening" [von Hanxleden and Kennedy, 1992]. On our benchmark, the benefits of loop-flattening are clear since it achieves a speedup of at least $7\times$ on all distributions except "Bimodal," on which it achieves a speedup of $5\times$. Our new techniques (EXCR and FLCRSU) also achieve similar speedups, indicating that they are well suited for the SIMD execution model as well.

However, a clear difference between the original loop-flattening technique and our new techniques arises when we look at the combined effect of VLIW efficiency and SIMD efficiency. In Figure 4.4, the EXCR and FLCRSU techniques, as expected, combine their superior single cluster performance and superior SIMD scaling to achieve speedups roughly between $2\times$–$5\times$. Simple loop-flattening (FL), however, never achieves a speedup of more than $2\times$. By comparing the results for simple loop-flattening (FL) and flattened conditional routing with state-unrolling (FLCRSU) in the two graphs in Figure 4.3 and Figure 4.4, we can see that loop-flattening provides FLCRSU with good SIMD performance, while our additions of conditional routing and state-unrolling provide it with efficient VLIW schedules.

4.3 RENDER Performance

If we circle back to the *geom_rast* kernel, Table 4.3 combines the results for existing techniques which we showed at the start of the chapter, and also adds the new results for CONDITIONAL ROUTING. We can see from the table that CONDITIONAL ROUTING does as well as PREDICATION in terms of speedup from one to eight clusters. Also, from last chapter (Section 3.4) we remember that CONDITIONAL ROUTING did better than COMMON on a single VLIW cluster. Therefore, conditional routing is our solution to combining the best of both worlds, providing a $1.6\times$ speedup over the next best existing technique on *geom_rast* on an eight-cluster SIMD machine.

There is still much room for improvement, though, since CONDITIONAL ROUTING is

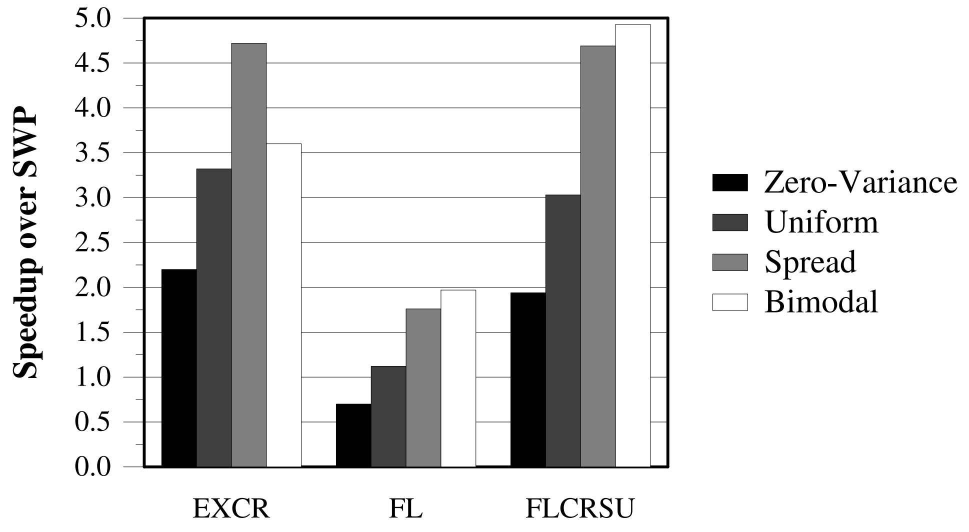


Figure 4.4: Comparison of if-statement techniques on an eight-cluster SIMD machine. The version of the kernel used is the same as in Figure 3.17(d), while the four probability distributions used are the same as in Figure 3.18.

still only achieving half the maximum speedup possible with eight clusters instead of one. As we will see in the next chapter, the one to eight cluster speedups of both *geom_rast* and the entire RENDER application can be improved by distributing work across all the clusters so as to evenly balance the load.

4.3.1 MIMD Comparison

Earlier we motivated the use for a SIMD execution model because it is more hardware efficient and because it is less complex. However we noted that MIMD organizations did have a performance advantage, since SIMD control flow limited performance: either clusters were forced to all follow the union of all required control-flow paths, or extra operations diluted ALU efficiency if we used predication instead. It turns out that, for *geom_rast*, a MIMD solution would have a speedup of 36% over the best existing algorithm (COMMON in Table 4.3).[1]

[1]The kernel immediately following *geom_rast* requires synchronization between all the clusters, so we assumed that all clusters would synchronize at the end of *geom_rast*. Hence the MIMD version still suffered from load-imbalance since some clusters finished executing *geom_rast* on their local batch before other clusters. This is why its one-to-eight cluster speedup is only $4.5\times$ instead of $8\times$.

Table 4.3 Performance of conditional techniques for the *geom_rast* kernel and the entire RENDER application on a stream processor with eight clusters.

Method	Machine type	1 to 8 Cluster Speedup		Normalized Comparison	
		geom_rast	RENDER	*geom_rast*	RENDER
ORIGINAL	SIMD	2.5	2.7	1.0	1.0
COMMON	SIMD	3.3	3.4	2.4	1.9
PREDICATION	SIMD	4.2	4.1	1.3	1.2
CONDITIONAL ROUTING	SIMD	4.2	3.8	3.8	2.4
COMMON	MIMD	4.5	4.0	3.2	2.2
CONDITIONAL ROUTING	MIMD	4.2	3.8	3.8	2.4

However, we now have a new alternative: conditional routing. It is interesting to see that CONDITIONAL ROUTING actually performs better than the best existing MIMD solution. Since CONDITIONAL ROUTING has the same performance on both the SIMD and MIMD machines, there is actually no benefit anymore to adding the MIMD hardware, for this application at least.

4.4 Summary

The restricted control-flow of the SIMD execution model affects any conditional techniques that do not remove all branch instructions from the loop body, such as speculation for if-statements, and software-pipelining for while-loops. While predication does eliminate branches, we have previously shown that it can be inefficient when the block size is large or when p is small. Conditional routing was the only technique that performed well under SIMD control without suffering the overhead of executing extraneous control-flow paths. However, conditionals still cause a problem for every technique we have discussed on machines with multiple clusters. In particular, each cluster has a different amount of work to do in their SRF bank, and hence some clusters may have to idle while others are still processing. The next chapter addresses this problem.

Chapter 5

Improving Load-balance with Conditional Streams

In the last chapter we saw that conditional routing is an important technique for achieving efficient performance on a stream architecture with SIMD clusters. However, we noted that after applying conditional routing, there was still load-imbalance between the clusters. This is because there is a different amount of work to process the entire input data stream in each cluster for each kernel. Conditional streams, which we will propose and evaluate in this chapter, are a mechanism for reducing the cost of this load-imbalance. The performance results will show that the conditional stream mechanism can improve the performance of the conditional routing technique up to 1.5x on the worst case situations for our set of micro-benchmarks.

5.1 Motivation

In order to motivate our efforts to improve load-imbalance, let's look back at the *geom_rast* kernel again. In the last chapter, we saw that even though the PREDICATION and CONDITIONAL ROUTING techniques don't suffer any slowdowns due to the restricted control-flow of SIMD, the speedup of *geom_rast* was only $4.2\times$ when going from a single-cluster to an eight-cluster machine. This is in comparison to between $6\times$–$8\times$ for kernels without conditionals [Rixner *et al.*, 1998; Kapasi *et al.*, 2002a]. The difference comes from load-imbalance in the dataset in each cluster's SRF bank. Even worse, since *geom_rast* produces a data-dependent number of outputs in each cluster, this causes load-imbalance further down the application pipeline as well. As a result, the remainder of the kernels in the RENDER application only had a speedup of around $4\times$ for these two methods, instead of closer to $8\times$ which is what we would expect. Thus, the problem we are attacking in this chapter is to improve the efficiency of applications with conditionals by distributing the workload among all the clusters, a process which is referred to as *load-balancing*.

5.2 Existing Techniques for Improving Load-Balance

This type of load-balancing can sometimes be achieved on a case by case basis via manual programmer intervention. Furthermore, intelligent partitioning at compile time might also improve load-balance. However, we would like to find a more general solution, that adapts dynamically so that it can successfully distribute the load for any set of input elements, even if the exact distribution is not known until run-time.

One such general technique is a global task-queue. Processing clusters must access a global data structure in order to obtain more work, and then must transfer the data from the global queue to their local memory. A notable technique that falls into this class of load-balancing is self-scheduling [Tang and Yew, 1986; Fang *et al.*, 1990]. While this clearly solves the load-imbalance issue, it does raise some other problems.

The most important issue is how to implement the global data structure using the banked SRF of a stream processor. We could implement the SRF as a global memory that is accessible by any cluster instead of banking it, but that would reduce its efficiency too much. Instead we would like to keep the SRF banked in order to retain its high bandwidth. Our solution is to stripe the queue across the banks, in order to allow consecutive accesses to the queue to be served by different SRF banks. We will show how to manage such a load-balancing scheme in this chapter.

Another important question stems from that fact that for the best load-balancing, the clusters should access the global data queue frequently (i.e., the granularity of work assigned to each requesting cluster should be small). However, this results in a high overhead, because too much time will be spent in the software routines that arbitrate who gets access to the queue and when. On a SIMD machine, this overhead can be particularly bad, because due to the lockstep execution model, many clusters may try to access the queue all at the same time. Previous researchers have suggested ways to combat the high access overhead by reducing the frequency with which clusters must access the global task queue. The basic idea is to find a way to allow each cluster to grab more work each time it accesses the queue, but without significantly affecting the load-imbalance. A good example is guided self-scheduling [Polychronopoulos and Kuck, 1987]. However, in this chapter, we present a different option that utilizes a hardware mechanism to access the global task queue. The idea is that using a hardware mechanism instead of a software routine to access the queue will reduce the overhead of accessing the queue. If the overhead is small enough, then the clusters can access the queue at a granularity of a single stream element, which gives the best load-balancing results.

The next section will first present the operation of the conditional streams mechanism—i.e., how the global task queue is striped across the clusters. Conditional streams are designed to be used in conjunction with conditional routing, and so the following example will show that combination. However, conditional streams can also be applied to other techniques as well, and we will analyze the performance of such combinations later. After we describe how they reduce load-imbalance, we will describe the hardware used to support conditional streams as well as their performance.

5.3 Conditional Streams

The main reason conditional routing causes load-imbalance between clusters is that streams are produced or consumed using conditional routing primitives. As we recall (Section 3.2), these conditional routing primitives are stream accessors that are modified to accept a case variable, whose value is determined locally in each cluster. In their basic form, this primitive will allow a cluster to produce or consume a stream in its own local SRF bank at a different rate from other clusters. Load-imbalance arises for a kernel when, for example, its input stream was produced by a previous kernel that used conditional output primitives, and each cluster had generated a different number of values in the stream in their local SRF bank.

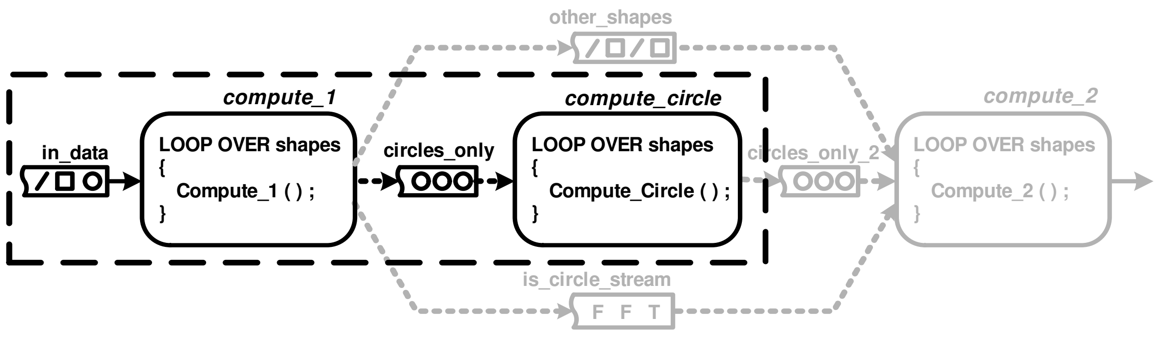
This situation is depicted in Figure 5.1, which shows the first two kernels from the original *shapes* example we used to illustrate conditional routing in Section 3.2. Notice that for the example dataset shown in the figure, it will take the second kernel, *compute_circle*, four loop iterations to process its input set, while it will take less for the other clusters. All clusters have to wait for the second cluster however. Ideally it would only take two loop iterations for all four clusters to process the eight circles. Thus, this dataset results in a 2x overhead for load-imbalance (ignoring kernel startup overhead). The remainder of this section presents how the conditional stream mechanism can eliminate the overhead of load-imbalance for kernels with if-statements (like the example we just discussed) and for kernels with while-loops.

