Stream Processor Architecture

by

Scott Rixner
Rice University

Contents

Foreword

We stand at the brink of a revolution in computer architecture. The conventional (von Neumann) processor architecture that has reigned for over 50 years is likely to be replaced over the next decade. This pending revolution is driven by two factors: the wire-limited nature of emerging semiconductor technology and the trend toward media applications. Professor Rixner, in this monograph, shows how architecture can evolve to match emerging technology to demanding media applications.

Advancing semiconductor technology enables us to pack an enormous amount of arithmetic capability on a single chip but also limits the communication latency and bandwidth both on and off chip. As wire widths shrink the delay of wires dominates the delay of transistors and gates. These trends place a premium on locality – to minimize communication – and concurrency – to make use of the large numbers of arithmetic units. Conventional processors can exploit neither locality nor concurrency. They have a single arithmetic unit and lump all storage at a given level (e.g., registers, cache, main memory) together — rather than localizing storage near its point of use.

Media applications also change the equation. Compared to conventional applications – for which processors have been tuned over decades – they have little spatial or temporal locality – a pixel of an image is often visited exactly once – and hence are poorly matched to conventional cache-based memory systems. They make use of low-precision fixed-point data types (16-bits is typical) as opposed to the high-precision (64-bit) data of scientific applications. They also have enormous amounts of data parallelism – all pixels can be processed in parallel – so they can tolerate considerable latency.

In this monograph, Professor Rixner shows how a stream processor architecture is ideally suited for applying emerging, wire-limited semiconductor technology to media applications. Stream processors operate by passing streams of data records through computation kernels. Operating on streams allows the concurrency of modern technology to be applied to the data parallelism of media applications. Explicitly dividing the application into kernels exposes locality on two levels. At one level, data within each kernel is entirely local to the kernel and can be kept entirely in local registers. At the next level, a stream generated by one kernel and consumed in the next can be passed through stream register file so it consumes no memory bandwidth — even if each pixel is only touched once. Stream processing shares its ability to exploit concurrency with vector processing. It is distinguished from vector processing in its ability to exploit locality.

Professor Rixner explores stream processing in the context of the Imagine stream processor which was developed in my research group at MIT and Stanford from 1996 to 2001. As described in Chapter 4 of this monograph, Imagine is a load-store architecture for streams. All operations pass streams into and out of a stream register file (SRF) that serves as a nexus for the processor. Load and store operations pass streams of records – not words or vectors – between memory and the SRF. Network operations send and receive streams of records. Finally, arithmetic kernels operate on one or more streams from the SRF generating new streams that are in turn written to the SRF. Professor Rixner explains this stream architecture in detail and shows how it efficiently supports the abstraction of stream programming.

After reviewing stream architecture and the Imagine stream processor, Professor Rixner addresses two aspects of optimizing bandwidth in stream processors: the use of a bandwidth hierarchy and the optimization of DRAM accesses. In Chapter 5, Professor Rixner shows how organizing storage into three levels – local registers, stream registers, and memory – can supply a large number of function units with data. The bandwidth hierarchy results in most data movement occurring where bandwidth is inexpensive – between local registers and adjacent ALUs. With order of magnitude reductions in bandwidth moving out to the stream registers and to memory respectively. This organization enables a relatively modest 2GB/s memory system in Imagine to support an ALU complex that can demand over 500GB/s of local bandwidth.

Memory access scheduling reorders memory operations to maximize the bandwidth realized from a conventional DRAM. Modern dynamic random access memories are in fact not random access. Their bandwidth and latency depends strongly on the access pattern. In Chapter 6, Professor Rixner shows that reordering memory operations to match the internal structure of the DRAMs can result in significant bandwidth improvement.

Stream processing is in its infancy, but it promises to be a major force in the coming decade. As the Imagine project moves toward completion and a working prototype the next steps in the evolution of stream processing are already taking shape. Major CPU vendors are evaluating the use of stream processors as media co-processors for conventional CPUs. Stream processing is being considered for a new generation of supercomputing. Also, researchers are looking at ways to generalize stream processing to make it applicable to a wider range of applications.

In this monograph, Professor Rixner gives the reader an accessible introduction to the emerging field of stream processing along with a scientific exploration of key bandwidth issues affecting the architecture of stream processing. Readers wanting to join the upcoming revolution in computer architecture are encouraged to read on.

William J. Dally
Stanford University
Palo Alto, California
August 1, 2001

Acknowledgements

A victorious army first wins and then seeks battle; a defeated army first battles and then seeks victory.

-Sun Tzu, *The Art of War*

I would like to thank Professor William J. Dally, to whom I am greatly indebted. He has an amazing ability to only fight the battles worth fighting which shows in the quality of his research program. I can only hope that some of that ability has rubbed off on me. Bill has provided me with countless opportunities to lead and to learn that I doubt I would have found otherwise. I am thankful to Bill for giving me the opportunity to take a leadership role on the Imagine project, which while not always easy, was a tremendous experience. Bill also gave me the opportunity to spend several years at Stanford University, and while I may not have embraced Stanford, the time I spent there was invaluable. Thanks also to the members of my thesis committee at MIT, Professors Leonard McMillan and Donald Troxel.

The Imagine project is the work of many, and this research would not have been possible without the contributions of others. I would like to thank the Imagine team: Ujval Kapasi, Brucek Khailany, Peter Mattson, John Owens, and many others who contributed along the way. We are all grateful to each other for pulling together to make such an ambitious project successful.

Many thanks go to my officemates Steve Keckler and Kelly Shaw. They not only put up with me all of those years, but also made my days as a graduate student enjoyable. I only hope I did the same for them.

I would also like to thank those who helped with the preparation of this document. Shelley Russell typed early versions of many chapters and proofread the final document. Without her help, my wrists may not have lasted to allow me to finish writing. Kelly Shaw spent countless hours proofreading all of the chapters at least once. Her comments significantly improved the overall quality of the writing.

My family has supported me throughout the process. There were many struggles and many accomplishments along the way, but through them all, they were always there. Thanks.

The research described here was supported by the Defense Advanced Research Projects Agency under ARPA order E254 and monitored by the Army Intelligence Center under contract DABT63-96-C-0037.

CHAPTER 1

Introduction

Media processing applications, such as three-dimensional graphics, video compression, and image processing, demand very high arithmetic rates. To meet the demands of these applications, a programmable media processor must support tens to hundreds of arithmetic units. In modern VLSI, hundreds of arithmetic units can easily fit on a modestly sized $1cm^2$ chip. The challenge, however, is to efficiently provide the necessary data bandwidth to keep those units busy. A stream architecture, which includes a *data bandwidth hierarchy*, enables a programmable media processor implemented in modern VLSI technology to efficiently provide data bandwidth for tens to hundreds of arithmetic units.

Applications must make efficient use of the available bandwidth in order to achieve high sustained computation rates. Media processing applications can naturally be expressed as a sequence of computation kernels that operate on data streams. These stream programs map easily and efficiently to the data bandwidth hierarchy of the stream architecture. This enables media processing applications to utilize inexpensive local data bandwidth when possible, and consume expensive global data bandwidth only when necessary.

The Imagine media processor implements a stream architecture to provide high sustained media processing performance. Imagine supports 48 arithmetic units with a three-tiered data bandwidth hierarchy, yielding a peak computation rate of 20 billion floating-point operations per second (GFLOPS). Applications are able to sustain a significant fraction of this peak rate. For example, QR matrix decomposition, useful in many signal processing applications, can be performed on Imagine at a sustained rate of 12.5GFLOPS.

1.1 Stream Architecture

The stream programming model exposes the locality and concurrency in media processing applications. In this model, applications are expressed as a sequence of computation kernels that operate on streams of data. A kernel is a small program that is repeated for each successive element in its input streams to produce output streams that are fed to subsequent kernels. Each data stream is a variable length collection of records, where each record is a logical grouping of media data. For example, a record could represent a triangle vertex in a polygon rendering application or a pixel in an image processing application. A data stream would then be a sequence of hundreds of these vertices or pixels. In the stream programming model, locality and concurrency are exposed both within a kernel and between kernels.

1.1.1 Stream Programming

To illustrate the stream programming model, consider the stream program for the intracoded frame (I-frame) encoder of an MPEG2 encoder. An MPEG2 encoder takes a video sequence as input and compresses the sequence of images into a single bit stream consisting of the following three types of frames: intracoded (I), predicted (P), and bidirectional (B). I-frames are compressed using only information contained in the current frame. P- and B-frames are compressed using information from the current frame and additional reference frames. This example application will be discussed further in subsequent chapters.

Figure 1.1 shows the kernel operations and stream data flow required to encode an I-frame. In the figure, the ovals represent computation kernels, the solid lines represent data streams, and the dashed lines represent scalar feedback. For example, the first kernel of the application, color conversion, takes one input stream, which is composed of the pixels of the original image in RGB format, and generates two output streams, which contain the luminance and the chrominance of the image.

The operations within each kernel are performed locally and independently on each stream element. Temporary data generated during the processing of each stream element does not need to be referenced during the processing of other stream elements or outside of the kernel. For example, the computations required to convert a pixel from

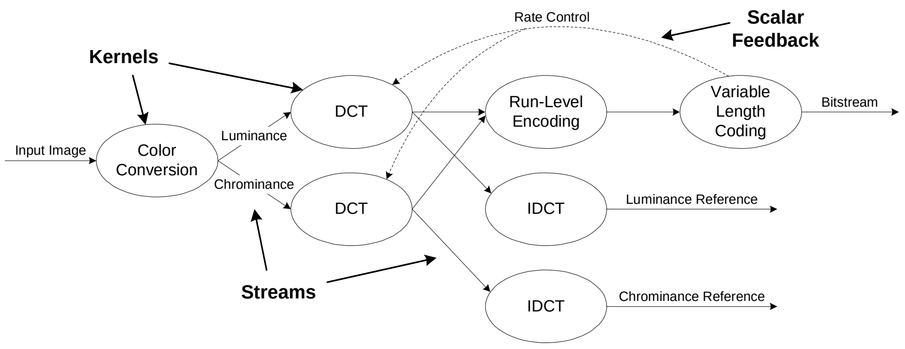


FIGURE 1.1 I-Frame Encoding from MPEG2 Encoder

RGB format to luminance and chrominance generate numerous temporary values. These temporaries do not affect other pixels and will never be referenced again after the pixel is converted. Therefore, that data is local to the kernel while it is processing each pixel in its input stream. Also, since the processing of each stream element is largely independent of the processing of other stream elements, multiple elements can be processed concurrently by the kernels. For instance, given enough hardware, the color conversion kernel could convert every pixel in the input image simultaneously, as the calculations for each pixel do not depend on the calculations for any other pixel.

As can be seen in the figure, kernels communicate between each other by passing data streams from one kernel to the next. The color conversion kernel, for example passes a stream of luminance values to the Discrete Cosine Transform (DCT) kernel. These localized stream transfers replace arbitrary global data references, yielding simple, regular communication patterns among kernels. Since kernels only pass data streams from one kernel to the next, multiple kernels can also operate concurrently on successive data streams in a pipelined fashion. One processor could convert the stream of RGB pixels into luminance and chrominance values in batches of several hundred pixels, while another processor could simultaneously transform previously converted batches of pixels into the frequency domain using the DCT.

1.1.2 Bandwidth Hierarchy

A stream architecture can exploit the locality inherent in media processing applications when they are expressed in the stream programming model. A stream architecture is organized around a three-tiered bandwidth hierarchy that includes a streaming memory system, a global stream register file, and local register files that feed the arithmetic units.

Figure 1.2 shows how the MPEG2 I-Frame encoder maps to the three-tiered bandwidth hierarchy. The figure also shows the number of bytes that are referenced by the application at each level of the hierarchy while a single frame is encoded. The seven computation kernels of the application execute on the arithmetic clusters, as shown in the figure. During the execution of each kernel, all of the temporary data that is computed and referenced during the course of the kernel is kept in the local register files. The stream register file is only referenced to read the kernel's inputs and to write the kernel's outputs. Also, data streams that are passed from kernel to kernel are stored within the stream register file and do not need to return to memory.

The highest level of the bandwidth hierarchy, the local distributed register files within the arithmetic clusters, exploits the locality within a kernel. Local register files are small, contain few ports, and are close to the arithmetic units. This enables them to collectively provide high data bandwidth cheaply and efficiently. Local data within a kernel can be stored in these register files so that it can be accessed frequently and quickly. The figure shows that the I-frame encoder references 154.4MB of data within these local register files. For example, all temporary values generated by the color conversion kernel in the process of converting pixels are stored in local registers. Since this data is not accessed outside of the kernel, it does not need to be stored in more expensive global storage. This allows kernels to utilize inexpensive local bandwidth when possible and only rely on more expensive global bandwidth when necessary.

In contrast, the arithmetic units in a conventional microprocessor are connected directly to a global register file. This forces the arithmetic units to consume expensive global data bandwidth to access local data. For example, to perform an FIR filter, Texas Instruments' TMS320C6203 ('C6) digital signal processor [TH97] accesses about 664 bytes of data per filter output in the global register file. The 'C6 consumes 24.9GB/s of data bandwidth out of that register file to achieve a sustained performance of 1 billion operations per second (GOPS). The Imagine stream processor, however, makes use of a bandwidth hierarchy to reduce the global data bandwidth demands. To perform the same FIR filter, Imagine only accesses about 4 bytes of data per filter output in the global register file. Instead of using global resources, tempo-

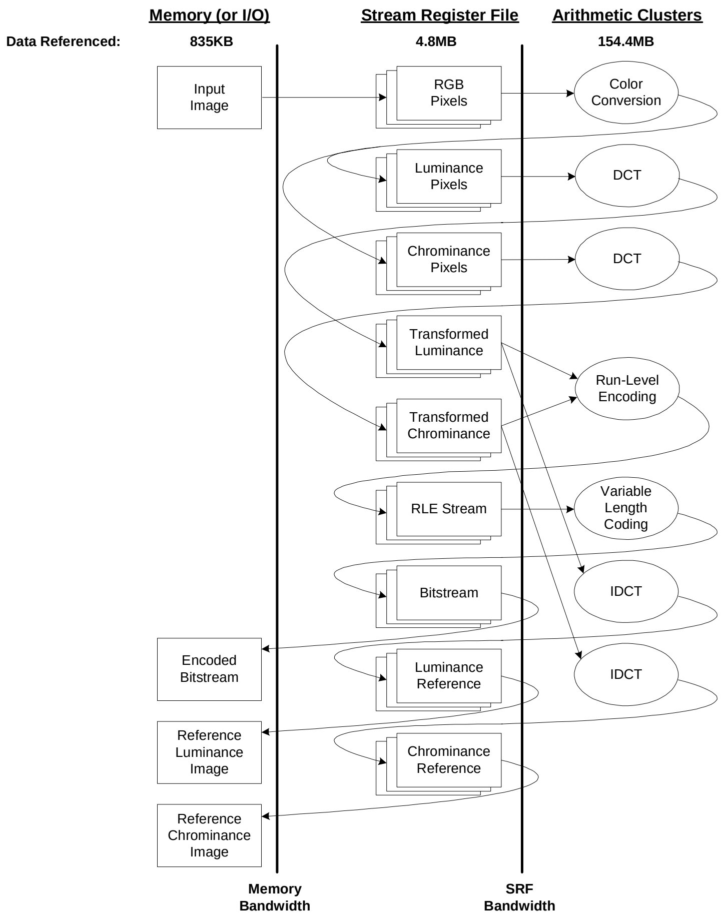


FIGURE 1.2 I-Frame Encoding from MPEG2 Encoder

rary data is stored and accessed in local register files. The consumption of global data bandwidth is drastically reduced to 2.6GB/s. By using over 273GB/s of inexpensive local data bandwidth, Imagine is able to achieve a performance of over 17GOPS.

The center of the bandwidth hierarchy, the stream register file, exploits the locality between kernels. The stream register file is a large global register file that allows data to be accessed sequentially as streams by the arithmetic clusters and the memory system. Kernels consume their input streams from the stream register file and store their produced output streams back to the stream register file. These output streams then become the input streams of subsequent kernels, enabling the stream register file to exploit the locality of stream recirculation. When a kernel finishes processing its streams, its outputs remain in the stream register file to be used as inputs for another kernel. For example, the luminance values that are generated by the color conversion kernel are stored in the stream register file until they can be processed by the DCT kernel; they never return to memory. The figure shows that the I-frame encoder references 4.8MB of data within the stream register file. This is 32 times fewer references than are made to the local register files. Storing intermediate streams in the global stream register file eliminates costly memory references, allowing stream programs to utilize global bandwidth when possible and rely on more expensive memory bandwidth only when necessary.

The lowest level of the bandwidth hierarchy, the streaming memory system, is utilized for global data storage. In the I-frame encoder, no intermediate data generated in the encoding process of the current frame is stored to memory. The only memory references that the application performs are to read the original, unencoded image, to write the final encoded bit stream, and to write the reference images that will be used to encode future P- or B-frames, resulting in 835KB of memory references. About six times less data is referenced in memory than in the stream register file and about 189 times less data is referenced in memory than in the local register files for this application.

1.1.3 Parallel Processing

A stream architecture can also exploit the concurrency inherent in media processing applications when they are expressed in the stream programming model. The bandwidth hierarchy scales the data bandwidth across multiple levels to support large numbers of arithmetic units. These arithmetic units are organized into single-instruction, multiple-data (SIMD) clusters that perform identical computations on different data elements. Each cluster contains multiple arithmetic units that are controlled by a very long instruction word (VLIW). The arithmetic clusters store temporary data locally, but transfer global data into and out of the stream register file. To enable multiple stream processors to be interconnected, the stream register file also connects to a network interface. The network interface allows entire data streams to be transferred from the stream register file of one processor to the stream register file of another processor through an external network.

The SIMD arithmetic clusters exploit the concurrency within kernels by operating on multiple stream elements simultaneously. A single, shared controller issues the same instructions to each cluster as they each operate on different elements of the kernel's input data streams. An arithmetic cluster contains multiple arithmetic units that exploit the instruction-level parallelism within the kernels. By using software pipelining, each cluster can also process multiple stream elements simultaneously. The combination of data parallelism, instruction-level parallelism, and software pipelining allows a stream processor to utilize large numbers of arithmetic units for media processing applications.

Multiple processors can exploit the concurrency among kernels by executing several kernels simultaneously. By splitting up the kernels in an application across the processors, multiple kernels can operate in parallel on different sections of the data. The first processor in the pipeline would execute one or more kernels to produce output streams that would then be passed to the next processor. As the next processor operates on those streams, the original processor could repeat its kernels on the next set of input data. Depending on the amount of parallelism in the program, this pipeline could be arbitrarily wide, in terms of the number of processors executing the same kernels across the data, or deep, in terms of the number of processors in the pipeline.

1.2 The Imagine Media Processor

Imagine is designed to be a proof-of-concept programmable stream processor showing the viability of structures such as the bandwidth hierarchy to enable scalable microprocessors that deliver high sustained multimedia performance. Imagine is organized around a three-tiered data bandwidth hierarchy consisting of a streaming memory system (2GB/s), a global stream register file (32GB/s), and a set of local distributed register files located near the arithmetic units (544GB/s). The bandwidth hierarchy of Imagine therefore provides memory bandwidth, global register bandwidth, and local register bandwidth with a ratio of 1:16:272. For each word accessed in memory, 16 words may be accessed in the global stream register file, and 272 words may be accessed in the clusters' local register file. At the lowest level, the memory system is designed to maximize the bandwidth of stream data transfers from external DRAM, rather than minimize the latency of individual references. The 128KB SRF at the center of the bandwidth hierarchy not only provides intermediate storage for data streams but also enables additional stream clients to be modularly connected to Imagine, such as a streaming network interface or a graphics texture caching unit. At the highest level, local, distributed register files feed 48 arithmetic units organized into a SIMD array of eight identical clusters.

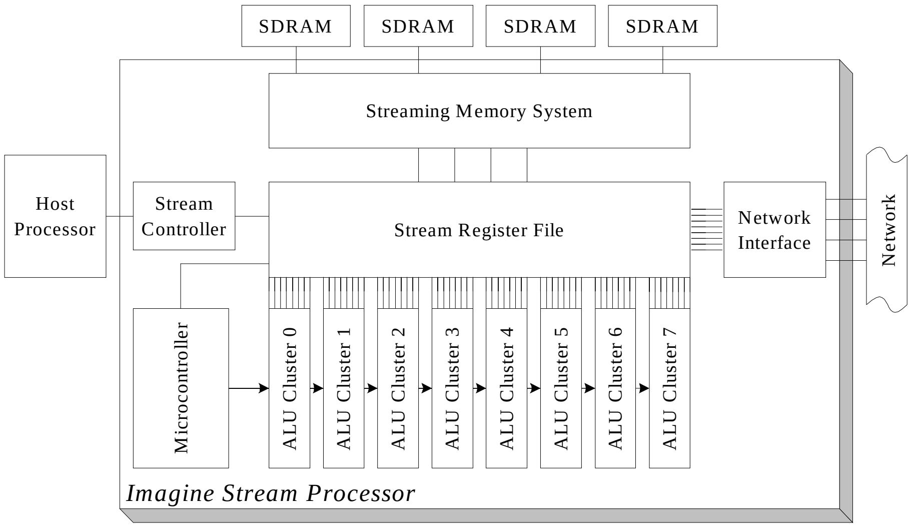


FIGURE 1.3 Imagine Block Diagram

Figure 1.3 shows a block diagram of Imagine's microarchitecture. Imagine is designed to be a coprocessor that operates on multimedia data streams. The stream register file (SRF) effectively isolates the arithmetic units from the memory system, making Imagine a load/store architecture for streams. As shown in the figure, eight arithmetic clusters, a microcontroller, a streaming memory system, a network interface, and a stream controller are connected to the SRF. The arithmetic clusters consume data streams from the SRF, process them, and return their output data streams to the SRF. The eight clusters operate simultaneously on interleaved data records transferred from the SRF, allowing eight elements to be processed in parallel. Each arithmetic cluster contains three adders, two multipliers, and one divide/square root unit. These units are controlled by statically scheduled VLIW instructions issued by the microcontroller.

Imagine is targeted to operate at 500MHz in a 0.15µm CMOS standard-cell implementation. The peak computation rate of the 48 arithmetic units is 20GOPS for both 32-bit integer and floating-point operations. For 16-bit and 8-bit parallel-subword operations, the peak performance increases to 40 and 80GOPS, respectively.

1.3 Contributions

The primary contributions of this research are as follows:

1. The concept of a bandwidth hierarchy which efficiently utilizes data bandwidth for media processing applications at three levels: local register files, global register files, and off-chip memory.
2. The architecture of a memory access scheduler to maximize the sustainable throughput of modern DRAM.
3. The architecture of a bandwidth-efficient stream processor that can sustain a significant fraction of its peak performance on media processing applications.
4. An experimental evaluation of the data bandwidth demands of media processing applications, the effectiveness of the bandwidth hierarchy, and the benefits of memory access scheduling.

1.4 Overview

This work focuses on the efficient use of data bandwidth for media processing applications. Chapter 2 presents background information on media processors and bandwidth management. The chapter shows that media processors have mirrored the designs of more conventional programmable processors with slight deviations. The storage hierarchy of a conventional processor, however, is not able to provide sufficient data bandwidth for media processing, as it is designed to minimize latency, rather than maximize bandwidth.

Chapter 3 presents the differentiating characteristics of media processing applications. These applications operate on streams of low-precision integer data, have abundant data parallelism, do not reuse global data frequently, and perform tens to hundreds of computations per global data reference. These characteristics differ from those of traditional applications for which modern programmable processors have been designed, motivating a departure from conventional microarchitectures for media processing.

Chapter 4 presents the architecture of the Imagine stream processor. Imagine is organized around a three-tiered bandwidth hierarchy that effectively supports 48 arithmetic units for media processing applications. Imagine is designed to demonstrate that an efficient bandwidth hierarchy enables high sustained multimedia performance. Imagine is expected to fit on a chip about 2.5cm^2 in size and deliver a peak performance of 20GFLOPS. Media processing applications can actually achieve and sustain from a quarter to over half of the peak computation rate.

Chapter 5 introduces the concept of a data bandwidth hierarchy that efficiently provides data bandwidth to the arithmetic units of a media processor. The chapter shows how technology constraints and media processing characteristics motivate microarchitectures for media processing that include storage hierarchies that scale the provided data bandwidth across multiple levels. A data bandwidth hierarchy enables a media processor to effectively utilize tens to hundreds of arithmetic units to sustain high computation rates.

Chapter 6 introduces the concept of memory access scheduling to maximize the sustained bandwidth of external DRAM at the lowest level of the data bandwidth hierarchy. The structure of modern DRAMs makes their achievable throughput and latency highly dependent on the access pattern. Memory access scheduling takes advantage of the internal structure of DRAM to reorder DRAM operations to maximize sustained throughput.

Finally, Chapter 7 reiterates the importance of careful bandwidth management to maximize media processing performance subject to the constraints of modern VLSI technology.

CHAPTER 2 Background

In order to provide the peak computation rates that media applications demand, modern media processors typically take advantage of special-purpose hardware. These processors sacrifice the flexibility of programmability but provide high multimedia performance by efficiently utilizing silicon resources. Special-purpose solutions manage bandwidth efficiently for the particular application they target. Programmable processors, in contrast, must rely on more flexible structures to support a wider range of applications. The data parallelism inherent to media processing applications enables the use of efficient SIMD and vector operations, but programmable processors still must be able to communicate data among arithmetic units and memory in a flexible manner. Therefore, these processors have traditionally included cache or register hierarchies. To be effective for media processing, these hierarchies require high sustained bandwidth from external memory. However, the techniques used to maximize bandwidth largely neglect the scatter/gather access patterns commonly found in complicated media processing applications, such as MPEG encoding and three-dimensional graphics.

2.1 Special-purpose Media Processors

Media processing systems typically employ dedicated, special-purpose hardware to meet the performance demands of media applications, especially three-dimensional graphics. For example, the InfiniteReality from SGI is a special-purpose multiprocessor system designed to provide high sustained graphics performance [MBDM97]. The InfiniteReality is composed of four custom Geometry Engines and 80-320 custom Image Engines. Combined, these special-purpose processors are able to deliver graphics rendering performance of up to 11 million triangles per second and 830 million pixels per second. Overall, the InfiniteReality achieves high performance graphics by distributing storage throughout the system which holds the appropriate data for different processing stages. This distributed storage provides a large amount of memory bandwidth at the cost of excessive amounts of memory capacity.

Recently, special-purpose, single-chip graphics processors have been able to provide the computational power of an InfiniteReality system. For example, the NVIDIA GeForce 256 provides a peak rendering performance of 15 million vertices per second and 480 million pixels per second [Gla99]. Special-purpose processors, such as the GeForce, are able to achieve high performance by customizing the processor to the demands of a single application and carefully managing bandwidth and computation resources on the chip. Both the InfiniteReality and the GeForce are specifically tailored for certain rendering algorithms, so performance rapidly degrades when there are any deviations from those algorithms.

2.2 Programmable Media Processors

Programmable media processors, including Chromatic's Mpact [Fol96], Philips' TriMedia [RS96], Philips' VSP [VPPL94], are far more flexible than special-purpose graphics systems. Media processors typically have architectures similar to digital signal processors but are augmented with special-purpose hardware pipelines and instructions for media processing. These processors also may have specialized storage hierarchies. For example, Mpact includes a 4KB global register file and makes use of high bandwidth Rambus DRAM to achieve the required data bandwidth to perform 3-D graphics. This reasonably large global register file significantly decreases the amount of memory bandwidth required by explicitly caching the working set of a graphics application, thereby minimizing transfers to memory.

Programmable media processors, however, have largely been replaced by multimedia extensions to general-purpose processors. These extensions, including MMX [PW96], MAX-2 [Lee96], and VIS [TONH96], perform SIMD operations on multiple narrow integer data values stored within the wide registers of general-purpose processors. More recently, SSE extends MMX to add support for prefetching stream data from memory and for SIMD floating point operations [Die99] [TH99]. Multimedia extensions such as these improve the performance of media processing kernels by exploiting fine-grained data parallelism. Furthermore, packed data types efficiently utilize the data bandwidth available in these processors by transferring several values in each wide data word.

