MEMORY HIERARCHY DESIGN FOR STREAM COMPUTING

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

Nuwan S. Jayasena

September 2005

Abstract

To be filled in...

Acknowledgements

To be filled in too...

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Recent application trends have made media and signal processing, and graphics some of the most important applications across a broad range of computing platforms ranging from desktop to mobile devices. At the same time, demands on high performance computing systems continue to grow as scientific applications in fields such as biotechnology and climate modeling attempt to simulate larger-scale phenomena for longer periods with increasing precision. A characteristic common to both these application classes, as well as several other important ones such as network processing and encryption, is the presence of *data parallelism* which enables the computation on many data elements to take place simultaneously.

The parallelism available in these applications provide an excellent match to the characteristics of modern VLSI implementation technology. The very large number of devices available on a single chip today enables many arithmetic units to be integrated on to a single processor, allowing several computations to proceed in parallel. However, bandwidth for providing operands for these computations must be managed carefully. While very high bandwidths can be sustained on-chip, only a small fraction of that potential bandwidth can be supplied from off-chip memory due to pin bandwidth limitations. Therefore, it is important to capture as much data reuse, or locality, available in applications in on-chip memory. *Stream programming* expresses data parallel applications in a manner that exposes the available parallelism and locality. *Stream processor* architectures exploit the parallelism and locality exposed by the programming model to achieve high performance and energy efficiency. Stream processing has been demonstrated to be effective for both media and scientific applications [RDK+98; DHE+03].

Stream processors implement a bandwidth hierarchy made up of registers and memory to sustain high operand bandwidth to arithmetic units. The registers closest to the arithmetic units provide the operands, and each successively further level of the hierarchy sustains decreasing bandwidth, matching the available off-chip bandwidth at the final level. Therefore, designing the bandwidth hierarchy in a manner that enables capturing the locality available in a broad range of data access patterns at levels close to the arithmetic units is critical for efficiently supporting a wide range of data parallel applications.

This thesis focuses on techniques for improving the efficiency of off-chip bandwidth utilization by capturing a broad range of locality on-chip, and by reducing non-essential off-chip traffic. We present a novel architecture for *stream register files* (SRF) that supports a wider range of access patterns than current stream processors through the use of explicitly indexed accesses. The SRF is a key component in the bandwidth hierarchy of a stream processor, and the additional flexibility afforded by indexed accesses enables more of the locality available in applications to be captured on-chip. We also compare the specialized bandwidth hierarchies of stream processors, which rely on software-managed memories, with the hardware-managed caches of general-purpose processors. We highlight the relative strengths of these two types of memories, and identify optimizations necessary in caches to better support streaming applications. Hybrid bandwidth hierarchies that incorporate characteristics of both stream register files and cache memories are also explored, and an organization that enables the mix of stream and cache memory to be tailored to suit application characteristics is presented.

We also explore the efficiency of stream processing in terms of register requirements, and DRAM and cache bandwidth utilization. These studies show that the ordering of computation and memory accesses in stream processing lead to lower register requirements and provide a better match to modern memory hierarchy characteristics relative to other mainstream, programmable data parallel execution models.

1.1 Research Contributions

The main contributions of this thesis to the field of computer architecture and stream processing are:

1. An analysis of the register capacity requirements of stream processing compared to other widely used, programmable data parallel processing techniques. This analysis demonstrates the reduced register capacity requirements that arise from structuring the parallel computation at the granularity of operations on entire data records, as is done in stream processing, compared to alternative techniques that express parallelism at a granularity smaller than entire records.

2. An evaluation of cache and DRAM performance of stream processing compared to other widely used, programmable data parallel processing techniques. This evaluation shows that stream processing, by exposing the record-granularity locality present in applications to the hardware, improves utilization in modern DRAM-based memory hierarchies, thereby achieving higher performance for bandwidth-constrained applications.

3. A novel architecture for stream register files that supports flexible access patterns through explicit indexing. This design enables a more complete *communication hierarchy* within the bandwidth hierarchy of stream processors, enabling a broader range of data reuse to be captured at the stream register file level. This directly results in reduced demands on scarce off-chip memory bandwidth for a variety of application classes. A microarchitecture for a low-area-overhead implementation of the proposed design is also presented.