5.3.1 If-Statements

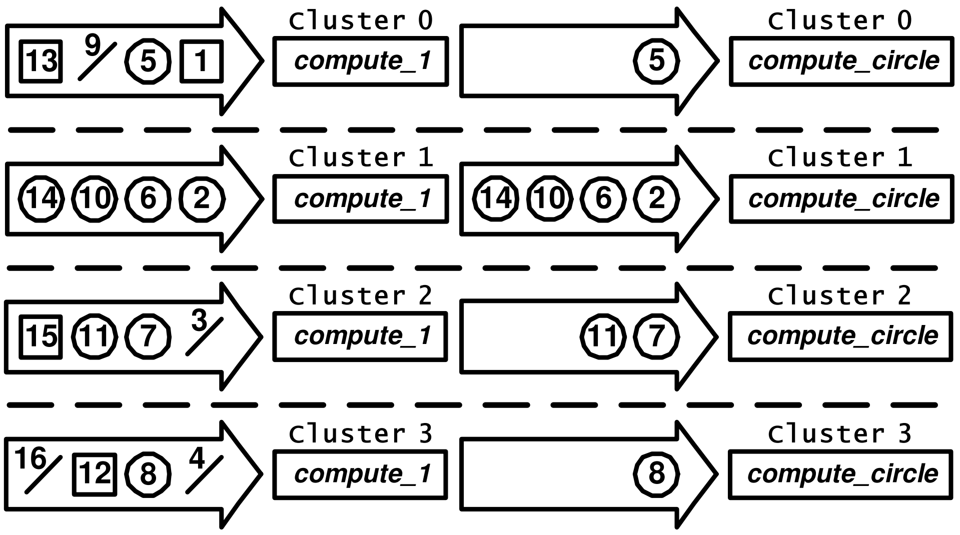
Conditional streams address this problem by using inter-cluster communication to load-balance work between the clusters. They achieve this by modifying the conditional routing primitives to ensure that the stream lengths remain balanced among all the clusters at all times (both during stream production and consumption). Instead of just reading a piece of data from the local SRF bank, the conditional stream mechanism does a global analysis between all the clusters to determine the best assignment of stream data to requesting clusters. In order to simulate a single large global task queue, the elements are ordered by striping them across the SRF. This ordering of stream elements and the process of load-balancing using conditional streams is illustrated in Figure 5.2, for the same example as in the previous figure.

On the first loop iteration, (a), only one cluster has a circle to output to the stream. The element, initially in the second cluster, is sent to the SRF bank of the first cluster over the inter-cluster switch. On the second iteration, (b), all four clusters have circles to output to the stream. Even though, in this case, each cluster could just output each element straight into their SRF bank without affecting the final load-imbalance, the conditional stream mechanism still sends them to SRF banks in different clusters. This is because the conditional stream mechanism imposes a specific ordering on clusters and stream elements:

- *Clusters* are ordered according to cluster number (that is, their cluster ID is also their


other_shapes
compute_1
compute_circle
compute_2
in_data
LOOP OVER shapes { Compute_1 ( ) ; }
circles_only
LOOP OVER shapes { Compute_Circle ( ) ; }
circles_only_2
LOOP OVER shapes { Compute_2 ( ) ; }
is_circle_stream
F F T

(a)



(b)

Figure 5.1: Load-imbalance due to conditional routing on a SIMD architecture. The example is based on a portion of the original *shapes* kernel from Section 3.2. Part (a) highlights the portion of the original example which we will focus on in this section. Part (b) shows the execution of the highlighted portion on a SIMD stream machine with four clusters. An example input dataset is given, as well as the results for that dataset.

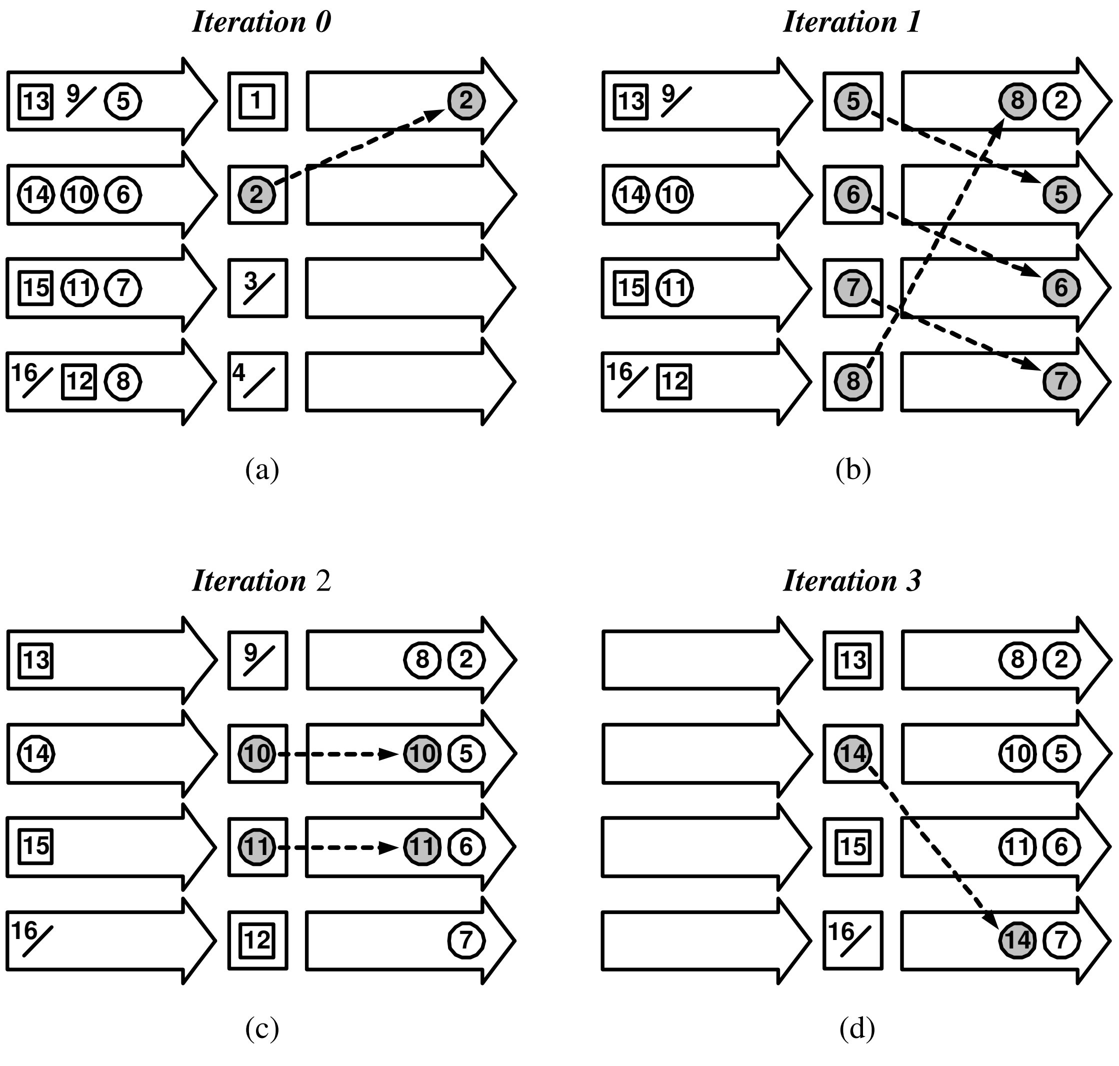


Figure 5.2: This series of figures shows how using a conditional output stream can eliminate load-imbalance for the same example as in Figure 5.1. (a)-(d) shows all four iterations of *compute_1*, after which the data are balanced between the four SRF banks.

ordinal number). Elements are read from or written to the clusters in this order.

- *Stream elements* are hierarchically ordered first according to their position within their local SRF bank, and then elements at the same position in different SRF banks are are ordered according to cluster number. Thus, if a stream element is identified by its (SRF position, SRF Bank #) tuple, then elements are ordered as such: $(0, 0), \ldots, (0, C-1), (1, 0), \ldots, (1, C-1), \ldots$. Elements are read from and written to the SRF in this order when using conditional streams.

Based on this ordering, when multiple clusters want to transfer a data record to/from a conditional stream (i.e., when the case variable is true in more than one cluster), the elements are transferred such that the record from the first cluster among the requesters will be transferred to/from the first stream location that has not been filled/read yet.

The next two loop iterations, (c) and (d), proceed in the same fashion as we have described. Eventually, due to the use of the conditional output stream, there are exactly two circle elements in the SRF bank of each cluster. Executing the next kernel, *compute_circle*, will only require two iterations, or one-half as many iterations required without the use of conditional streams. In general, the only remaining load-imbalance that can arise when executing *compute_circle* would be due to the fact that the number of circles in the input dataset may not be an exact multiple of the number of clusters. Hence, some clusters might execute a maximum of one extra loop iteration of the kernel.

The example so far has focused on the conditional output stream mechanism required for the *compute_1* kernel (Kernel 3.1). The process, however, is quite similar for input conditional streams. For example, the *compute_2* kernel (Kernel 3.2) uses the conditional input routing primitive (“popif”) in order to restore the ordering in the original kernel. It turns out that when we apply the conditional input stream mechanism using the ordering we described earlier, it will cause the input elements from the `circles_only_2` and `other_shapes` streams to be distributed to the clusters in such a way that the original ordering is restored. That is, the *compute_2* kernel will produce the elements of the final output stream, `out_data`, in the same order as would have been produced had we not used conditional streams. Furthermore, notice that the intermediate streams `circles_only_2` and `other_shapes` are also produced and processed in order.

5.3.2 While-Loops

Conditional streams can be used in conjunction with any of the methods we introduced for while-loops as well, and in particular with expanded conditional routing (EXCR) and flattened conditional routing with state-unrolling (FLCRSU). As we discussed earlier, expanded conditional routing simply considers a while-loop as a series of if-statements. It runs a separate kernel for each iteration of the while-loop, and for each iteration creates a new stream of elements that needs continued processing and appends the others to a stream of finished elements. Recall that even the original expanded conditional routing method does not maintain the ordering of the output stream. Thus we do not have to address ordering here. The benefit that conditional streams provide to expanded conditional routing is to load-balance each iteration of the while-loop. That is, for each iteration of the original while-loop, the stream of elements that require further processing will be equally spread out in the SRF banks of all the clusters. The only load-imbalance that can arise is because the number of elements remaining to be processed for any iteration of the while-loop is not a multiple of the number of clusters.

FLCRSU uses conditional routing in order to load-balance on the input side of the kernel. That is, even though each cluster may start out with the same number of elements in their SRF bank, there can still be load-imbalance because each cluster consumes their data elements at a different rate. These techniques use a conditional input stream mechanism to deliver the next element(s) in the SRF banks to the next cluster(s) that have finished processing their previous stream element. Conditional streams will reduce the total load-imbalance between the clusters, but some imbalance may remain. This imbalance would be due to the fact that all clusters may not finish their final elements at the same time. This could be expensive, especially if an element that required a large number of iterations was one of the last elements in the stream.

5.4 Implementation

So far in this chapter we have discussed how conditional streams work and how they are used by the various conditional routing techniques. Now, before we analyze the impact on performance, we will focus on how to implement the conditional stream mechanism. Implementing conditional streams requires intercepting the cluster request signals (i.e., the signals that cause a SRF bank to push or pop an element) from each cluster, and processing them globally. These signals are set to the case value in each cluster. If the local case value is TRUE in x clusters (i.e., there are x requesters), we need to figure out which of the x SRF banks (out of the C total banks) to access in order to service that particular conditional stream access. After the conditional stream hardware figures out the mapping from requesting clusters to servicing SRF banks, data values must be transferred across a switch to get them from the correct cluster to SRF bank, or vice versa. First we will present a solution to do this on a stream architecture. Then we will discuss the scalability of conditional stream hardware to configurations with large numbers of clusters.

5.4.1 Hardware Overview

The basic organization of top level hardware required for inserting the conditional stream unit into a stream processor is shown in Figure 5.3. The diagram shows the case for C clusters and S stream buffers (i.e., a maximum of S logical streams). Essentially, the conditional stream unit intercepts signals that go between a cluster and its local SRF bank. This new unit, however, is monolithic and not segregated into banks because it needs to combine information from all the clusters (the case values) and transfer data between clusters. As is shown on the bottom portion of the diagram, normal stream transfers occur without modification. To clarify, if `cstream` is set to FALSE, then the data and control signals pass through the multiplexers directly between the SRF banks and their associated clusters without intervention. Conditional stream logic only intervenes when the one of the stream accesses is a conditional stream access, indicated via the `cstream` signal. Individual signals in the `cstream` array could be set, for instance, by a micro-controller instruction that is issued at the start of the kernel indicating that a particular logical stream will be accessed as a conditional stream for the remainder of the execution of the kernel.

When `cstream` is set to TRUE, the case values are sent to the global conditional stream logic. This logic is shown in detail in Figure 5.4. Essentially there are four main blocks.