2.3 Vector Processors

Vector processors are a natural choice to exploit the data parallelism inherent in media processing applications [Koz99] [LS98]. A vector consists of a fixed-length set of data words. A single vector instruction performs multiple, identical operations on these vectors. To efficiently exploit the data parallelism in vectors, the architecture of a vector processor commonly includes some combination of a vector register file, deeply pipelined arithmetic units, and a SIMD organization [Lin82] [Oed92] [Rus78]. A vector register file stores vectors of data, rather than individual words, that are transferred sequentially during vector operations. Vector processors, whether or not they include a vector register file, transfer entire vectors, rather than individual words, between the processor and memory. Vector memory transfers hide the memory access latency of individual word references. The arithmetic units of a vector processor are usually deeply pipelined and organized in a SIMD fashion. The data parallelism inherent in vector operations allows deep pipelining and SIMD processing, since it is guaranteed that all of the operations on the elements of a vector will be independent of each other. Therefore, the arithmetic unit pipelines will remain full for the duration of the vector operation.

2.4 Stream Processors

Media processing applications can naturally be expressed as a series of computations performed on streams of data. Systems such as Cheops [BW95] and SYDAMA [GMG89] directly map the stream dataflow graphs of media processing applications to one or more stream processing elements. For example, Cheops consists of a set of specialized stream processors that each accept one or two data streams as inputs and generate one or two data streams as outputs. Each stream processor is designed to perform a specific video processing function. To allow some flexibility, a general-purpose programmable processor is also included in the system. Functions not provided by the specialized stream processors can then be implemented in software on the general-purpose processor. Cheops stores data streams in VRAM[1]. Direct memory access controllers use the random access VRAM ports to reorganize data that will later be processed using the sequential access port. Data streams may either be forwarded directly from one stream processor to the next based on the application's dataflow graph or transferred between the VRAM and the stream processors. Stream processors are able to exploit the inherent streaming nature of media processing applications by directly mapping the dataflow graph of the application to hardware.

2.5 Storage Hierarchy

Since programmable processors cannot tailor communication and storage resources to specific applications or algorithms, a storage hierarchy is required to provide flexible communication and storage for large amounts of data. The storage hierarchy in a programmable processor can be controlled either by hardware or software. Hardware managed caches appeared as early as in the IBM 360/85 [Lip68]. Today, almost all microprocessors include a hardware managed cache hierarchy to reduce memory latency and improve performance [Die99] [HL99] [Kes99]. Caching takes advantage of spatial and temporal locality of reference to minimize average memory latency. Applications that exhibit large amounts of locality benefit from a cache's ability to dynamically store frequently accessed data close to the arithmetic units.

Several processors incorporate a software managed register hierarchy to reduce the latency to access frequently referenced data. Such processors have a small global register file connected to the arithmetic units and a larger register file that holds additional data. For example, the Cray-1 included a 64-entry *T* register file which acted as backup storage to an 8-entry *S* register file [Rus78]. The Cray-1's memory system did not include a cache; the compiler stores frequently accessed values in the *T* registers so that they do not need to be refetched from memory. Texas Instruments TMS320C6000 digital signal processors, representative of modern DSPs, include two small register files that feed the arithmetic units and a large, on-chip memory that can be configured either as cache or as addressable storage [TH97]. By configuring the on-chip memory as addressable storage, media data can be staged from memory and held during computations when the register files near the arithmetic units do not have enough space.

1. Video RAM (VRAM) is two-ported DRAM with one port allowing random access and the other port allowing fast sequential access.

2.6 DRAM Access Scheduling

Storage hierarchies minimize the amount of data that must be referenced in external DRAM. However, DRAM bandwidth can still be a performance bottleneck, especially for media processing applications. Several mechanisms have been employed to improve the performance of external DRAM on streaming reference patterns, which are common to media processing.

Stream buffers prefetch data structured as streams or vectors to hide memory access latency [Jou90]. Stream buffers do not, however, reorder the access stream to take advantage of the internal structure of DRAM. For streams with small, fixed strides, references from one stream tend to make several column accesses for each row activation, giving good performance on a modern DRAM. However, conflicts with other streams and non-stream accesses often evict the active row of the DRAM, thereby reducing performance. McKee's Stream Memory Controller (SMC) extends a simple stream buffer to reduce memory conflicts among streams by issuing several references from one stream before switching streams [HMS+99] [MW95]. The SMC, however, does not reorder references within a single stream.

The Command Vector Memory System (CVMS) [CEV98] reduces the required processor to memory address bandwidth by transferring *commands* to the memory controllers, rather than individual references. A command includes a base and a stride which is expanded into the appropriate sequence of references by each off-chip memory bank controller. The bank controllers in the CVMS schedule accesses among commands to improve the bandwidth and latency of the SDRAM. The Parallel Vector Access unit (PVA) [MMCD00] augments the Impulse memory system [CHS+99] with a similar mechanism for transferring commands to the Impulse memory controller. Neither of these systems reorder references within a single stream. Conserving address bandwidth, as in the CVMS and the PVA, is important for systems with off-chip memory controllers but is largely orthogonal to scheduling memory accesses.

The SMC, CVMS, and PVA do not handle indirect (scatter/gather) streams. These references are usually handled by the processor cache, as they are not easily described to a stream prefetching unit. However, indirect stream references do not cache well because they lack both spatial and temporal locality. These references also do not typically make consecutive column accesses to the same row, severely limiting the sustainable data bandwidth when those references are satisfied in order.

Several memory access schedulers have been proposed as part of systems-on-a-chip [WAM+99] [YHO97]. Hitachi has built a test chip of their access optimizer for embedded DRAM that contains the access optimizer and DRAM [WAM+99]. A simple scheduler is implemented which performs accesses for the oldest pending reference that can access the DRAM subject to timing and resource constraints. The access optimizer is 1.5mm^2, dissipates 26mW, and runs at 100MHz in a 0.18μm process. More aggressive scheduling would require more logic and slightly increase the area and power of such an access optimizer.

2.7 Summary

A stream architecture, which will be described here, bridges the performance gap between special-purpose and programmable media processors. A stream architecture exploits the data parallelism and instruction-level parallelism inherent in media processing applications by directly operating on the applications' data streams in a single processor.

Previous stream processors have passed data streams among processing elements and have stored streams in dedicated memories or external DRAM. The stream architecture that will be described here differs from previous stream processors in that the entire stream dataflow from an application can be mapped to a single processor by using an efficient storage hierarchy. The storage hierarchy of such a stream architecture differs from both the conventional cache hierarchy and register hierarchy in that it is optimized for data bandwidth and is organized expressly to transfer data streams.

A stream processor differs from a vector processor in a few key ways. First, a *stream* consists of a set of data records, where each record is one or more related data words. Each stream can have a different length. In contrast, vectors all have the same length and are composed of a set of single data words. Second, a single stream instruction performs multiple, independent computations on these streams. Stream instructions perform an entire function, or *kernel*, on successive stream elements, whereas vector instructions perform primitive arithmetic operations on successive vector elements. Finally, the arithmetic units within a stream processor are organized into arithmetic clusters that include local storage. An arithmetic cluster performs the operations within a kernel on each stream element, so the arithmetic units within a cluster exploit the instruction-level parallelism and the locality of the kernel. Similar to a vector processor, multiple SIMD arithmetic clusters exploit the data parallelism of the stream operation by performing identical processing on interleaved stream elements.

The streaming memory system of a stream processor transfers streams of data between the processor and external DRAM, rather than individual words as in a conventional memory system. Many previous methods of scheduling memory accesses have reordered references among such streams to maximize the sustained bandwidth from external DRAM. They typically have not, however, reordered references within a single stream to maximize the bandwidth of scatter/gather streams. The memory access scheduling techniques presented here reorder memory accesses independent of the streams from which they were generated, allowing scatter/gather stream accesses to be scheduled as effectively as strided accesses. Furthermore, the hardware organization and scheduling algorithms that will be introduced more aggressively exploit the internal structure of the DRAM to maximize sustained memory bandwidth.

CHAPTER 3

Media Processing Applications

Media processing applications operate on streams of low-precision data, have abundant data-parallelism, rarely reuse global data, and perform tens to hundreds of operations per global data reference. These characteristics differ from the traditional applications for which modern programmable microprocessors are designed. This chapter describes the distinguishing characteristics of media processing applications and presents five applications that will be studied throughout this monograph.

3.1 Media Processing

Media processing applications, such as video compression, three-dimensional graphics, and image processing, are prominent consumers of computing cycles. These applications have entered the mainstream computer market and users have come to expect high quality multimedia computing. Media applications demand very high arithmetic rates, 10-100 billion operations per second, and these demands will continue to grow as applications become more realistic and compelling.

Media processing applications, which have requirements similar to signal processing applications, differ significantly from the applications for which traditional microprocessors are designed. Conventional applications operate on high-precision data, have complex control flow and many data dependencies, exhibit large amounts of temporal and spatial locality of reference, and perform very few operations per global data reference. In contrast, media applications operate on low-precision data, have abundant data-parallelism, rarely reuse global data, and perform tens to hundreds of operations per global data reference. The differences between these two families of applications limit the performance of media processing applications when they are run on modern programmable microprocessors optimized for traditional applications.

3.2 Sample Applications

This section briefly describes five representative media processing applications: stereo depth extraction, MPEG2 encoding, QR matrix decomposition, space-time adaptive processing and polygon rendering. Stereo depth extraction (DEPTH) is a representative image processing application that computes the depth of pixels within an image. MPEG2 encoding (MPEG) is a representative video compression application that generates a compressed bit stream from a video sequence. QR matrix decomposition (QRD) and space-time adaptive processing (STAP) are representative signal processing applications that can be used for radar processing. Finally, polygon rendering (RENDER) is a representative graphics application that synthesizes images from polygonal models. In the following section, these applications will be characterized to quantify some important properties of media processing applications.

3.2.1 Stereo Depth Extraction

DEPTH takes a pair of 320x240 8-bit grayscale camera images of the same scene as input and produces a depth map of that scene using Kanade’s multi-baseline stereo algorithm [KKK95]. The depth map encodes the distance from one of the cameras, designated as the reference, to each pixel in the scene.

3.2.2 MPEG2 Encoding

MPEG takes three frames of 360x288 24-bit color video and compresses them into an MPEG2 compliant bit stream [Jac96]. An MPEG encoder can encode sequence of images into a single bit stream consisting of the following three types of frames: intracoded (I), predicted (P), and bidirectional (B). The three types of frames differ in which images are used to compress them. I-frames are compressed using only information contained in the current frame. P-frames are compressed using information from the current frame and the previous I- or P-frame, allowing still and slowly moving objects to be encoded more compactly than they would be in an I-frame. Finally, B-frames are compressed using information from the current frame, the previous I- or

P-frame, and the next I- or P-frame. B-frames achieve the highest compression ratio and decrease noise by using two images as references. The MPEG application encodes its three input frames into one I-frame and two P-frames. The output of this application is the run-level encoding that would be fed into a Huffman encoder.

3.2.3 QR Matrix Decomposition

QRD uses the compact-WY [SV89] representation of the blocked-Householder transform to compute an orthogonal Q matrix and an upper triangular R matrix such that Q·R is equal to the input matrix [GV96]. This can be used to solve linear equations that are found in a variety of signal and radar processing applications. QRD operates on a 192x96 element matrix of floating-point numbers.

3.2.4 Space-time Adaptive Processing

STAP is used to cancel clutter and interference in airborne radar images. STAP includes preprocessing, Doppler processing, weight computation, and weight application. The preprocessing steps prepare the raw data from the antenna for adaptive processing. The Doppler processing step attenuates signals that are moving at the same velocity and in the opposite direction of the radar, eliminating ground clutter from the return. The weight computation step involves performing several QR matrix decompositions to find the adaptive filter weights. Finally, these weights are applied to the radar signal to generate the output of the STAP algorithm. The MITRE RT_STAP *easy* benchmark is used for this implementation [CTW97]. The easy benchmark uses a post-Doppler adaptive displaced phase center antenna algorithm [Nit99] to compensate for the motion of the antenna and is equivalent to current airborne radar systems.

3.2.5 Polygon Rendering

RENDER takes three-dimensional polygonal models of objects as input and produces a 24-bit color image composed of these objects projected onto a two-dimensional 720x720 pixel viewing area. RENDER performs back-face culling to eliminate polygons that are facing away from the viewer and applies modelview, projection, and viewport transformations to generate polygons in the two-dimensional screen space. These polygons are then rasterized and textured using depth buffering and mipmapping to generate the final image. A complete description of polygon rendering can be found in [FvFH96] and a description of the RENDER application (called ADVS-8 in the paper) can be found in [ODK+00].

3.3 Application Characteristics

These five representative media processing applications share four important characteristics: they operate on low-precision data, they exhibit abundant data parallelism, they rarely reuse global data, and they perform tens to hundreds of operations per global data reference. This section shows the precision and type of operations used in these applications, the size of their global data, and their computational intensity. The numbers collected to support these characteristics are from the implementation of these applications for the Imagine stream processor. However, the data is collected independently of the applications' execution on the Imagine processor.

3.3.1 Operations

Table 3.1 shows the precision and type of operations performed by the media processing applications. The five applications perform between 15 and 154 million operations over the course of their execution. The table shows the percentage of operations in each application by precision and type. The types of operations are divided into three categories: add/sub, multiply, and other. The add/sub category includes all addition and subtraction operations. The multiply category includes all types of multiplication operations. Finally, the other category includes all other operations, such as divide, shift and logical operations. The table shows that the bulk of these operations are on low-precision data. For instance, over 56% of the operations performed by MPEG are 8-bit integer addition or subtraction operations.

Both DEPTH and MPEG perform only integer operations. Their data is mostly 8-bit and 16-bit integers. A few 32-bit integer operations are performed, but these operations are mostly for bookkeeping purposes and do not manipulate image data. In contrast, QRD and STAP perform mostly floating point operations. Again, a few 32-bit integer operations are performed by these applications, but they are for bookkeeping purposes and do not manipulate media data. Even though these applications use floating point data for its range, they still use low-precision data, as they are able to make effective use of single-precision (32-bit) floating point operations, instead of the double-precision (64-bit) operations found in many modern microprocessors.

The table shows that almost half of the operations performed collectively by these applications are 8-bit and 16-bit addition and subtraction operations. Most of the rest of the operations are 32-bit floating point addition, subtraction, and multiplication. Therefore, the bulk of media operations found in these applications operate on low-precision integer and floating point data.

		DEPTH	MPEG	QRD	STAP	RENDER	*Total*
Operations (Millions)		54.5	154.0	15.1	82.4	57.2	*363.2*
32-bit Floating Point	add/sub (%)	0	0	50.0	49.7	18.3	*16.2*
	multiply (%)	0	0	49.0	46.2	15.8	*15.0*
	other (%)	0	0	0	0.5	6.7	*1.2*
32-bit Integer	add/sub (%)	0.2	3.2	0.2	0.6	10.9	*3.2*
	multiply (%)	0	0	0	0	0.3	*0.1*
	other (%)	4.8	9.5	0.8	3.0	34.7	*10.9*
16-bit Integer	add/sub (%)	70.9	21.6	0	0	5.2	*20.6*
	multiply (%)	19.2	4.2	0	0	8.2	*6.0*
	other (%)	4.9	5.2	0	0	0	*2.9*
8-bit Integer	add/sub (%)	0	56.3	0	0	0	*23.9*
	multiply (%)	0	0	0	0	0	*0*
	other (%)	0	0	0	0	0	*0*

TABLE 3.1 Precision and Type of Operations in Media Processing Applications

3.3.2 Global Data

Table 3.2 shows the size of the global data in these media applications, how much of that data is accessed, and how many times it is accessed. The size of the global data in these applications ranges from only 144KB up to almost 6MB. All of the applications, except for RENDER, reference all of their global data during the course of execution. The global data in the RENDER application includes a mipmapped texture map and the frame buffer. The texture map can be very large, and much of the map is not accessed during the rendering of a single image. This results in a large amount of global data that is never used. There is no reason why all of the data from all of the levels of the mipmap would be referenced to draw any particular frame of an image, and in fact, studies have shown that usually only a small fraction of the texture map is referenced [HG97]. RENDER also includes a large frame buffer that stores both depth and color information for each pixel in the final image. The RENDER application does not draw the background of the image, so only pixels that appear in the foreground are referenced in the frame buffer. In this scene, almost 89% of the frame buffer is never referenced.

	DEPTH	MPEG	QRD	STAP	RENDER	*Total*
Global Data (KB)	450.0	2633.0	144.0	2160.0	6020.9	*11407.9*
Unique Global Data Referenced (KB)	450.0	2633.0	144.0	2160.0	1896.1	*7283.1*
Total Global Data Referenced (KB)	450.0	3875.0	793.0	2160.0	3858.8	*11136.8*
Accesses per Unique Data Referenced	1.00	1.47	5.51	1.00	2.04	*1.53*

TABLE 3.2 Size and Reuse of Global Data in Media Processing Applications

The table also shows the total amount of global data that must be referenced over the course of execution of these applications. Additional accesses performed by the particular implementation of the programs are ignored, as are aggressive optimizations that eliminate global data accesses. This eliminates the effects of inadequately sized register files and other hardware limitations. By dividing the total amount of referenced data by the unique amount of referenced data, the average number of accesses per global data element can be determined. As shown in the table, global data is referenced between 1 and 5.5 times for these applications. Therefore, the bulk of the global data is rarely reused.

3.3.3 Compute Intensity

Table 3.3 shows the computational requirements of these media processing applications. The number of operations and the total amount of global data referenced are repeated from Tables 3.1 and 3.2. The bottom row of the table shows the number of operations that are performed for each word of global data that is referenced, where a word is assumed to be four bytes. As can be seen in the table, between 58 and 473 operations are performed by these applications for each word of global data that is referenced.

3.3.4 Summary

The five applications considered in this chapter operate mostly on 8-bit and 16-bit integers, contain abundant data parallelism, access each unique global data item an average of only 1.53 times, and perform 127.4 operations per global data reference.

	DEPTH	MPEG	QRD	STAP	RENDER	*Total*
Operations (Millions)	54.5	154.0	15.1	82.4	57.2	*363.2*
Total Global Data Referenced (KB)	450.0	3875.0	793.0	2160.0	3858.8	*11136.8*
Operations per Referenced Word (4 bytes)	473.3	155.3	74.3	149.0	57.9	*127.4*

TABLE 3.3 Operations per Global Data Reference

To achieve high sustained media performance, programmable processors must be designed to exploit these characteristics, rather than those of traditional applications for which most microprocessors have been designed.

CHAPTER 4

The Imagine Stream Processor

The Imagine stream processor is a programmable media processor designed to process data streams. Imagine is organized around a three-tiered data bandwidth hierarchy consisting of a streaming memory system (2GB/s), a global stream register file (32GB/s), and a set of local distributed register files near the arithmetic units (544GB/s). This hierarchy therefore provides memory bandwidth, global register bandwidth, and local register bandwidth with a ratio of 1:16:272. For each word accessed in memory, 16 words may be accessed in the global stream register file (SRF), and 272 words may be accessed in the clusters' local register files. At the lowest level, the memory system is designed to maximize the bandwidth of stream data transfers from external DRAM, rather than minimize the latency of individual references. The 128KB SRF at the center of the bandwidth hierarchy not only provides intermediate storage for data streams, but also enables additional stream clients to be modularly connected to Imagine, such as a streaming network interface or a graphics texture caching unit. At the highest level, local distributed register files feed 48 arithmetic units organized into a single-instruction, multiple-data (SIMD) array of eight identical clusters. Imagine is designed to be a proof-of-concept stream processor showing the viability of structures such as the bandwidth hierarchy and the streaming memory system to enable scalable microprocessors that deliver high sustained multimedia performance. In a 0.15μm CMOS standard-cell implementation, Imagine will operate at 500MHz, yielding a peak performance of 20GFLOPS on a 2.5cm^2 chip. This chapter explains the Imagine architecture, its stream programming model, and its scalability.

4.1 Stream Processing

Technology and application trends together motivate a shift away from the scalar, general-purpose register architecture in wide use today toward a stream-based architecture with a bandwidth-efficient register organization. A 32-bit floating-point multiplier requires less than 0.25mm^2 of chip area in a contemporary 0.18µm CMOS technology, so hundreds of arithmetic units can fit on an inexpensive 1cm^2 chip. The challenge, however, is keeping these units fed with instructions and data. It is infeasible to support tens to hundreds of arithmetic units with a single register file and instruction issue unit. Therefore, partitioned structures that use local communication are required to take advantage of the available computation resources.

Media processing applications, including rendering 2-D and 3-D graphics, image and video compression and decompression, and image processing, contain significant amounts of locality and concurrency. These applications operate on large *streams* of low-precision integer data and share the following three key characteristics: operations on elements within streams are data-parallel, global data is rarely reused, and as many as 100-200 operations are performed per global data reference. These applications are poorly matched to conventional architectures that exploit very little parallelism, depend on data reuse, and can perform very few operations per memory reference.

Fortunately, these applications are well matched to the characteristics of modern VLSI technology. The data parallelism inherent in these applications allows a single instruction to control multiple arithmetic units and allows intermediate data to be localized to small clusters of units, significantly reducing communication demands. Even though global data is rarely reused, forwarding streams of data from one processing kernel to the next localizes data communication. Furthermore, the computation demands of these applications can be satisfied by keeping intermediate data close to the arithmetic units, rather than in memory, and by organizing multiple arithmetic units to take advantage of both data parallelism and instruction-level parallelism.

The Imagine architecture matches the demands of media applications to the capabilities of VLSI technology by supporting a stream programming model which exposes the desired locality and concurrency within the applications. Imagine is organized around a large (128KByte) stream register file (SRF). Memory load and store operations move entire streams of data between memory and the SRF. To the programmer, Imagine is a load/store architecture for streams; an application loads streams into the SRF, passes these streams through a number of computation kernels, and stores the results back to memory.

This chapter describes the basic elements of the Imagine architecture and the stream programming model. It provides the basis for the in-depth explanation of the bandwidth hierarchy and memory access scheduling presented in Chapters 5 and 6. Imagine will also act as the baseline architecture for the experimental evaluation of those features.

4.1.1 Stream Programming

Media processing applications are naturally expressed as a sequence of computation kernels that operate on long data streams. A kernel is a small program that is repeated for each successive stream element in its input streams to produce output streams that are fed to subsequent kernels. Each data stream is a variable length collection of records, where each record is a logical grouping of media data. For example, a record could represent a triangle vertex in a polygon rendering application. A data stream, then, could be a collection of hundreds of these vertices.

Records within a data stream are accessed sequentially and processed identically. This greatly simplifies the movement of data through a media processor by allowing the instruction overhead to be amortized over the length of these homogeneous data streams. For instance, a single memory stream transfer operation can collect hundreds of records from memory to form a homogeneous stream that is stored sequentially in a stream register file on the processor. This stream can then be transferred across the network, again with a single instruction, or processed by a computation kernel.

By organizing media processing applications in this stream model, the following characteristics that were enumerated in the previous chapter are exposed: data parallelism is abundant, very little data is reused, and many operations are required per memory reference. These properties can easily be exploited by a media processor designed to operate on data streams. The abstraction of a data stream maps naturally to the streaming data types found in media processing applications. The inputs to most media processing applications are already data streams and the expected outputs are data streams as well. Streams expose the fine-grained data parallelism inherent in media applications as well. Each record of a stream, whether it is 8 bits or 24 words, will be processed identically, so multiple records can be processed in parallel using the same instructions.

Kernels naturally expose the coarse-grained control parallelism in media processing applications, as they form a pipeline of tasks. Multiple kernels can therefore operate in parallel on different sections of the application's data. The first kernel in the pipeline would produce output streams that would then be passed to the next kernel. As

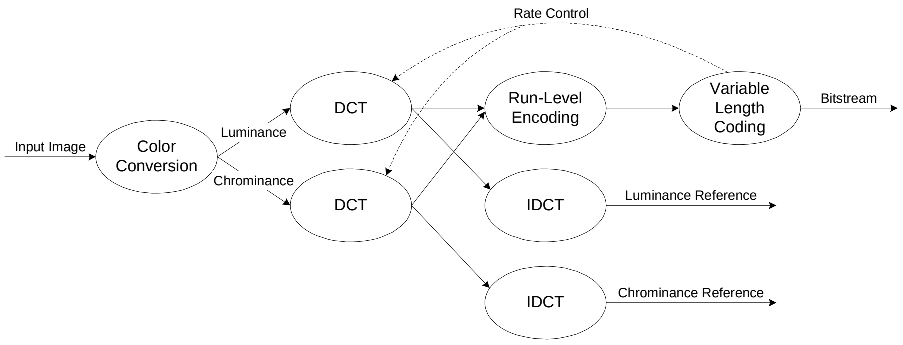


FIGURE 4.1 I-Frame Encoding from MPEG2 Encoder

the next kernel operates on those streams, the original kernel could operate on the next set of input data. Finally, the memory and network bandwidth demands of media processing applications can also be met using this stream model. Since all data is organized as streams, single memory or network transfer operations initiate long transfers with little control overhead that can be optimized for bandwidth.

4.1.2 MPEG2 I-frame Encoding

To illustrate the stream programming model, consider the stream program for the intracoded frame (I-frame) encoder of an MPEG2 encoder [Jac96]. An MPEG2 encoder takes a video sequence as input and compresses the sequence of images into a single bit stream consisting of three types of frames: intracoded (I), predicted (P), and bidirectional (B). I-frames are compressed using only information contained in the current frame. P-frames are compressed using information from the current frame and the previous I- or P-frame, allowing small amounts of motion to be encoded more compactly. Finally, B-frames are compressed using information from the current frame, the previous I- or P-frame, and the next I- or P-frame. B-frames achieve the highest compression ratio and decrease noise by using two images as references.

Figure 4.1 shows the kernel operations and stream dataflow required to encode an I-frame.[1] In the figure, the ovals represent computation kernels, the solid lines represent data streams, and the dashed lines represent scalar feedback. The first kernel of the application, color conversion, takes the original image in RGB format as input and

1. Note that the I-frame encoder is used for illustrative purposes, since it is simpler. The MPEG application described and analyzed earlier encodes both I- and P-frames.

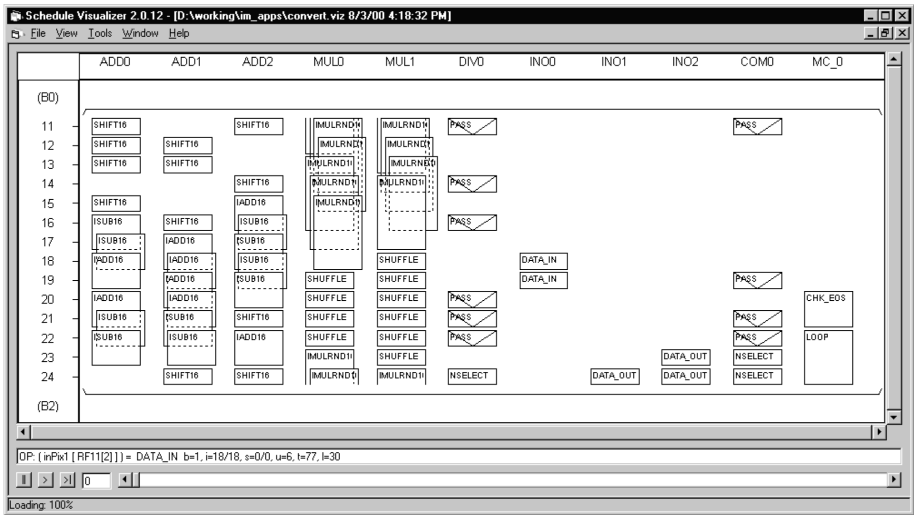
```
kernel convert(istream<RGB>  RGB_pixels,
               ostream<Y>    Y_pixels,
               ostream<CbCr> CbCr_pixels)
{
  RGB  inPix;
  Y    outY;
  CbCr outCbCr;

  loop_stream(RGB_pixels) {
    // Input next pixel
    RGB_pixels  >> inPix;

    // Convert RGB to YCbCr
    outY       =  0.257*inPix.r + 0.504*inPix.g + 0.098*inPix.b + 16;
    outCbCr.cb = -0.148*inPix.r - 0.291*inPix.g + 0.439*inPix.b + 128;
    outCbCr.cr =  0.439*inPix.r - 0.368*inPix.g - 0.071*inPix.b + 128;

    // Output resulting pixel
    Y_pixels    << outY;
    CbCr_pixels << outCbCr;
  }
}
```

FIGURE 4.2 Color Conversion Kernel

generates two output streams: the luminance (*Y*) and the chrominance (*CbCr*) of the image.