4. An evaluation of software- and hardware-managed on-chip memories for stream computing, which provides insights on the key benefits and sources of inefficiencies in each of these two types of memories. Hardware-managed (cache) memories benefit from fine-grain memory management implemented in hardware,

but suffer due to inefficiencies in replacement policies, name translation, and bandwidth limitations. Software-managed memories suffer due to coarse-grain memory management in software, but benefit from application-aware management policies and greater flexibility in name translation. This evaluation also establishes design requirements for hardware-managed (cache) memories when used in the context of a streaming bandwidth hierarchy.

5. *Epoch-based cache invalidation*, a novel technique for improving hardware-managed memory performance for stream computing based on identifying and actively invalidating cached state that is no longer live. This enables cache lines with dead data to be evicted without generating memory writes even if the lines are marked as dirty in the cache, resulting in reduced memory traffic.

6. A microarchitecture for a hybrid memory that supports both software and hardware management while preserving the key requirements of each type of memory for stream computing. The proposed structure allows the available capacity to be dynamically allocated between hardware- and software-managed memories based on application requirements identified at compile-time. Evaluations of this memory demonstrate that no single allocation of capacity between hardware- and software-managed memory is optimal over different applications, and customizing the allocation on a per-application basis leads to improved performance, particularly as off-chip bandwidth becomes increasingly scarce.

1.2 Thesis Roadmap

Chapter 2 provides an overview of background material including a brief introduction to stream programming, stream processor architecture, and alternative data parallel architectures.

Chapter 3 evaluates the impact that the granularity at which data parallelism is exploited at - individual operations or entire loop bodies - has on intermediate state storage requirements and off-chip memory system performance. This distinction in granularity correlates closely to a key difference between vector and stream processing - two common data parallel execution models used in a number of recent programmable data-parallel architectures.

Chapter 4 explores the spatial and temporal communication requirements of data parallel applications, and the importance of a communication hierarchy in enabling the bandwidth hierarchy to efficiently capture a broad range of application data reuse patterns. Chapter 4 also introduces and evaluates a novel stream register file architecture that efficiently supports significantly more communication freedom compared to prior designs. This added flexibility enables efficient support for a wider range of access patterns, improving the performance of several classes of applications both in terms of execution time and reduced off-chip memory traffic.

While on-chip memories of traditional data parallel architectures have often been software-managed, using hardware managed memories for data parallel computing is becoming increasingly important due to a number of recent trends. Chapter 5 addresses these trends and studies the tradeoffs in software and hardware management of on-chip memory for data parallel computing. We characterize application data access patterns, and correlate these to memory system performance in order to identify the types of accesses that benefit from each type of memory. This study also highlights the sources of performance loss in the hardware and software managed memories, and establishes a set of requirements for each type of memory for sustaining high performance for the application classes studied. Chapter 5 also introduces epoch-based cache invalidation, a technique to improve the performance of hardware managed memories by identifying and explicitly invalidating dead data in the cache.

Chapter 6 introduces and evaluates a hybrid memory architecture that supports both software and hardware management while retaining the desirable characteristics of each type of memory identified in chapter 5. This design allows the available on-chip memory capacity to be dynamically allocated between software and hardware management, enabling the memory hierarchy to be better tailored to each application's access characteristics. The microarchitecture proposed for this design is a logical extension of the stream register file architecture introduced in chapter 4, enabling a low overhead implementation in the context of stream processing.

Finally, chapter 7 presents conclusions and discusses future research directions.

Chapter 2

Background and Motivation

Today's VLSI circuit technology enables billions of transistors to be integrated on a single chip. Therefore, the potential exists for thousands of arithmetic units to be incorporated on a single processor to achieve very high performance. However, in order to realize this potential, two technology constraints must be overcome. First, the long latency of memory accesses, which can be hundreds to thousands of cycles for an off-chip access in modern high performance processors, must be tolerated. Second, off-chip memory bandwidth, which is significantly lower than that available to on-chip memory structures, must not become a performance bottleneck.

Traditional general-purpose processors rely on instruction-level parallelism (ILP) to achieve high performance across a broad range of applications. However, extracting ILP from instruction sequences at compile-time is hindered by the lack of run-time information, and doing so at run-time requires complex and expensive hardware structures. Therefore, resources of processors designed for ILP are allocated to anticipate a relatively small amount of simultaneous computation. Significantly greater resources are spent on discovering the parallelism and on cache hierarchies designed to reduce the average latency of memory accesses. As a result, only a small fraction of the raw capability of the underlying implementation technology is realized in terms of true computational performance in such architectures.