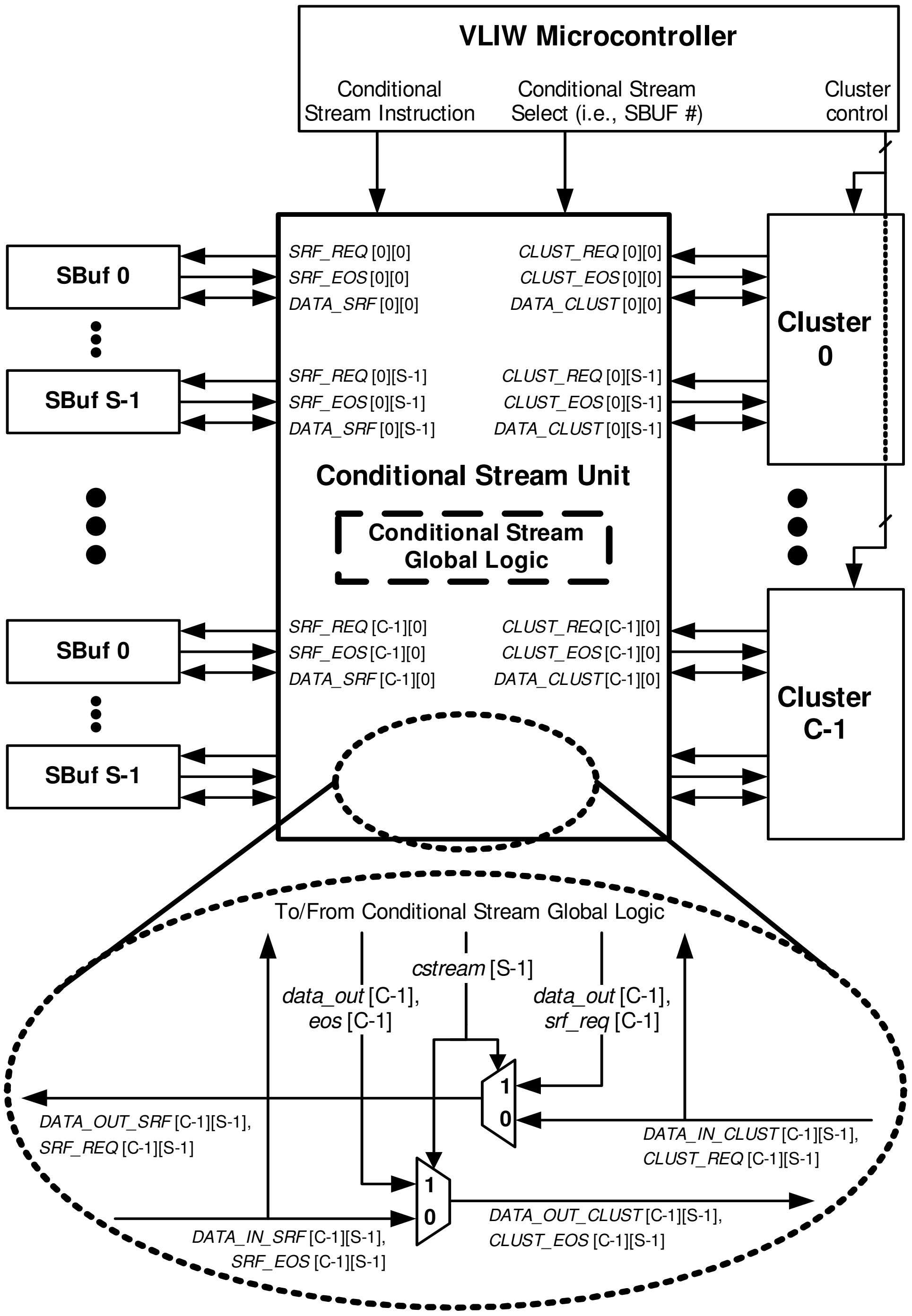


Figure 5.3: Conditional stream implementation overview for a SIMD architecture. The interface between the clusters, stream buffers, and conditional stream logic is shown. EOS = end of stream. The conditional stream logic is shown in more detail in Figure 5.4.

- The central controller receives the case values from the clusters and responds to instructions from the micro-controller.
- The second main block is a set of state registers for each stream buffer, which can be read and written to by the central controller.
- The third block is a data switch, which can transfer data values from a processing cluster to any SRF bank. The switch is assumed to be a full crossbar, and can thus handle arbitrary communication permutations between the clusters and SRF banks.
- Finally, the fourth block is a set of multiplexers in front of the switch inputs that can be configured to let the switch handle data flowing from the SRF banks back to the clusters or vice versa. There are also multiplexers that select one of S stream buffers for each input port of the switch.

5.4.2 Hardware Details

We will now delve a little deeper into the details of the hardware implementation. The goal of this section is two-fold. The first is to serve as a reference example of how to integrate the conditional streams hardware functions into the kernel ISA of a stream processor. The second goal is to serve as a primer to the following section on the scalability of the hardware that implements the conditional streams instructions. We will first present an ISA for the conditional streams unit, and then go through the instruction sequences required for both a conditional input and conditional output stream access. Readers interested less in the conditional streams hardware and more interested in their performance may wish to skip directly to Section 5.5.

The instructions that are needed to successfully execute a conditional stream access are shown in Table 5.1. The CLEAR_CS_STATE and INIT_*CX*_STATE instructions are used to indicate which streams are accessed using conditional streams, and to initialize the state for those streams. The GEN_*CX*_STATE instructions read the `clust_req` signals for the stream buffer being accessed (the stream buffer number is provided by the micro-controller along with the instruction). These instructions will process the request signals to

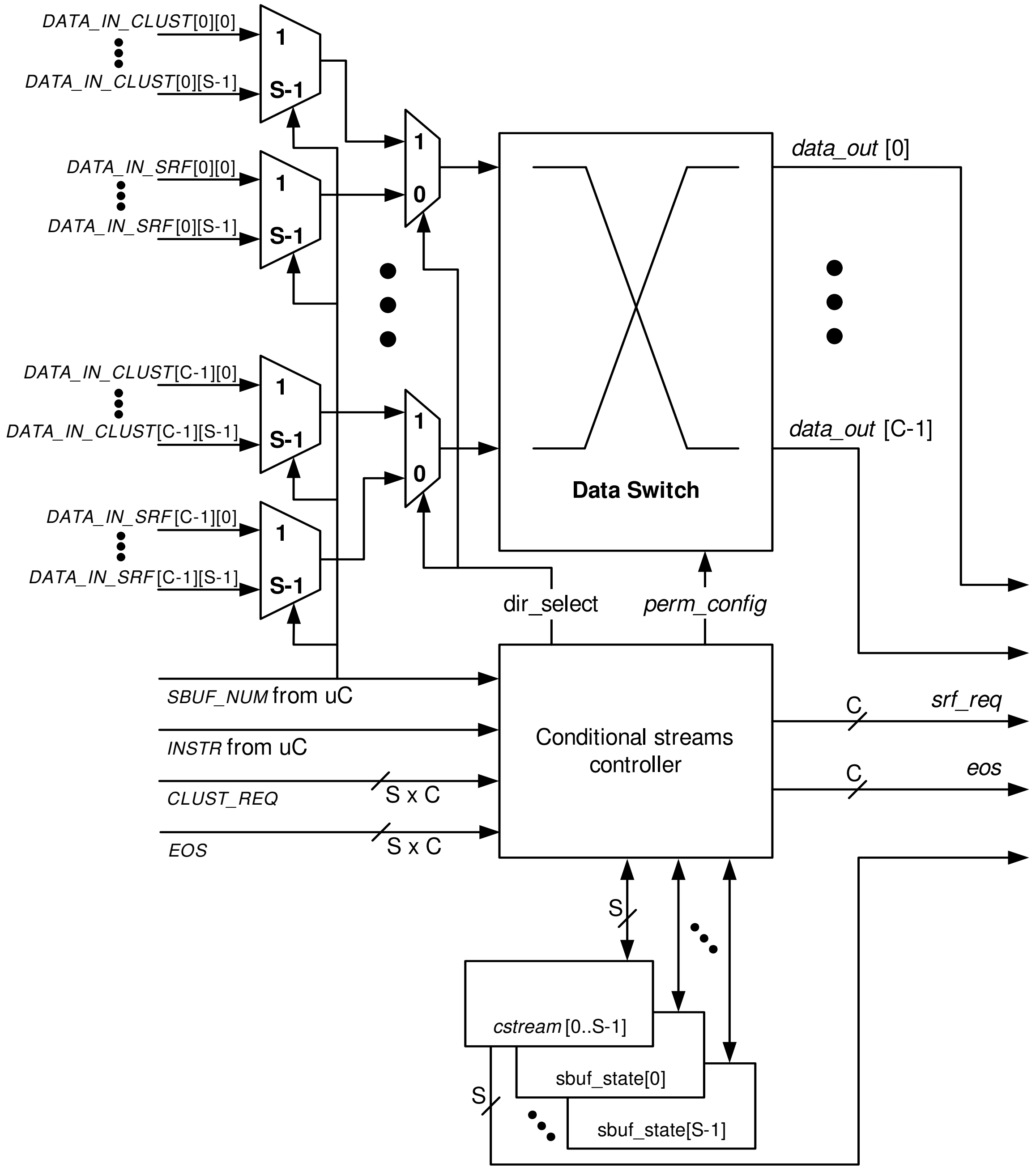


Figure 5.4: Conditional stream global logic details.

Table 5.1 Instructions supported by the conditional streams unit.

Instruction(s)	Description
CLEAR_CS_STATE	Resets all streams to default to a non-conditional stream. Generally this instruction is issued automatically by the micro-controller at the outset of every kernel.
INIT_CI_STATE INIT_CO_STATE	Initializes state for a conditional stream. These instructions are run once at the start of a kernel for every input or output conditional stream needed by that kernel (after the CLEAR_CS_STATE operation).
GEN_CI_STATE GEN_CO_STATE	Generates signals for the next SRF data transfer. These instructions accept the case values from each cluster and combine them in order to generate all necessary control signals to read or write the data records associated with the case values.
GET_CS_EOS	Returns end of stream signals after applying the correct SRF bank to cluster mapping for the current conditional stream access. This instruction must be called after GEN_CI_STATE and before the COND_DATA_IN operation(s) for each individual record access.
COND_DATA_OUT COND_DATA_IN	Transfer a data word to or from the SRF. These instructions actually transfers a word of data between the cluster banks and the SRF banks. Must be executed once for each word in a data record.

compute the proper SRF bank to cluster mapping, taking into account the current state of the stream buffer in `sbuf_state`. Based on this mapping, the instructions generate the permutation for transferring the data, as well as the request signals for the SRF banks. It stores updated state back into the `sbuf_state` register. The GET_CS_EOS instruction returns the proper stream empty condition to each cluster by applying the necessary SRF bank to cluster mapping computed by the GEN_CI_STATE operation. The COND_DATA_*X* operations use the results of the GEN_*CX*_STATE instructions in order to actually perform the data transfers. These instructions must be called once for each word in a data record. Table 5.2 shows the sequence of operations that must be executed in order to access a record using a conditional input and conditional output stream.

Checking whether an input stream is empty when using conditional streams requires

Table 5.2 Instruction sequences for executing conditional stream accesses. Both the instructions required to execute once per kernel, and those required for each conditional access stream are shown. The code assumes that there are two data words per input record and three data words per output record.

Code to execute once at the start of a kernel

INSTR	SBUF_NUM
CLEAR_CS_STATE	—
INIT_CI_STATE	0
INIT_CO_STATE	1

Operation sequence to execute for each conditional input stream access

INSTR	SBUF_NUM
GEN_CI_STATE	0
GET_CS_EOS	0
COND_DATA_IN	0
COND_DATA_IN	0

Operation sequence to execute for each conditional output stream access

INSTR	SBUF_NUM
GEN_CO_STATE	1
COND_DATA_OUT	1
COND_DATA_OUT	1
COND_DATA_OUT	1

some changes. We can no longer use the `empty( )` accessor to get the stream state, since each cluster does not know which SRF bank it is going to be accessing. For instance, consider executing the example first introduced in Kernel 3.6, which is a loop-flattened version of our initial while-loop example. When executing this kernel on a SIMD architecture without using conditional streams, our target architecture would selectively stall clusters once the loop stopping condition was met locally. Once the loop stopping condition was satisfied in all clusters, execution would resume starting with the first instruction after the loop. We will display the loop condition here again for reference:

```
    loop while $\neg$in_stream.empty() $\vee$ $\neg$done begin
```

Now, consider the execution of this kernel using a conditional input stream. The portion of the condition that checks whether `in_stream` is empty will not necessarily be accurate. This is because the stream empty condition is not a purely local operation anymore, and must be performed globally across the clusters. Furthermore, even though the conditional stream might not be empty, there *still may not* be enough elements to satisfy all the requesting clusters. In this case, some of the clusters will not get valid data. Thus, the case values must be known before checking for the end of stream condition. However, the case values may not necessarily be known at the start of the loop, and may in fact be calculated after the start of the loop and before the first conditional access. A solution to this is to check the stream empty condition explicitly for every conditional stream record access using the GET_CS_EOS instruction, and then qualify code in the rest of the loop with this condition. The loop-flattened kernel we referred to above, with this new modification for conditional input streams, is shown for reference in Kernel 5.1. We updated the API to return the `EOS` signal as part of the conditional access, and in this example we assigned the result to `stream_done`.

More details on the exact pieces of stream buffer state that we need to store and the exact logic that each instruction requires are provided for reference in Table 5.3 and Table 5.4.

Table 5.3 Stream buffer specific state stored in the conditional stream unit. There are S sets of stream buffer states—one per stream buffer.

`sbuf_state` field	Size (bits)	Description
`dir`	1	Indicates direction of stream buffer for the duration of this kernel. TRUE indicates an input stream (read from SRF), and FALSE indicates an output stream (write from SRF).
`start`	$\lg C$	Which SRF bank should service the next read or write. The order of bank access is determined by the conditional stream ordering, discussed in Section 5.3.1.
`srf_acc[C]`	$1 \times C$	Indicates which of the SRF banks need to advance their stream pointer and read or write a data word.
`perm[C]`	$\lg C \times C$	Configuration for the data switch. We are assuming the switch is source routed—that is, `perm[i]` indicates which cluster's data should be sent to the output port for cluster i.

Table 5.4 Conditional stream unit instruction details. All the referenced signals and state are listed in Table 5.3 or shown in Figure 5.3 and Figure 5.4. Note that signals listed in small caps are external inputs and outputs for the conditional stream unit.