Each kernel depicted in Figure 4.1 is written in the KernelC language, described in Section 4.3.2. Figure 4.2 shows a simplified version of the color conversion kernel (*convert*).[2] This kernel takes one input stream and produces two output streams, corresponding directly to the streams indicated by the arrows in Figure 4.1. Figure 4.3 graphically depicts the VLIW code for the color conversion kernel compiled by the kernel compiler, *iscd* [MDR+00]. For clarity, the preprocessing before the loop and the postprocessing after the loop are not shown in either figure. In Figure 4.3, the columns are the functional units in the arithmetic clusters, the rows are instructions, and each box indicates an operation. The box indicating each operation spans the appropriate number of instructions given that particular operation's latency. Since the units are pipelined, operations can overlap. Using a three-stage software pipeline and processing two pixels per loop iteration per cluster, this particular unoptimized kernel achieves 74% utilization of the adders and multipliers. KernelC eliminates the need to program in assembly, as this visualization of the compiled code can be used to drive source code optimizations to improve kernel performance.

2. The actual color conversion kernel also filters and subsamples the chrominance values.



```
OP: ( inPix1 [ RF11[2] ] ) = DATA_IN  b=1, i=18/18, s=0/0, u=6, t=77, l=30
```

FIGURE 4.3 Schedule Visualization of the Color Conversion Kernel Inner Loop

As shown in Figure 4.1, after the image is separated into a luminance and chrominance stream, each 8x8 pixel block is transformed by a two-dimensional discrete cosine transform (DCT) to convert the image to the frequency domain. The DCT kernel also quantizes its results to compress the blocks in the frequency domain. The resulting values are run-level encoded to eliminate the numerous zero entries in the quantized data. Finally, the results are Huffman encoded by a variable length coding (VLC) kernel to produce the final MPEG2 bit stream. In order to regulate the bit stream rate, the VLC kernel provides scaling feedback to the DCT kernels to raise or lower the bit rate, as necessary, with a corresponding increase or decrease in image quality. An overview of the run-level encoding and Huffman encoding functions can be found in [Jac96] and [Sed90].

In order to encode future frames relative to this one, the transformed and quantized values are decoded by performing the inverse quantization and inverse DCT to create a reference frame of luminance and chrominance values. This reference frame is identical to the frame that the decoder will generate, preventing quantization differences from propagating from frame to frame.

The MPEG application is written in the StreamC language, described in Section 4.3.1. Figure 4.4 shows the StreamC for the MPEG2 I-frame encoder. The StreamC executes on the host processor which would also be running a *stream scheduler.* This

```
for (row=0; row<NROWS; row++) {
  // update quantization factor for rate control
  quantizerScale = newQuantizerScale;

  // setup streams for this row
  ...

  // Perform I-Frame encoding
  convert(InputRow, &YRow, &CbCrRow);
  dct(YRow, dctIconstants, quantizerScale, &DCTYRow);
  dct(CbCrRow, dctIconstants, quantizerScale, &DCTCbCrRow);
  rle(DCTYRow, DCTCbCrRow, rleConstants, &RunLevelsRow);
  vlc(RunLevelsRow, &bitStream, &newQuantizerScale);

  // Store generated bit stream
  ...

  // Generate reference image for subsequent P or B frames
  idct(DCTYRow, idctIconstants, quantizerScale, &RefYRow);
  idct(DCTCbCrRow, idctIconstants, quantizerScale, &RefCbCrRow);

  // Store reference rows
  ...
}
```

FIGURE 4.4 I-frame Encoding StreamC

stream scheduler is a run-time memory manager that manages data streams on the stream processor. The kernel calls shown in the figure will actually call StreamC library functions that will initiate kernel operations on Imagine. For example, the call to the *convert* kernel with three stream arguments will issue a kernel instruction to Imagine which includes the address of the color conversion kernel in the microcontroller's microcode store and descriptors for the input stream, *InputRow,* and two output streams, *YRow* and *CbCrRow.*

In addition, the stream scheduler will transfer the dependencies among stream operations to Imagine, so that the on-chip stream controller can issue the kernel instruction without further intervention from the host processor when all dependent stream operations have completed. For example, the stream scheduler will inform the stream controller that an input row of pixels must be loaded before the *convert* kernel can commence. If the color conversion kernel has not yet been loaded into the microcode store, the stream scheduler would first load the kernel program from memory and transfer it into the microcode store. Similarly, if the stream descriptors for the data streams are not already located on the chip, the stream scheduler would first update those descriptors before initiating the kernel. The rest of the I-frame encoder's StreamC after *convert* would be similarly converted into Imagine stream operations.

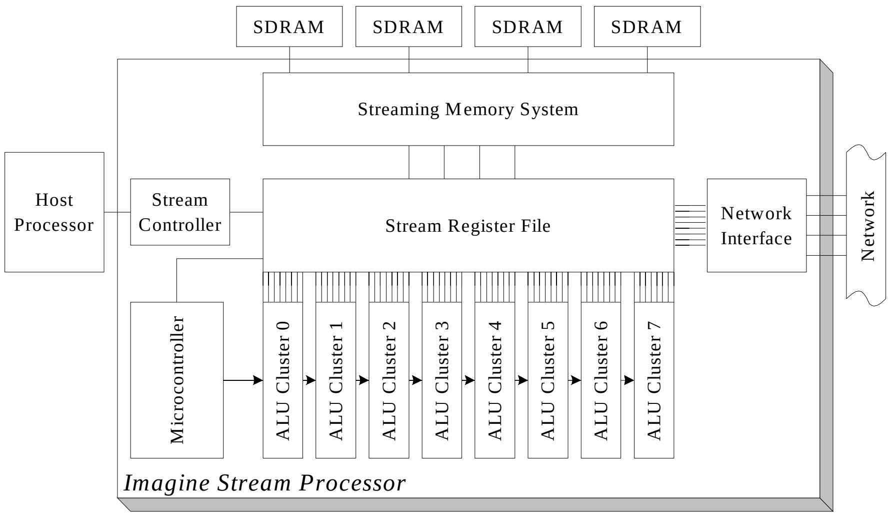


FIGURE 4.5 Imagine Block Diagram

This I-frame encoder illustrates the three key properties of media processing applications mentioned earlier: very little data is reused, data parallelism is abundant, and many operations are required per memory reference. First, data is rarely reused in this application; for example, once the image is color converted, the original input image is never referred to again. Second, most computations within the encoder are data parallel. If there were enough hardware, all of the blocks in the image could be color converted in parallel, since there is no dependence from one block to another. Finally, assuming that only the original image, the final coded bit stream, and the reference images are actually stored in memory, this I-frame encoder would perform 49.5 arithmetic operations per memory reference.

4.2 Architecture

The stream architecture of the Imagine media processor effectively exploits the desirable application characteristics exposed by the stream model. Imagine meets the computation and bandwidth demands of media applications by directly processing the naturally occurring data streams within these applications. Figure 4.5 shows a block diagram of Imagine's microarchitecture. Imagine is designed to be a coprocessor that operates on multimedia data streams. The stream register file (SRF) effectively isolates the arithmetic units from the memory system, making Imagine a load/store architecture for streams. All stream operations transfer data streams to or from the SRF. For instance, the network interface transfers streams directly out of and into the SRF, isolating network transfers from memory accesses and computation. This simplifies the design of the processor and allows the clients (the arithmetic clusters, the memory system, the network interface, etc.) to tolerate the latency of other stream clients. In essence, the SRF enables the streaming data types inherent in media processing applications to be routed efficiently throughout the processor.

As shown in Figure 4.5, eight arithmetic clusters, a microcontroller, a streaming memory system, a network interface, and a stream controller are connected to the SRF. The arithmetic clusters consume data streams from the SRF, process them, and return their output data streams to the SRF. The microcontroller is connected to the SRF to allow compiled kernel programs to be loaded from the SRF into the microcontroller's microcode store. The streaming memory system transfers streams between the SRF and the off-chip DRAM. The network interface transfers streams between the SRF and other processors or devices connected to the network. The stream controller allows the host processor to transfer data and microcode programs into or out of the SRF, although large data transfers would be made through the Imagine network.

4.2.1 Stream Register File (SRF)

The SRF is a 128KB (1Mbit) memory optimized for stream transfers. The SRF can hold multiple variable-length data streams, subject only to capacity constraints. A data stream is an arbitrary length sequence of records, such as a row of pixels from an image or a set of vertices from a polygonal graphics model. Streams are referenced using stream descriptors which are held in a 64-entry stream descriptor register file (SDRF). Each stream descriptor register (SDR) includes a base address in the SRF and a stream length. The memory array of the SRF has 1024 rows of 1024 bits with a single port.

The stream register file supports numerous clients, including the arithmetic clusters and memory system, through a single physical port. However, each has its own dedicated logical port(s) into the SRF. To support this abstraction, all clients are connected to the SRF by *stream buffers*. Each stream buffer is composed of two 1024-bit half-buffers. A half-buffer of a single stream buffer may be filled or drained each cycle through the 1024-bit port into the SRF memory array. If a stream buffer that is supporting a client reading from the SRF has an empty half-buffer, then that stream buffer requests access to the memory array. Similarly, if a stream buffer that is supporting a client writing to the SRF has a full half-buffer, then it also requests access to

Client	**Streams**	**Type**	**Words per Cycle**
Network	8	read/write	2W per stream
Stream Controller	1	read/write	1W per stream
Microcontroller	1	read only	1W per stream
Memory Index	2	read only	1W per stream
Memory Data	2	read/write	1W per stream
Arithmetic Clusters	8	read/write	8W per stream

TABLE 4.1 SRF Streams

the memory array. When granted access to the memory array, a stream buffer transfers the next consecutive 1024 bits in its associated data stream. This enables each client connected to the SRF to act as if it has its own dedicated stream port into the SRF. Since the clients access fewer than 1024 bits in the SRF at a time, the stream buffers are able to distribute the bandwidth of the single port across the multiple stream clients dynamically. However, the clients may only access stream data sequentially. This is matched to the stream programming model, as stream applications do not reference arbitrary data. Rather, they only access data streams sequentially.

There are 22 stream buffers connected to the SRF memory array. These 22 stream buffers are associated with the clients listed in Table 4.1. Eight streams are associated with the network interface, so there can be eight active network stream transfers at any given time. The network interface transfers two 32-bit words per cycle to/from one of these stream buffers in order to match the available network bandwidth. The stream controller has access to a dedicated stream buffer to allow the host processor to transfer streams to/from Imagine. One stream buffer is connected to the microcode store to allow kernel microcode to be loaded into the microcontroller. Two sets of stream buffers connect to the memory system. Each set consists of a data stream and an index stream which provides indices for indirect (gather/scatter) transfers. Finally, eight stream buffers are allocated to the arithmetic clusters. This allows kernels executing on the arithmetic clusters to have up to eight total input/output streams. All eight streams are connected to all of the arithmetic clusters. Each of these streams can transfer eight 32-bit words per cycle (one word per cluster).

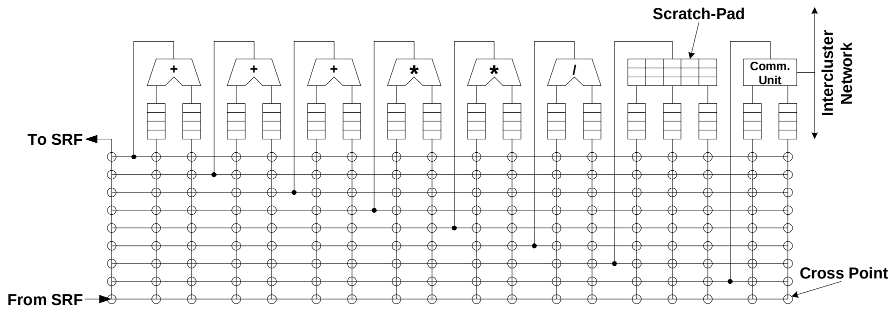


FIGURE 4.6 Arithmetic Cluster

4.2.2 Arithmetic Clusters

Imagine contains eight arithmetic clusters controlled by a single microcontroller in a SIMD fashion. The arithmetic clusters operate simultaneously on interleaved data records transferred from the SRF, allowing eight data records to be processed in parallel. As shown in Figure 4.6, each arithmetic cluster contains three adders, two multipliers, one divide/square root unit, one 256-entry scratch-pad memory, and one intercluster communication unit. These units are controlled by statically scheduled VLIW instructions issued by the microcontroller. Across the eight arithmetic clusters, Imagine's 48 arithmetic units achieve the high sustained computation rates required by media processing applications. The mix of arithmetic units in Imagine are matched to the demands of the targeted media applications, but the architectural concept is compatible with other mixes of arithmetic units within the clusters.

All of the arithmetic units support both 32-bit single precision floating point and 32-bit integer operations. In addition, the adders and multipliers also support 16-bit and 8-bit parallel-subword operations for a subset of the integer operations. These arithmetic operations are well-matched to the low-precision data types commonly found in media processing applications. These parallel-subword instructions, similar to the multimedia instructions commonly added to general-purpose processors [Lee96] [PW96] [TONH96], exploit the fine-grained data parallelism inherent in media applications. The adders, multipliers, scratch-pad, and communication unit are fully pipelined, allowing a new operation to issue every cycle. The divide/square root unit has two SRT cores, so no more than two divide or square root operations can be in flight at any given time.

Data cannot be transferred directly from memory to the arithmetic clusters. All data from memory must first be loaded into the SRF and then transferred to the arithmetic clusters. Sometimes, however, kernels executing in the arithmetic clusters benefit from being able to index into small arrays or lookup tables. The scratch-pad units in each cluster provide this functionality; they are accessed with a base address provided by the microcontroller added to an offset provided locally by each cluster. This enables each cluster to access different array elements within its scratch-pad.

Rather than the conventional centralized register file organization, the arithmetic units utilize a distributed register file (DRF) structure. As shown in Figure 4.6, each arithmetic unit has a small dedicated two-port register file connected to each of its inputs. A switch then connects the outputs of all of the arithmetic units to the 17 register files within the cluster. Effectively, the DRF organization eliminates the implicit switch that would be replicated within each register of a conventional register file. Only one explicit switch is needed in the DRF organization. The cross points in Figure 4.6 show where connections from the outputs of arithmetic units to register files can be made. On the Imagine chip there is enough area over the arithmetic cluster to allow a complete switch, so all of the cross points are connected. However, a sparse interconnect could be used and is fully supported by the compiler tools [MDR+00]. By making the communication among arithmetic units explicit through this switch, rather than implicit through a centralized register file, the area, delay, and power dissipation of the register file structure is drastically reduced with only a slight degradation in performance [MDR+00] [RDK+00b].

Media applications are not always perfectly data parallel, and in fact often require extensive data reorganization. To facilitate this reorganization, the communication unit, shown on the right of Figure 4.6, allows data to be transferred among arithmetic clusters through the intercluster network, whereas the intracluster switch of the DRF organization only allows data to be communicated among arithmetic units within a cluster. The communication unit within each cluster is able to drive one word per cycle onto its own dedicated bus. The microcontroller also has a dedicated bus onto which it can drive data, to allow immediates and other data to be broadcast to the clusters. Collectively, these buses form the intercluster network, shown in Figure 4.7. Each cluster's communication unit may read from any one of these nine buses each cycle. A permutation descriptor controls which bus will be read by the communication units in each cluster. This descriptor can either be provided globally by the microcontroller or generated locally within each cluster. The ability to communicate among the arithmetic clusters significantly improves performance of applications that are not perfectly data-parallel, since each cluster may only access one eighth of the memory in the SRF.

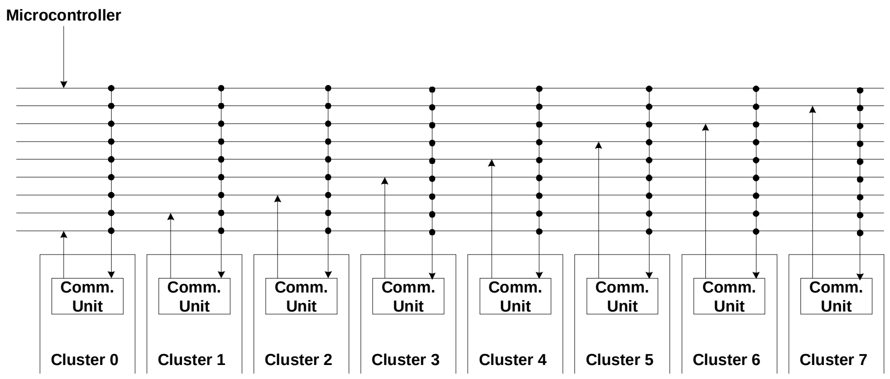


FIGURE 4.7 Intercluster Communication Network

4.2.3 Microcontroller

The microcontroller is a very long instruction word (VLIW) control engine that issues instructions to the arithmetic clusters. Each cycle, the microcontroller broadcasts a single 568-bit VLIW instruction to all eight clusters. Since instructions are statically scheduled by the compiler, the only dynamic behavior is that the microcontroller may be forced to stall the arithmetic clusters when a stream buffer is not ready for a cluster access. This occurs when the kernel attempts a stream input from an empty stream buffer or a stream output to a full stream buffer. In both of these cases, the microcontroller stalls the arithmetic clusters until that stream buffer receives access to the SRF memory array and is consequently made ready for cluster accesses.

Kernels are stored in a 2K instruction by 568 bit (1.1Mbit) microcode store located within the microcontroller. The microcontroller takes advantage of the high instruction reference locality in small loops found in media processing applications by storing kernels directly in this microcode store, rather than fetching them from memory or a cache. Consequently, kernel instructions are never fetched from memory during the course of kernel execution, so kernels never need to stall due to instruction fetch latencies. Frequently, the microcode store is large enough to hold all of the kernels for an application. In this case, kernels only need to be loaded once from memory for the entire execution of the application. However, if the kernels do not all fit, then they are loaded into the microcode store prior to their execution. In this case, the loading of one kernel can be overlapped with the execution of other kernels in the application.

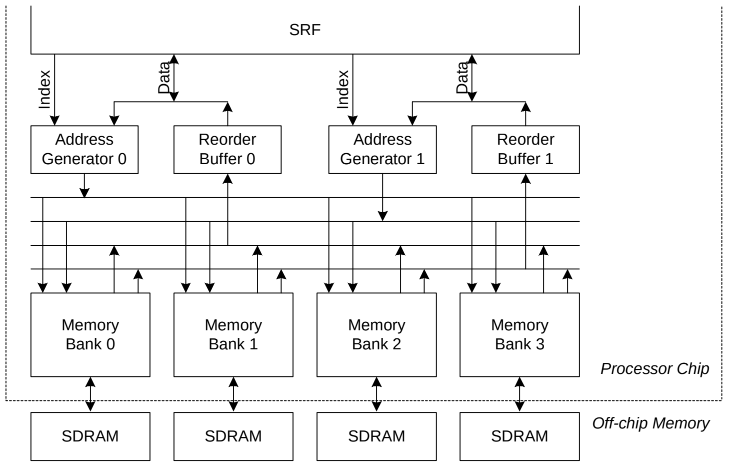


FIGURE 4.8 Streaming Memory System

4.2.4 Streaming Memory System

The memory system supports data stream transfers between the SRF and off-chip DRAM. This stream load/store architecture simplifies programming and minimizes stalling in the arithmetic clusters by isolating them from variable memory access latencies. Because the granularity of memory references is streams of data and not individual words, the memory system can be optimized for throughput, rather than reference latency, which will be explored further in Chapter 6. Furthermore, accesses to external DRAM can be reordered to maximize the sustained bandwidth of the off-chip memory. Since a consumer of a stream, such as a kernel executing in the arithmetic clusters, cannot commence until the entire stream is available in the SRF, the time it takes to load the entire stream is far more important than the latency of any particular reference. Moreover, these references can easily be overlapped with computation. The memory system can be loading streams for the next set of computations and storing streams for the previous set of computations while the current set of computations is occurring.

As shown in Figure 4.8, the Imagine streaming memory system consists of a pair of address generators, four interleaved memory banks, and a pair of reorder buffers that place stream data in the SRF in the correct order. All of these units are on the same chip as the Imagine processor core, except for the off-chip SDRAM memory.

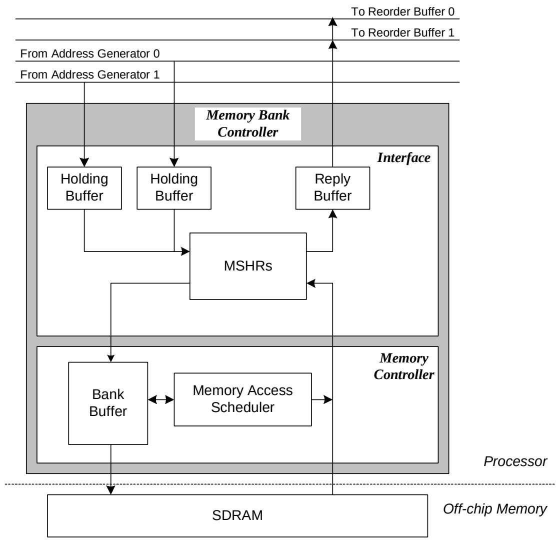


FIGURE 4.9 Memory Bank Architecture

The address generators may generate memory reference streams of any length, as long as the data fits in the SRF, using one of the following three addressing modes: constant stride, indirect, and bit-reversed. Elements within a record are stored contiguously in memory, so the addressing modes dictate how the starting memory address of each record in a stream is calculated, and the record length dictates how many consecutive words are accessed for each record. For constant stride references, the address generator takes a base, stride, and length, and computes successive record addresses by incrementing the base address by the stride. For indirect references, the address generator takes a base address and an index stream from the SRF and calculates record addresses by adding each index to the base address. Bit-reversed addressing is used for FFT memory references and is similar to constant stride addressing, except that bit-reversed addition (carries are propagated to the right instead of to the left) is used to calculate addresses [OSB99]. The addressing information needed by the address generators is stored in a memory address register file (MARF). A stream load/store operation, issued by the stream controller, indicates which memory address register (MAR) contains the addressing mode, base address, stream length, and other information needed for that transfer.

Figure 4.9 shows the architecture of the memory banks within the streaming memory system. References arriving at the memory banks from the address generators are stored in a small holding buffer until they can be processed. Despite the fact that there is no cache, a set of registers similar in function to the *miss status holding registers* (MSHRs) of a non-blocking cache [Kro81] exist to keep track of in-flight references and to do read and write coalescing. When a reference arrives for a location that is already the target of another in-flight reference, the MSHR entry for that reference is updated to reflect that this reference will be satisfied by the same DRAM access. When a reference to a location that is not already the target of another in-flight reference arrives, a new MSHR is allocated and the reference is sent to the *bank buffer*. The memory controller schedules DRAM accesses to satisfy the pending references in the bank buffer and returns completed accesses to the MSHRs. The MSHRs send completed loads to a reply buffer which holds them until they can be sent back to the reorder buffers. As the name implies, the reorder buffers receive out of order references and transfer the data to the SRF in order.

In this streaming memory system, memory consistency is maintained in the following two ways: conflicting memory stream references are issued in dependency order, and the MSHRs ensure that references to the same address complete in the order that they arrive. This means that a stream load that follows a stream store to overlapping memory locations may be issued as soon as the address generators have sent all of the store's references to the memory banks.

4.2.5 Network Interface

The network interface connects the SRF of the Imagine stream processor to an external network, enabling source routed data stream transfers through that network. Source routing information is held in a network route register file (NRRF) within the network interface. Each network route register (NRR) can be written with a source route that can then be used to control the path taken by network stream transfers. The network interface transfers entire data streams from one processor to another through the Imagine network. This drastically increases the ratio of data to control that must go through the network, as well as eliminating instructions to send and receive each data element within the stream.

The Imagine processor has four bidirectional network channels that can be interconnected in an arbitrary topology. Each channel is 16-bits wide in each direction and uses differential signaling, requiring a total of 256 data pins in the network interface. Not only can other Imagine processors be connected to the external network, but I/O devices can also be connected to the network, with some additional logic, to send and receive streams using the Imagine network protocol. Since the channels may be interconnected in an arbitrary topology, multiprocessor solutions can be constructed that provide network bandwidth tailored to the communication demands of the targeted applications. For instance, all four channels of two Imagine processors could be connected to each other to provide four times the network bandwidth between those two processors. This flexible topology combined with the amortization of instructions and control over entire data streams enable Imagine to deliver the high network bandwidth demanded by media processing applications.

4.2.6 Stream Controller

The stream controller receives stream instructions from the host processor and issues them to the units of Imagine, described in Sections 4.2.1-4.2.5, subject to dependency constraints. The stream controller is similar in function to the issue unit in a superscalar processor; it issues stream instructions, possibly out of order, to available stream units without violating the dependencies among stream instructions. The stream controller accepts instructions from the host processor, stores them in a pending instruction queue, and determines which instructions are available for issue. Status signals from the stream units are used to determine when operations complete, allowing the stream controller to know when an instruction's dependencies have been satisfied. The stream controller simply selects an instruction that is ready to issue and issues it to the appropriate stream unit. The stream controller also includes a general-purpose register file (SCTRF) that can be accessed by stream register read, write, and move instructions, which will be described in Section 4.3.1. These registers are used to transfer data between the stream controller and other modules, such as the host processor or control registers within the stream units.

4.2.7 Host Processor

An external host processor executes all scalar code and transfers stream instructions to Imagine, which acts as a coprocessor. A stream processor could include a serial host processor on the same chip. This would allow the two processors to be tightly integrated so that the overhead of issuing instructions to the stream processor would be negligible. However, since Imagine is a proof-of-concept research project, an off-the-shelf host processor is used to control Imagine. The host processor is able to execute small serial sections of code that exist in any real-world application. Imagine is tailored to take advantage of the characteristics of media processing applications, as described in Chapter 3. Code that is not actually operating on media data, but is rather orchestrating the control flow of the overall application, is much better suited to a conventional serial processor than a SIMD stream processor like Imagine. Therefore, each processor is able to execute the code for which it was designed — the host processor executes small sections of control intensive serial code and Imagine executes large data parallel stream programs.

4.3 Programming Model

The stream programming model exposes the following three key properties of media processing applications: very little data is reused, data parallelism is abundant, and many arithmetic operations are required per memory reference. In media processing applications expressed in this form, most data is passed directly from kernel to kernel through data streams. Kernels are usually perfectly data-parallel, or nearly so, allowing multiple stream elements to be processed simultaneously. Finally, data streams can be recirculated from one kernel to the next through the SRF, without requiring data to be returned to memory for intermediate storage.

Stream applications are programmed at two levels: stream and kernel. As described in Section 4.2.7, a stream processor acts as a coprocessor to a sequential host processor. Stream-level programs execute on this host processor and orchestrate the flow of data streams through the stream processor. These programs are written in StreamC, a derivative of C++ that includes library functions that issue stream instructions to the stream processor. Kernel-level programs operate on these data streams and execute on the microcontroller and arithmetic clusters of the stream processor. Kernels are written in KernelC, a subset of the C language, and may access local variables, read the head elements of input streams, and write the tail elements of output streams. Kernels may not, however, make arbitrary memory references. Kernels loop over their input streams, perform computations on each stream record element in turn, and produce successive data elements of their output streams.

4.3.1 Stream-level Programming

StreamC programs make calls to library functions in order to manipulate streams on the stream processor. These library functions can send operations to the stream processor from the stream instruction set, which includes the eleven basic instructions in Table 4.2. All of the instructions, except READ, are asynchronous, in that the host processor may issue them to the stream processor and then continue execution of the stream program. The host processor must wait for the return value of the READ instruction. The stream instructions can be grouped into the following four categories: control word movement, synchronization, stream movement, and computation.