Several important application classes exhibit data parallelism where computation can be applied to a large number of data items simultaneously. Unlike ILP, data parallelism can often be analyzed and scheduled at compile-time, eliminating the need for expensive hardware techniques to dynamically discover parallelism. Application classes with large amounts of data parallelism include media and signal processing, where long sequences of samples are operated on, and scientific applications where the behavior of physical systems are approximated by modeling a large number of small segments of the overall system. These application classes are also some of the most demanding consumers of compute cycles. It is widely accepted that media applications will continue to consume an increasingly greater fractions of the computing capability on personal computing devices [DD97]. At the same time, scientific computing applications often drive the requirements for high-end supercomputer designs. The parallelism available in these applications can be exploited by processor architectures specialized for extracting data parallelism to sustain very high performance, realizing a greater fraction of the capabilities enabled by implementation technology.

2.1 Exploiting Data Parallelism

The presence of data parallelism in an application provides two important opportunities to achieve high performance – the parallelism itself, and latency tolerance. It is important to note that the memory hierarchy design plays a critical role in enabling both these opportunities.

First, computation on large numbers of data elements can proceed in parallel, increasing the compute rate. This is further facilitated by the fact that operations on one data element are largely independent of operations on other data elements in most media and scientific applications. However, the levels of the memory hierarchy closest to the arithmetic units (usually the register files) must be designed to provide sufficient operand bandwidth to sustain the parallel computation. Furthermore, the overall compute rate is ultimately limited by the rate at which application inputs and outputs can be read or written from off-chip memory through any intermediate stages of the memory hierarchy. Therefore, efficient utilization of the bandwidth available at all levels, especially off-chip memory, is crucial for sustaining high performance.

Second, the available parallelism allows long memory latencies to be tolerated. Unlike ILP processors, whose ability to continue beyond a long latency memory operation is limited by available ILP and the sizes of the issue window and commit queue, a data parallel architecture can continue to operate on the many other independent data elements available. However, the memory hierarchy must be designed to support continued execution while the long latency accesses are outstanding. While some degree of such support is found in most modern high-performance memory systems in the form of non-blocking caches with multiple outstanding misses, the requirements are more stringent for data parallel architectures. The cost of a memory stall in terms of wasted instruction issue slots is extremely high in data parallel architectures due to the highly-parallel execution capabilities. In order to minimize the probability of memory stalls, the number of concurrently outstanding memory accesses that must be supported by data parallel architectures to the i^{th} level of the memory hierarchy is given by *Little's Law* as expressed by equation (2.1), where A_i, BW_i, and L_i are the outstanding accesses, bandwidth, and latency for the i^{th} level of the memory hierarchy. In modern systems where off-chip memory accesses take several hundred cycles to complete, this requires the memory system to support thousands of in-flight accesses.

$$A_i = BW_i \times L_i \tag{2.1}$$

The latency tolerance of data parallel applications shifts the focus of memory hierarchy design of architectures targeted at these applications from reducing average latency (which is the critical concern in general-purpose CPUs) to achieving high bandwidth. The ability to sustain many in-flight memory operations is crucial from this perspective as well, since it allows the entire memory pipeline to be filled, enabling a high fraction of the peak memory bandwidth to be sustained.

The techniques presented in this thesis build on these capabilities enabled by data parallelism in order to further improve the off-chip memory bandwidth utilization for a variety of application classes.