CLEAR_CS_STATE

```
$cstream[0] = \cdots = cstream[S-1] = \text{FALSE}$
```

INIT_CI_STATE and INIT_CO_STATE

```
$cstream[\text{SBUF\_NUM}] = \text{TRUE}$
$sbuf\_state[\text{SBUF\_NUM}].dir = (op == INIT\_CI\_STATE)$
$sbuf\_state[\text{SBUF\_NUM}].start = 0$
```

Common logic for GEN_CI_STATE and GEN_CO_STATE

```
**let** $clust\_req = \text{CLUST\_REQ}[\,][\text{SBUF\_NUM}]$
**let** $numreq = clust\_req[0] + \cdots + clust\_req[C-1]$
**let** $next\_start = (sbuf\_state[\text{SBUF\_NUM}].start + numreq) \mod C$
**let** $wrap = (nxt\_start <= sbuf\_state[\text{SBUF\_NUM}].start)$
**with** $sbuf\_state[\text{SBUF\_NUM}]$ **do**,
    $srf\_acc[i] := ((i >= start)$ && $(i < nxt\_start))$
                    $\|$ $(!((i >= nxt\_start)$ && $(i < start))$ && $wrap)$
    $start := nxt\_start$
```

GEN_CI_STATE

```
**let** $clust\_req = \text{CLUST\_REQ}[\,][\text{SBUF\_NUM}]$
**let** $mynum(cl) = clust\_req[0] + \cdots + clust\_req[cl-1]$
**with** $sbuf\_state[\text{SBUF\_NUM}]$ **do**,
    $perm[i] := (start + mynum(i)) \mod C$
```

GEN_CO_STATE

```
**let** $clust\_req = \text{CLUST\_REQ}[\,][\text{SBUF\_NUM}]$
**let** $mynum(cl) = (cl + C - sbuf\_state[\text{SBUF\_NUM}].start) \mod C$
**with** $sbuf\_state[\text{SBUF\_NUM}]$ **do**,
    $perm[i] :=$ smallest $j$, s.t. $mynum(i) + 1 = clust\_req[0] + \cdots + clust\_req[j]$
```

GET_CS_EOS

```
$eos[i] = \text{EOS}[perm[i]][\text{SBUF\_NUM}]$
```

COND_DATA_IN and COND_DATA_OUT

```
**with** $sbuf\_state[\text{SBUF\_NUM}]$ **do**,
    $dir\_select = dir$
    $perm\_config = \{perm[0], \ldots, perm[C-1]\}$
    $srf\_req = \{srf\_acc[0] \ldots srf\_acc[C-1]\}$
```

Kernel 5.1 *compute_iterative* with loop-flattening, updated for conditional input streams. Compared to the original, Kernel 3.6, this version has a modified loop condition since the stream empty state must be returned by the conditional stream access function. Note that the semantics of the kernel shown are such that an extra iteration will be required at the end of the kernel for the last cluster(s), since they have to attempt to read an input record from the stream in order to figure out if it is empty or not.

```
$done \leftarrow$ true;
$stream\_done \leftarrow in\_stream.empty()$;
loop while $\neg stream\_done \lor \neg done$ begin
  /* stream_done == in_stream.empty() for this cluster */
  $State, stream\_done \xleftarrow{\text{popif}} in\_stream(done)$;
  $State, done \leftarrow COMPUTE\_ITERATIVE(State)$;
  /* only output if input stream */
  /* wasn't empty for this cluster */
  $out\_stream(done \land \neg stream\_done) \xleftarrow{\text{pushif}} State$;
end
```

5.4.3 Scalability

The complexity, size and delay of the hardware necessary to implement conditional streams grows as the number of clusters (C) increases. We will consider each potentially problematic piece of logic in turn, and show that the most expensive piece of logic to scale is the hardware to implement the GEN_CO_STATE instruction. The problem with the straightforward hardware implementation of this operation is that each cluster must generate a different 1-bit request signal for every other cluster. This amounts to transposing C^2 wires. In order to reduce the wiring area, therefore, we will employ binary encoding instead of one-hot, and will distribute the priority encoding to occur in stages located near the request sources instead of routing all the inputs to one global encoder. The details of these optimizations will be made clearer later in the section.

Inter-Cluster Switch Scaling

The two pieces of hardware we need to worry about when scaling are the inter-cluster communication paths and the conditional stream global logic. Let's consider both in turn.

The first important communication path is the inter-cluster data switch. It turns out that by using a 2-D grid layout of the processing clusters, we can minimize the area impact of this switch [Khailany *et al.*, 2003]. This is achieved by implementing the switch in two stages: in the first stage, each cluster puts a data value onto a vertical bus dedicated to it; in the second stage, each horizontal bus (there is one per cluster) selects which of the vertical buses to read from. This scheme keeps the C^2 area impact of the expensive switch points at each column-row intersection small. This allows good scaling at least up to 256 clusters (with 5 ALUs per cluster). Furthermore, the same study [Khailany *et al.*, 2003] indicates that the inter-cluster data switch can be fully pipelined as well, so that the bandwidth of the switch is not sacrificed. Therefore, this hardware should not be a limiting factor, up to 256 clusters at least. The other important communication paths we need to consider are those that gather control signals from each cluster and deliver them to the global logic, and those paths that distribute output control signals and switch permutations from the global logic back to the clusters. Using the grid layout given in the above study, these do not significantly increase area as well.

GEN_CI_STATE Scaling

We identified the conditional stream global logic as the other important piece of hardware to consider when scaling the number of clusters. The most complex pieces of logic are those for calculating the switch permutation for the GEN_*CX*_STATE operations. Let's look at the one for GEN_CI_STATE first. We basically need to calculate a running sum of all the `clust_req` signals, where each intermediate output is added to `start` to produce a result that is $\lg C$ bits wide. A simplistic implementation of this operation would require serial execution of C additions, and hence $O(C)$ time. However, we can use a parallel prefix scan-+ operation [Cormen *et al.*, 1990] in order to implement this operation in $O(\lg C)$ time.

GEN_CO_STATE Scaling

The implementation of the permutation generation for GEN_CO_STATE is much more complicated, however. This is because each cluster needs to do a search, not just a deterministic calculation, as we can see from the pseudo-code in Table 5.4 for calculating `perm` in the GEN_CO_STATE operation. A reasonable approach for implementing this logic requires first using a parallel prefix scan-+ operation, just as before. For each cluster, this will produce the destination cluster (`dest`) that data must be sent to. Unfortunately, the inter-cluster switches on stream processors to date have been built to be input-routed.[1] Thus, at this point, each cluster must instead figure out which other cluster is trying to write to it (`src`), hence the need for a search.

This search can be implemented with relatively little hardware using a bisection search. This would only require adding another $\lg C$-bit inter-cluster switch. Each cluster writes their local value of `dest` to the switch during each communication operation. Since `dest` is the result of a running sum, is it ordered across the clusters. Each cluster maintains its own search pointer, and does a bisection search to locate the cluster it should read from (i.e., the first other cluster whose value of `dest` points to the cluster doing the searching). A total of $\lg C$ communication operations would be required, however, and these could not be pipelined either since they are dependent on each other. Thus the latency and throughput of this iterative method are poor, and hence we would like to find a more direct approach in hardware to speed things up.

To this end, a naïve hardware-only implementation of the circuitry required to implement the logic to calculate `src` in each cluster is shown in Figure 5.5. The area of this wiring scales roughly as $O(C^4)$, making this logic the most expensive to scale of all the conditional stream hardware modules. Using the area models from [Khailany *et al.*, 2003] again, we can estimate the impact of this logic on the stream processor. This impact is shown in Table 5.5 for several different machines, each with a different number of clusters. Up until 64 clusters, the area is not a prohibitive factor; however beyond that this logic starts to occupy a large fraction of the total area and is impractical.

However, we can actually improve the implementation in order to generate a solution

[1]This is partially so that non-permutation traffic, such as multi-cast, can also be efficiently routed.

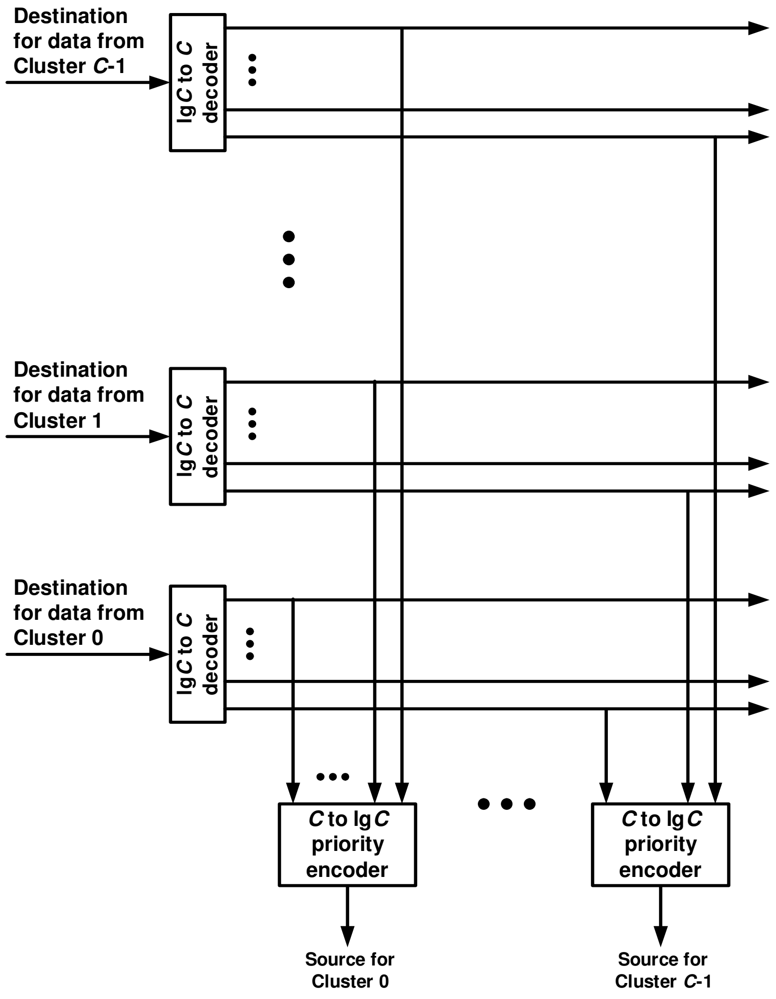

Destination for data from Cluster C-1
lg C to C decoder
Destination for data from Cluster 1
lg C to C decoder
Destination for data from Cluster 0
lg C to C decoder
C to lg C priority encoder
C to lg C priority encoder
Source for Cluster 0
Source for Cluster C-1

Figure 5.5: Naïve GEN_CO_STATE implementation.

Table 5.5 Area impact of naïve GEN_CO_STATE implementation. We used the area models from [Khailany *et al.*, 2003]. Each cluster consists of five arithmetic ALUs. The total area includes the processing clusters, micro-controller, SRF, as well as the GEN_CO_STATE logic.

# Clusters (C)	Fraction of Total Area
8	1%
16	3%
32	8%
64	20%
128	43%
256	73%

that scales as $O(C^2 \lg^2 C)$, instead of $O(C^4)$. This improved implementation is shown in Figure 5.6. Since each cluster only generates one destination for its data, we can use binary encoding instead of one-hot encoding. Also, instead of routing all the priority encoder inputs to the bottom, we distribute the encoding with a stage located at the source of every encoder input. Essentially, these two optimizations are trading-off extra gates for reduced wiring area. We evaluated the wiring area for this new method using the same models as in Table 5.5, and found that the GEN_CO_STATE logic only occupies 5% of the area of the 256 cluster machine now, as opposed to 73% with the straightforward method. Unfortunately, note that the latency of this hardware grows as $O(C)$ as shown. However, the latency can be improved to $O(\lg C)$, at the expense of adding even more extra gates, by doing the vertical searches in a tree fashion, as shown in Figure 5.7[2]. Ultimately, these two implementation optimizations could result in a hardware implementation of GEN_CO_STATE that is practical beyond configurations with 64 clusters. However, further study is required to quantify the exact area of this improved solution. We do not provide this analysis here.

[2]Technically, this optimization will increase the rate at which area grows to $O(C^2 \lg^3 C)$

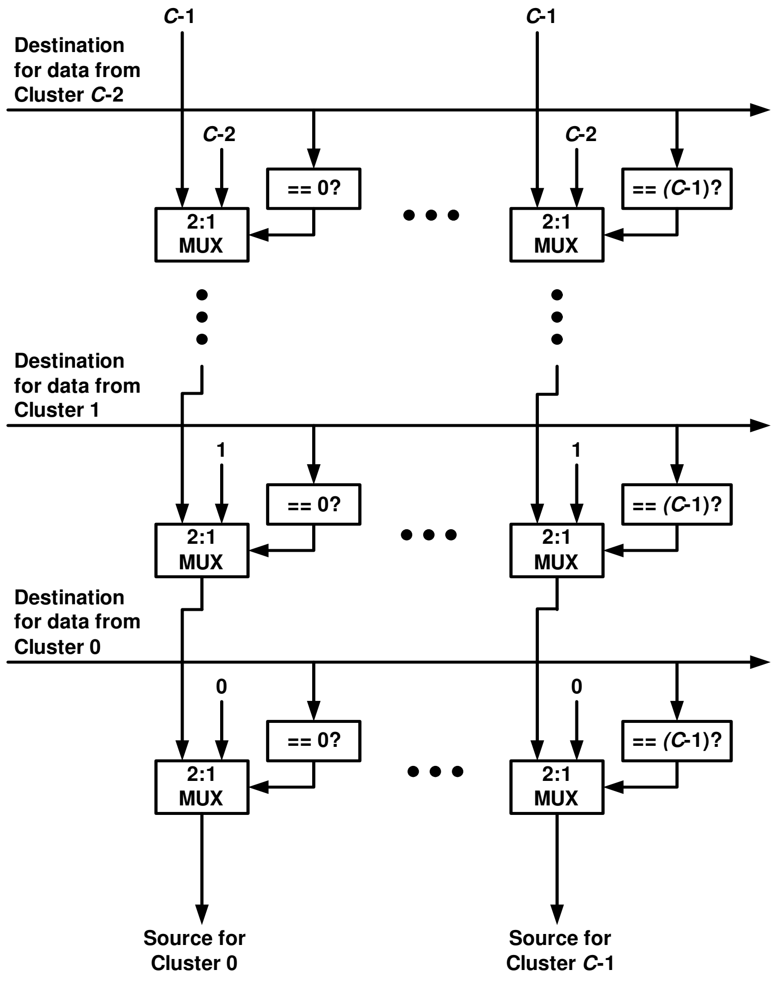


Figure 5.6: Improved GEN_CO_STATE implementation for large numbers of clusters.