Instruction	Description
MOVE *dst_reg src_reg*	Move the contents of *src_reg* to *dst_reg*
WRITE_IMM *reg data*	Write *data* into *reg*
READ *reg*	Return the contents of *reg* to the host processor
barrier	Block all subsequent instructions from issuing until all previous instructions have issued
synch	Synchronize microcontroller with stream controller
LOAD *mar dat_sdr idx_sdr*	Load a stream from memory described by *mar* into a stream in the SRF described by *dat_sdr* — *idx_sdr* describes an index stream in the SRF for indexed loads
STORE *mar dat_sdr idx_sdr*	Store a stream from the SRF described by *dat_sdr* to memory described by *mar* — *idx_sdr* describes an index stream in the SRF for indexed store
SEND *nrr tag sdr*	Send a stream from the SRF described by *sdr* to the network using the route held in *nrr* and identified by *tag*
RECEIVE *tag sdr*	Receive a stream from the network identified by *tag* into a stream in the SRF described by *sdr*
LOAD_UCODE *mpc sdr*	Load microcode program from a stream in the SRF described by *sdr* to location *mpc* in the microcode store
CLUSTOP *mpc sdr0..sdr7*	Initiate a kernel on the arithmetic clusters starting at location *mpc* in the microcode store — *sdr0-sdr7* describe the input/output data streams in the SRF

TABLE 4.2 Imagine Stream Instruction Set Architecture

Control Word Movement. The MOVE, WRITE_IMM, and READ instructions allow the host processor to transfer control information to/from the non-cluster registers on the stream processor. The MOVE instruction allows data values to be moved from one register to another. This is useful if one stream instruction generates a value that will be used by another stream instruction. The WRITE_IMM instruction writes a value to a register in the stream processor. The READ instruction similarly reads a value from a register in the stream processor. The target of these three instructions are usually the microcontroller register file (UCRF), the stream descriptor register file (SDRF), the memory address register file (MARF), the network route register file (NRRF), and the stream controller register file (SCTRF), described in Section 4.2.

Synchronization. The BARRIER and SYNCH instructions enable the stream controller, host processor, and microcontroller to synchronize with each other. The host processor sends a BARRIER instruction to indicate that all instructions that were sent prior to the barrier instruction should issue before any instruction that was sent after the barrier instruction is issued. The SYNCH instruction synchronizes the stream controller with the microcontroller, leaving other stream transfers unaffected. When a kernel issues a synchronization instruction, the microcontroller stalls and the stream controller is notified that the microcontroller is waiting for a SYNCH instruction to resume execution. While the microcontroller is stalled, the stream controller can perform any number of stream operations, such as transferring information to or from the microcontroller through the UCRF. When the stream controller is done issuing the required stream operations, it issues the SYNCH instruction, which frees the microcontroller to continue execution.

Stream Movement. The LOAD, STORE, SEND, and RECEIVE instructions transfer data streams throughout the system. LOAD and STORE instructions move data streams between the SRF and memory. These instructions take a stream descriptor (*dat_sdr*) which specifies a starting location and length of a stream in the SRF, and an address descriptor (*mar*) that provides the base address in memory, addressing mode (constant stride, indexed, or bit-reversed), and the record size of data elements in the stream. Indexed memory transfers take an additional stream descriptor (*idx_sdr*) which specifies the start and length of the index stream in the SRF. SEND and RECEIVE instructions allow streams to be transferred from the SRF of one Imagine processor to the SRF of another Imagine processor through the network. SEND and RECEIVE instructions can also be used for I/O transfers through the network.

Computation. Finally, the CLUSTOP instruction initiates a kernel operation on a set of data streams. This instruction specifies a kernel to execute using an address into the on-chip microcode storage (*mpc*) and stream descriptors for up to eight input and/or output streams in the SRF (*sdr0-sdr7*). Kernels are loaded from the SRF (they are first loaded from memory into the SRF using a normal stream LOAD instruction) to the microcode store by a LOAD_UCODE instruction. If the microcode storage is large enough to hold all of the kernels needed by an application (as it frequently is), then each kernel need only be loaded into the microcode store once and may then be reused by subsequent CLUSTOP instructions.

4.3.2 Kernel-level Programming

KernelC programs are written in a subset of the C language. All C mathematical expressions are supported, but pointers are not allowed and the control flow constructs are restricted. The C++ stream operators, >> and <<, are used to read records into the clusters from input streams and write records to output streams in the SRF. Kernels are optimized for stream computation, so they may not use pointers to make arbitrary memory references. To allow limited memory addressing, either an index stream can be generated that can be used to perform an indirect stream load or store, or small arrays may be stored within the clusters' scratch-pads. All instructions in a KernelC program are executed simultaneously on all eight clusters, thereby processing eight stream elements identically in parallel. On such a SIMD machine, data dependent control flow constructs, such as if-then-else, are not allowed. The only form of control flow allowed in a KernelC program is a loop which can be terminated by a variety of conditions, such as a counter in the microcontroller or a combination of cluster condition codes.

A KernelC program must be written with the knowledge that it will execute simultaneously on all eight clusters, since the Imagine compiler tools do not currently perform automatic SIMD parallelization. Because this parallelization is not automatic, the programmer must orchestrate the movement of data between the arithmetic clusters. A typical kernel will be written to first perform some preprocessing, then loop over its input stream(s), and finally perform some postprocessing. The preprocessing stage typically consists of setting up constants and reading in coefficients from SRF streams. The main loop will then process eight elements of the kernel's input stream(s) per iteration. This loop can be unrolled and/or software pipelined to improve performance. If the kernel operation is completely data-parallel, the loop simply reads its inputs, performs some processing, and writes its outputs. However, many media processing kernels involve some interaction among stream elements. Consequently, the main loop may require communication among the clusters and/or storage of previous stream elements. A set of *communication* instructions exist in KernelC to transfer data among the clusters. These instructions compile to operations that are executed by the communication units in the clusters. Finally, the post processing stage performs any end-case cleanup that is necessary for the kernel. Kernels, however, do not need to follow this structure since KernelC is flexible enough to allow multiple loops or nested loops, as the algorithm requires.

Despite the abundant data parallelism available in media processing applications, data dependent conditional control flow is sometimes necessary. Since the clusters operate in a SIMD fashion, such data dependent conditional execution can be difficult. Imagine provides the following three mechanisms that convert data dependent control flow

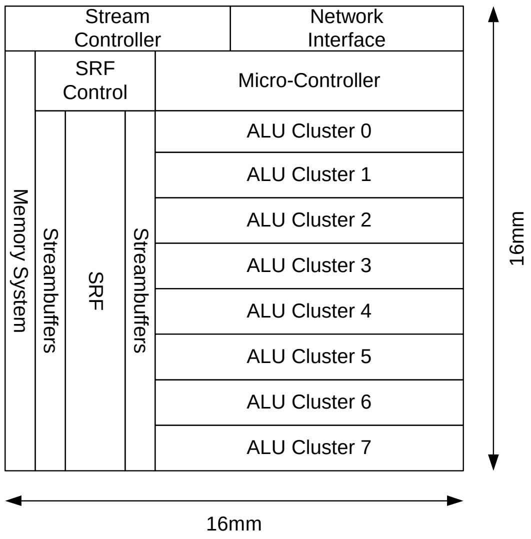


FIGURE 4.10 Imagine Floorplan

into conditional data movement: the select operation, scratch-pad access, and conditional streams. First, the select operation is simply the C “?:” operator implemented as an arithmetic primitive in hardware. The select operation allows data to be accessed based on data-dependent conditions within each cluster. Second, each cluster can access a different location within its scratch-pad by using a locally computed offset. The scratch-pad therefore allows different clusters to operate on different array elements stored within the scratch-pads. Finally, conditional streams allow each cluster to conditionally transfer data to/from the SRF, independent of the other clusters, based on condition codes computed within each cluster [KDR+00].

4.4 Implementation

Imagine is designed to operate at 500MHz in a Texas Instruments 0.15µm CMOS process. As shown in the floorplan in Figure 4.10, the chip will measure slightly less than 16x16mm, including pads. Imagine requires approximately 456 signal pins and 21 million transistors. The SRF and microcode store are approximately six million transistors each, the arithmetic clusters are roughly 750 thousand transistors each, and the remaining components are about three million transistors. To facilitate the implementation of a research prototype, not all of the chip need run at 500MHz. In fact, many units are designed such that they can run slower with negligible performance degradation.

At the base of the bandwidth hierarchy, the memory system can supply 2GB/s of peak data bandwidth. The four memory banks are each attached to their own external 32-bit wide SDRAMs. The memory controllers and SDRAMs operate at 125MHz, yielding a peak memory bandwidth of 500MB/s per bank. Data can also be fed into the SRF from the network. The network interface has a peak bandwidth of 8GB/s. The four 16-bit network channels operate at 500MHz, yielding a peak bandwidth of 1GB/s per channel in each direction.

In the center of the bandwidth hierarchy, the 128KB SRF memory array is designed to operate at 250MHz, yielding a peak bandwidth of 32GB/s. The 22 stream buffers, however, operate at 500MHz, yielding instantaneous peak bandwidth of 172GB/s. The stream buffers therefore scale the SRF bandwidth to enable short bursts of higher bandwidth than the 32GB/s memory array can provide. In practice, the clients rarely need an average bandwidth of more than the 32GB/s supplied by the memory array.

Finally, the arithmetic clusters contain 48 total arithmetic units (6 per cluster) and 136 local register files (17 per cluster) that yield a peak computation rate of 20GFLOPS and a peak data bandwidth of 544GB/s. The clusters’ peak computation rate is 20GOPS for both 32-bit integer or floating-point operations. For 16-bit and 8-bit parallel-subword operations, the peak performance increases to 40 and 80GOPS, respectively.

The three-tiered bandwidth hierarchy of Imagine therefore provides DRAM bandwidth, global register bandwidth, and local register bandwidth with a ratio of 1:16:272. That is, for each word accessed in memory, 16 words may be accessed in the SRF, and 272 words may be accessed in the clusters' local register files. Viewed differently, Imagine can perform forty 32-bit arithmetic operations per 4-byte word accessed in memory, or 2.5 operations per word accessed in the SRF. This is well-matched to the compute intensive nature of media processing applications. For comparison, conventional architectures can typically perform about five operations per word accessed in memory, and require three global register accesses per operation [HL99] [Kes99].

4.5 Scalability and Extensibility

Media processing applications currently demand very high computation rates, and as media processing algorithms and applications become more realistic and sophisticated those rates will continue to increase. Therefore, any useful media processor must be able to scale beyond its current capabilities. The modularity of the Imagine architecture easily lends itself to scaling and extension. Internally, the Imagine processor can easily be scaled or extended, as the SRF isolates the various stream units from each other. The SRF also allows new stream clients to be added to the processor without disrupting the rest of the architecture. The network interface allows multiple Imagine processors to be interconnected without disrupting the programming model. Applications that run on a single Imagine processor can be ported to a multiprocessor configuration by inserting network send and receive instructions. These instructions transfer streams from one processor to the next in the middle of the kernel processing pipeline.

4.5.1 Scalability

The number of arithmetic clusters can be increased or reduced in order to provide higher or lower stream throughput with a corresponding change in cost. The width of the SRF and the peak memory bandwidth also needs to be scaled equivalently to maintain the correct bandwidth ratios to keep the arithmetic units' utilization high. While increasing the number of arithmetic clusters does not incur any major costs inside each cluster, some global communication may become prohibitive. For instance, the microcode instructions must be broadcast to all of the clusters. Instructions could be buffered in clusters closer to the microcontroller so that they are delayed long enough for the distant clusters to receive them. However, this would increase the number of required branch delay slots. The microcontroller could be placed in the center of the cluster array to allow a trivial doubling of the number of clusters, but this introduces its own set of problems since it is more difficult for the microcontroller to communicate and synchronize with the stream controller. The intercluster communication network also has long global wires that would be affected by scaling. The intercluster switch could be pipelined to continue to allow generalized intercluster communication with increased latency and significant buffering.

A better way to scale the Imagine architecture is to place multiple Imagine cores on a single chip. Figure 4.11 shows four 4-cluster Imagines on a single chip. The SRFs are connected via an on-chip stream network that interfaces to an off-chip network. Applications could also easily take advantage of such a structure, since the stream programming model lends itself to multiprocessor configurations. Kernels could be

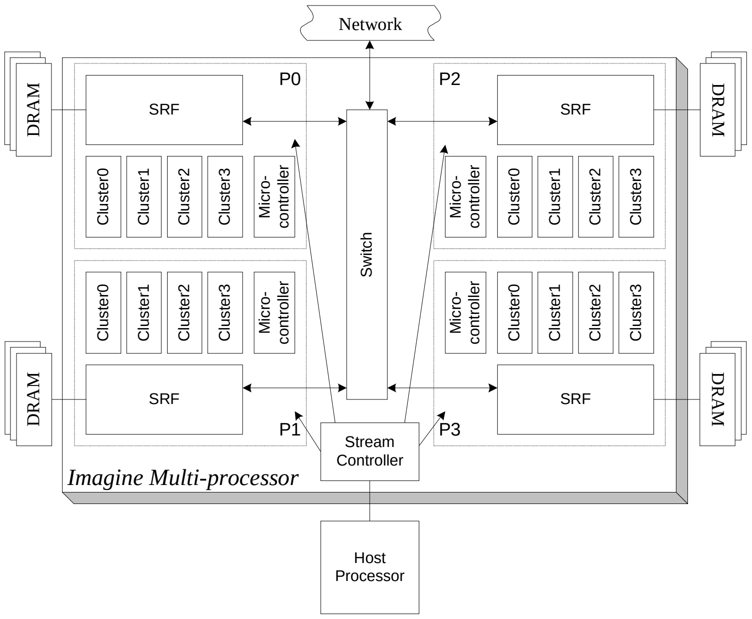


FIGURE 4.11 Multi-Imagine Processor

split across multiple stream cores in many different ways. For instance, kernels could be pipelined such that one core could perform some amount of kernel processing on batches of data and then forward the resulting streams to the next core while processing the next batch of data. Similarly, multiple cores could run the same kernels on different subsets of the application data.

4.5.2 Extensibility

The modularity of the architecture also easily lends itself to extension. The stream register file effectively isolates the stream units from each other. This allows additional stream clients to be added to the architecture by adding the appropriate number of stream buffers to the SRF and adding the appropriate stream instruction(s) to pass streams through the new unit. For example, a variable-length bit-coding unit could be added to simplify Huffman encoding/decoding for MPEG processing. If given some programmability, then such a unit could also perform any number of serial bit operations, such as 5-6-5 RGB pixel conversion to 8-8-8 RGB data. Another example stream unit could be a texture caching unit that takes a stream of texture addresses as input and returns both a stream of texels that were found in the cache and a stream of addresses for texels which must be acquired from memory. Any number of such streaming units could be added to the basic core of the processor, only requiring the appropriate scaling of SRF bandwidth and capacity. While each specialized streaming unit is likely to have a limited range of applications for which it is useful, processors could be designed such that the set of stream units they contain are tailored to the targeted application domain.

The arithmetic clusters themselves can also be easily modified. The mix and composition of arithmetic units could easily be changed or extended, and units that support special DSP or media processing operations could easily be added.

The stream register file at the center of the bandwidth hierarchy of the Imagine architecture allows the processor to be modularly scaled or extended. The SRF provides a well defined interface to arbitrary stream processing units which also fit into the stream programming model. To scale or extend the architecture, the bandwidth at the different levels of the hierarchy simply needs to be adjusted to ensure that all of the computation resources can be effectively utilized.

CHAPTER 5

Data Bandwidth Hierarchy

Currently, real-time media processing applications demand 10-100 GOPS of sustained computation. As these applications become more compelling and realistic, their computation demands will continue to grow. To meet the current demands, a 500MHz processor would have to support 20-200 arithmetic units at nearly 100% utilization. In modern VLSI, this amount of computation is cheap. Hundreds of arithmetic units can easily fit on a modestly sized 1cm^2 die — a 32-bit floating point multiplier is only 0.25mm^2 in a 0.18μm CMOS process. Communication is the expensive resource. Technology constraints prohibit conventional microarchitectural organizations from supporting enough arithmetic units to meet the computational requirements of media processing applications. However, a microarchitecture that incorporates a data bandwidth hierarchy can efficiently scale to exploit the abundant data parallelism inherent in these applications. This chapter will discuss the communication bottlenecks of modern microprocessors, the organization of a data bandwidth hierarchy, and the impact of such a storage structure on media processing performance.

5.1 Overview

Media processing applications demand computation rates of 10-100 GOPS. A memory system, such as that of the Imagine media processor, which provides a peak bandwidth of 2GB/s could not effectively support enough arithmetic units to achieve the

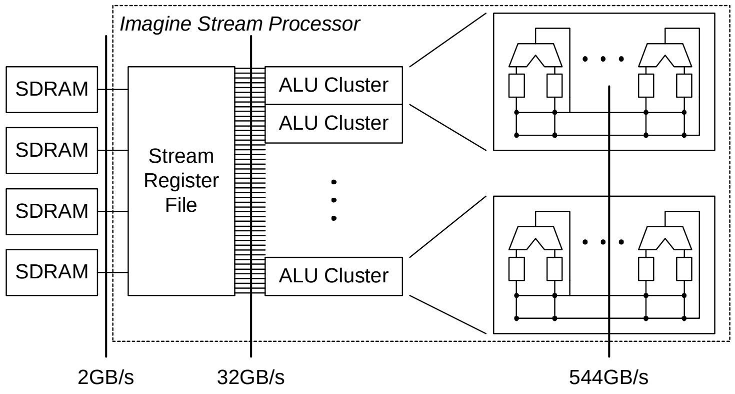


FIGURE 5.1 Imagine's Bandwidth Hierarchy

required computation rates. Traditionally, programmable microprocessors have bridged the bandwidth gap between DRAM and the arithmetic units by using a cache hierarchy and a centralized register file to supply the arithmetic units with data. Unfortunately, technology constraints and media application behavior prohibit these structures from effectively scaling to support the tens to hundreds of arithmetic units required for media processing.

A data bandwidth hierarchy, however, can bridge the bandwidth gap between DRAM and the arithmetic units by scaling the provided bandwidth across the levels of the storage hierarchy. Figure 5.1 shows the bandwidth hierarchy of the Imagine stream processor. The external DRAM provides a peak bandwidth of 2GB/s. The next level of the storage hierarchy, the stream register file (SRF), provides a peak bandwidth of 32GB/s, which is 16 times higher than the memory bandwidth. Finally, the distributed register file structure within the arithmetic clusters provides a peak bandwidth of 544GB/s, which is 17 times higher than the SRF bandwidth. The external DRAM, SRF, and local register files form a three-tiered bandwidth hierarchy in which the bandwidth is scaled by a ratio of 1:16:272 across the levels.

A bandwidth hierarchy allows media processing applications to use data bandwidth efficiently. Temporary data is stored locally in the arithmetic clusters where it may be accessed frequently and quickly. Intermediate streams are stored on-chip in the SRF where they can be recirculated between kernels without requiring costly memory references. Finally, input, output, and other global data is stored in external DRAM, since it is referenced infrequently and requires large amounts of storage space.

However, to take advantage of an explicitly managed storage hierarchy, such as the bandwidth hierarchy shown in Figure 5.1, a programming model that exposes the locality and concurrency of applications is required. The traditional sequential programming model hides parallelism and locality, making it difficult to extract parallel computations and manage data movement. The stream programming model, however, enables parallelism to be readily exploited and makes most data movement explicit. This enables a media processor that incorporates a bandwidth hierarchy to store data and perform computations more efficiently.

A cache hierarchy dynamically manages data movement and storage. However, media applications have regular, predictable memory behavior, so the programmer and compiler have enough knowledge to manage the memory hierarchy effectively at compile-time. The programmer and compiler are usually unable to take advantage of this knowledge because of the complexity of detailed memory management at the granularity of data words.

The stream programming model, however, simplifies memory management. Storage in the stream register file is allocated to entire streams, rather than individual words. The programmer or compiler can manage the small number of streams that are active at any given time. This leads to far more efficient communication and storage, as data is transferred only when necessary and the overhead required by a cache to manage memory dynamically is eliminated.

Furthermore, media processing applications map naturally to a bandwidth hierarchy. Kernels operate on data stored locally within the arithmetic clusters, data streams are transferred through the SRF, and global data is stored in external DRAM. Therefore, in this model the programmer allocates large data streams in the SRF, rather than individual data words. This frees the programmer from detailed memory management, making it easy to utilize the large SRF. The bandwidth provided at each level of the hierarchy is well-matched to application demands, as well. For instance, STAP, a radar processing application, has a ratio of memory references to SRF references to local register file references of 1:11:276. This matches the bandwidth provided by the levels of the hierarchy, enabling STAP to achieve a sustained performance of almost 5.5GFLOPS, which is over 25% of Imagine's peak computation rate. Other applications perform even better, such as MPEG which achieves 17.9GOPS.

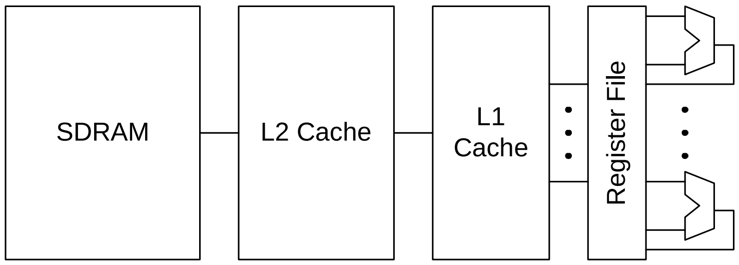


FIGURE 5.2 Conventional Storage Hierarchy

5.2 Communication Bottlenecks

Figure 5.2 shows the conventional storage hierarchy used to feed a processor's arithmetic units with data. At the lowest level, the arithmetic units are fed directly from a global register file connected to all of the arithmetic units. The next level of the storage hierarchy is one or more levels of cache memory. These caches get successively larger and slower the further they get from the arithmetic units. Ultimately, data is stored in large, slow DRAM, which has traditionally been located off the processor chip.

A conventional storage hierarchy is optimized to minimize the arithmetic units' latency to access data. The register file is typically small enough that it can be accessed in a single cycle. The cache hierarchy keeps data that is likely to be accessed soon close to the arithmetic units to minimize the latency of memory operations. The slow DRAM is used as a large storage area which is only accessed when the cache hierarchy does not hold the referenced data. This structure has been quite successful for several reasons:

1. For small numbers of arithmetic units, the global register file is able to provide general storage and communication among arithmetic units allowing data to be quickly forwarded from one arithmetic unit to another.
2. Many applications have small working sets and large amounts of spatial and temporal locality which allow a small cache to provide enough storage to hold most of the useful data required by the program at any given time.
3. The sustainable bandwidth out of modern DRAM exceeds the bandwidth demands of most applications because of the effectiveness of the cache hierarchy.

Conventional storage hierarchies, however, are not well matched to the demands of media processing applications. The structures within the conventional hierarchy are optimized to minimize data access latency. The global register file is designed for fast communication among a small number of arithmetic units, the cache hierarchy is optimized to minimize memory latency, and the DRAM is utilized to provide large amounts of storage. Rather than low latency, media processing applications demand high data bandwidth to sustain large amounts of independent processing. They require far more arithmetic units to achieve their requisite computation rates, tolerate longer memory latencies, and demand high bandwidth from off-chip memory. These characteristics are incompatible with the conventional storage hierarchy and render it ineffective in the following ways:

1. The large computation demands of media applications make a centralized global register file inefficient, if not infeasible, because of the massive bandwidth required by the tens to hundreds of arithmetic units accessing it.
2. Media applications have very large working sets (such as images) and have very little temporal or spatial locality, making caches ineffective. Additionally, the increased number of arithmetic units create bandwidth demands higher than a cache hierarchy could deliver.
3. Media applications have extremely high memory bandwidth demands because they operate on large amounts of raw data (such as video sequences) continuously transferred from an external source that must be buffered in memory.

All three elements of the storage hierarchy prevent the required data bandwidth from being supplied to the arithmetic units. To support numerous arithmetic units, a global register file must provide a large amount of storage and provide high data bandwidth. Technology constraints make such a register file costly. Furthermore, the cache hierarchy cannot provide enough bandwidth to support the additional arithmetic units. Caches are burdened by the fact that they are transparent to the programmer and compiler. Since the program does not direct data movement within the hardware, the cache must dynamically transfer and store data which may or may not be used in the future. The hardware overhead required to manage this memory limits the effective bandwidth that a cache can provide. Without sufficient bandwidth from the cache hierarchy, the arithmetic units will often sit idle waiting for data from the cache. Finally, when the DRAM is managed to minimize the latency of references that must access it, the sustained bandwidth of the DRAM is lowered. The following sections further describe the communication bottlenecks of the global register file, cache hierarchy, and DRAM.

5.2.1 Register File Structure

Programmable processors have typically combined storage and communication among arithmetic units into a centralized global register file. These register files have been designed to supply the peak bandwidth required by the arithmetic units. Since most processors have few arithmetic units (typically less than 10), this register file design has provided the simplest and most general mechanism for communicating data among arithmetic units. However, current technology constraints make such centralized register files costly and inefficient, even for supporting relatively small numbers of arithmetic units. A centralized global register file used to interconnect four arithmetic units would be approximately 0.2mm^2 in a 0.18μm CMOS process. To connect 16 arithmetic units, a centralized register file would be around 6.4mm^2. In fact, the size of a centralized register file grows as N^3 for N arithmetic units. Similarly, the power dissipated by a centralized register file grows as N^3 and the delay grows as $N^{3/2}$ [RDK+00b].

A centralized register file cannot efficiently support tens to hundreds of arithmetic units because of its cost, in terms of area, power, and delay. To support the large number of arithmetic units demanded by media processing applications, the register file must be partitioned to reduce its cost. Register file partitioning has been used effectively in architectures such as the Compaq Alpha 21264 microprocessor [Kes99] and the Texas Instruments 'C6000 digital signal processors [TH97]. Partitioning enables increased bandwidth from the register file structure at reduced cost, allowing the register files to support larger numbers of arithmetic units efficiently. A partitioned register file can be incorporated into the conventional storage hierarchy; however, conventional caches cannot scale enough to supply data bandwidth for the tens to hundreds of arithmetic units demanded by media processing applications.

5.2.2 Cache Hierarchy

In order for a cache hierarchy to be effective, applications must exhibit large amounts of spatial and temporal locality. By keeping frequently accessed data closer to the arithmetic units, the latency to access that data is drastically reduced. Media processing applications do not exhibit much temporal locality, since they typically only reference data once, process it, and never reference it again. Therefore, holding recently accessed data close to the arithmetic units in a cache is useless. Media applications also do not exhibit much spatial locality, as they tend to access data in strided patterns. For non-unit strides, this leads to many words of each cache line being unreferenced, especially if the stride is larger than the cache line size. This renders the cache's ability to prefetch data located in the vicinity of referenced data useless, as that prefetched data will not be accessed. Also, most media processing applications have extremely large working sets, which can easily exceed the capacity of a cache and cause thrashing.

Media processing applications are far more sensitive to bandwidth than latency. So, even if caching were effective for these applications, a cache's ability to reduce memory latency will not necessarily improve performance. Increasing the sustainable memory bandwidth would be far more beneficial for these bandwidth intensive applications. An effective cache can reduce the bandwidth demands on external DRAM. The cache prevents unnecessary off-chip memory references from occurring by holding recently accessed data on the chip. However, an ineffective cache can actually increase the bandwidth demands on the DRAM. A cache, by its very nature, speculatively prefetches data on every memory access. If that data is never referenced, which is the case in applications without spatial locality, then precious DRAM bandwidth was wasted transferring that data. This is increasingly problematic as memory bandwidth becomes more scarce compared to compute resources [BGK97].

Furthermore, a cache is an inefficient way to stage data from memory. Address translation is required on every reference, accesses are made with long memory addresses, some storage within the cache must be allocated to address tags, and conflicts may evict previously fetched data that will be referenced in the future. Despite this overhead, caches reduce memory latency dynamically at run-time and allow hardware controlled management of on-chip memory. In conventional processors, these benefits offset the inefficiencies of a cache. However, media processing applications have predictable memory behavior which allows entire streams of data to be non-speculatively prefetched long before they are needed. Therefore, it is more efficient to provide sufficient registers to hide the latency of memory and to prefetch data directly into those registers rather than into a cache.