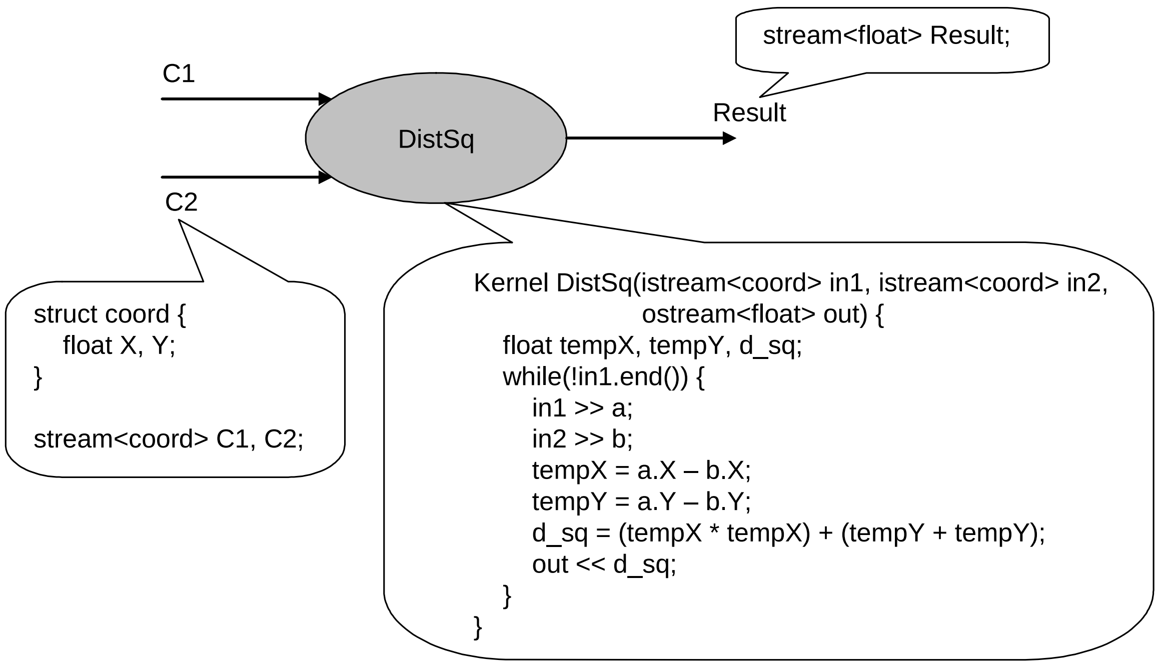
2.2 Stream Programming

The stream programming model can be used to express data parallel applications in a manner that exposes the available locality and parallelism. In this model, an application is composed of a collection of data *streams* passing through a series of computation *kernels*. Each stream is a sequence of homogeneous data records. Each kernel is a loop body that is applied to each record of the input stream(s). A trivial example is a kernel that computes the square of the distance between two points in 2D space and is shown in figure 2.1. The compute kernel is represented by the oval and streams are represented by the arrows in the figure. The kernel takes two input streams, each a sequence of 2D coordinates, and outputs a single stream, the squared distances between each pair of input coordinates. In general, kernels may have zero or more input streams and one or more output streams. Kernels that perform *reductions* may generate scalar outputs instead of, or in addition to, output streams. The computation performed within the kernel loop bodies may be arbitrarily complex.

A more complex example – a simplified portion of a finite element method (FEM) expressed as a stream program – is shown in figure 2.2. The first kernel in this example (*K1*) receives an input stream of grid cells and generates a stream of indices in to an array of flux values stored in memory. These indices are used to lookup a sequence of flux values that form the input stream to kernel *K2*. *K2* and subsequent kernels all produce output streams that are fed to later kernels as inputs, until the final kernel (*K4*) produces the application's outputs.

Locality

This stream representation allows the programmer to express multiple levels of locality [RDK+98] that can be captured at various levels of the bandwidth hierarchy. *Kernel locality* is exploited by keeping all intermediate values generated during a kernel's execution in registers without writing them to higher levels of the memory hierarchy. Only the input and output streams of a kernel are read from or written to the memory hierarchy. *Producer-consumer locality* is exposed by forwarding the streams produced



```
stream<float> Result;
```

```
struct coord {
   float X, Y;
}

stream<coord> C1, C2;
```

```
Kernel DistSq(istream<coord> in1, istream<coord> in2,
                  ostream<float> out) {
   float tempX, tempY, d_sq;
   while(!in1.end()) {
      in1 >> a;
      in2 >> b;
      tempX = a.X – b.X;
      tempY = a.Y – b.Y;
      d_sq = (tempX * tempX) + (tempY + tempY);
      out << d_sq;
   }
}
```

Figure 2.1: Simple kernel computing the square of the distance between two sequences of coordinates