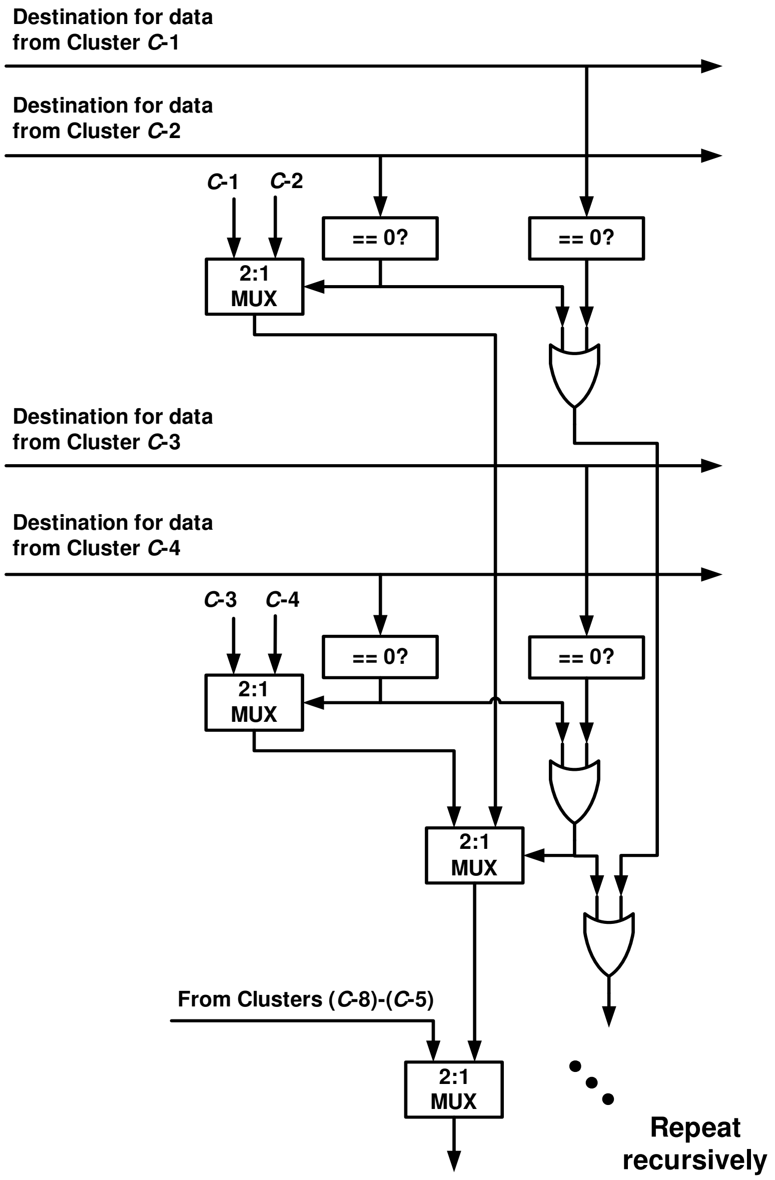

Destination for data from Cluster C-1
Destination for data from Cluster C-2
C-1
C-2
== 0?
== 0?
2:1 MUX
Destination for data from Cluster C-3
Destination for data from Cluster C-4
C-3
C-4
== 0?
== 0?
2:1 MUX
2:1 MUX
From Clusters (C-8)-(C-5)
2:1 MUX
Repeat recursively

Figure 5.7: Logarithmic-time hardware search implementation for GEN_CO_STATE. The idea shown in this figure is applied recursively in order to achieve the log-time result. The logic shown only produces the result for one cluster—this would have to be duplicated C times for a complete solution.

Even Further Scaling

The above area analysis was done for 5 ALUs per cluster, where each ALU supports 32-bit floating point operations. If the number of ALUs per cluster or number of bits of precision per ALU decreases, or floating-point support is dropped, then the area of each cluster will decrease. For example, one could envision a cluster with only 2 ALUs that only supported 16-bit fixed-point operations. This might be targeted at mobile low-power applications, perhaps. Since the area of the conditional stream hardware would have remained unchanged, its relative area overhead would be more significant, and we might want to search for ways to reduce it. Furthermore, for machines with this type of lean processing cluster, or for machines with more than 64 clusters, we can expect that an inter-cluster switch with limited connectivity would be used (i.e., one that is not a full crossbar). Since conditional streams need to route arbitrary permutations, their performance could degrade with limited connectivity switches since these switches may take multiple cycles to route arbitrary permutations. In these situations, we will probably want to use a scheme that uses conditional streams to load-balance with some set of neighboring clusters only. Of course, this would not result in perfect load-balancing, but might be a good trade-off between performance and hardware efficiency.

5.5 Performance

As we mentioned at the beginning of the chapter, we developed the conditional streams hardware in order reduce the overhead of accessing the task queue in the SRF. We should verify two things, then. First, is the speed of the software solution slow enough to warrant the extra hardware? A purely software implementation of conditional streams required 26 and 51 extra operations for a conditional input and conditional output access, respectively, for a single-word record on an eight-cluster machine (larger machines would require more operations). Clearly, executing this number of extra operations every time we access a record in the SRF will significantly affect the performance of conditional streams. In contrast, the hardware-accelerated approach only requires three and two operations, respectively, for the same conditional stream accesses.

The second thing we need to verify is if the hardware solution actually does reduce the overhead enough to allow us to reap the performance benefits of any improvements in load-balance. This is covered in the rest of this section. We will show that conditional streams incur a small fixed overhead for use, but provide the desired immunity against load-imbalance between clusters.

5.5.1 If-statements

We can apply conditional streams to *synthetic_if* in the same way as we did for the *shapes* example earlier in the chapter. The variability arises because of statistical variations—that is, even though the average fraction of elements that execute the body of the if-statement is p across all the clusters, not every cluster will have exactly $(BatchSize/C) * p$ elements that execute the body of the if-statement in their batch. Thus, statistical variations will cause each cluster to have differing amounts of work, and conditional streams can even out the distribution of this work between all the clusters.

The results for *synthetic_if* are shown in Figure 5.8, for three experiments. The difference between each experiment was how dominant the basic block `BODY` was in the kernel. The $1\times$ case is analogous to $W_{\text{body}} = 32$ operations. We also tried the $10\times$ and $100\times$ cases, which correspond to $W_{\text{body}} = 320$ and $W_{\text{body}} = 3{,}200$, respectively. The last case is analogous to a large kernel, where the bulk of it is nested within the if-statement. The reason we tried these different cases is because load-imbalance only affects the *body* kernel when using conditional routing, thus the larger this kernel is relative to the other kernels the larger the cost of load-imbalance, as given by Amdahl's Law. The $100\times$ version is representative of the asymptotic case when the body of the if-statement completely dominates the runtime of the kernel.

The graph shows two sets of speedups for each case: one for IDEAL, which assumes that we can achieve perfect load-balancing between the processing clusters, and one for the CONDITIONAL STREAMS technique. Each curve is a speedup over the plain (non-load-balanced) implementation of each kernel using conditional routing. For the $1\times$ case, using conditional streams actually degrades performance slightly compared to the base case. This is because the *body* kernel only accounts for at most one-half the total execution time of the

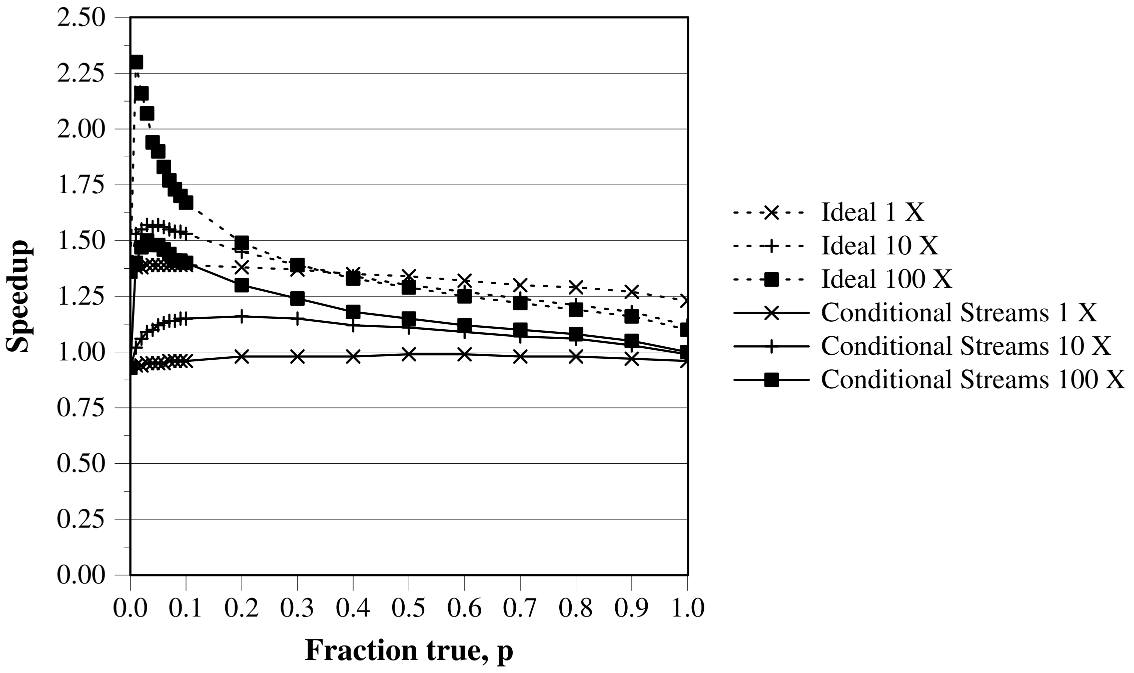


Figure 5.8: Conditional stream speedup for if-statements on an eight cluster SIMD machine. The various curves differ in how large the body of the if-statement was (in terms of operations). In the legend, $N\times$ refers to a test where the size of W_{body} was N times larger than the base $1\times$ case. The IDEAL curves assume perfect load-balancing between the processing clusters. The batch size used was 64 elements per cluster. $1 \times W_{\text{body}} = 32$ operations; $W_{\text{input}} = W_{\text{output}} = 17$ operations; $S = 2$ words.

kernel. Furthermore, when $p = 0.0$ and $p = 1.0$, conditional streams also always degrades performance because there is no load-imbalance in the input dataset. The slowdown due to conditional streams, though, is always limited to less than 8%, over all values of p and over all the experiments. However, as we start to increase the size of the basic block `BODY`, conditional streams start to benefit overall performance. This is especially true when we execute kernels with if-statements that are expensive, but rarely taken ($10\times$ and $100\times$ with small p). At the point of biggest disparity, conditional streams offers a roughly 50% speedup. The IDEAL curves show that conditional routing and conditional streams in concert can reduce the difference between actual kernel performance and ideal performance to within 30% for most values of p, and to within 65% in the worst case for the experiments shown.

5.5.2 While-loops

The results for the speedup that conditional streams provide to our base conditional routing techniques is shown in Table 5.6. As expected, the biggest benefit of conditional streams is gained when the amount of variance in the data set is high. In fact, conditional streams provides roughly a 40%-50% speedup for the dataset with the highest variance. We should also expect that when we have little or no variance in the data set we would see a slowdown due to the overhead of inserting conditional streams. While we do see a small overhead for FLCRSUCS of 4% for the “Zero-Variance” dataset, we actually see a speedup for the same dataset on EXCRCS. This anomaly occurred because the compiler happened to schedule the main-loop of the *body* kernel for the conditional streams version with less software-pipelining stages. Thus, the reduced loop-priming overhead results in a slight speedup even for the case with no variance in the dataset.

The second set of data, shown in Figure 5.9 shows how well the conditional stream versions of the kernels do compared to the ideal performance of the machine. Conditional streams combined with conditional routing is able to achieve more than 67% of ideal performance (i.e., within 1.5x of the ideal execution time), for the entire range of datasets shown with the FLCRSUCS technique. The EXCRCS technique still pays a hefty penalty for executing a dataset with a small number of elements that require a large number of

Table 5.6 Conditional stream speedup for while-loops on an eight cluster SIMD machine. The version of the kernel used is the same as in Figure 3.17(d), while the four probability distributions used are the same as in Figure 3.18.

Speedup compared to the same method without conditional streams

Method	Zero-Variance	Uniform	Spread	Bimodal
Expanded Conditional Routing with Conditional Streams (EXCRCS)	1.01	1.10	1.20	1.42
Flattened Conditional Routing with State Unrolling and Conditional Streams (FLCRSUCS)	0.96	1.04	1.11	1.47

iterations, as we saw in Section 3.3.4. Therefore, while it comes within 15% of the ideal performance of the machine (1.14x of the ideal execution time) when there is no variance in the dataset, it performs at less than 50% of ideal with the “Bimodal” dataset.

5.5.3 Impact of SRF Communication

As we noted in Section 3.2.3, the amount of live state that is transferred into and out of the basic blocks can have an impact on performance. A large amount of state will imply a proportionally larger amount of extra SRF communication that needs to occur with conditional routing. With the addition of conditional streams, however, this communication impacts performance even more. Each conditional SRF access requires an inter-cluster switch traversal. The switch bandwidth now becomes the limiting factor for the performance of kernels that have to transfer a lot of state to/from the SRF. Figure 5.10 shows how the schedule length of the conditional streams version of the *input* kernel from the *synthetic_if* benchmark (Section 3.2.1) degrades faster compared to the plain conditional routing version as the amount of live state increases.