5.2.3 DRAM

Off-chip DRAM bandwidth is fundamentally limited by the pins on the chip, both in terms of the number of available pins and the achievable bandwidth per pin. Conventional synchronous DRAM (SDRAM) chips currently deliver a peak transfer rate of up to about 133Mb/s per data pin. The peak data bandwidth of off-chip DRAM is further related to the number of pins that the processor can allocate to such memory. Somewhat recent improvements in DRAM have increased that rate to about 266Mb/s per pin in Double Data Rate SDRAM (DDR SDRAM) and about 800Mb/s per pin in Rambus Direct RDRAM (DRDRAM [Cri97]).

However, existing DRAM architectures cannot sustain these rates on random memory accesses. To maximize memory bandwidth, modern DRAM components allow pipelining of memory accesses, provide several independent memory banks, and cache the most recently accessed row of each bank. While these features increase the peak supplied memory bandwidth, they also make the performance of the DRAM highly dependent on the access pattern. Depending on the reference pattern, 14-97% of off-chip DRAM bandwidth can actually be utilized on continuous memory stream transfers [RDK+00a]. Therefore, to maximize DRAM bandwidth, DRAM accesses must be carefully managed to take advantage of the internal DRAM structure, rather than attempting to minimize latency by satisfying references in order. Similarly, on-chip storage and bandwidth must be organized and managed so as to minimize transfers to this off-chip memory and thereby avoid wasting precious off-chip memory bandwidth.

5.3 Register Organization

The communication bottlenecks of modern VLSI together with the streaming nature of media processing applications motivate an explicitly managed bandwidth hierarchy to effectively utilize tens to hundreds of arithmetic units for media processing tasks. A bandwidth hierarchy allows media processing applications to utilize communication-limited technology efficiently. Temporary data is stored locally in the arithmetic clusters where it may be accessed frequently and quickly. Intermediate streams are stored on-chip in the SRF where they can be recirculated between kernels without requiring costly memory references. Finally, input, output, and global data is stored in external DRAM, since it is referenced infrequently and requires large amounts of storage space. Data is referenced far more frequently in the levels of the hierarchy that are closer to the arithmetic units. Therefore, each level of the hierarchy must provide successively more bandwidth than the level below it. Media applications map naturally to such a bandwidth hierarchy, enabling them to utilize inexpensive, local bandwidth when possible, and only consume expensive, global data bandwidth when absolutely necessary.

As shown in Section 5.2, the register file, cache, and DRAM are all communication bottlenecks in modern VLSI. As explained in Section 5.2.3, modern DRAM bandwidth is constrained by the peak bandwidth of the pins devoted to accessing that DRAM. Therefore, the only thing that can be done to improve the sustained bandwidth of the DRAM is to manage the accesses to that DRAM in order to take advantage of its internal structure. Maximizing the sustained bandwidth of modern DRAM will be discussed in Chapter 6. Since caching is not an effective method of improving media processing performance, the register file must be structured such that it can hide the latency of memory. This section will therefore focus on how modern VLSI technology constrains register file organization.

The cost of a register file, in terms of area, power, and delay, is directly related to the number of registers in the file, R, and the number of ports that access the file, p. As shown in [RDK+00b], the area of a register file is proportional to p^2R, the power dissipation is also proportional to p^2R, and the delay is proportional to $pR^{1/2}$. For a centralized register file, both p and R are proportional to the number of arithmetic units in the processor, leading to area and power proportional to N^3 and delay proportional to $N^{3/2}$ for N arithmetic units [RDK+00b]. Such a register file clearly does not scale well to support large numbers of arithmetic units.

Media processing applications require more registers per arithmetic unit than a general-purpose processor since a conventional cache hierarchy is ineffective. These additional registers are used to hide the latency of memory directly within the register file structure. This makes it even more difficult to scale the register file to support large numbers of arithmetic units. As shown previously, media processing applications can easily be expressed in the stream programming model to expose the parallelism in their computations and the regularity of their communications. The large data streams that must be transferred to/from memory and between kernels could easily interfere with each other within a cache, make inefficient use of cache lines, and exceed a cache's capacity. However, these same data streams can efficiently be stored in a large register file, as their access is regular and predictable at compile time. Unfortunately, the combined effect of these additional registers and the large number of arithmetic units needed by media processing applications further increases the cost of the register file. A register file must, therefore, be partitioned in order to reduce its area, power, and delay to acceptable levels.

One efficient way to partition a register file is shown in Figure 5.3. A distributed register file (DRF) organization such as this can provide inexpensive data bandwidth to a large number of arithmetic units. Each arithmetic unit has a small two-ported register file connected directly to each of its inputs. A single switch then routes data among arithmetic units and register files, in contrast to the centralized register file structure commonly used in practice which effectively contains this switch inside of every register. The DRF organization reduces the area of the register file for N arithmetic units to be proportional to N^2, instead of N^3. Similarly, the power and delay of such a structure is reduced N^2 and N, respectively, as opposed to N^3 and $N^{3/2}$ [RDK+00b].

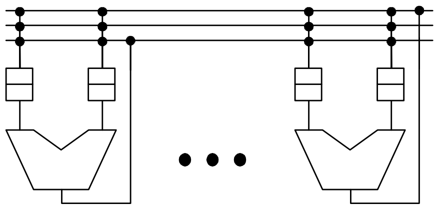


FIGURE 5.3 Distributed Register File Organization

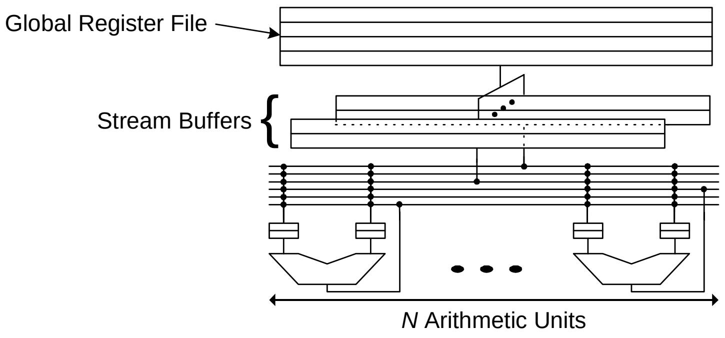


FIGURE 5.4 Stream Register File Organization

A stream register file organization, as shown in Figure 5.4, provides inexpensive storage for data streams, which can be transferred sequentially to/from local DRF storage near the arithmetic units. The global register file stores data streams that will be recirculated through the cluster of arithmetic units for successive kernel operations, so it only needs to support sequential stream accesses. Since large, multiported register files are expensive, the cost of this register file can be reduced by organizing it with a single wide port and a set of stream buffers. Each stream buffer only allows sequential accesses. The single port to the register file is shared in time by all the stream buffers which transfer wide data words when they obtain access to the port. The arithmetic units may then access the stream buffers at the granularity of individual data words. Effectively, this organization separates the register file into two parts: a large register file with a single port to stage data from memory and smaller register files that feed the arithmetic units.

Data from memory is only transferred to/from the global stream register file (SRF), similar to a conventional load/store architecture except that the unit of data is an

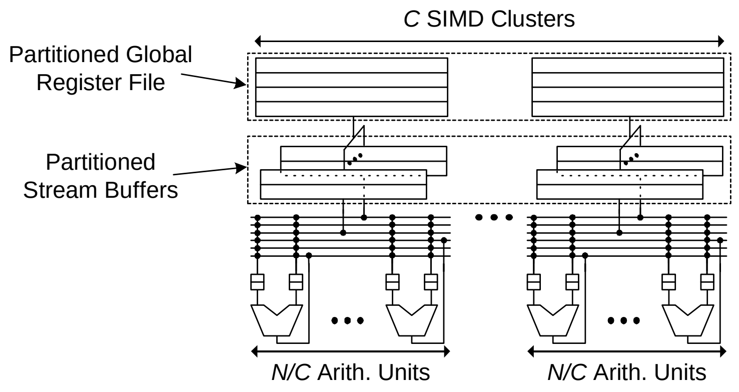


FIGURE 5.5 Stream/SIMD Register File Organization

entire stream. The SRF effectively isolates the arithmetic units from the long, variable latencies of DRAM accesses. Since kernels are not started until their input streams are available in the SRF, their stream input and output operations cannot stall the arithmetic units. Therefore it is not only practical, but advantageous, to statically schedule the arithmetic units at compile time. Since the compiler knows that stream references will not stall the arithmetic clusters, it can schedule arithmetic operations more efficiently than a dynamic scheduler. The compiler can consider a larger number of operations and does not need to consider variable dynamic latencies due to memory, since all global data references access the stream register file. Additionally, the hardware cost of a dynamic superscalar operation scheduler is eliminated.

The arithmetic units can be further partitioned into single-instruction, multiple-data (SIMD) clusters, as shown in Figure 5.5. Because of the abundant data parallelism in media processing applications, the arithmetic units and their distributed register files can be grouped into identical clusters. Each arithmetic cluster can only access its dedicated portion of the SRF and operates on subsequent data records in parallel. By reducing the number of arithmetic units that must be interconnected by a switch in this manner, the area, power, and delay of the register file structure is reduced. Additionally, since the clusters are performing data-parallel computations, they can share a single controller, further reducing the overhead associated with each arithmetic unit. This structure reduces the area and power of the register file for N arithmetic units organized into C clusters to be proportional to N^2/C and the delay to be proportional to N/C [RDK+00b].

A partitioned register organization, including local distributed register files and a global stream register file divided into SIMD clusters, and external DRAM combine to form an efficient bandwidth hierarchy. A 32-bit wide single data rate SDRAM operating at 125MHz yields a peak bandwidth of 500MB/s and a single Rambus channel operating at 400MHz yields a peak bandwidth of 1.6GB/s. Therefore, by allocating pins to four external banks of DRAM, modern DRAM can provide 2-6.4GB/s of peak data bandwidth. A stream register file spanning most of the width of a 1cm^2 chip provides a peak bandwidth of 32-64GB/s, an order of magnitude higher than DRAM. An eight cluster, distributed register file structure supporting 48 total arithmetic units can provide a peak bandwidth in excess of 500GB/s, an order of magnitude higher than a global stream register file.

5.4 Evaluation

The bandwidth hierarchy described in the previous section is well-matched to modern VLSI technology, but to be useful, applications must be able to take advantage of it. If applications perform only a single operation on each unique word referenced in memory, then the bandwidth hierarchy would be rendered useless and the computation rate would be entirely limited by memory bandwidth. Ideally, the ratio among memory references, global data references, and local data references would exactly match the supplied bandwidth of the three levels of the hierarchy. In practice, the application demands lie somewhere between these two extremes.

This section will evaluate the efficiency of the bandwidth hierarchy. First, Section 5.4.1 will show that kernels can make efficient use of local register files for temporary storage to feed the arithmetic units and of an SRF to feed input streams to the arithmetic units and store output streams from the arithmetic units. Second, Section 5.4.2 will show that when kernels are considered as part of an application, rather than in isolation, the SRF can be efficiently used to capture the locality of stream recirculation within media processing applications, thereby limiting the bandwidth demands on off-chip memory. Third, Section 5.4.3 will show that media processing applications demand successively increasing amounts of bandwidth when they are mapped to the three-tiered storage hierarchy described in this chapter. Fourth, Section 5.4.4 will show that the hierarchical bandwidth scaling of the storage hierarchy is in fact what enables these media processing applications to achieve 77-96% of the performance that they would sustain given infinite instantaneous data bandwidth. Finally, Section 5.4.5 will show the sustained performance and arithmetic utilization that the data bandwidth hierarchy enables Imagine to achieve.

```
for (n=0; n<NUMOUTPUTS; n++)
{
  sum = 0;                          // 32-bit accumulator
  for (i=0; i<13; i++)
    sum += k[i] * input[n+i];       // MAC for convolution
  output[n] = (sum + 0x4000) >> 15;  // Round and shift to 16 bits
}
```

FIGURE 5.6 C Code for a 13-tap, 16-bit Fixed-point, Real FIR Filter

5.4.1 Kernel Comparison

A comparison of kernel performance across programmable architectures shows the advantages of the bandwidth hierarchy within a stream processor. Consider a simple media processing kernel, a Finite Impulse Response (FIR) filter. An FIR filter is a one dimensional convolution of a small kernel over a long data stream. The following equation describes the operation (*M* is the number of coefficients in the filter):

$$y(n) = \sum_{i=0}^{M-1} k_i \cdot x(n-i) \tag{5.1}$$

Input values that do not exist on the boundaries of the computation are defined to be 0. In this section, a 13-tap filter is used consisting of 16-bit fixed-point coefficients with 15 bits of fraction and one sign bit, making the coefficients, *k*, lie in the range $-1 \le k < 1$. The input stream also consists of 16-bit fixed-point numbers with the output having the same precision as the input. For accuracy, the 32-bit products are summed at 32-bit precision and then rounded and shifted to generate the final 16-bit results. The input stream can be zero padded to simplify boundary cases, although this is not necessary.

Figure 5.6 shows the C code for a simple implementation of such an FIR filter. It assumes that there are extra zeros at the beginning and end of the input stream so that bounds checking does not need to be performed. Also, the coefficient array is reversed to simplify the index calculations. All of the implementations of the FIR filter presented in this section are functionally equivalent to this C code, except that the implementations either do not require zero padding or only require it on one end of the input.

The number of global data references and the number of arithmetic operations required by this kernel can be determined directly from the C code. Assuming an infi-

	DSP	Parallel-subword	Stream
Memory	36.0	49.9	≤ 4.03
Global Register File	664.1	296.7	4.03
Local Register File	N/A	N/A	420.02

TABLE 5.1 Bytes Referenced per 16-bit FIR Filter Output (2048 Output, 13-tap filter)

nite amount of register storage, each input would have to be loaded once, each output would have to be stored once, and the 13 coefficients would have to be loaded into the register file once. This would lead to a total of slightly more than 4 bytes of memory referenced per output. For each output, 13 multiplies, 13 additions, and one shift are required (the C code performs one extra addition operation for simplicity). If the only register file references made by the code were to support the minimum required transfers between memory and the register file and to support the input and outputs of the computations, this would lead to a total of 276 bytes referenced in the register file per output.

Table 5.1 shows the bandwidth demands of this FIR filter for a scalar digital signal processor (Texas Instruments TMS320C6203 [TH97]), a parallel-subword enhanced general-purpose processor (Intel Pentium II MMX [PW96]), and a stream processor (Imagine). The 'C6 DSP has a similar architecture to the conventional architecture presented at the beginning of this chapter. However, instead of a single centralized register file, the eight arithmetic units are split into two groups of four, each with its own register file. The on-chip memory of the 'C6 can be configured either as cache or as addressable memory, with equivalent bandwidth in both organizations. The architecture of the Pentium II is a very close approximation to the conventional storage structure. The parallel-subword extensions (MMX [PW96]) allow SIMD operations to be performed on data within the 64-bit floating-point registers. This improves performance and efficiency when performing the 16-bit and 32-bit calculations of the FIR filter. Imagine, described in Chapter 4, utilizes a data bandwidth hierarchy.

As can be seen in Table 5.1, all of the processors use more bandwidth than is strictly necessary to perform the FIR filter. The DSP references 36 bytes of memory per output, rather than the 4 bytes that are necessary, because 16 coefficients and 18 inputs are loaded for every pair of outputs, leading to 32 bytes of additional memory traffic. For each pair of outputs, 16 coefficients and 18 inputs are loaded, instead of 13 coefficients and 14 inputs, because the filter procedure is unrolled such that the filter length must be a multiple of eight. FIR filters with lengths that are not a multiple of eight would achieve the same performance as if their length were the next multiple of eight, so the code for a 16-tap filter was used with three of the taps being zero. Over 664 bytes are referenced within the register file per output for this version of the code, as opposed to the 276 bytes that are strictly necessary, for the following reasons: the extra memory traffic also generates additional register accesses, the 32-bit registers are used to hold 16-bit values causing useless bytes to be accessed for 16-bit operations, and addressing and loop counting operations require register usage.

The MMX version of the code suffers similar problems as the DSP in terms of memory references. For every pair of outputs, 52 coefficients and 16 inputs are loaded. Since the inputs are packed into 64-bit words, four copies of the 13 filter coefficients are required. Each copy is stored in four 64-bit words, aligned to start at a different 16-bit boundary. Unlike the DSP version of the code, this version is optimized for a 13-tap filter, so the additional coefficients are only needed to solve alignment issues. For each pair of outputs, 16 inputs are loaded, rather than 14, as the inputs are also stored in 64-bit words. This leads to the additional 46 bytes of memory references over the required 4 bytes per output. The MMX version of the code, however, is very efficient in its register usage, requiring only about 297 bytes of register accesses in comparison to the required 276. This is due to several factors: the Pentium allows one operand of many operations to come from memory and the 64-bit words are used to efficiently hold four 16-bit samples or two 32-bit accumulators. Otherwise, the MMX code suffers from similar addressing and loop counting overhead.

The table clearly shows the benefits of the bandwidth hierarchy in the Imagine column. Imagine avoids unnecessary memory traffic, separates global and local data, and does not require addressing calculations to access streams. Imagine makes very close to the minimum required number of memory references (4.03 bytes). Each input is loaded once, each output is stored once, and 26 coefficients are loaded once for the entire filter (the 16-bit coefficients are packed into 32-bit words, so two differently aligned copies are required, as in the MMX code). In an application, the input to the FIR is likely to have been produced by a previous kernel and the output from the FIR is likely to be consumed by a subsequent kernel. Therefore, the table lists Imagine’s memory references as less than or equal to 4.03 bytes because, in practice, the input to the FIR will already be in the SRF and the output will stay in the SRF. Imagine also only references these same 4.03 bytes in the global register file. Once the coefficients are moved into the arithmetic clusters, they are stored locally for the duration of the computation. Similarly, since there is enough register storage provided by the DRF structure of the local register files, inputs do not need to be loaded repeatedly. Therefore, only 4.03 bytes per output are referenced in the SRF. Imagine references 420 bytes of local register storage per output, compared to the required 276 bytes, in order to transfer data around from one register file to another within the DRF structure. Data must be transferred among register files in the DRF structure to facilitate software pipelining and to duplicate data that is needed by multiple functional units. This local register bandwidth, however, is much cheaper to provide than global register bandwidth, so these additional referenced bytes do not hurt performance.

Figure 5.7 shows the FIR filter kernel written in the stream programming model. The inner loop from the C code implementation is completely unrolled to enable better performance when the outer loop is software pipelined. The 16-bit inputs are packed into 32-bit words (the *half2* data type consists of two 16-bit values packed into a single 32-bit word), so in each loop iteration, two inputs values are read per cluster and two output values are written per cluster. The code in the first page of the figure is the overhead required to correctly compute the FIR on an eight cluster SIMD machine. At the top of each loop iteration, each cluster reads two new input values. Since the inputs are interleaved across the eight clusters, they are rotated (using the *commucperm* instruction), and delayed for one loop iteration if necessary (using the *?:* operator and *cluster_id()* value), in order to provide each cluster with the previous twelve inputs that will be needed to produce that cluster’s two outputs.

Since the setup code on the first page of the figure places the correct 14 inputs into *in01* through *inAB*, all eight clusters are able to compute their outputs in parallel. The code in the second page of the figure is the unrolled computations needed to compute two outputs of the FIR filter. The *muld* operator takes two packed 16-bit inputs and multiplies the high 16-bit values of each word to produce one 32-bit result and multiplies the low 16-bit values of each word to produce another 32-bit result. The *hi_lo* operator – which is a language construct, not a hardware instruction – places these two 32-bit products into its two arguments. These products are then summed and rounded to produce two output values, *outC* and *outD*. These 16-bit values are then packed into a single 32-bit word and output to the SRF.

Table 5.2 shows the performance and bandwidth usage of the three processors on the FIR filter. The results in the table are generated assuming that all three processors have memory systems with infinite bandwidth that never cause the processors to stall. The number of operations that an architecture can perform per byte referenced in register files or memory can be calculated by dividing the sustained performance in the table by the appropriate data bandwidth in the table. For example, Imagine is able to perform 6.7 operations per byte referenced in the global register file, whereas the 'C6 and Pentium II are only able to perform 0.04 and 0.09 operations per byte, respec-

	DSP	Parallel-subword	Stream
Processor	TMS320C6203	Pentium II MMX	Imagine
Technology (µm)	0.15	0.25	0.15
Frequency (MHz)	300	450	500
Performance (GOPS)	1.01	1.47	17.57
Inner Loop ALU Utilization (%)	100	94	77
Local RF BW (GB/s)	N/A	N/A	273.25
Global RF BW (GB/s)	24.88	16.20	2.62
Memory BW (GB/s)	1.42	2.73	≤ 2.62

TABLE 5.2 FIR Filter Performance and Bandwidth (2048 Output, 13-tap Filter)

tively. Therefore, to achieve a given level of performance, the bandwidth hierarchy of Imagine requires two orders of magnitude less global register file bandwidth. By offloading the job of feeding the arithmetic units from the global register file to the local register files, the bandwidth hierarchy is able to support a much larger number of arithmetic units using a global register file with approximately the same peak bandwidth. Imagine achieves a similar number of operations per byte of local register file bandwidth (0.06) as the other architectures do per byte of global bandwidth. However, since local register bandwidth is inexpensive in terms of area, power, and delay, far more arithmetic units can be supported. This allows Imagine to achieve over 17GOPS on the FIR filter, while the ’C6 and Pentium II can only achieve about 1 and 1.5GOPS, respectively. Even if the Pentium II’s clock speed is increased to 750MHz to account for technology differences, it would still only achieve about 2.5GOPS with 0.09 operations per byte of global register bandwidth. Therefore, the bandwidth hierarchy of a stream processor efficiently scales data bandwidth that can be effectively utilized by media processing applications to achieve higher performance.

Table 5.2 also shows the utilization of the arithmetic units of each processor in the inner loop of the FIR filter kernel. The 'C6, which is optimized for such digital signal processing functions as the FIR filter, achieves 100% utilization of its eight arithmetic units. This is possible because the types of arithmetic units are an exact match for this kernel. The Pentium II is only able to achieve 94% utilization of its two arithmetic units. Data dependencies prevent a few of the issue slots in the inner loop from being

```
kernel firfx13(istream<half2> data,
               istream<half2> krnl,
               ostream<half2> output)
{
  // Declare variables
  half2 k01, k23, k45, k67, k89, kAB, kC_; // Kernel for high output
  half2 k_0, k12, k34, k56, k78, k9A, kBC; // Kernel for low output
  half2 in01, in23, in45, in67, in89, inAB, inCD; // Input Data
  half2 com01, com23, com45, com67, com89, comAB; // Communicated data
  half2 dly01, dly23, dly45, dly67, dly89, dlyAB; // Delayed data;
  int p00, p11, p22, p33, p44, p55, p66, p77, p88, p99, pAA, pBB, pCC;
  int p01, p12, p23, p34, p45, p56, p67, p78, p89, p9A, pAB, pBC, pCD;
  int sum00, sum01, sum02;
  int sum10, sum11, sum12;
  int outC, outD;
  half2 outCD;

  // Input kernel (1.15 fixed point numbers)
  krnl >> k01 >> k23 >> k45 >> k67 >> k89 >> kAB >> kC_;
  krnl >> k_0 >> k12 >> k34 >> k56 >> k78 >> k9A >> kBC;

  // Initialized delayed data to 0
  dly01 = 0;  dly23 = 0;  dly45 = 0;  dly67 = 0; dly89 = 0; dlyAB = 0;

  // Mask to pack two 16-bit results into a 32-bit word
  cc packcc = itocc(0xFFFF0000);

  loop_stream(data) pipeline(1) {
    // Get next data elements (2 packed 16-bit values per 32-bit word)
    data >> inCD;

    // Communicate data to other clusters
    comAB = commucperm(rotateRightBy1, inCD);
    com89 = commucperm(rotateRightBy2, inCD);
    com67 = commucperm(rotateRightBy3, inCD);
    com45 = commucperm(rotateRightBy4, inCD);
    com23 = commucperm(rotateRightBy5, inCD);
    com01 = commucperm(rotateRightBy6, inCD);

    // Select between current and delayed data (for end clusters)
    inAB = (cluster_id() < 1) ? dlyAB : comAB;
    in89 = (cluster_id() < 2) ? dly89 : com89;
    in67 = (cluster_id() < 3) ? dly67 : com67;
    in45 = (cluster_id() < 4) ? dly45 : com45;
    in23 = (cluster_id() < 5) ? dly23 : com23;
    in01 = (cluster_id() < 6) ? dly01 : com01;

    // Delay data for end clusters
    dlyAB = comAB; dly89 = com89; dly67 = com67;
    dly45 = com45; dly23 = com23; dly01 = com01;
```

FIGURE 5.7 KernelC Code for a 13-tap, 16-bit Fixed-point, Real FIR Filter

```
        // Calculate Products
        hi_lo(p00, p11) = muld(k01, in01);
        hi_lo(p22, p33) = muld(k23, in23);
        hi_lo(p44, p55) = muld(k45, in45);
        hi_lo(p66, p77) = muld(k67, in67);
        hi_lo(p88, p99) = muld(k89, in89);
        hi_lo(pAA, pBB) = muld(kAB, inAB);
        pCC             = hi(muld(kC_, inCD));

        p01             = lo(muld(k_0, in01));
        hi_lo(p12, p23) = muld(k12, in23);
        hi_lo(p34, p45) = muld(k34, in45);
        hi_lo(p56, p67) = muld(k56, in67);
        hi_lo(p78, p89) = muld(k78, in89);
        hi_lo(p9A, pAB) = muld(k9A, inAB);
        hi_lo(pBC, pCD) = muld(kBC, inCD);

        // Calculate sums
        sum00 = ((p00 + p11) + (p22 + p33));
        sum01 = ((p44 + p55) + (p66 + p77));
        sum02 = ((p88 + p99) + (pAA + pBB));

        sum10 = ((p01 + p12) + (p23 + p34));
        sum11 = ((p45 + p56) + (p67 + p78));
        sum12 = ((p89 + p9A) + (pAB + pBC));

        outC = (sum00 + sum01) + (sum02 + pCC);
        outD = (sum10 + sum11) + (sum12 + pCD);

        // Round 32-bit results and shift back to 16-bit precision
        outC = (outC + 0x4000) << 1;
        outD = (outD + 0x4000) >> 15;

        // Pack results and append to output stream
        outCD = select(packcc, outC, outD);
        output << outCD;
      }
    }
```

FIGURE 5.7 KernelC Code for a 13-tap, 16-bit Fixed-point, Real FIR Filter

used. Finally, Imagine achieves 77% utilization of it's 48 arithmetic units. The eight dividers go unused in this kernel, and the communication unit is 100% utilized, preventing further improvement. Even though the mix of arithmetic units in Imagine is not as ideally suited to the FIR filter kernel as the mix of units in the 'C6, the bandwidth hierarchy still enables Imagine to keep over three-fourths of its compute resources busy.

The performance of the FIR filter kernel presented here does not show the actual effects of memory latency and bandwidth. Table 5.2 shows the amount of memory bandwidth that would be needed in order to sustain the given computation rates. The structure of the memory system and the reference locality of the memory accesses determine how much of that bandwidth would actually need to be provided by external DRAM. The 'C6 and Pentium II would be able to satisfy many of their memory references directly out of their on-chip caches. Depending on data placement, however, conflicts may evict some reusable data. The 'C6 would be able to avoid these evictions by configuring on-chip memory as addressable storage, rather than a cache. External DRAM must provide the bandwidth that is not provided by these on-chip memories and caches. If the majority of the memory bandwidth demand is not satisfied by on-chip resources, then these applications will become memory bound and their sustained computation rates will drop.

If the FIR filter was the only processing being performed on the data streams in Imagine, then all of the required memory bandwidth given in Table 5.2 would have to come from external DRAM. However, when entire applications are considered, the bandwidth hierarchy of Imagine enables many of the memory references required by the FIR filter kernel to be eliminated. Streams would be recirculated through the SRF from one kernel to the next, so the inputs and outputs of the FIR would not need to be stored in memory. Rather, the output of a previous kernel would be recirculated through the SRF and input to the FIR, and the output of the FIR would be recirculated through the SRF to another kernel. This recirculation of streams through the SRF dramatically reduces the bandwidth demands on memory in Imagine.