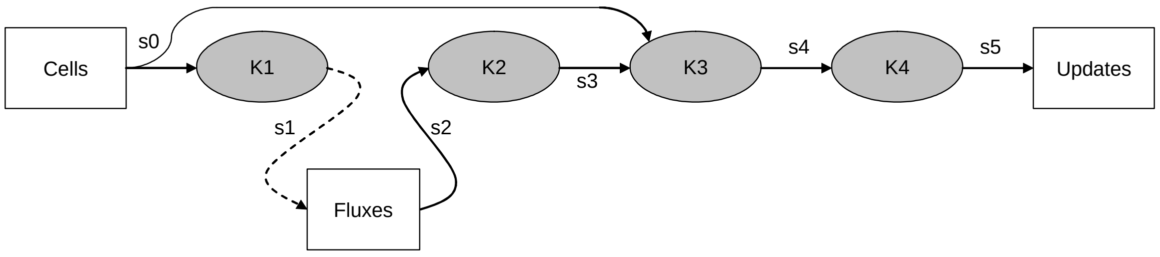


Figure 2.2: Stream representation of simplified finite element method

by one kernel to subsequent kernels as shown by the arrows in the example of figure 2.2. Ideally, these intermediate streams are captured entirely in on-chip memory, and do not generate off-chip memory accesses. Stream-level temporal locality is expressed by a single stream being consumed by multiple kernels. Once generated or fetched from memory, such streams are held in on-chip memory whenever possible for the entire duration of all uses. Finally, off-chip memory accesses are performed for reading essential application inputs and writing final application outputs. This programming model matches well with the hierarchy of registers, on-chip memory, and off-chip memory present in most architectures. The registers are typically optimized for high bandwidth to sustain computation while the on-chip memory is optimized for large capacity to capture the working sets of applications and to hide memory latency.

The partitioning of applications in to kernels is driven by the available register and memory resources of the target architecture. Kernels are sized to minimize register spills, potentially partitioning loop bodies that generate more intermediate state than can be held in registers in to multiple kernels. At the application level, data sets that are too large to fit in on-chip memory are partitioned using a technique known as *strip-mining* [Lov77]. Strip-mining partitions the input data set in to segments known as *strips* such that all of the intermediate state for the computation on a single strip fits in on-chip memory. Thus multiple strips can be operated on in sequence, still limiting off-chip memory accesses to only the essential application inputs and outputs.

Parallelism

The stream programming model also exposes multiple levels of parallelism inherent in the application. Data parallelism is expressed since the kernel loop body may be applied to many records of an input stream at once. In the solver example, the *K1* kernel can be applied to all of the cells of its input stream in parallel, assuming the processor the stream program is being executed on has sufficient compute resources. ILP may also be exploited within the stream programming model by executing multiple independent operations among those that must be applied to a single data element (i.e. within a single iteration of the kernel loop body) in parallel. This may be achieved via pipelining and/or issuing multiple instructions per cycle.

Task-level parallelism across kernels is also exposed by the stream programming model. For example, kernels *K1* and *K2* in the solver example may execute in parallel on different processors, with the results of *K1* being passed on to *K2*. In addition, the execution of a kernel may also be overlapped with the loading of application inputs for a subsequent kernel or storing of results generated by a previous kernel.

The stream programming model does not allow arbitrary accesses to global memory within kernels. This restriction enables inter-kernel access disambiguation to be performed at the stream level. In this context, applications that require data-dependent address generation must first generate a set of indices and use those to *gather* data in to contiguous streams which can then be used as inputs to subsequent kernels. Such an example is shown by the access to the flux array in the example of figure 2.2, where *K1* generates the indices and the gathered stream is consumed by *K2*. Similarly, data dependent writes my be performed by *scattering* a result stream using an index stream after the completion of the kernel(s) that generates the results.

Stream Programming Languages

Applications can be expressed in the stream programming model through specialized high-level languages. [Mat02] describes *KernelC*, a language for representing compute kernels, and *StreamC*, a language for orchestrating sequences of kernels and stream manipulations along with serial portions of code that are not data parallel. [BFH$^+$04] presents *Brook*, a unified language for expressing kernel and serial code as well as the orchestration of kernels. [TKA02] describes *streamit*, a language for expressing stream computation as a collection of filters connected via pipelines, split-joins, and feedback loops. All of these languages are based on the familiar syntax of mainstream languages such as C/C++ and Java and add additional semantics to express streams and impose restrictions to enable data flow and dependency analysis among kernels or filters.

The benchmarks used for evaluations in this thesis were implemented in the stream programming model using the *StreamC* and *KernelC* programming languages. This allows the locality and parallelism in the applications to be exploited across a variety of processor architectures. In addition, novel techniques introduced in this thesis, such as indexed SRF access, epoch-based cache invalidation, and hybrid memory hierarchies, are evaluated within the context of stream programming. While these techniques also have potential applications outside the domain of stream programming, the locality exposed by this programming model enable these techniques to achieve a high degree of efficiency.