As the amount of operations in the *input* kernel increases, however, the point at which the knee in the curve occurs will move to the right, since there will be more cycles of arithmetic to overlap with the communication operations. Also, recall that the kernel that

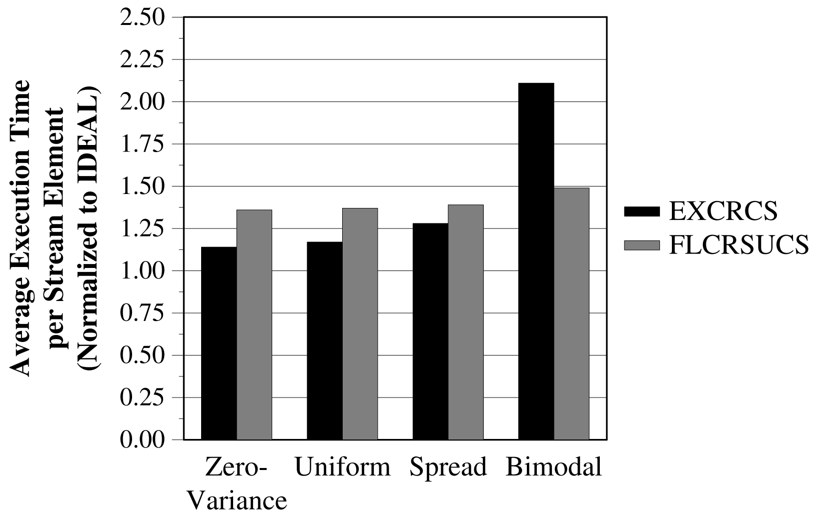


Figure 5.9: Conditional stream results, normalized to the execution time of IDEAL, are shown for the *synthetic_while* kernel. EXCRCS = Expanded Conditional Routing with Conditional Streams; FLCRSUCS = Flattened Conditional Routing with State Unrolling and Conditional Streams.

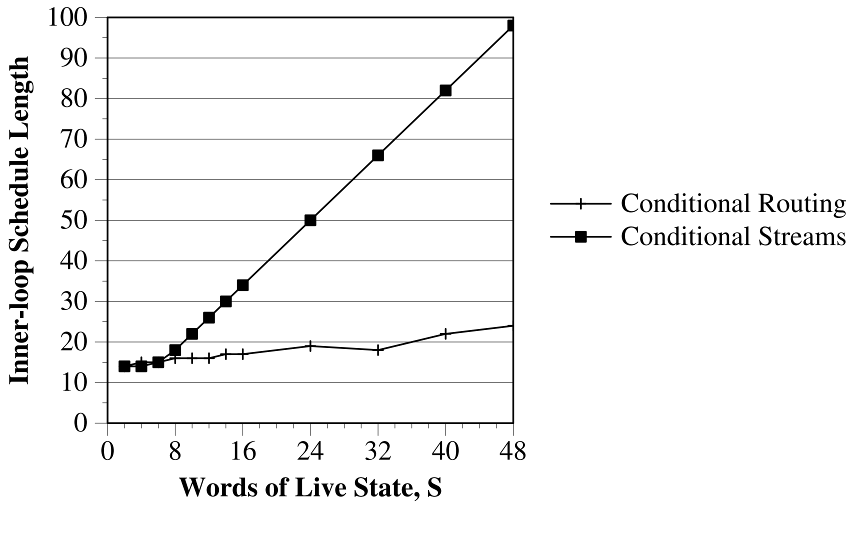


Figure 5.10: Impact of SRF communication on conditional stream version of *input*, which is a kernel from the *synthetic_if* benchmark, introduced in Section 3.2.1. The *input* kernel requires two conditional stream accesses per stream element: one for the TRUE stream and the other for the FALSE stream.

implements the body of the if-statement does not use conditional streams, and thus will not have this drastic reduction in performance as the amount of live state increases. Hence, if the majority of the execution time is spent in the *body* kernel, then the impact of live state will be minimized. For clarity, we can codify our observations above in the following expression for the performance impact of conditional streams on *synthetic_if* (i represents execution time in cycles, SU represents speedup, SD represents slowdown):

$$Speedup = SU = \frac{i_{cs} - i}{i}$$

$$i = i_{in} + i_{body} + i_{out}$$

$$i_{cs} = \frac{i_{body}}{SU_{body}} + (SD_{in} i_{in} + SD_{out} i_{out})$$

In this expression, i_{cs} is the execution time of the version with conditional streams. The value of i_{cs} depends on the speedup or slowdown of each of the three constituent kernels: *input*, *body*, and *output*. We expect the execution time of the *body* kernel to decrease because of improved load-balance, and to increase for the *input* and *output* kernels because of increased inter-cluster communication. So, if the *body* is much larger than the other two kernels, the improved load-balance will probably outweigh scheduling inefficiencies due to inter-cluster communications. However, if there is a lot of internal live state and i_{body} is not very large, then the slowdowns of the other blocks may negate any improvements due to better load-balance.

5.6 Other Uses of Conditional Streams

We have shown so far that the conditional stream hardware mechanism is useful for load-balancing work when using conditional routing techniques. An observation we can make at this point is that in general, conditional streams can provide the abstraction of a single stream that all the clusters are accessing—i.e., a global queue, that under hood happens to be distributed across the SRF banks. Hence, the utility of the hardware implementation is not limited to just conditionals. It can also accelerate any program that needs to use the abstraction of a single-stream that is accessed by all the clusters (as opposed to C separate sets of data all accessed independently by each cluster). Let's look at some examples.

We can improve the process of handling reduction outputs. When all the clusters work together to produce one output for every iteration, then extra software has to ensure that the outputs are properly interleaved across the SRF banks. Conditional streams can be used instead to avoid adding this extra control code. Another example comes from a depth extractor application. In this application, a kernel compares two images at several different horizontal shifts (disparities). The kernel is called repeatedly with a different horizontal shift between the two rows. Assuming the data for each row is interleaved across the SRF banks, control code has to be inserted to perform the shifting of the two rows relative to each other. This code would need to calculate the necessary permutations to find the correct SRF bank to cluster mapping for the current disparity. Instead, conditional streams handles this shift elegantly, and simply requires a loop that drains a specified number of pixels from one of the streams before starting the main loop.

Merging two sorted streams into one longer stream is another good example. In one implementation, all clusters work together to output C elements every loop iterations, of which C_1 elements are from the first stream and C_2 are from the second. C_1 and C_2 are data-dependent. In order to read more values for the next iteration, therefore, a complicated function has to be solved that generates the SRF to cluster mappings to refill data from both input streams. The conditional stream hardware handles this operation efficiently, and again avoids the addition of extra software to calculate the SRF bank to cluster mappings every iteration. For example, the main loop schedule length for the fully software implementation of a merge sort kernel was 25 cycles long, whereas the same loop with hardware conditional stream primitives was only 14 cycles long. This translates to a roughly 80% overhead for the software version.

5.7 RENDER Performance

Let's look at the performance on an entire application (RENDER) now. Applying conditional streams to the *geom_rast* kernel actually alleviates two different instances of load-imbalance in this application. Firstly, by using conditional streams with conditional routing, as discussed in this chapter, we can distribute the work evenly across the clusters for *geom_rast*. Secondly, by using a conditional stream to store the outputs of the *geom_rast* kernel to the SRF banks, we can ensure that kernels that come after *geom_rast* will have a load-balanced input set to work on. Let's look at the impact of both instances of load-balancing in turn.

The results in the first column of Table 5.7 show the improvement of the *geom_rast* kernel due to conditional streams, where each row is for the combination of a different conditional technique with conditional streams. It turns out conditional streams don't help ORIGINAL or COMMON. This is because all clusters always read another input at exactly the same time, due to SIMD. A conditional input stream doesn't need to redistribute any elements, because a cluster cannot go on its own and start a new element without waiting for the rest. The performance of PREDICATION doesn't improve either, even though we used loop-flattening (since loop-flattening allows a cluster to start processing a new element before the other clusters have finished). Unfortunately, early in the loop there is an update of some loop-carried state which is needed by the next stream element; therefore each cluster can only process elements from within its local SRF bank. CONDITIONAL ROUTING however, jettisons the portion of code with the loop-carried dependency into a separate kernel before we start to optimize the while-loop in *geom_rast*. Therefore we can load-balance the input to the while-loop, as well as the work for the different if-statements in *geom_rast*. We achieve a 10% speedup on *geom_rast* when we use conditional streams with CONDITIONAL ROUTING.

The second source of load-imbalance we mentioned arises because the *geom_rast* kernel produces a potentially different number of output elements for each input element. Since each cluster can produce a different number of outputs into their local SRF bank, this will cause all the kernels downstream to suffer load-imbalance, even if these later kernels do not contain any conditionals themselves. Therefore, by writing the outputs of *geom_rast* to the SRF using a conditional stream we can ensure that the load will be distributed evenly for the remainder of the kernels in the RENDER application. Every conditional technique took advantage of this, and there was between $1.1\times$–$1.4\times$ speedup in the overall application by using conditional streams. We should note that the improvement achieved by RENDER is the highest when using CONDITIONAL ROUTING, partly because CONDITIONAL ROUTING was the only method that increased the performance of the *geom_rast* itself.

Table 5.7 Impact of conditional streams on the performance of the *geom_rast* kernel and the entire RENDER application on a stream processor with SIMD clusters.

Method	Machine type	Conditional Streams Speedup		Normalized Comparison	
		geom_rast	RENDER	*geom_rast*	RENDER
ORIGINAL	SIMD	1.0	1.1	1.0	1.0
COMMON	SIMD	1.0	1.2	2.4	2.1
PREDICATION	SIMD	1.0	1.1	1.3	1.2
CONDITIONAL ROUTING	SIMD	1.1	1.4	4.2	3.1
COMMON and CONDITIONAL ROUTING	MIMD	—	—	4.6	3.3

The MIMD results are shown in the last row of the table. Recall that MIMD was not able to provide any performance advantage in the last chapter, when we looked at the RENDER application without conditional streams. However, the situation is slightly different when we add conditional streams. The CONDITIONAL ROUTING implementation incurs a large overhead for inter-cluster communication when using conditional streams. For this reason, combining it with the COMMON technique can help on a MIMD machine, especially to implement the if-statements that have a smaller operations to communication ratio. However, even so, MIMD only provides a 10% performance advantage on this application.[3] An interesting future extension of this work would be to see how the actual performance per unit area compares between the SIMD and MIMD machines.

[3]A MIMD implementation of conditional streams must be able to handle requests to multiple conditional streams on the same cycle (from different clusters). On any real MIMD implementation, only one conditional access would be handled at a time, causing a stall on the clusters requesting access to other streams. However, our simulation infrastructure ignored this and assumed multiple conditional requests could be handled each cycle. This unfairly improves the MIMD result, but we estimate the impact of this simulation inaccuracy to be small on final performance.

5.8 Summary

A conditional stream is a mechanism that provides the abstraction of a single global queue, despite the fact that the SRF is actually banked along with the clusters. Conditional streams can be used in conjunction with the conditional routing technique in order to provide immunity to the load-imbalance in an input data stream, and to generate output data streams in a way that eliminates any more load-imbalance further downstream in the application. When faced with the decision of whether to employ conditional streams, we must first make sure that any benefits gained by improved load-balance between the clusters is not offset by overheads, both due to executing conditional stream control operations (such as GEN_*CX*_STATE) and due to poorer schedules that result from limited inter-cluster switch bandwidth. For the if-statement and while-loop benchmarks, there was less than 8% overhead for using conditional streams when no load-imbalance was in the input dataset. On the flip side, conditional streams achieved up to a 1.5x speedup for the worst-case situations in our experiments. We were able to keep the overhead so low because we used a hardware mechanism for implementing our scheme. We presented the hardware implementation details, and noted that some of the logic can get expensive when the number of processing clusters in the machine gets large. Finally, besides being useful for implementing conditional routing, the conditional stream functionality is useful in a variety of other situations, such as merging two streams. Since the conditional stream mechanism is implemented in hardware, we can eliminate control logic that would otherwise dilute the kernel code in these situations.

Chapter 6

Conclusions

Stream processors offer a compelling combination of area and power-efficiency along with programmability. They excel on a wide range of media processing applications, which tend to be quite regular. However, applications that contain conditionals are challenging to execute on stream processors (and other explicitly parallel processors such as SIMD, vector, and VLIW machines as well). We make the observation however, that conditionals within kernel main loops do not necessarily break the data-parallel paradigm—that is we can still operate on many stream elements in parallel. However, conditionals do require different calculations for each element, so each processing cluster may not execute the exact same code for each element. Our goal in this dissertation was to develop techniques to efficiently execute kernels with conditionals, without sacrificing the efficiency of the hardware. To this end, we wanted to avoid adding dynamic scheduling hardware to improve ILP and we wanted to avoid providing a separate micro-controller for each processor cluster (MIMD) to improve data-parallelism.

Before we summarize our contributions, let's review how bad the problem was to start with. Let's look at the *geom_rast* kernel one more time on two machines: 1) a single-cluster stream processor, with only one ALU in the cluster that is not pipelined; and 2) an eight-cluster SIMD stream processor with five ALUs per cluster that are pipelined four cycles deep. We made the observation that the *geom_rast* kernel can be compiled and executed on the first machine with almost perfect efficiency. However, without using any of the techniques we introduced, the kernel executes at most $18\times$ as fast on the larger stream processors. Compare this to the potential speedup:

$$\frac{\text{ops/cycle of large machine}}{\text{ops/cycle of small machine}} = \frac{(1\ \frac{op/ALU}{cycle})(5\ \frac{ALUs}{cluster})(8\ clusters)}{\frac{1}{4}\ op/cycle} = 160\times$$

We are still roughly nine times as slow as the peak possible speedup. Our new techniques, conditional routing and conditional streams, however are able to increase the speedup to $31\times$, which is roughly five times as slow as the peak possible speedup. So while our techniques improved upon existing techniques by $1.9\times$, there is still a lot of room for improvement. The following discussion will review our specific contributions and identify potentially fruitful opportunities to increase efficiency even further via future research.