5.4.2 Application Evaluation

The MPEG2 I-frame encoder, introduced in Section 4.1.2, efficiently utilizes the bandwidth hierarchy to minimize references to the SRF and off-chip memory. Figure 5.8 shows how the MPEG2 I-Frame encoder maps to the three-tiered bandwidth hierarchy. The figure also shows the number of bytes that are referenced by the application at each level of the hierarchy. The seven computation kernels of the application execute on the arithmetic clusters, as shown in the figure. During the execution of

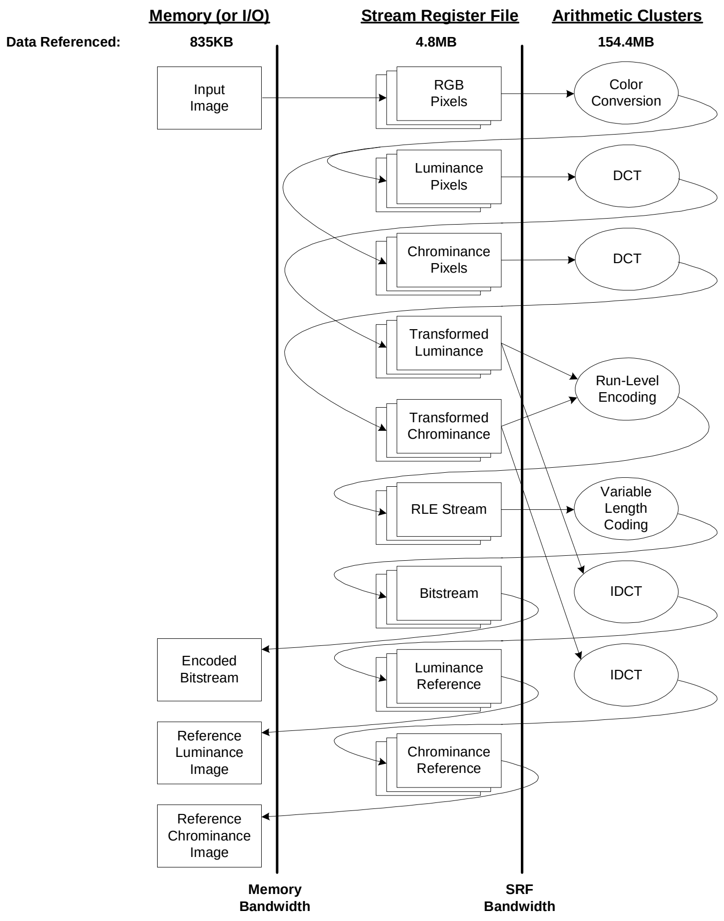


FIGURE 5.8 I-Frame Encoding from MPEG2 Encoder

each kernel, the SRF is only referenced to read the kernel's inputs and write the kernel's outputs. For example, the color conversion kernel reads RGB pixels from its input stream, performs operations on local data, and writes luminance and chrominance pixels to its output streams. All intermediate data is stored in the local register files within the clusters, resulting in 154.4MB of data being referenced within those register files. As each pixel is converted by the color conversion kernel, for instance, all of the partial sums and products are stored within these local register files. Data streams that are passed from kernel to kernel are stored within the SRF and do not need to return to memory, as shown in the figure. This results in 4.8MB of data being

	Unconstrained Bandwidth	Constrained SRF Bandwidth	Bandwidth Use
Performance (GOPS)	10.89	10.83	10.58
Local Register File Bandwidth (GB/s)	167.47	166.53	162.64
Global Register File Bandwidth (GB/s)	5.23	5.20	5.08
Memory Bandwidth (GB/s)	0.88	0.88	0.86

TABLE 5.3 Bandwidth Demands and Use of MPEG2 I-frame Encoder

referenced within the SRF. No intermediate data that is used in the encoding process of the current frame is stored to memory. The only memory references that are performed by the application are to read the original unencoded image, to write the final encoded bit stream, and to write the reference images that will be used to encode future P- or B-frames, resulting in 835KB referenced in memory. In practice, the unencoded images could enter the SRF directly from an input device connected to the Imagine network. Similarly, the encoded bit stream could leave the SRF directly for the Imagine network, rather than being stored in memory.

Table 5.3 shows the bandwidth demanded by and used by the I-frame encoder. The first column shows the bandwidth demands of the application when the bandwidth of the memory system and the SRF are unconstrained. This demand is the amount of data bandwidth that would be consumed by the application if it were executed on the Imagine architecture with an SRF and memory system that could provide infinite instantaneous bandwidth. This means that streams can be transferred between memory and the SRF instantaneously, and that the arithmetic clusters never stall when reading from or writing to the SRF. The second column shows the bandwidth that would be consumed by the application if it were executed on the Imagine architecture with a normal SRF and a memory system that could provide infinite instantaneous bandwidth. This means that memory transfers are only limited by the available SRF bandwidth. Finally, the third column shows the sustained data bandwidth that the application is able to achieve on the Imagine architecture given the actual limits of both the SRF and memory system. As can be seen from the table, this application uses almost six times more SRF bandwidth than memory bandwidth and about 32 times more local register file bandwidth than SRF bandwidth.

This illustrates the effectiveness of the bandwidth hierarchy at all three levels. The local register files are able to meet the demands of the arithmetic units and provide far more data bandwidth than would be available in traditional architectures. The SRF is only utilized for stream storage between kernels and therefore only needs to sustain an order of magnitude less data bandwidth. Finally, the memory system holds global data and is referenced only when necessary, keeping the overall memory demands to a minimum.

The local register files effectively supply the arithmetic units with data. The arithmetic clusters demand about 167.5GB/s of data bandwidth to achieve 10.9GOPS of computational throughput. In practice, the local register files actually provide about 162.6GB/s of bandwidth. However, the slight decrease from the demand occurs because of SRF and memory bandwidth limitations, not bandwidth limitations of the local register files. The SRF limits the sustainable performance slightly; the arithmetic clusters are idle for about 0.5% of the run-time because the SRF is unable to sustain bandwidth bursts above 32GB/s for extended periods of time. The table shows that constraining the SRF bandwidth reduces the application's performance to 10.8GOPS. The arithmetic clusters are also idle for about 2.5% of the run-time waiting for memory transfers to complete, further reducing performance to about 10.6GOPS.

The SRF is utilized for intermediate stream storage between kernels. This results in a sustained demand over the entire application of about 5GB/s. The SRF is actually able to provide over 97% of this bandwidth. As previously mentioned, the drop in bandwidth from the demand occurs because of limited memory bandwidth and because the application has bursty SRF access patterns that sometimes exceed the 32GB/s of average bandwidth the SRF can provide. The utilization of the SRF for stream recirculation keeps the memory bandwidth demands at a realizable level.

The application demands only 0.88GB/s of memory bandwidth to sustain 10.9GOPS of computation. This is a ratio of 49.5 operations per word of memory referenced. It is not difficult for the Imagine memory system to sustain a significant fraction of this demand. In fact, it is able to achieve about 97% of this bandwidth, almost one half of the peak bandwidth of the Imagine memory system.

5.4.3 Application Bandwidth Demands

For a given arithmetic capacity, a stream processor can achieve a peak computation rate based on the available parallelism within the applications. This rate can be measured by providing infinite instantaneous data bandwidth from the SRF and memory system. Table 5.4 shows the peak computation rate on the Imagine stream processor, with 48 arithmetic units organized into eight SIMD clusters and a 128KB SRF, for the

Application	Operations per Memory Reference	Performance	
DEPTH	60.1	14.94 GOPS	274 frames per second
MPEG	164.8	18.69 GOPS	364 frames per second
QRD	131.3	13.79 GOPS	1.1 ms per QRD
STAP	43.3	7.03 GOPS	11.7 ms per interval
RENDER	25.1	4.77 GOPS	83 frames per second

TABLE 5.4 Computation Rate with Unlimited Data Bandwidth

five applications (DEPTH, MPEG, QRD, STAP, and RENDER) that were introduced in Section 3.2. As can be seen in the table, these five applications can achieve sustained computation rates from between 5 and 19 billion operations per second, if the 48 arithmetic units are provided with infinite data bandwidth. These values are a function of the available parallelism within the applications, the throughput and latency of the arithmetic units, and the separation of the arithmetic units into SIMD clusters. The table also shows that these applications perform between 25 and 165 operations per word accessed in memory. This shows the computational intensity of these applications and further motivates the necessity of a bandwidth hierarchy to enable such large amounts of computation per memory reference.

In order to achieve the computation rates presented in Table 5.4, each application demands a certain amount of bandwidth from each level of the bandwidth hierarchy. These demands are shown in Table 5.5. As can be seen from the table, all of the applications demand less than 1GB/s of memory bandwidth. This illustrates the effectiveness of the bandwidth hierarchy in eliminating unnecessary memory references in media processing applications. Without the global register file recirculating streams, the memory system would have to sustain from 2.4 to 22.5 GB/s of data bandwidth to enable these applications to achieve their peak computation rate. While it is possible for the streaming memory system of the Imagine stream processor to sustain around 1GB/s of bandwidth out of modern DRAMs (see Chapter 6), it is impossible to exceed the peak DRAM bandwidth of 2GB/s. Therefore, without the bandwidth hierarchy, these applications would all be limited by memory bandwidth.

The table further shows that the bandwidth demands of the applications increase by an order of magnitude at each tier of the bandwidth hierarchy. For example, the ratio

Application	Memory Bandwidth (GB/s)	Global Register Bandwidth (GB/s)	Local Register Bandwidth (GB/s)
DEPTH	0.99	22.45	247.51
MPEG	0.45	2.40	197.51
QRD	0.42	4.23	298.72
STAP	0.65	6.94	179.07
RENDER	0.76	5.43	161.23

TABLE 5.5 Application Demands on a Stream Processor with Infinite Data Bandwidth

of memory references to global register file references to local register file references in the STAP application is roughly 1:11:276. Across all of the applications, the demands on the SRF are 5 to 23 times higher than the demands on the memory system, and the demands on the local register files are 10 to 80 times higher than the demands on the SRF. Therefore, the applications would clearly benefit from a bandwidth scaling that increases by roughly an order of magnitude at each level of the hierarchy.

Figure 5.9 shows the sensitivity of four of the applications to the size of the SRF. The figure shows the computation rate of each of the applications when the SRF and memory system can provide infinite instantaneous bandwidth. As can be seen in the figure, an SRF size of 20KW (80KB) is sufficient for the applications to achieve their peak rate for the problem sizes given in Section 3.2. MPEG benefits slightly from more SRF space and is able to increase performance by 2% with a 36KW (144KB) SRF.

The performance of each application plateaus as the size of the SRF is increased because the SRF becomes large enough to hold the applications' working sets. As the SRF size is increased, so is the length of streams that can be processed by the applications' kernels. Each kernel has some overhead associated with starting and stopping the kernel. This overhead includes the time it takes the stream controller to initiate the kernel and set up its streams in the SRF, the code outside of kernel loops that is executed once each time a kernel is called, and the time spent priming and draining software pipelined loops within the kernels. However, all of these applications have a natural maximum useful stream length based on the input data. For instance, DEPTH

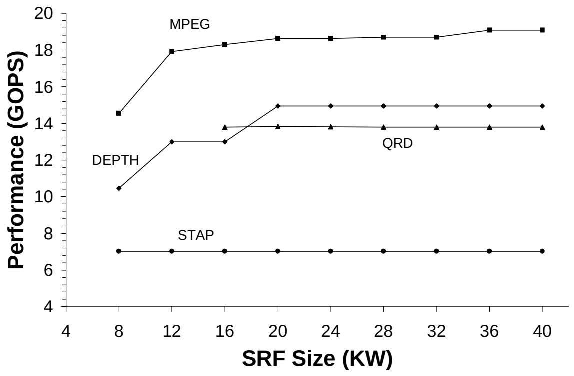


FIGURE 5.9 Computation Rate with Unlimited Bandwidth vs. SRF Size

and MPEG only benefit from operating on an entire row of their input images, as their kernels cannot operate on more than one row at a time. Beyond that, they cannot take advantage of longer streams. The graph shows that for the size of the data sets in these experiments, a 20KW (80KB) SRF allows the maximum useful stream sizes for all of the applications, except for MPEG. Larger problem sizes would benefit from larger SRF sizes. However, the performance of these applications would always plateau after the SRF becomes large enough to hold streams of the natural length for each application. The RENDER application is not shown in Figure 5.9 because the polygon renderer requires support from the stream scheduler which is not currently available to correctly operate at different SRF sizes.

5.4.4 Bandwidth Sensitivity

The bandwidth hierarchy is not effective if the SRF does not provide significantly more bandwidth than the memory system. By scaling the provided bandwidth across the levels of the hierarchy, however, a stream processor is able to sustain a significant fraction of the performance of media processing applications when they are provided with infinite bandwidth.

Figure 5.10 shows the performance of the media applications for a 32KW SRF with a peak bandwidth varying from 4 to 64GB/s. The 4GB/s SRF case corresponds roughly to forcing the applications to rely solely on the memory system. With a 4GB/s SRF, the memory system consumes from 0.5 to 1GB/s of that bandwidth to transfer streams to/from the external DRAM, leaving about 3GB/s for transferring data to and from

Application	Arithmetic Unit Utilization (%)	Performance (GOPS)
DEPTH	33.4	13.09
MPEG	36.8	17.88
QRD	60.7	12.49
STAP	26.6	5.47
RENDER	17.9	3.95

TABLE 5.6 Application Arithmetic Utilization and Performance on Imagine

the arithmetic clusters. This is slightly more bandwidth than would be available from the 2GB/s memory system if the SRF were not part of the bandwidth hierarchy but is low enough to show the degradation in performance without an SRF that provides significantly increased bandwidth over the memory system.

As can be seen in the figure, the sensitivity to the bandwidth of the SRF varies by application. DEPTH shows the most sensitivity, with a performance of over 5.6 times higher with a 64GB/s SRF compared to a 4GB/s SRF. This is not surprising, given that DEPTH is the application that demands the most SRF bandwidth at 22.45GB/s. To sustain over 20GB/s across the entire application, the SRF must have a significantly higher peak bandwidth. The other applications show more modest improvements, ranging from 20% to 60%. These applications achieve most of their peak performance with an SRF that can provide 16GB/s, which is still eight times higher than the peak memory bandwidth. Finally, the gap between the unlimited bandwidth performance and the actual achieved performance of all of the applications, except DEPTH, is due to memory bandwidth limitations. While DEPTH is able to overlap almost all of its stream memory transfers with computation, dependencies prohibit the other applications from hiding all of their memory transfers. Therefore, while the memory system is able to sustain a significant fraction of the average demand from these applications, it is not always able to provide the applications with their peak bandwidth demands.

5.4.5 Arithmetic Unit Utilization

Table 5.6 shows the utilization of the arithmetic units on Imagine and the resulting application performance. The various applications are able to utilize the arithmetic units between 18% and 61%. This is lower than the 77% utilization that is achieved in

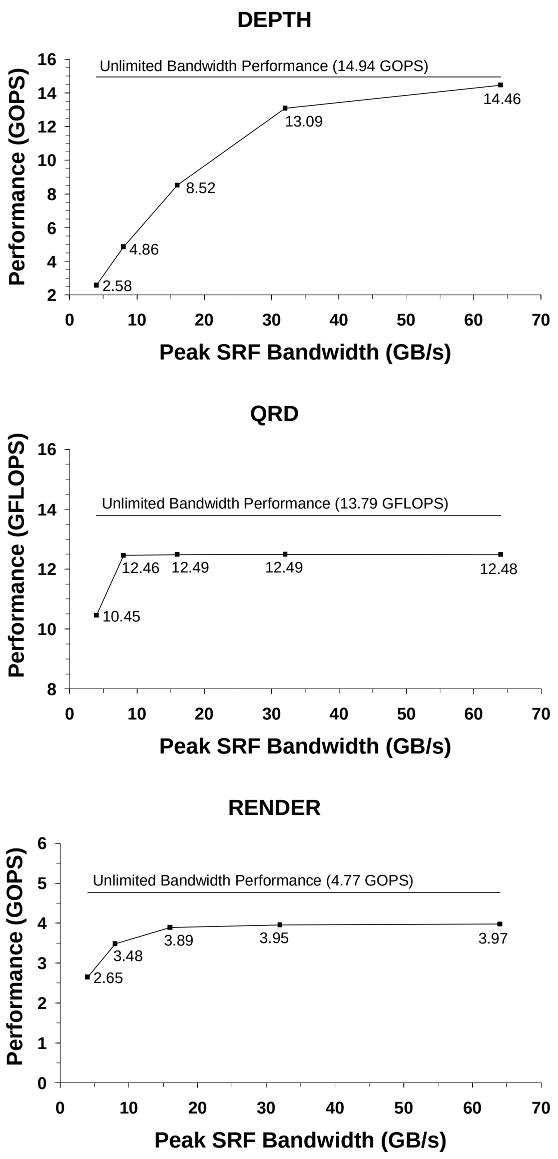


FIGURE 5.10 Sustained Performance vs. Bandwidth of a 32KW SRF

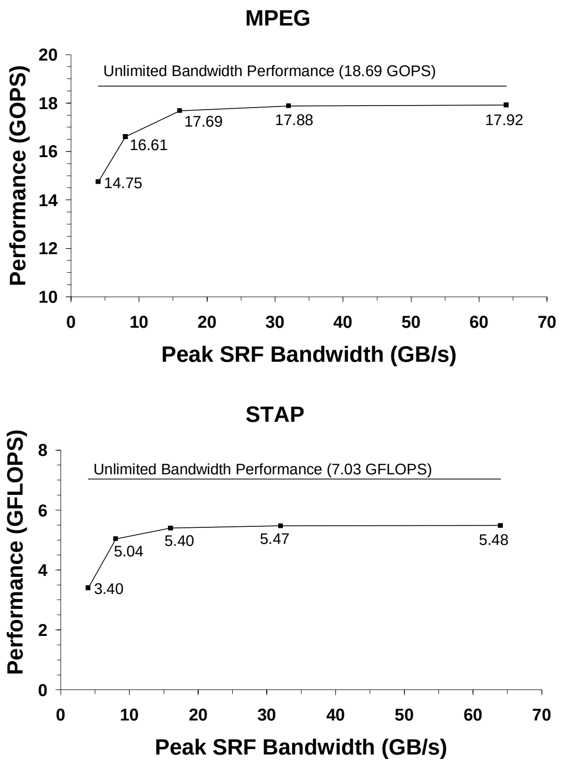


FIGURE 5.10 Sustained Performance vs. Bandwidth of a 32KW SRF

the inner loop of the FIR filter kernel. During the execution of an application, the arithmetic units are left idle when stalled waiting for data from memory of the SRF, and there is far less computation outside of the inner loops of kernels. These factors greatly decrease the achievable arithmetic utilization of entire applications. However, because the data bandwidth hierarchy enables Imagine to support 48 arithmetic units with very high utilization during the inner loops of kernels, applications are still able to achieve from 4 to 18GOPS of sustained computation.

5.5 Summary

Media processing applications benefit from a bandwidth scaling across multiple levels of the storage hierarchy to bridge the gap from modern DRAM bandwidth to the data bandwidth required by the arithmetic units. Without this bandwidth scaling, media applications are memory bandwidth limited, which severely reduces the number of arithmetic units that can be utilized efficiently. A three-tiered bandwidth hierarchy, including distributed local register files, a global stream register file, and external DRAM, can effectively meet the needs of these applications by scaling the bandwidth by over an order of magnitude at each level.

The bandwidth hierarchy enables media processing applications to utilize bandwidth efficiently. Kernels make efficient use of local register files for temporary storage to supply the arithmetic units with data and of an SRF to supply input streams to the arithmetic units and store output streams from the arithmetic units. When kernels are considered as part of an application, rather than in isolation, the SRF can be efficiently used to capture the locality of stream recirculation within media processing applications, thereby limiting the bandwidth demands on off-chip memory. When mapped to the three-tiered storage hierarchy described in this chapter, media processing applications demand successively increasing amounts of bandwidth. By matching these demands, the bandwidth scaling of the storage hierarchy enables these media processing applications to achieve 4-18GOPS, which is 77-96% of the performance that they would sustain given infinite instantaneous data bandwidth.

CHAPTER 6

Memory Access Scheduling

A streaming memory system, which forms the base of a stream processor's data bandwidth hierarchy, transfers entire streams of data, rather than individual words. Sustained memory bandwidth directly contributes to the overall performance of a stream processor, so a streaming memory system is optimized for memory bandwidth, rather than memory latency. To maximize bandwidth, the memory system must take advantage of the characteristics of modern DRAM. The bandwidth and latency of a memory system are strongly dependent on the manner in which accesses interact with the "3-D" structure of banks, rows, and columns characteristic of contemporary DRAM chips. Modern DRAM components allow memory accesses to be pipelined, provide several independent memory banks, and cache the most recently accessed row of each bank. These features increase the peak performance of the DRAM, but also make the sustained performance highly dependent on the access pattern. There is nearly an order of magnitude difference in bandwidth between successive references to different columns within a row and different rows within a bank. Memory access scheduling improves the performance of a memory system by reordering memory references to exploit locality within the 3-D memory structure. This chapter will discuss the organization of modern DRAM, the architecture of a memory access scheduler, and their impact on sustained memory bandwidth for media processing.

6.1 Overview

As discussed previously, media applications require up to 300GB/s of data bandwidth to utilize 48 arithmetic units. An effective bandwidth hierarchy can bridge the gap between modern DRAM bandwidth and the bandwidth required by the arithmetic units, which is over two orders of magnitude. However, for such a hierarchy to be successful, the memory system must be able to sustain a significant fraction of the peak DRAM bandwidth. Chapter 5 showed that the five representative media processing applications demand from 0.4 to 1.0GB/s of memory bandwidth to achieve their peak computation rates. On the Imagine stream processor, this corresponds to 20-50% of the peak DRAM bandwidth. This can be achieved with a streaming memory system which employs memory access scheduling to take advantage of the locality and concurrency of modern DRAM.

6.1.1 Stream Memory Transfers

A streaming memory system transfers streams of data between the processor and external DRAM, rather than individual words or cache lines as in a conventional memory system. A data stream is a sequence of records, where each record consists of one or more words. Records represent media data, such as an audio sample, a complex frequency value, or a triangle vertex. The words that compose an individual record are stored contiguously in memory, but a stream may consist of records scattered throughout the memory space. The starting addresses of the records that compose a particular data stream can be calculated using a constant stride, bit-reversed arithmetic, or an index stream. The streaming memory system of the Imagine media processor was described in Section 4.2.4.

Since entire streams of data are transferred by the memory system, the latency of any individual reference is not important. Rather, the bandwidth of the transfer is what affects performance. For example, consider the execution of the MPEG2 I-frame encoder on Imagine. Figure 6.1 shows the kernel operations and memory stream transfers for a portion of the application's execution. The figure shows parts of the processing of three batches of macroblocks. The left column shows kernel execution on the arithmetic clusters and the two columns on the right show stream memory transfers. The load from memory of the macroblocks for batch *i-1* occurs prior to cycle 6000, so is not shown. First the color conversion kernel is executed to convert the original macroblocks into luminance and chrominance streams. Then the DCT, run-level encoding, and inverse DCT execute. The run-level encoded blocks are stored to memory after they are encoded. The kernel labelled "reference" in the figure simply reorganizes data from the IDCT kernels to be in the proper format for use in P-

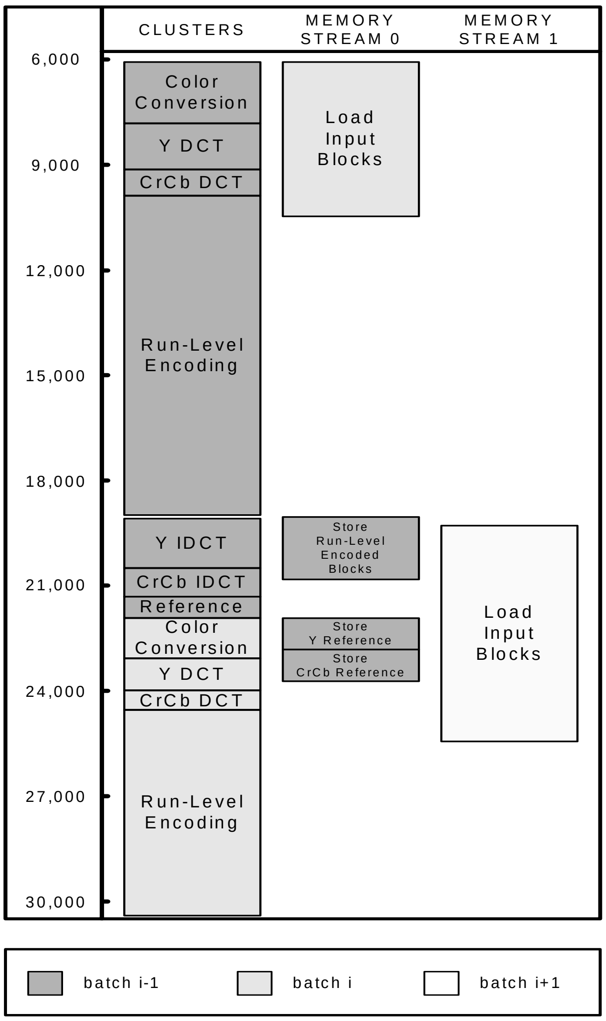


FIGURE 6.1 Excerpt of Execution of an MPEG2 I-frame Encoder

and B-frame encoding. After the reference kernel executes, the reference luminance and chrominance images are stored to memory as well. The macroblock stream load for batch *i* is shown in the figure, as are the first four kernels.

As can be seen in the figure, the memory stream transfers are overlapped with computation kernels that are processing other data streams. Notice that while the processing for batch *i-1* is occurring, the macroblocks for batch *i* are simultaneously loaded into the SRF. Similarly, the results of processing batch *i-1* are stored concurrently with both processing and the macroblock load for batch *i+1*.

Since the SRF isolates the arithmetic kernels from memory stream transfers, batches of macroblocks may be loaded into the SRF while arithmetic computations are being performed on streams that are already in the SRF. Because a stream processor can overlap memory stream transfers with kernel execution, applications can tolerate large amounts of memory latency. Therefore, the streaming memory system of such a processor can be optimized to maximize bandwidth, rather than latency, in order to improve media processing performance.

6.1.2 Memory Access Scheduling

To maximize memory bandwidth, modern DRAM components allow pipelining of memory accesses, provide several independent memory banks, and cache the most recently accessed row of each bank. While these features increase the peak supplied memory bandwidth, they also make the performance of the DRAM highly dependent on the access pattern. Modern DRAMs are not truly random access devices (equal access time to all locations) but rather are three-dimensional memory devices with dimensions of bank, row, and column. Sequential accesses to different rows within one bank have high latency and cannot be pipelined, while accesses to different banks or different words within a single row have low latency and can be pipelined.

The three-dimensional nature of modern memory devices makes it advantageous to reorder memory operations to exploit the non-uniform access times of the DRAM. This optimization is similar to how a superscalar processor schedules arithmetic operations out of order. As with a superscalar processor, the semantics of sequential execution are preserved by reordering the results.

Memory access scheduling reorders DRAM operations, possibly completing memory references out of order, to maximize sustained memory bandwidth. Figure 6.2 illustrates the benefits of memory access scheduling. The figure shows eight memory references scheduled to access external DRAM in three different ways. Each reference is denoted by its bank, row, and column address for clarity. The references are listed in the order that they arrived at the memory controller from top to bottom.