2.3 Stream Processor Architecture

Stream processors are programmable processors that are optimized for executing programs expressed using the stream programming model. A block diagram of a stream processor is shown in figure 2.3.

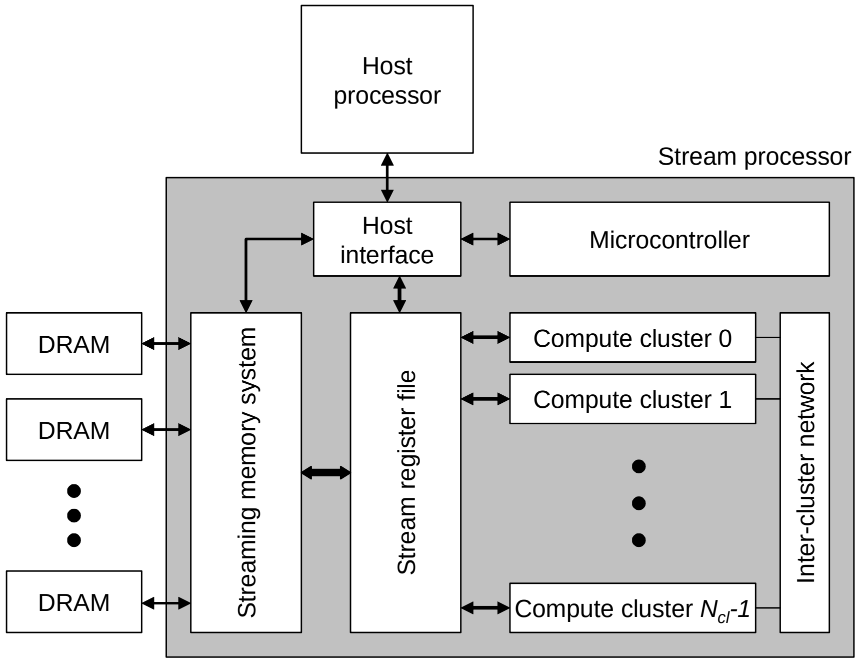


Figure 2.3: Block diagram of a stream processor

The stream processor operates as a coprocessor under the control of the host processor, which is often a standard general-purpose CPU. A *stream program* executing on the host processor orchestrates the sequence of kernels to be executed and the necessary transfer of input and output data streams between the stream processor and off-chip memory. Kernel execution takes place directly on the stream processor from instructions stored in the microcontroller. New kernels may be loaded into the microcontroller as needed, possibly under explicit control of the host processor. The sequence of operations initiated by the host processor to orchestrate the stream program of figure 2.2 is shown in figure 2.4. The host interface of the stream processor issues the commands received from the host to the appropriate units as resources become available, subject to dependencies among the commands.

```
load_microcode(K1);
load_microcode(K2);
load_microcode(K3);
load_microcode(K4);
load_stream(Cells, s0);
K1(s0, s1);
load_stream(Fluxes[s1], s2);
K2(s2, s3);
K3(s0, s3, s4);
K4(s4, s5);
store_stream(s5, Updates);
```

Figure 2.4: Sequence of host processor operations for simple FEM example

The arithmetic units of the stream processor are grouped in to n identical compute clusters. Each cluster consists of several functional units and associated registers. A block diagram of an example cluster organization is shown in figure 2.5. The local register files (LRFs) attached to each functional unit provide the input operands for that unit, and results are written to one or more of the LRFs via the intra-cluster network. Loop-carried state and other shared data may be communicated among compute clusters over the inter-cluster network via the *COMM* unit.