6.1 Conditional Routing

Conditional routing replaces conditional control-flow with communication. This type of data-steering creates streams of elements that all require similar processing, and hence each stream can be processed quite efficiently. Conditional routing is thus quite amenable to software-pipelining to improve VLIW schedules, and to execution on hardware-efficient SIMD machines. We did make the observation however that when the size of the data records starts to get large, the required SRF communication starts to slow the kernel down. Furthermore, we noted that the output of certain optimizations reordered the output of a kernel with a while-loop. In this regard, we found that the addition of an indexable-SRF was the best solution for restoring order, if necessary.

Because of these drawbacks, we found that conditional routing was not always the best technique to use, but rather that it offered a large performance improvement when the situation was amenable. Therefore, in addition to presenting raw performance, we analyzed a series of parameter studies for if-statements, in order to understand when we should choose one technique over the other. For example, PREDICATION can not be beat when the conditional is small and frequently executed.

Handling complicated conditionals, such as if-statements with many else-if clauses or with many levels of nesting, could be handled by flattening the whole conditional construct

(Section 3.2.2), and then choosing the best technique for each resulting top-level clause. However, this may not be the optimal algorithm, since we are ignoring some global information contained in the original structure of the if-statements. Therefore, an interesting area for further research would be to develop algorithms for finding the best combinations of techniques to apply to complex conditional constructs.

6.2 Conditional Streams

Conditional streams attacked the problem of balancing load across all the clusters of a stream processor. We proposed a hardware mechanism for accessing the SRF that evenly distributes elements between the various SRF banks and the clusters. While conditional streams offered good performance when record sizes were small, the effect of larger record sizes can potentially reduce its efficiency. This is the main reason why we improved the performance of the *geom_rast* kernel by only 10% when we added conditional streams. In order to address this, we recommend further work in evaluating methods of implementing conditional streams that don't require every record that is read from or written to the SRF to be transferred across the global inter-cluster switch. For example, we might consider incorporating some of the ideas from existing load-balancing techniques that employ *distributed task-queues*.

A distributed task-queue is set up so that processing clusters can make as many references as possible to the local SRF bank. As imbalance starts to appear, only the communication necessary to remedy the imbalance is incurred. Keckler classifies distributed task-queue algorithms into two types: those that load-balance via *offloading* and those that load-balance via *work stealing* [Keckler, 1994]. Offloading schemes require two phases: a computation phase and communication phase. Processors operate on data in their local queue in the computation phase. The communication phase is triggered on a node based on how many tasks are currently in its local task queue. A good comparison of several of these types of techniques is given provided in [Eager *et al.*, 1986]. In work stealing systems, only processors with empty local task queues can participate in load-balancing, which they do by requesting work from other processors. An example of this type of scheme is *lazy task creation* [Mohr *et al.*, 1991]. We suggest exploring how to combine some of these methods with conditional streams, in order reap the benefits of improved load-balancing even when record sizes are large.

6.3 A Compiler Framework

While conditional routing and conditional streams are tremendously useful in their current forms, we have shown that there is still much room for improvement. Some of the suggestions above can help in this regard. However, we feel that probably the most important piece of technology required to proceed will be a framework for implementing these conditional techniques in an automatic compiler tool. This would accelerate the process of converting a greater number of irregular applications into the stream model. Currently, the choice of conditional technique has to be made by the programmer, and switching techniques involves manual effort to change kernel and stream code. Even if the compiler couldn't analytically determine the best technique, it would at least expedite the process of testing various techniques to experimentally determine which one is the best.

A useful compiler framework could use a variety of input, from programmer hints to profiling information. Whether a particular stream must be ordered or not can be included as a keyword in the stream language. Many of the results presented in this dissertation will contribute directly to the effort of developing a compiler framework for handling kernel conditionals. Specific parameter studies, such as the effect of basic-block size and stream length, identify the relative importance that each characteristic should play in compiler heuristics. This dissertation also outlines the available options for different situations which might arise in real program. For example how to sort the output streams of a while-loop kernel, or how to compile more complex conditional-statements.

One interesting piece of technology that must be developed for the compiler framework to be successful is the ability to split kernels. This is required to implement the conditional routing optimizations. The algorithm for splitting kernels must identify which internal live state goes to the different clauses of the conditional, and must route that state to and from the corresponding streams. Interestingly, a tool that could split kernels would be useful for more than just applying conditional routing. For instance, kernels must be split when a data structure is randomly accessed, and that data structure resides in off-chip DRAM because it cannot fit in the on-chip memories (SRF or scratch-pads). Furthermore, splitting a kernel might be the best solution for implementing a single large kernel that contains too much state to allocate within the local register files (LRFs).

Appendix A

The *IMAGINE* Stream Processor

Throughout this thesis, we have evaluated our conditional routing results on a prototypical stream processor. This is not necessarily an idealized model (i.e., realistic latencies were used for performance), but at the same time the particular combination of architectural parameters we used has never actually been implemented. This appendix presents some details on a stream processor that has been implemented in silicon (*IMAGINE*), in order to provide some concrete numbers. Furthermore, the appendix will explore how the conditional streams hardware was implemented on *IMAGINE*, and will present the set of changes and compromises that were made to the conditional streams implementation on a real implementation.

The architecture shown in Figure A.1 contains all the basic stream hardware of the canonical stream processor we introduced in Section 2.2, which consisted of a stream register file (SRF) and stream clients. The stream clients on *IMAGINE* include a kernel execution unit (KEU) and a DRAM interface, as well as a network interface. The KEU consists of eight arithmetic VLIW clusters, each of which consists of six arithmetic ALUs (three adders, two multipliers, and one divide/square-root unit).

In order to simplify the design and implementation of the *IMAGINE* architecture, the functionality of the application processor has been split into two portions. The on-chip stream controller only handles the sequencing of operations to the stream clients. An external processor executes the application level code and hands off any operations destined for stream clients to the stream controller through the host interface.

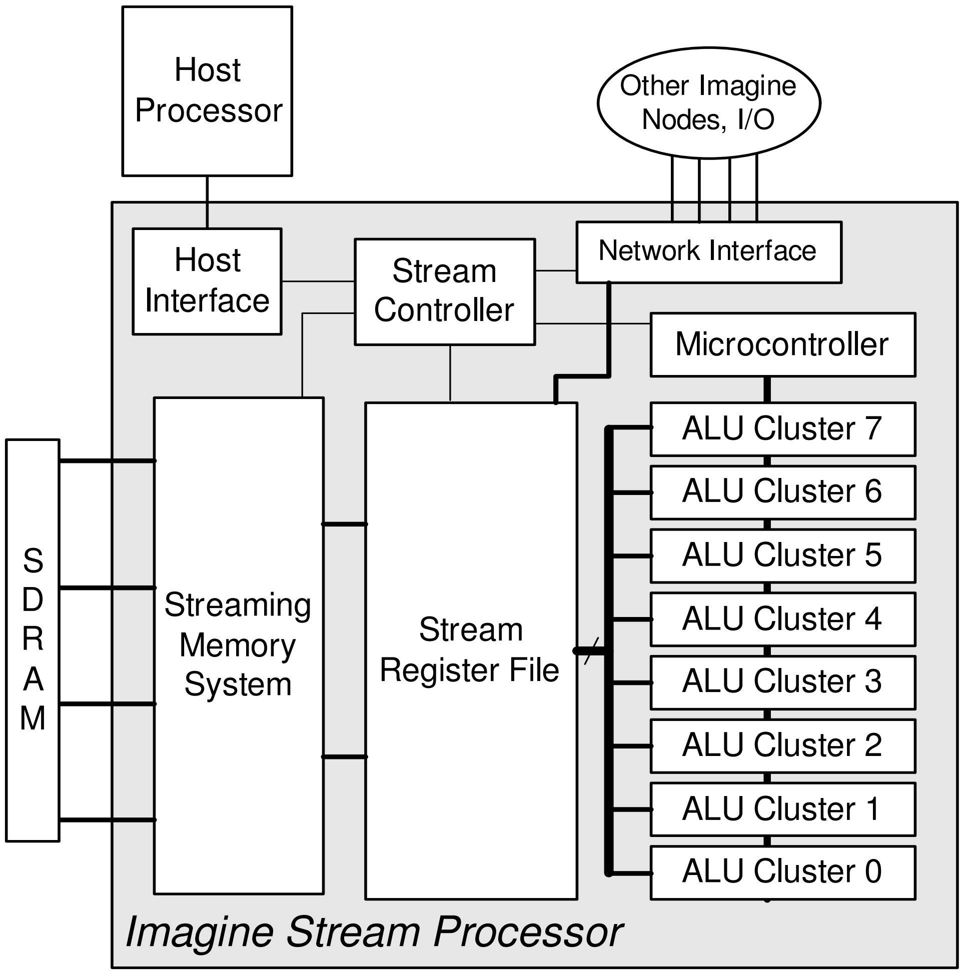


Figure A.1: The Imagine Processor architecture block diagram and floor-plan.

This processor supports 48 ALUs on a single die to provide a high peak performance. At a frequency of 180 MHz, *IMAGINE* delivers 7.4 GFLOPS or 14.4 16-bit GOPS. On a range of media processing applications, the processor sustains roughly half of this peak performance [Kapasi *et al.*, 2003]. Furthermore, when supplied with a lower voltage and clocked at a lower frequency (1.2 V, 96 MHz), the processor achieves a peak of 2.4 GFLOPS/W or 4.7 16-bit GOPS/W.

A.1 Conditional Streams Implementation on Imagine

Implementing conditional streams on the *IMAGINE* architecture presented some additional challenges, since we wanted to meet these two additional constraints:

1. In order to reduce complexity, the SRF control on the *IMAGINE* architecture is not as flexible as the target SIMD architecture we have been evaluating throughout the dissertation so far. In particular, the clusters on *IMAGINE* are not capable of accessing their individual SRF banks independently, and can only push or pop stream elements together. Thus, even though only one cluster may request an element, all eight SRF banks will have to pop an element from the stream in order to service that request.

2. We wanted to minimize the impact of conditional streams on the eventual area of the VLSI implementation of the architecture.

In order to satisfy the first constraint, we had to implement a buffer outside of the SRF. For a conditional input stream, if initially less than eight clusters requested an input element, then the remainder would be stored in the buffer ($C = 8$ on *IMAGINE*). On the next access, data would first be supplied from this buffer, and only if there were not enough elements in the buffer would more elements be read from the stream buffers. Again, any remaining elements would be kept in the buffer for future accesses. The use of a buffer to do this is shown in Figure A.2. In this implementation, there must be room for two words per record element of the stream datatype. The figure shows a sequence of three conditional stream accesses, where the stream buffers are only accessed for two of the three accesses. This is because the there is sufficient data in the buffers to completely satisfy the second conditional stream access. The end-of-stream logic was also modified to take into account any elements that might be in the buffers at the end of the kernel. As for output streams, if less than eight clusters wanted to initially output data, then these data elements would be stored in the buffer with no stream buffer transfer. On the next access, again, elements would first be stored to the buffers, and only if more than eight elements have been accumulated will a stream buffer transfer occur. Furthermore, *IMAGINE* did not have the ability to track stream lengths that were not a multiple of C. Thus, at the end of conditional output streams, NULL data elements had to be used to fill any extra stream slots required to bring the stream length to the next highest multiple of eight. Future kernels downstream in the application have to specifically case out on these NULL elements in software to make sure they were not mistaken for real data.

The second constraint, minimizing the impact on area, was met by reusing many modules already present in the *IMAGINE* architecture for other purposes. Thus, the inter-cluster communication switch that was used by kernels to swap data between clusters, served double-duty as the data switch for conditional streams as well. Furthermore, the extra buffers required for conditional streams, as discussed just above, were implemented within the scratch-pad register files in each cluster. These existing modules, named the COMM unit and SCRATCHPAD unit, are controlled independently by each cluster on the *IMAGINE* architecture. Thus, in order to simplify interfacing with these modules, we factored the conditional streams global logic into eight identical parts, one for each cluster. This logic was implemented as two ALUs in each cluster, the JUKEBOX and VALID units. These two units controlled the COMM and SCRATHPAD units in each cluster in order to do the necessary buffering and data transfers for conditional streams. Of course, in order to distribute the conditional stream logic efficiently, some of the computation has to be duplicated on each cluster. Nonetheless, these ALUs occupied less than 5% of the area of a cluster on the *IMAGINE* prototype chip, less than 1% of the area of the entire processor. Further details and illustrations of the implementation of conditional streams on *IMAGINE* can be found in [Kapasi *et al.*, 2000; Khailany, 2003].