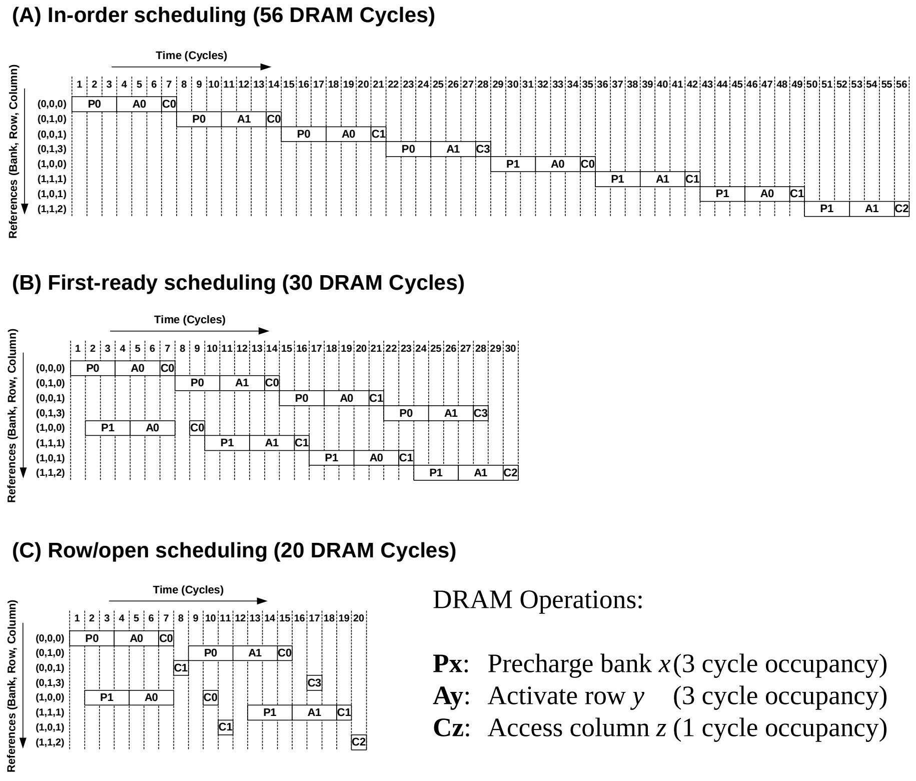


FIGURE 6.2 Memory Access Schedules with Different Scheduling Algorithms

Figure 6.2(A) shows the DRAM operations that would be required to satisfy these eight references in order. Modern systems commonly satisfy memory references in this manner, possibly with slight variations such as automatically precharging the bank when a cache line fetch is completed. Since no two subsequent references in the example target the same row of the same bank, every reference must precharge a bank, activate a row, and perform a column access. For a DRAM with a three cycle bank precharge latency, a three cycle row activation latency, and a single cycle column access latency, these eight references require 56 DRAM cycles to complete in order.

Using a first-ready access scheduling algorithm, the same eight references can be completed sooner, as shown in Figure 6.2(B). The first-ready scheduler considers all pending references and schedules a DRAM operation for the oldest pending reference that does not violate the timing and resource constraints of the DRAM. The most obvious benefit of this scheduling algorithm over the in-order scheduler is that accesses to other banks can be made while waiting for a precharge or an activate operation to complete for the oldest pending reference. This relaxes the serialization of the in-order scheduler and allows multiple references to progress in parallel. In this example, the first-ready scheduler is able to exploit the bank parallelism in the reference stream. The accesses for the references that target bank 1 can mostly be overlapped with accesses for the references that target bank 0. The first-ready scheduler is able to complete the eight references in only 30 cycles. It is not always the case that first-ready scheduling exploits all of the available bank parallelism and none of the row locality. If the same eight references had arrived in a different order, a first-ready scheduler would be able to exploit varying degrees of each.

More sophisticated scheduling algorithms can maximize DRAM bandwidth by exploiting more of the bank parallelism and row locality found in the pending memory references. Figure 6.2(C) shows the DRAM accesses performed by a row/open scheduler, which will be explained later in the chapter. As can be seen in the figure, a row/open scheduler completes the eight references in only 20 DRAM cycles. This scheduler is able to exploit both the bank parallelism and row locality of the reference stream by overlapping accesses to banks 0 and 1 and by performing all of the column accesses to each row that is activated before precharging the bank for subsequent row activations.

6.2 Modern DRAM

The previous example illustrates that the order in which DRAM accesses are scheduled can have a dramatic impact on memory throughput and latency. To maximize memory bandwidth, modern DRAM components allow pipelining of memory accesses, provide several independent memory banks, and cache the most recently accessed row of each bank. While these features increase the peak supplied memory bandwidth, they also make the performance of the DRAM highly dependent on the access pattern. Therefore, a memory controller must take advantage of the characteristics of modern DRAM to improve performance.

Figure 6.3 shows the internal organization of modern DRAMs. These DRAMs are three-dimensional memories with the dimensions of bank, row, and column. Each bank operates independently of the other banks and contains an array of memory cells

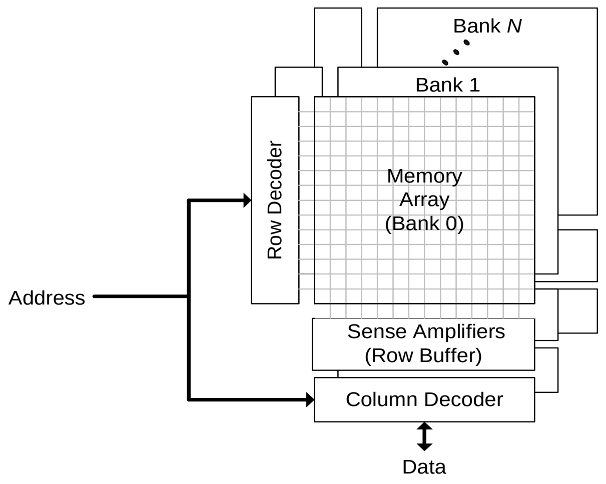
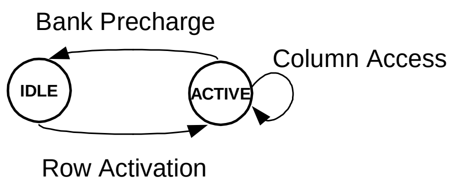


FIGURE 6.3 Modern DRAM Organization

that are accessed an entire row at a time. When a row of this memory array is accessed (*row activation*), the entire row of the memory array is transferred into the bank's row buffer. The row buffer serves as a cache to reduce the latency of subsequent accesses to that row. While a row is active in the row buffer, any number of reads or writes (*column accesses*) may be performed, typically with a throughput of one per cycle. After completing the available column accesses, the cached row must be written back to the memory array by an explicit operation (*bank precharge*), which prepares the bank for a subsequent row activation. An overview of several different modern DRAM types and organizations, along with a performance comparison for in-order access, can be found in [CJDM99].

For example, the 128Mb NEC µPD45128163 [NEC98], a typical SDRAM, includes four internal memory banks, each composed of 4096 rows and 512 columns. This SDRAM may be operated at 125MHz, with a precharge latency of 3 cycles (24ns) and a row access latency of 3 cycles (24ns). Pipelined column accesses that transfer 16 bits may issue at the rate of one per cycle (8ns), yielding a peak transfer rate of 250MB/s. However, it is difficult to achieve this rate on non-sequential access patterns for several reasons. A bank cannot be accessed during its precharge/activate latency, a single cycle of high impedance is required on the data pins when switching between read and write column accesses, and a single set of address lines is shared by all DRAM operations (bank precharge, row activation, and column access). The

(A) Simplified bank state diagram



(B) Operation resource utilization

	Cycle	1	2	3	4
Precharge:	Bank	■	■	■	□
	Address	■	□	□	□
	Data	□	□	□	□
Activate:	Bank	■	■	■	□
	Address	■	□	□	□
	Data	□	□	□	□
Read:	Bank	□	□	□	□
	Address	■	□	□	□
	Data	□	□	□	■
Write:	Bank	□	□	□	□
	Address	■	□	□	□
	Data	■	□	□	□

FIGURE 6.4 Internal DRAM Bank State Diagram and Resource Utilization

amount of bank parallelism that is exploited and the number of column accesses that are made per row access dictate the sustainable memory bandwidth out of such a DRAM.

A memory controller must generate a schedule that conforms to the timing and resource constraints of these modern DRAMs. Figure 6.4 illustrates these constraints for the NEC SDRAM with a simplified bank state diagram and a table of operation resource utilization. Each DRAM operation makes different demands on the three DRAM resources (the internal banks, a single set of address lines, and a single set of data lines). The memory controller must ensure that the required resources are available for each DRAM operation it issues.

Each DRAM bank has two stable states, IDLE and ACTIVE, as shown in Figure 6.4(A). In the IDLE state, the DRAM is precharged and ready for a row access. It will remain in this state until a row activation operation is issued to the bank. To issue a row activation, the address lines must be used to select the bank and the row being activated, as shown in Figure 6.4(B). Row activation requires 3 cycles, during which no other operations may be issued to that bank, as indicated by the utilization of the *bank* resource for the duration of the operation. During that time, however, operations may be issued to other banks of the DRAM. Once the DRAM's row activation latency has passed, the bank enters the ACTIVE state, during which the contents of the selected row are held in the bank's row buffer. Any number of pipelined column accesses may be performed while the bank is in the ACTIVE state. To issue either a read or write column access, the address lines are required to indicate the bank and the column of the active row in that bank. A write column access requires the data to be transferred to the DRAM at the time of issue, whereas a read column access returns the requested data three cycles later. Additional timing constraints not shown in Figure 6.4, such as a required cycle of high impedance between reads and writes, may further restrict the use of the data pins.

The bank will remain in the ACTIVE state until a precharge operation is issued to return it to the IDLE state. The precharge operation requires the use of the address lines to indicate the bank which is to be precharged. Like row activation, the precharge operation utilizes the *bank* resource for 3 cycles, during which no new operations may be issued to that bank. Again, operations may be issued to other banks during this time. After the DRAM's precharge latency, the bank is returned to the IDLE state and is ready for a new row activation operation. Frequently, there are also timing constraints that govern the minimum latency between a column access and a subsequent precharge operation. DRAMs typically also support column accesses with automatic precharge, which implicitly precharges the DRAM bank as soon as possible after the column access.

The address and data resources are usually shared among the banks, thereby serializing access to the internal banks.While the state machines for the individual banks are independent, only a single bank can perform a transition requiring a particular shared resource each cycle. For many DRAMs, the bank, row, and column addresses share a single set of pins. Hence, the scheduler must arbitrate between precharge, row, and column operations that all need to use this single resource. Other DRAMs, such as Direct Rambus DRAMs (DRDRAMs) [Cri97], provide separate row and column address lines (each with their own associated bank address) so that column and row accesses can be initiated simultaneously. To approach the peak data rate with serialized resources, there must be enough column accesses to each row to hide the precharge/activate latencies of other banks. Whether or not this can be achieved is dependent on the data reference patterns and the order in which the DRAM is accessed to satisfy those references. The need to hide the precharge/activate latency of the banks in order to sustain high bandwidth cannot be eliminated by any DRAM architecture without reducing the precharge/activate latency, which would likely come at the cost of decreased bandwidth or capacity, both of which are undesirable.

6.3 Memory Access Scheduling

The three-dimensional nature of modern memory devices makes it advantageous to reorder memory operations to exploit the non-uniform access times of the DRAM. However, reordering memory operations can potentially increase the latency of some references. References can be delayed in order to allow other references to access the DRAM, improving overall performance. As mentioned earlier, the streaming nature of media processing applications makes them less sensitive to memory latency than memory throughput. Also, the stream register file of a stream processor further isolates the arithmetic units from the effects of memory latency, so the latency of any individual reference will not affect overall performance. Furthermore, a good reordering will actually reduce the average latency of memory references because of the differences in throughput and latency of the DRAM based on the reference pattern. Therefore, a streaming media processor will benefit greatly from reordering memory operations to achieve the maximum throughput out of modern DRAM.

Memory access scheduling is the process of ordering the DRAM operations (bank precharge, row activation, and column access) necessary to complete the set of currently pending memory references. Used here, the term *operation* denotes a command, such as a row activation or a column access, issued by the memory controller to the DRAM. Similarly, the term *reference* denotes a memory reference generated by the processor, such as a load or store to a memory location. A single reference generates one or more memory operations depending on the schedule.

6.3.1 Architecture

Given a set of pending memory references, a memory access scheduler may choose one or more row, column, or precharge operations each cycle, subject to resource constraints, to advance one or more of the pending references. The simplest, and most common, scheduling algorithm only considers the oldest pending reference, so that references are satisfied in the order that they arrive. If it is currently possible to make progress on that reference by performing some DRAM operation, then the memory controller performs the operation. While this does not require a complicated access scheduler in the memory controller, it is clearly inefficient.

If the DRAM is not ready for the operation required by the oldest pending reference, or if that operation would leave available resources idle, it makes sense to consider operations for other pending references. Figure 6.5 shows the structure of a more sophisticated access scheduler. As memory references arrive, they are allocated storage space while they await service from the memory access scheduler. In the figure,

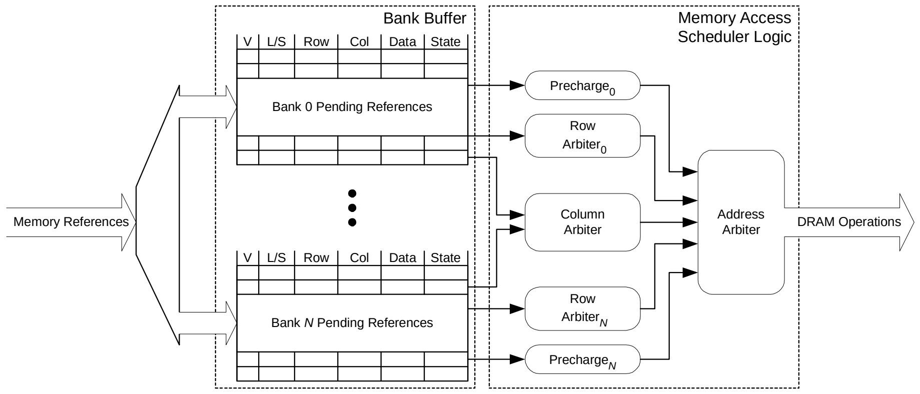


FIGURE 6.5 Memory Access Scheduler Architecture

references are initially sorted by internal DRAM bank. Each pending reference is represented by the following six fields: valid (V), load/store (L/S), address (Row and Col), data, and whatever additional state is necessary for the scheduling algorithm. Examples of state that can be accessed and modified by the scheduler are the age of the reference and whether or not that reference targets the currently active row. In practice, the pending reference storage could be shared by all the banks (with the addition of a bank address field) to allow dynamic allocation of that storage at the cost of increased logic complexity in the scheduler.

Each internal DRAM bank has an associated precharge manager and row arbiter. The precharge manager decides when its associated bank should be precharged. Similarly, the row arbiter for each bank decides which row, if any, should be activated when that bank is idle. Each set of data and address pins on the DRAM has a corresponding column arbiter and address arbiter within the memory access scheduler. These arbiters are shared by all of the banks. All current DRAMs have a single set of data pins, so there is a single column arbiter that grants the shared data line resources to a single column access out of all the pending references to all of the banks. Finally, the precharge managers, row arbiters, and column arbiter send their selected operations to the address arbiter(s), which grant the shared address resources to one or more of those operations.

The memory access scheduler hardware, shown in Figure 6.5, is incorporated into the memory controller of the processor. Since the processor clock is likely to be faster than the DRAM clock, it is possible for many new references to enter the bank buffer each DRAM cycle. On each DRAM cycle, all of the arbiters update their decisions, and the address arbiter makes a final decision as to what DRAM operation to perform, if any. These decisions may or may not include the most recently arrived references, depending on the degree of pipelining within the memory access scheduler.

6.3.2 Algorithms

The precharge managers, row arbiters, and column arbiter can use several different policies to select DRAM operations. The combination of policies used by these arbiters determine the memory access scheduling algorithm. As described in the previous section, the precharge managers must decide whether or not their associated banks should be precharged. The row arbiters must decide which row to activate in a precharged bank, and the column arbiter must decide which column operation from which bank should be granted access to the data line resources. Finally, the address arbiter must decide which of the selected precharge, activate, and column operations to perform subject to the constraints of the address line resources. If the address resources are not shared, it is possible for both a precharge operation and a column access to the same bank to be selected. This is likely to violate the timing constraints of the DRAM. Ideally, this conflict can be handled by having the column access automatically precharge the bank upon completion, which is supported by most modern SDRAMs. This section describes the various policies that can be used by the arbiters of a memory access scheduler.

In-order. This policy applies to any of the arbiters. A DRAM operation will only be performed if it is required by the oldest pending reference. While used by almost all memory controllers today, this policy yields poor performance compared to policies that look ahead in the reference stream to better utilize DRAM resources.

Priority. This policy applies to any of the arbiters. The operation(s) required by the highest priority ready reference(s) are performed. Three possible priority schemes are as follows: *ordered*, older references are given higher priority; *age-threshold*, references older than some threshold age gain increased priority; and *load-over-store*, load references are given higher priority. These priority schemes can also be combined to form more sophisticated schemes. Age-threshold prevents starvation while allowing greater reordering flexibility than ordered. Load-over-store decreases load latency to minimize processor stalling on stream loads.

Open. This policy applies to the precharge managers. A bank is only precharged if some pending references target other rows in the bank and no pending references target the active row. The open policy should be employed if there is significant row locality, making it likely that future references will target the same row as previous references did.

Closed. This policy applies to the precharge managers. A bank is precharged as soon as no more pending references target the active row. The closed policy should be employed if it is unlikely that future references will target the same row as the previous set of references.

Most-pending. This policy applies to the row or column arbiters. The row or column access to the row with the most pending references is selected. This allows rows to be activated that will have the highest ratio of column to row accesses, while waiting for other rows to accumulate more pending references. By selecting the column access to the most demanded row, that bank will be freed up as soon as possible to allow other references to make progress. This policy can be augmented by one of the priority schemes described above to prevent starvation.

Fewest-pending. This policy applies to the column arbiter. The fewest pending policy selects the column access to the row targeted by the fewest pending references. This minimizes the time that rows with little demand remain active, allowing references to other rows in that bank to make progress sooner. A weighted combination of the fewest pending and most pending policies could also be used to select a column access. This policy can also be augmented by one of the priority schemes described above to prevent starvation.

Precharge-first. This policy applies to the address arbiter. The precharge-first policy always selects a precharge operation to issue to the DRAM if any are available. It would then select a row activation operation, if any are available, and then finally select a column operation to send to the DRAM. This policy can increase the amount of bank parallelism that is exploited by initiating precharge/row operations first and then overlapping the latency of those operations with available column accesses. This policy can also be augmented by one of the priority schemes described above to select which operation to perform if more than one is available.

Column-first. This policy applies to the address arbiter. The column-first policy always selects a column operation to issue to the DRAM if any are available. It would select a precharge or row operation if no column operation is available. This policy can reduce the latency of pending references that target active rows. One of the priority schemes can be used to determine which precharge or row operation would be selected if more than one is available.

Row-first. This policy applies to the address arbiter. It is nearly identical to the precharge-first policy, except that instead of always favoring precharge operations, it favors precharge and row activation operations equally. This policy is a compromise between the precharge-first and column-first policies, as it increases the amount of bank parallelism that is exploited by overlapping column accesses with the precharge/activate latency of the DRAM, but it also attempts to reduce the latency of references by activating rows targeted by higher priority references quickly. One of the priority schemes described above can be used to determine which precharge or row activation to select. Column accesses will be selected only if no precharge or row activation operations are available.

6.4 Evaluation

Memory access scheduling exploits locality and concurrency in modern DRAM components. However, some access patterns, such as unit stride stream accesses, already interact well with the internal DRAM structure. If these patterns are common in applications, then scheduling memory accesses will have little effect. In practice, media processing applications contain a mix of reference patterns, some of which interact well with the DRAM's structure, and some of which do not.

This section will evaluate the effect of memory access scheduling on the sustained memory bandwidth of the Imagine stream processor. First, Section 6.4.1 will show how different types of memory stream accesses interact with modern DRAM. Five distinct microbenchmarks will be used to show the sustained memory bandwidth for different types of stream accesses with and without memory access scheduling. Memory access scheduling can improve the sustained bandwidth of these microbenchmarks by 144%. Second, Section 6.4.2 will examine the memory behavior found in media processing applications, giving an indication of the potential gains of memory access scheduling. Finally, Section 6.4.3 will show that memory access scheduling is able to improve the sustained memory bandwidth for media processing applications by 8%, making several of the applications compute-bound. Furthermore, memory access scheduling increases the sustained bandwidth achievable for the applications' memory traces by 51%.

6.4.1 Microbenchmarks

Table 6.1 shows five microbenchmarks that will be used to understand the effects of memory access scheduling. For these microbenchmarks, no computations are performed outside of the address generators. This allows memory references to be issued at their maximum throughput, constrained only by the buffer storage in the memory banks. All of the microbenchmarks perform ten stream transfers, each of which contain 512 references.

Name	Description
Unit Load	Unit stride load stream accesses with parallel streams to different rows in different internal DRAM banks.
Unit	Unit stride load and store stream accesses with parallel streams to different rows in different internal DRAM banks.
Unit Conflict	Unit stride load and store stream accesses with parallel streams to different rows in the same internal DRAM banks.
Constrained Random	Random access load and store streams constrained to a 64KB range.
Random	Random access load and store streams to the entire address space.

TABLE 6.1 Microbenchmarks

A memory controller that performs no access reordering will serve as a basis for comparison. This controller uses an in-order policy, described in Section 6.3.2, for all decisions. A column access will only be performed for the oldest pending reference, a bank will only be precharged if necessary for the oldest pending reference, and a row will only be activated if it is needed by the oldest pending reference. No other references are considered in the scheduling decision. This algorithm, or slight variations, such as automatically precharging the bank when a cache line fetch is completed, can commonly be found in systems today.

The gray bars in the graph in Figure 6.6 show the performance of the benchmarks using the baseline in-order access scheduler. *Unit load* performs very well with no access scheduling, achieving 97% of the peak bandwidth (2GB/s) of the DRAMs. Almost all of the references in the *unit load* benchmark access rows in the DRAM that are active. The 3% overhead is the combined result of infrequent precharge/activate cycles and the start-up/shutdown delays of the streaming memory system.

The 14% drop in sustained bandwidth from the *unit load* benchmark to the *unit* benchmark shows the performance degradation imposed by forcing intermixed load and store references to complete in order. Each time the references switch between loads and stores, a cycle of high impedance must be left on the data pins, decreasing the sustainable bandwidth. The *unit conflict* benchmark further shows the penalty of swapping back and forth between rows in the DRAM banks, which drops the sustainable bandwidth down to 51% of the peak. The random benchmarks sustain about 15% of the bandwidth of the *unit load* benchmark. This loss roughly corresponds to the

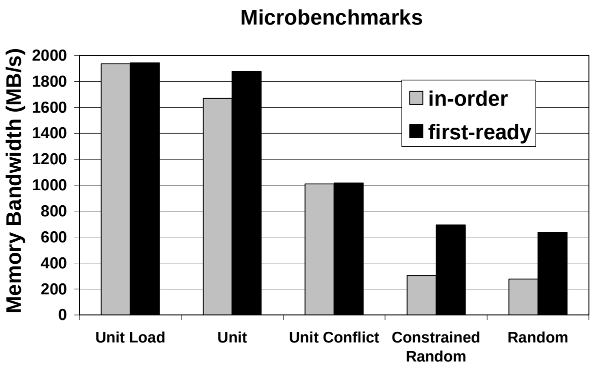


FIGURE 6.6 Microbenchmark Performance using In-order and First-ready Schedulers

degradation incurred by performing accesses with a throughput of one word every seven DRAM cycles (the random access throughput of the SDRAM) compared to a throughput of one word every DRAM cycle (the column access throughput of the SDRAM).

The use of a very simple first-ready access scheduler improves the performance of the microbenchmarks by 0-132%. First-ready scheduling uses the ordered priority scheme, as described in Section 6.3.2, to make all scheduling decisions. The first-ready scheduler considers all pending references and schedules a DRAM operation for the oldest pending reference that does not violate the timing and resource constraints of the DRAM. The most obvious benefit of this scheduling algorithm over the baseline is that accesses targeting other banks can be made while waiting for a precharge or activate operation to complete for the oldest pending reference. This relaxes the serialization of the in-order scheduler and allows multiple references to progress in parallel.

The black bars in the graph in Figure 6.6 shows the sustained bandwidth of the first-ready scheduling algorithm for each benchmark. *Unit load* shows no improvement as it already sustains almost all of the peak SDRAM bandwidth, and the random benchmarks show an improvement of over 125%, as they are able to increase the number of column accesses per row activation significantly.

When the oldest pending reference targets a different row than the active row in a particular bank, the first-ready scheduler will precharge that bank even if it still has pend-

Algorithm	Column Access	Precharging	Row Activation	Access Selection
col/open	priority (ordered)	open	priority (ordered)	column first
col/closed	priority (ordered)	closed	priority (ordered)	column first
row/open	priority (ordered)	open	priority (ordered)	row first
row/closed	priority (ordered)	closed	priority (ordered)	row first

TABLE 6.2 Memory Access Scheduling Algorithms

ing references to its active row. More aggressive scheduling algorithms are required to further improve performance. Table 6.2 presents four aggressive scheduling algorithms that will further increase sustained memory bandwidth. The policies for each of the schedulers in Table 6.2 are described in Section 6.3.2. The range of possible memory access schedulers is quite large, and covering all of the schedulers examined in Section 6.3 would be prohibitive. These four schedulers, however, are representative of many of the important characteristics of an aggressive memory access scheduler.

Figure 6.7 presents the microbenchmarks' sustained memory bandwidth for each memory access scheduling algorithm. The aggressive scheduling algorithms significantly improve the memory bandwidth of the microbenchmarks over in-order scheduling. Again the *unit load* benchmark shows no improvement. The *unit conflict* benchmark, however, improves by 115% with more aggressive scheduling. All of the memory access scheduling algorithms considered are able to reorder accesses to the DRAM, so that references to conflicting banks to not interfere with each other in the DRAM. The random benchmarks improve by 125-250% with the more aggressive scheduling algorithms. They favor a closed precharge policy, in which banks are precharged as soon as no more pending references target their active row, because it is unlikely that there will be any reference locality that would make it beneficial to keep the row open. By precharging as soon as possible, the access latency of future references is minimized. For most of the other benchmarks, the difference between an open and a closed precharge policy is slight. *Unit load* is a notable exception, as it performs worse with the col/closed algorithm. In this case, column accesses are satis-

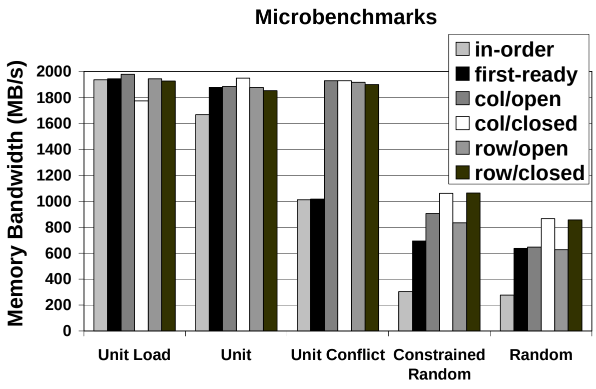


FIGURE 6.7 Microbenchmark Performance Using Aggressive Schedulers

fied rapidly, emptying the bank buffer of references to a stream, allowing the banks to be precharged prematurely in some instances.

6.4.2 Memory Demands

The memory performance of the microbenchmarks gives an indication of how much bandwidth an application will be able to sustain, given that application's memory behavior. Table 6.3 shows the number and length of stream memory transfers in the five applications presented in Chapter 3. The table shows that the applications make from 114 thousand to 2.3 million memory references over the course of their execution. These references are part of over 25 thousand total stream transfers. The bandwidth demands on the memory system, presented previously in Section 5.4.3, are also shown in the table. These applications demand from 0.4 to 1.0GB/s of memory bandwidth to achieve their peak performance on Imagine.