Notice that conditional routing *cannot be used on IMAGINE without the conditional stream mechanism*. Conditional routing requires that all clusters be able to fill and drain their SRF banks independently. However, each cluster cannot independently access their

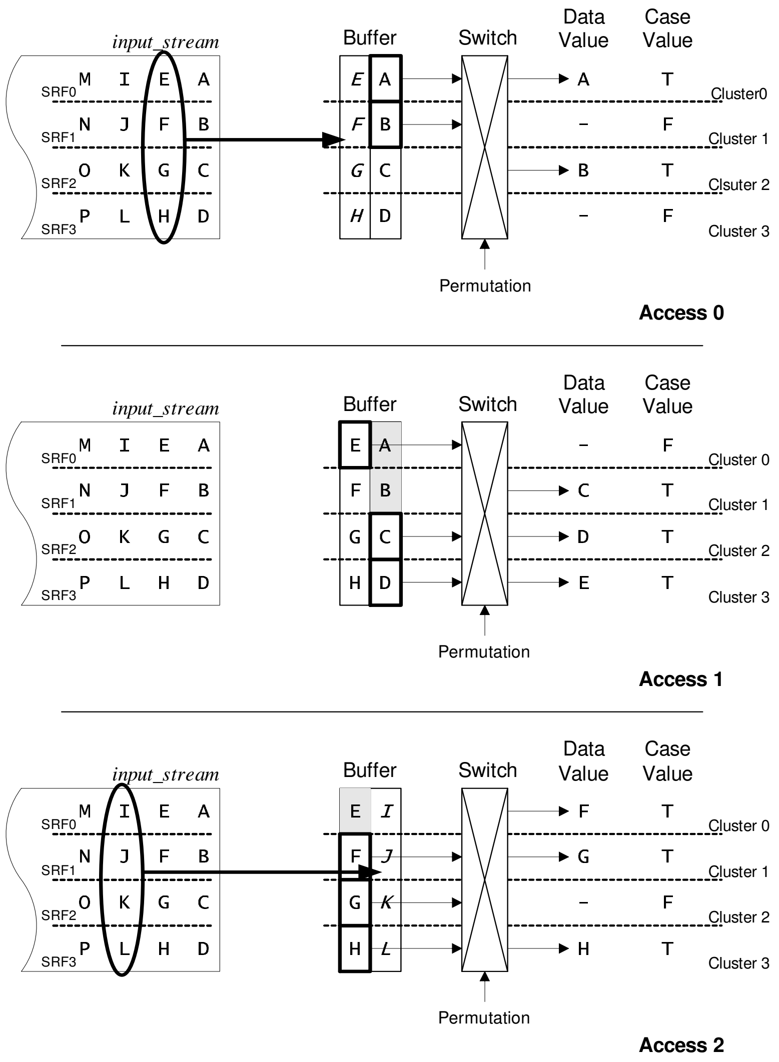


Figure A.2: Use of a buffer for conditional streams with Imagine-style stream buffers. The stream buffers are only read for the first and last conditional input stream operations.

own SRF on *IMAGINE*. Conditional streams solves the problem by buffering data and balancing the SRF accesses across the banks. Thus, the benefits of conditional routing we have discussed for SIMD machines in Chapter 4 would not be applicable to a stream processor with *IMAGINE*-style limited SRF control. This increases the importance of conditional streams on an *IMAGINE*-style architecture, because they are necessary, not only to achieve good load-balancing, but to also reap the benefits that conditional routing offer.

Bibliography

[Agarwala *et al.*, 2002] S. Agarwala, P. Koeppen, T. Anderson, A. Hill, M. Ales, R. Damodaran, L. Nardini, P. Wiley, S. Mullinnix, J. Leach, A. Lell, M. Gill, J. Golston, D. Hoyle, A. Rajagopal, A. Chachad, M. Agarwala, R. Castille, N. Common, J. Apostol, H. Mahmood, M. Krishnan, D. Bui, Q. An, P. Groves, L. Nguyen, N. Nagaraj, and R. Simar. A 600 MHz VLIW DSP. In *2002 International Solid-State Circuits Conference Digest of Technical Papers*, pages 56–57, 2002.

[Ahn *et al.*, 2003] Jung Ho Ahn, William J. Dally, Brucek Khailany, Ujval J. Kapasi, and Abhishek Das. Evaluating the Imagine stream architecture. Submitted to the 31st Annual International Symposium on Computer Architecture, November 2003.

[Allen *et al.*, 1983] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings 10th Annual Symposium on Principles of Programming Languages*, pages 177–189, January 1983.

[Cormen *et al.*, 1990] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, pages 665–667, 695–697. MIT Electrical Engineering and Computer Science Series. MIT Press, 1990.

[Dehnert and Bratt, 1989] James C. Dehnert and Peter Y.-T. Hsu Joseph P. Bratt. Overlapped loop support in the Cydra 5. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–38, Boston, MA, April 1989. ACM Press.

[Dulong, 1998] Carole Dulong. The IA-64 architecture at work. *IEEE Computer*, 31(7):24–32, July 1998.

[Eager *et al.*, 1986] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, 12(5):662–675, May 1986.

[Ebcioglu and Nakatani, 1990] K. Ebcioglu and T. Nakatani. A new compilation technique for parallelizing loops with unpredictable branches on a vliw architecture. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Research Monographs in Parallel and Distributed Computing*, pages 213–229. MIT Press, 1990.

[Eden and Mudge, 1998] A. N. Eden and T. Mudge. The YAGS branch prediction scheme. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 69–77, Dallas, TX, November 1998. IEEE Computer Society Press.

[Ellis, 1986] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. ACM doctoral dissertation award; 1985. The MIT Press, 1986.

[Fang *et al.*, 1990] Zhixi Fang, Peiyi Tang, Pen-Chung Yew, , and Chuan-Qi Zhu. Dynamic processor self-scheduling for general parallel nested loops. *IEEE Transactions on Computers*, 39(7):919–929, July 1990.

[Fisher, 1981] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–489, July 1981.

[Hwu *et al.*, 1993] W. W. Hwu, S. A. Mahlke, W. Y. Chen, N. J. Warter P. P. Chang, R. A. Bringmann, R. E. Hank R. G. Ouellette, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7:229–248, January 1993.

[Jayasena *et al.*, 2004] Nuwan Jayasena, Mattan Erez, Jung Ho Ahn, and William J. Dally. Stream register files with indexed access. In *To appear in Proceedings of the 10th International Symposium on High Performance Computer Architecture*, February 2004.

[Kanade *et al.*, 1996] Takeo Kanade, Atsushi Yoshida, Kazuo Oda, Hiroshi Kano, and Masaya Tanaka. A stereo machine for video-rate dense depth mapping and its new applications. In *Proceedings of the 15th Computer Vision and Pattern Recognition Conference*, pages 196–202, June 1996.

[Kapasi *et al.*, 2000] Ujval J. Kapasi, William J. Dally, Scott Rixner, Peter R. Mattson, John D. Owens, and Brucek Khailany. Efficient conditional operations for data-parallel architectures. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, pages 159–170, Monterey, CA, December 2000. ACM Press.

[Kapasi *et al.*, 2002a] Ujval J. Kapasi, William J. Dally, Scott Rixner, John D. Owens, and Brucek Khailany. The Imagine stream processor. In *Proceedings 2002 IEEE International Conference on Computer Design*, pages 282–288, Freiburg, Germany, September 2002.

[Kapasi *et al.*, 2002b] Ujval J. Kapasi, Peter Mattson, William J. Dally, John D. Owens, and Brian Towles. Stream scheduling. Concurrent VLSI Architecture Tech Report 122, Computer Systems Laboratory, Stanford University, Stanford, CA, March 2002.

[Kapasi *et al.*, 2003] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. Programmable stream processors. *IEEE Computer*, pages 54–62, August 2003.

[Keckler, 1994] Stephen W. Keckler. The importance of locality in scheduling and load balancing for multiprocessors. Concurrent VLSI Architecture Memo 61, MIT, February 1994.

[Khailany *et al.*, 2001] Brucek Khailany, William J. Dally, Ujval J. Kapasi, Peter Mattson, Jin Namkoong, John D. Owens, Brian Towles, Andrew Chang, and Scott Rixner. Imagine: Media processing with streams. *IEEE Micro*, pages 35–46, March/April 2001.

[Khailany *et al.*, 2003] Brucek Khailany, William J. Dally, Scott Rixner, Ujval J. Kapasi, John D. Owens, and Brian Towles. Exploring the VLSI scalability of stream processors. In *Proceedings of the Ninth Symposium on High Performance Computer Architecture*, pages 153–164, Anaheim, CA, February 2003.

[Khailany, 2003] Brucek Khailany. *The VLSI Implementation and Evaluation of Area- and Energy-Efficient Streaming Media Processors*. PhD thesis, Stanford University, Stanford, CA, June 2003.

[Lam, 1988] Monica S. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 318–328, Atlanta, GA, June 1988. ACM.

[Lowney *et al.*, 1993] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. ODonnell, and John C. Ruttenberg. The Multiflow trace scheduling compiler. *Journal of Supercomputing*, 7(1–2):51–142, May 1993. Special Issue on Instruction-Level Parallelism.

[Mahlke *et al.*, 1992] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 45–54. IEEE Computer Society Press, December 1992.

[Mattson *et al.*, 2000] Peter Mattson, William J. Dally, Scott Rixner, Ujval J. Kapasi, and John D. Owens. Communication scheduling. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 82–92, Cambridge, MA, November 2000.

[Mattson, 2001] Peter Mattson. *A Programming System for the Imagine Media Processor*. PhD thesis, Stanford University, Stanford, CA, 2001.

[McFarling and Hennessy, 1986] Scott McFarling and John Hennessy. Reducing the cost of branches. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 396–403, Tokyo, Japan, June 1986. IEEE Computer Society Press.

[Mohr *et al.*, 1991] Eric Mohr, David A. Kranz, and Robert H. Halstead Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[Montrym and Moreton, 2002] John Montrym and Henry Moreton. Nvidia GeForce4. In *Hotchips 14*, August 2002.

[Naffziger *et al.*, 2002] Samuel D. Naffziger, Glenn Colon-Bonet, Timothy Fischer, Reid Riedlinger, Thomas J. Sullivan, and Tom Grutkowski. The implementation of the Itanium 2 microprocessor. *IEEE Journal of Solid-State Circuits*, 37(11):1448–1460, November 2002.

[Owens *et al.*, 2000] John D. Owens, William J. Dally, Ujval J. Kapasi, Scott Rixner, Peter Mattson, and Ben Mowery. Polygon rendering on a stream architecture. In *Proceedings of the 2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 23–32, August 2000.

[Owens *et al.*, 2002] John D. Owens, Brucek Khailany, Brian Towles, and William J. Dally. Comparing Reyes and OpenGL on a stream architecture. In *2002 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 47–56, September 2002.

[Owens, 2002] John D. Owens. *Computer Graphics on a Stream Architecture*. PhD thesis, Stanford University, November 2002.

[Polychronopoulos and Kuck, 1987] Constantine D. Polychronopoulos and David J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputer. *IEEE Transaction on Computer*, 36(12):1425–1439, December 1987.

[Rai, 2003] Jathin Sanoor Rai. A feasibility study on the application of stream architectures for packet processing applications. Master's thesis, North Carolina State University, Raleigh, NC, 2003.

[Rajagopal *et al.*, 2002] Sridhar Rajagopal, Scott Rixner, and Joseph R. Cavallaro. A programmable baseband processor design for software defined radios. In *Proceedings of the 45th IEEE International Midwest Symposium on Circuits and Systems*, volume 3, pages 413–416, Tulsa, OK, August 2002. Invited Paper.

[Rajagopal, 2004] Sridhar Rajagopal. *Scalable Wireless Application-Specific Processors (SWAPs) For Emerging Wireless Systems*. PhD thesis, Rice University, Houston, TX, May 2004. (Expected).

[Rau *et al.*, 1989] B. Ramakrishna Rau, David W. L. Yen, Wei Yen, and Ross A. Towle. The Cydra 5 departmental supercomputer. *IEEE Computer*, 22(1):12–35, January 1989.

[Rixner *et al.*, 1998] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter R. Mattson, John D. Owens, Brucek Khailany, and Abelardo Lopez-Lagunas. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 3–13. IEEE Computer Society Press, 1998.

[Rixner *et al.*, 2000] Scott Rixner, William J. Dally, Brucek Khailany, Peter Mattson, Ujval J. Kapasi, and John D. Owens. Register organization for media processing. In *Proceedings of the Sixth International Symposium on High Performance Computer Architecture*, pages 375–387, January 2000.

[Rixner, 2001] Scott Rixner. *Stream Processor Architecture*. Kluwer Academic Publishers, Boston, MA, 2001.

[Russell, 1978] Richard M. Russell. The Cray-1 Computer System. *Communications of the ACM*, 21(1):63–72, January 1978.

[Seznec *et al.*, 2002] André Seznec, Stephen Felix, Venkata Krishnan, and Yiannakis Sazeides. Design tradeoffs for the Alpha EV8 conditional branch predictor. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 295–306, Anchorage, AL, May 2002. IEEE Computer Society Press.

[Sherryl and Pappas, 1990] Tomboulian Sherryl and Matthew Pappas. Indirect addressing and load balancing for faster solution to mandlebrot set on simd architectures. In *Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation*, pages 443–450, College Park, MD, October 1990.

[Slotnick *et al.*, 1962] Daniel L. Slotnick, W. Carl Borck, and Robert C. McReynolds. The Solomon computer. In *Proceedings of the Fall 1962 Eastern Joint AFIPS Computer Conference*, volume 22, pages 97–107, Philadelphia, PA, December 1962.

[Smith, 1981] James E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 135–148, Minneapolis, MN, May 1981. IEEE Computer Society Press.

[Stoodley and Lee, 1996] Mark G. Stoodley and Corinna G. Lee. Software pipelining loops with conditional branches. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 262–273, Paris, France, December 1996. IEEE Computer Society Press.

[Tang and Yew, 1986] P. Tang and P.-C. Yew. Processor self-scheduling for multiple-nested parallel loops. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 528–535, August 1986.

[TI, 2001] Texas Instruments. *TMS320C6713, Floating-Point Digital Signal Processors*, December 2001. Report SPRS186E, Revised July 2003.

[Towle, 1976] R. A. Towle. *Control and Data Dependence for Program Transformations*. PhD thesis, University of Illinois, Urbana-Champaign, 1976. Technical Report R-76-788, Dept. of Computer Science.

[von Hanxleden and Kennedy, 1992] Reinhard von Hanxleden and Ken Kennedy. Relaxing SIMD control flow constraints using loop transformations. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 188–199, 1992.

[Warter *et al.*, 1992] Nancy J. Warter, Grant E. Haab, Krishna Subramanian, and John W. Bockhaus. Enhanced modulo scheduling for loops with conditional branches. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 170–179, Portland, OR, December 1992. IEEE Computer Society Press.

[Warter-Perez and Partamnian, 1995] Nancy J. Warter-Perez and Noubar Partamnian. Modulo scheduling with multiple initiation intervals. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 111–118, Ann Arbor, MI, November 1995. IEEE Computer Society Press.

[Yeh and Patt, 1991] Tse-Yu Yeh and Yale N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 51–61. ACM Press, November 1991.