The table shows that the vast majority of stream transfers use constant strides. While many of these strided transfers use a unit stride, especially in DEPTH and MPEG, many do not. For example, QRD and STAP frequently access the columns of matrices, using very large constant strides. The only applications that perform indirect stream transfers are MPEG and RENDER. In both of these applications, those accesses roughly correspond to the *constrained random* microbenchmark presented in the previous section. MPEG performs indexed loads into the reference images during the motion estimation phase of P-frame encoding. As the motion in images is somewhat random, these references will also be somewhat random, but they will be constrained to lie within the region of the reference image that was searched. RENDER performs indexed

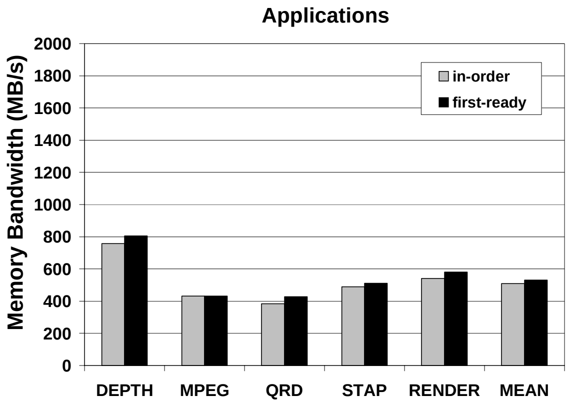
		DEPTH	MPEG	QRD	STAP	RENDER	*Total*
References (Thousands)		907.0	934.6	114.4	1900.6	2278.2	*6134.7*
Bandwidth Demand (GB/s)		0.99	0.45	0.42	0.65	0.76	*0.64*
Stride Stream Transfers	Loads	3131	83	19	3413	2630	*9276*
	Avg. Length	185.4	842.0	3984.8	452.4	150.8	*287.5*
	Stores	1440	324	78	2112	4050	*8004*
	Avg. Length	226.7	1360.3	495.6	168.7	256.0	*274.7*
Indirect Stream Transfers	Loads	0	252	0	0	4074	*4326*
	Avg. Length	---	1682.3	---	---	172.8	*260.7*
	Stores	0	0	0	0	3768	*3768*
	Avg. Length	---	---	---	---	37.4	*37.4*

TABLE 6.3 Number and Length (in Words) of Stream Memory Transfers by Type

loads and stores into the frame buffer and texture map. As pixels are rendered into arbitrary locations, these references will also be somewhat random but constrained to the frame buffer and texture map. There is some locality, in that adjacent pixels are likely to lie close to each other in the frame buffer and access texels that are close to each other as well. However, they may not necessarily reference pixels or texels that are in consecutive locations in memory, as the frame buffer and texture map are stored in row major order and the renderer may traverse those structures in arbitrary directions.

6.4.3 Application Performance

Application level experiments were performed both with and without the applications' computation. When running just the memory traces, dependencies from the application were maintained by assuming that the computation occurred at the appropriate times but was instantaneous. The applications show the performance improvements that can be gained by using memory access scheduling on the Imagine stream processor. The applications' memory traces, with instantaneous computation, show the potential of the scheduling techniques as processing power increases and the applications become entirely limited by memory bandwidth.



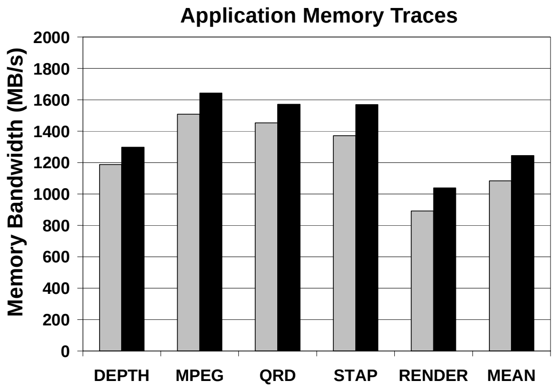


FIGURE 6.8 Application Performance Using In-order and First-ready Schedulers

Figure 6.8 shows the sustained memory bandwidth of the applications when using an in-order and first-ready memory access scheduler. The figure shows the memory performance of both the applications and the applications' memory traces. Even with in-order scheduling, the memory system sustains very high throughput on the applications' memory traces. The weighted mean of the traces' performance is just over 50% of the peak memory system performance. RENDER's memory trace, which includes the most indirect stream transfers, achieves the lowest sustained memory bandwidth at just over 890MB/s. The memory trace of MPEG, even though it also contains indirect memory transfers, is able to achieve the highest sustained bandwidth at over 1500MB/s. This results from the fact that the indirect references found in MPEG are far more constrained than those found in RENDER, so they exhibit much more row locality within the DRAMs.

When using an in-order scheduler, the applications are not always able to overlap memory and computation successfully. The processor is stalled waiting for memory an average of 21% of the time on these applications. However, these applications are compute-bound for large portions of their execution. During those times, there are no outstanding memory transfers, so the sustained memory bandwidth of the applications is far lower than the sustained bandwidth of the memory traces. Across the applications, the weighted mean of the sustained bandwidth for in-order scheduling is just above 500MB/s, which is 25% of the peak memory bandwidth. The in-order scheduler therefore provides 79% of the bandwidth demanded by the applications.

The improvements of first-ready scheduling are modest, both on the applications and the memory traces. First-ready scheduling increases the memory bandwidth of the applications by 4.5% and the memory traces by 14.8%. With a first-ready scheduler, the memory system supplies 82% of the bandwidth demanded by the applications.

Figure 6.9 shows the sustained memory bandwidth of the applications when using the aggressive memory access scheduling algorithms presented earlier in Table 6.2. Again, the figure shows the memory performance of both the applications and the applications' memory traces. The row/open scheduling algorithm performs best, improving the memory bandwidth of the applications by 8%, and the applications' memory traces by 51%. Some applications, such as MPEG, show very little improvement in performance with aggressive memory access scheduling. On the Imagine stream architecture, MPEG efficiently utilizes the SRF to capture locality. This makes MPEG compute-bound, so increasing the memory system bandwidth cannot improve performance. However, the performance of the memory trace of MPEG achieves 94% of the peak bandwidth of the memory system when memory access scheduling is employed.

All of the applications and traces slightly favor an address arbiter policy which selects row accesses first, rather than one that selects column accesses first. Also, algorithms that include an open page precharging policy outperform those with a closed page policy, suggesting that these applications exhibit significant row locality in their reference streams. The col/closed policy performs poorly across all of the applications and traces in comparison with the other scheduling algorithms. As mentioned previously, this algorithm satisfies column accesses rapidly, emptying the bank buffer of references to a stream, allowing the banks to be precharged prematurely in some instances. The row/open scheduler delivers an average of 547MB/s to the applications and

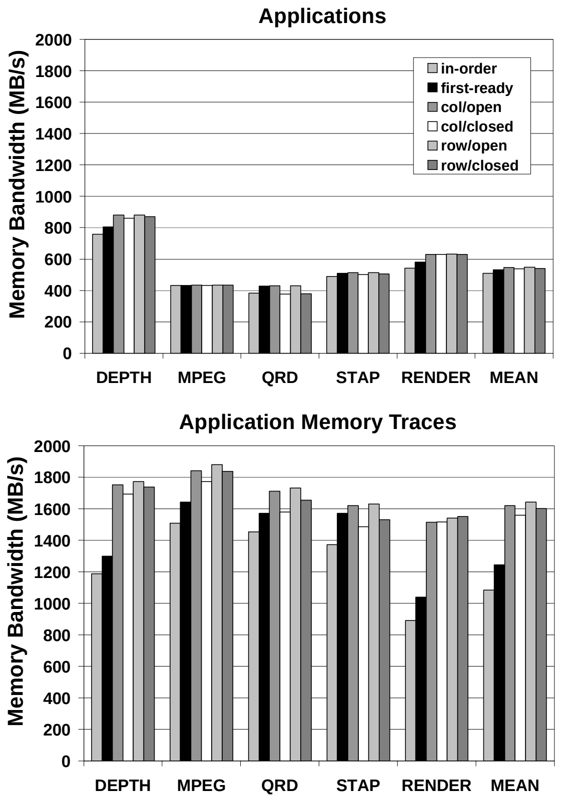


FIGURE 6.9 Application Performance Using Aggressive Schedulers

1641MB/s to the application traces. With this algorithm, DEPTH, MPEG, and QRD are compute-bound, and would require additional computation resources to improve performance. The memory system is able to deliver 79.5% and 87% of the memory bandwidth demanded by STAP and RENDER, respectively. Data dependencies constrain these applications from further overlapping memory and computation, limiting their peak performance.

6.5 Summary

An effective bandwidth hierarchy can bridge the gap of over two orders of magnitude between modern DRAM bandwidth and the bandwidth required by the arithmetic units. However, for such a hierarchy to be successful, the memory system must be able to sustain a significant fraction of the peak DRAM bandwidth. Five representative media processing applications demand from 0.4 to 1.0GB/s of memory bandwidth to achieve their peak computation rates. On the Imagine stream processor, this corresponds to 20-50% of the peak DRAM bandwidth. A streaming memory system which employs memory access scheduling to take advantage of the structure of modern DRAM can make many of these applications compute-bound, even on a processor with 48 arithmetic units.

Simple unit stride accesses can easily achieve 97% of the bandwidth of modern DRAM. However, random accesses are only able to sustain 14% of the DRAM bandwidth. The difference in performance corresponds to the degradation incurred by performing an access once every seven DRAM cycles, which is the random access throughput of the DRAM, compared to a throughput of one word every DRAM cycle, which is the column access throughput of the DRAM. Memory access scheduling closes this gap, and improves the performance across several microbenchmarks by an average of 144%. In practice, media processing applications make many stream references with constant strides, but they also access large two-dimensional buffers with arbitrary reference patterns. By scheduling these stream references to take advantage of the locality and concurrency of modern DRAM, the sustained memory bandwidth for media processing applications can make many media applications compute-bound and can increase the sustained bandwidth for the applications' memory traces by an average of 51%.

Memory access scheduling improves the sustainable throughput of modern DRAM by reordering memory references to exploit the available bank parallelism and row locality. As DRAM evolves, techniques such as memory access scheduling will continue to be necessary to overcome the non-uniform access times of the memory arrays. However, the techniques and algorithms described here could be integrated into the DRAM itself, with some modifications to enable the memory controller to interact with the scheduler.

CHAPTER 7

Conclusions

The major contribution of this work is a bandwidth-efficient stream architecture for media processing. VLSI constraints motivate a data bandwidth hierarchy to provide enough data bandwidth to support the demands of media processing. In modern VLSI, hundreds of arithmetic units can fit on an inexpensive chip, but communication among those arithmetic units is expensive in terms of area, power, and delay. Conventional storage hierarchies do not provide enough data bandwidth to support large numbers of arithmetic units. In contrast, a bandwidth hierarchy effectively feeds numerous arithmetic units by scaling data bandwidth across multiple levels. This bridges the gap between the bandwidth available from modern DRAM and the bandwidth demanded by tens to hundreds of arithmetic units.

This work also contributes the design and evaluation of the Imagine stream processor architecture to take advantage of the characteristics of media processing applications subject to the constraints of modern VLSI. The Imagine stream processor implements a data bandwidth hierarchy and memory access scheduling to efficiently provide data bandwidth to 48 arithmetic units. When expressed in the stream programming model, media processing applications map naturally to the data bandwidth hierarchy and are able to efficiently utilize data bandwidth that is scaled across the levels of the hierarchy. Imagine is, therefore, able to execute media processing applications at high sustained computation rates.

The stream programming model exposes the locality and concurrency of media processing applications so these characteristics can effectively be exploited by a stream processor. In this model, applications are expressed as a sequence of computation kernels that operate on streams of data. This exposes the locality and concurrency both within and among kernels, and enables the applications to effectively utilize a three-tiered data bandwidth hierarchy. At the base of the bandwidth hierarchy, memory access scheduling maximizes the throughput of modern DRAM enabling applications to sustain 79-96% of the memory bandwidth they demand. At the center of the hierarchy, the global stream register file provides a factor of 16 times more bandwidth than the memory system and enables data streams to be recirculated from kernel to kernel without consuming costly memory bandwidth. At the top of the bandwidth hierarchy, numerous local distributed register files feed 48 arithmetic units efficiently with a factor of 17 times more bandwidth than the stream register file. These local register files enable temporary storage of local kernel data so that more expensive global register bandwidth is only consumed to recirculate data streams. Imagine achieves 4 to 18GOPS of sustained computation on representative media processing applications using both the bandwidth hierarchy and memory access scheduling.

7.1 Imagine Summary

Imagine is a programmable media processor that matches the demands of media processing applications to the capabilities of modern VLSI technology. Imagine implements a stream architecture, which includes an efficient data bandwidth hierarchy and a streaming memory system. The bandwidth hierarchy bridges the gap between the 2GB/s memory system and the arithmetic units that require 544GB/s to achieve their full utilization. At the base of the bandwidth hierarchy, the streaming memory system sustains a significant fraction of the available DRAM bandwidth, enabling media processing applications to take full advantage of the bandwidth hierarchy.

The bandwidth hierarchy enables Imagine to achieve 77-96% of the performance of a stream processor with infinite memory and global data bandwidth. Kernels make efficient use of local register files for temporary storage to supply the arithmetic units with data and of the stream register file to supply input streams to the arithmetic units and store output streams from the arithmetic units. The stream register file efficiently captures the locality of stream recirculation within media processing applications, thereby limiting the bandwidth demands on off-chip memory. When mapped to the three-tiered storage hierarchy, the bandwidth demands of media processing applications are well-matched to the provided bandwidth which is scaled by a ratio of 1:16:276 across the levels of the hierarchy.

Memory access scheduling improves the performance of media processing applications on Imagine by an average of 8%, making many applications compute-bound, even with 48 arithmetic units. Simple unit stride accesses can easily achieve 97% of the bandwidth of modern DRAM. However, random accesses are only able to sustain 14% of the DRAM bandwidth without memory access scheduling. The difference in performance corresponds to the degradation incurred by performing an access once every seven DRAM cycles (the random access throughput of the DRAM), compared to a throughput of one word every DRAM cycle (the column access throughput of the DRAM). Memory access scheduling closes this gap and improves the performance across several representative microbenchmarks by an average of 144%. Using an aggressive scheduler that exploits the row locality and bank parallelism of the reference stream, Imagine is able to improve the bandwidth of application memory traces by an average of 51% and to provide 79-96% of the memory bandwidth demanded by media processing applications.

Imagine is designed to fit on a 2.5cm^2 chip and operate at 500MHz in a 0.15µm CMOS standard-cell implementation. The peak computation rate of the 48 arithmetic units is 20GOPS for both 32-bit integer and floating-point operations. For 16-bit and 8-bit parallel-subword operations, the peak performance increases to 40 and 80GOPS, respectively. Applications achieve a significant fraction of this peak rate. For example, an MPEG encoder can sustain 18GOPS on 16-bit data.

7.2 Future Architectures

Microprocessors have traditionally included a storage hierarchy consisting of DRAM, caches, and a global register file. This organization is optimized to minimize the arithmetic units' latency to access data. The register file is typically small enough that it can be accessed in a single cycle. The cache hierarchy keeps data that is likely to be accessed soon close to the arithmetic units to minimize the latency of memory operations. The slow DRAM is used as a large storage area that is only accessed when the cache hierarchy does not hold the referenced data.

The effectiveness of the cache hierarchy is deceptive, as many of the advantages of a cache arise from the fact that register files are too small. A cache relies on data reuse to reduce latency. An inadequately sized register file artificially inflates the amount of data reuse within an application. For example, the FIR filter example of Chapter 5 showed that the coefficients of the filter needed to be reloaded from memory for the computation of every set of outputs. If the register file were larger, then these coefficients could be kept in registers, rather than the cache, and would then only be accessed outside the register file once, at the start of execution. Similarly, temporary data produced during the course of a computation is frequently spilled to memory because the register file is not large enough. Again, the cache captures this artificial data reuse by holding that data until it is reloaded into the register file.

For applications with latency tolerance, as found in media processing, data can be prefetched far in advance of its use. Ideally, that data would be loaded into registers, but small register files do not have space for both the current data and the prefetched data. A cache can hold this prefetched data, again making the cache appear beneficial. Therefore, a cache's effectiveness is largely due to the fact that modern VLSI constrains global register files that directly feed arithmetic units to be small.

A cache can also limit the available bandwidth for the arithmetic units and waste external DRAM bandwidth, making it difficult to bridge the data bandwidth gap. A cache is an inefficient way to stage data from memory because address translation is required on every reference, accesses are made with long memory addresses, some storage within the cache must be allocated to address tags, and conflicts may evict previously fetched data that will be referenced in the future. These inefficiencies make it difficult to scale the bandwidth of a cache without dramatically increasing its cost. Furthermore, an ineffective cache actually wastes external DRAM bandwidth. Cache line fetches retrieve more than one word from the DRAM on every cache miss. If these words are not all accessed, then DRAM bandwidth was wasted transferring the entire line. Similarly, writing back data can waste bandwidth. A write-through policy will always write stored data back to the DRAM. This is particularly wasteful if the data is actually temporary data that was spilled to memory due to limited register space. That temporary data will never be referenced again after it is reloaded into the register file, so it did not need to return to DRAM. A write-back policy will usually write the entire cache line back to DRAM, again wasting bandwidth if not all of the words in the line are dirty.

As the number of arithmetic units in a processor rises, data bandwidth becomes increasingly important. Since a cache is optimized to minimize latency rather than maximize bandwidth, the traditional storage hierarchy needs to be reconsidered for processors that contain tens to hundreds of arithmetic units. For example, an explicitly managed bandwidth hierarchy is a far more efficient use of resources for media processing applications. A global stream register file does not incur the overhead of a cache and is able to provide far more data bandwidth through a single, wide port. Similarly, local distributed register files do not suffer from the bandwidth bottlenecks of a global register file, so they can effectively support larger numbers of arithmetic units.

A programming model that exposes the locality and concurrency of applications is required to utilize an explicitly managed storage hierarchy effectively. The traditional sequential programming model hides parallelism and locality, making it difficult to extract parallel computations and manage data movement. The stream programming model, however, enables parallelism to be readily exploited and makes most data movement explicit. This enables a media processor to store data and perform computations more efficiently.

This work demonstrates the need for future media processors to be designed to maximize the data bandwidth delivered to the arithmetic units. Without sufficient data bandwidth, media processors will not be able to support enough arithmetic units to meet the demands of compelling media applications. The traditional storage hierarchy found in most programmable processors is unable to provide enough data bandwidth to support tens to hundreds of arithmetic units, both because the hardware structures are inefficient and because the supported programming model hides the locality and concurrency of the applications. Communication-limited VLSI technology makes it difficult to extract locality and concurrency dynamically.

Therefore, future media processors must provide an explicitly managed storage hierarchy, scale data bandwidth between external DRAM and the arithmetic units, and provide local communication and storage resources. An explicitly managed storage hierarchy can minimize global communication by allowing the compiler to be aware of expensive communication and storage. This motivates the compiler to utilize inexpensive bandwidth when possible and resort to more expensive bandwidth only when necessary. The bandwidth between external DRAM and the arithmetic units must be scaled in order to support large numbers of arithmetic units. Without this scaling, utilization will ultimately be limited by memory bandwidth. Finally, local communication and storage resources are required to enable temporary data to be efficiently stored and communicated among arithmetic units. The traditional global register file organization severely limits the sustainable computation rate by requiring global data bandwidth to be used for local data transfers and memory bandwidth to be used for intermediate data storage. Local register files can be utilized to store local data and provide efficient bandwidth to the arithmetic units, which frees up the global register file to hold intermediate values and prevent them from accessing external DRAM unless absolutely necessary. Bandwidth-efficient architectures are therefore required to overcome the communication-limited nature of modern VLSI.

References

[BW95] V. Michael Bove, Jr. and John A. Watlington. Cheops: A Reconfigurable Data-Flow System for Video Processing. *IEEE Transactions on Circuits and Systems for Video Technology* (April 5, 1995), pp. 140-149.

[BGK97] Doug Burger, James R. Goodman, and Alain Kägi. Limited Bandwidth to Affect Processor Design. *IEEE Micro* (November/December 1997), pp. 55-62.

[CTW97] Kenneth C. Cain, Jose A. Torres, and Ronald T. Williams. RT_STAP: Real-Time Space-Time Adaptive Processing Benchmark. Technical Report MTR 96B0000021, The MITRE Corporation, Bedford MA, February 1997.

[CHS+99] John Carter, Wilson Hsieh, Leigh Stoller, Mark Swanson, Lixin Zhang, Erik Brunvand, Al Davis, Chen-Chi Kuo, Ravindra Kuramkote, Michael Parker, Lambert Schaelicke, and Terry Tateyama. Impulse: Building a Smarter Memory Controller. *Proceedings of the Fifth International Symposium on High Performance Computer Architecture* (January 1999), pp. 70-79.

[CEV98] Jesus Corbal, Roger Espasa, and Mateo Valero. Command Vector Memory Systems: High Performance at Low Cost. *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques* (October 1998), pp. 68-77.

[Cri97] Richard Crisp. Direct Rambus Technology: The New Main Memory Standard. *IEEE Micro* (November/December 1997), pp. 18-28.

[CJDM99] Vinodh Cuppu, Bruce Jacob, Brian Davis, and Trevor Mudge. A Performance Comparison of Contemporary DRAM Architectures. *Proceedings of the International Symposium on Computer Architecture* (May 1999), pp. 222-233.

[Die99] Keith Diefendorff. Pentium III = Pentium II + SSE. *Microprocessor Report* (March 1999), pp. 1-7.

[FvFH96] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice,* Addison-Wesley Publishing Company: Menlo Park, California, 1996.

[Fol96] Pete Foley. The Mpact Media Processor Redefines the Multimedia PC. *Proceedings of COMPCON* (February 1996), pp. 311-318.

[Gla99] Peter N. Glaskowsky. Most Significant Bits: NVIDIA GeForce offers acceleration. *Microprocessor Report* 13(12), 1999.

[GV96] Gene H. Golub and Charles F. Van Loan. *Matrix Computations, Third Edition*, The John Hopkins University Press: Baltimore, Maryland, 1996.

[GMG89] A. Gunzinger, S. Mathis, and W. Guggenbühl. The SYnchronous DAtaflow MAchine: Architecture and Performance. *Proceedings of Parallel Architectures and Languages Europe* (June 1989), pp. 85-99.

[HG97] Ziyad S. Hakura and Anoop Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. *Proceedings of the International Symposium on Computer Architecture* (June 1997),pp. 108-120.

[HMS+99] Sung I. Hong, Sally A. McKee, Maximo H. Salinas, Robert H. Klenke, James H. Aylor, and William A. Wulf. Access Order and Effective Bandwidth for Streams on a Direct Rambus Memory. *Proceedings of the Fifth International Symposium on High Performance Computer Architecture* (January 1999), pp. 80-89.

[HL99] Tim Horel and Gary Lauterbach. UltraSPARC-III: Designing Third-Generation 64-Bit Performance. *IEEE Micro* (May/June 1999), pp. 73-85.

[Jac96] Keith Jack. *Video Demystified: A Handbook for the Digital Engineer*, LLH Technology Publishing: Eagle Rock, Virginia, 1996.

[Jou90] Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. *Proceedings of the International Symposium on Computer Architecture* (May 1990),pp. 364-373.

[KDR+00] Ujval Kapasi, William J. Dally, Scott Rixner, Peter Mattson, and John D. Owens. Efficient Conditional Operations for Data-Parallel Architectures. *Proceedings of the International Symposium on Microarchitecture* (December 2000).

[Kes99] R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro* (March/April 1999), pp. 24-36.

[KKK95] Takeo Kanade, Hiroshi Kano, and Shigeru Kimura. Development of a Video-Rate Stereo Machine. *Proceedings of the International Robotics and Systems Conference* (August 1995), pp. 95-100.

[Koz99] Christoforos Kozyrakis. A Media-Enhanced Vector Architecture for Embedded Memory Systems. Technical Report UCB/CSD-99-1059, University of California Computer Science Division, Berkeley, CA, July 1999.

[Kro81] David Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. *Proceedings of the International Symposium on Computer Architecture* (May 1981), pp. 81-87.

[LS98] Corinna G. Lee and Mark G. Stoodley. Simple Vector Microprocessors for Multimedia Applications. *Proceedings of the International Symposium on Microarchitecture* (December 1998), pp. 25-36.

[Lee96] Ruby B. Lee. Subword Parallelism with MAX-2. *IEEE Micro* (August, 1996), pp. 51-59.

[Lin82] N. R. Lincoln. Technology and Design Tradeoffs in the Creation of a Modern Supercomputer. *IEEE Transactions on Computers* (May 1982), pp. 363-376.

[Lip68] J. S. Liptay. Structural Aspects of the System/360 Model 85, Part II: The Cache. *IBM Systems Journal* 7:1 (1968), pp. 15-21.

[MMCD00] Binu K. Matthew, Sally A. McKee, John B. Carter, and Al Davis. Design of a Parallel Vector Access Unit for SDRAM Memory Systems. *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture* (January 2000), pp. 39-48.

[MDR+00] Peter Mattson, William J. Dally, Scott Rixner, Ujval J. Kapasi, and John D. Owens. Communication Scheduling. *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (November 2000).

[MW95] Sally A. McKee and William A. Wulf. Access Ordering and Memory-Conscious Cache Utilization. *Proceedings of the First Symposium on High Performance Computer Architecture* (January 1995), pp. 253-262.

[MBDM97] John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal. InfiniteReality: A Real-Time Graphics System. *Proceedings of SIGGRAPH* (August 1997), pp. 293-302.

[NEC98] NEC Corporation. *128M-bit Synchronous DRAM 4-bank, LVTTL Data Sheet*. Document No. M12650EJ5V0DS00, 5th Edition, Revision K (July 1998).

[Nit99] Ramon Nitzberg. *Radar Signal Processing and Adaptive Systems*, Artech House: Boston, MA, 1999.

[Oed92] Wilfried Oed. Cray Y-MP C90: System features and early benchmark results. *Parallel Computing* (August 1992), pp. 947-954.

[OSB99] Alan V. Oppenheim and Ronald W. Shafer with John R. Buck. *Discrete-Time Signal Processing, Second Edition*, Prentice Hall: Upper Saddle River, NJ, 1999.

[ODK+00] John Owens, William J. Dally, Ujval J. Kapasi, Scott Rixner, Peter Mattson, and Ben Mowery. Polygon Rendering on a Stream Architecture. *Proceedings of the Eurographics/SIGGRAPH Workshop on Graphics Hardware* (August 2000).

[PW96] Alex Peleg and Uri Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro* (August, 1996), pp. 42-50.

[RS96] Selliah Rathnam and Gerrit A. Slavenburg. An Architectural Overview of the Programmable Multimedia Processor, TM-1. *Proceedings of COMPCON* (February 1996), pp. 319-326.

[RDK+98] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo López-Lagunas, Peter R. Mattson, and John D. Owens. A Bandwidth-Efficient Architecture for Media Processing. *Proceedings of the International Symposium on Microarchitecture* (December 1998), pp. 3-13.

[RDK+00a] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory Access Scheduling. *Proceedings of the International Symposium on Computer Architecture* (June 2000), pp. 128-138.

[RDK+00b] Scott Rixner, William J. Dally, Brucek Khailany, Peter Mattson, Ujval J. Kapasi, and John D. Owens. Register Organization for Media Processing. *Proceedings of the Sixth Symposium on High Performance Computer Architecture* (January 2000), pp. 375-386.

[Rus78] Richard M. Russell. The Cray-1 Computer System. *Communications of the ACM* (January 1978), pp. 63-72.

[SV89] R. Schreiber and C. Van Loan. A storage-efficient WY representation for products of Householder transformations. *SIAM J. Scientific and Statistical Computing*, 10(1):53--57, 1989.

[Sed90] Robert Sedgewick. *Algorithms in C*, Addison-Wesley Publishing Company, 1990.

[TH99] Shreekant Thakkar and Tom Huff. Internet Streaming SIMD Extensions. *Computer* (December 1999), pp. 26-34.

[TONH96] Marc Tremblay, J. Michael O'Connor, Venkatesh Narayanan, and Liang He. VIS Speeds New Media Processing. *IEEE Micro* (August, 1996), pp. 10-20.

[TH97] Jim Turley and Harri Hakkarainen. TI's New 'C6x DSP Screams at 1,600 MIPS. *Microprocessor Report* (February 17, 1997), pp. 14-17.

[VPPL94] H. Veendrick, O. Popp, G. Postuma, and M. Lecoutere. A 1.5 GIPS video signal processor (VSP). *Proceedings of the IEEE Custom Integrated Circuits Conference* (May 1994), pp. 95-98.

[WAM+99] Takeo Watanabe, Kazushige Ayukawa, Seiji Miura, Makoto Toda, Tetsuya Iwamura, Kouichi Hoshi, Jun Sato, and KazumasaYanagisawa. Access Optimizer to Overcome the "Future Walls of Embedded DRAMs" in the Era of Systems on Silicon. *IEEE International Solid-State Circuits Conference Digest of Technical Papers* (February 1999), pp. 370-371.

[YHO97] Tadaaki Yamauchi, Lance Hammond, and Kunle Olukotun. The Hierarchical Multi-bank DRAM: A High-Performance Architecture for Memory Integrated with Processors. *Proceedings of the Conference on Advanced Research in VLSI* (September 1997), pp. 303-319.

Index