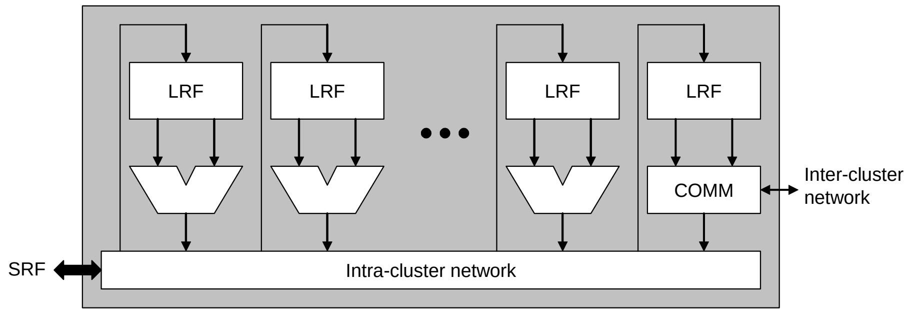


Figure 2.5: Block diagram of a single compute cluster

The microcontroller executes the instruction streams for the kernels and broadcasts the control for arithmetic operations to the compute clusters. Therefore all compute clusters execute the same sequence of operations in single-instruction multiple-data (SIMD) fashion. While SIMD execution is not fundamental to stream computation, it amortizes the overhead of instruction storage and control sequencing hardware over all clusters, leading to implementation efficiencies. The data parallelism in applications is exploited by each compute cluster operating on a separate element of the input stream(s). Techniques such as predication and *conditional streams* [KDR+00] are used to provide efficient support for conditional execution within this SIMD framework.

The units within a compute cluster are controlled in VLIW fashion, exploiting the ILP available within a single iteration of the kernel loop. Loop unrolling and software pipelining may be performed at compile time to increase arithmetic unit utilization in kernels that do not have sufficient ILP to fill a large fraction of the VLIW issue slots.

The *stream register file* (SRF) is a large on-chip memory that provides storage for intermediate streams, capturing the producer-consumer and temporal locality that exists between kernels at stream granularity.

The *streaming memory system* includes one or more stream load/store units for transferring streams between off-chip memory and the SRF, memory controllers for off-chip DRAM, and optionally, an on-chip cache.

2.3.1 Bandwidth Hierarchy

The bandwidth hierarchy of a stream processor is formed by the LRFs, SRF, and the streaming memory system. Table 2.1 lists the bandwidths available at each level of the hierarchy for the Imagine [RDK+98] and Merrimac [DHE+03] stream processors. In these processors, each level of the hierarchy closer to the arithmetic units provides approximately an order of magnitude or more bandwidth than the previous level. By capturing data reuse in high-bandwidth levels, these architectures have demonstrated sustained high utilization on many arithmetic units despite limited off-chip memory bandwidth.

	Imagine (400 MHz)	**Merrimac (1 GHz)**
Local register files, aggregate (GB/s)	435.2	3072
Stream register file (GB/s)	25.6	512
Memory system (GB/s)	1.6	64

Table 2.1: Bandwidth hierarchy of Imagine and Merrimac processors

The bandwidth hierarchy of a stream processor also forms a close match to the levels of locality exposed by the stream programming model. The LRFs capture kernel locality while the SRF captures stream-level producer-consumer and temporal locality among kernels. Figure 2.6 shows how the example application of figure 2.2 maps to the bandwidth hierarchy.

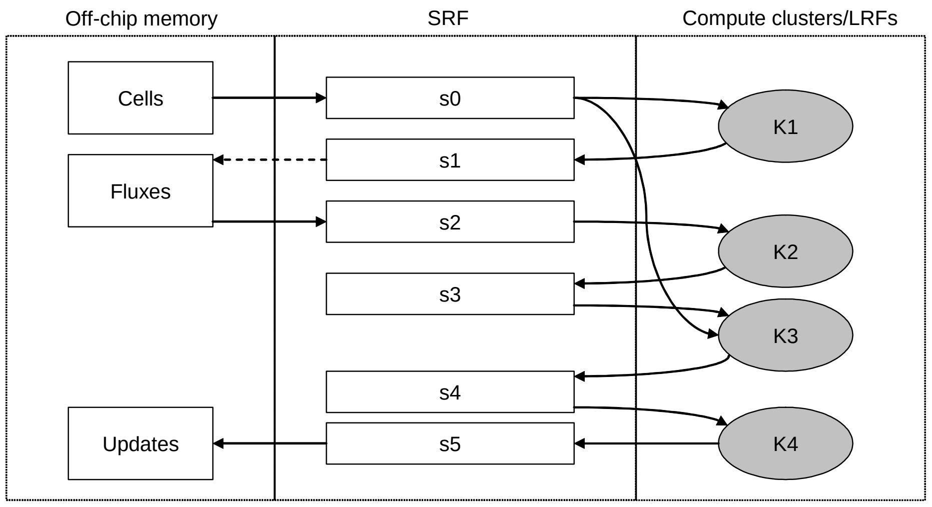


Figure 2.6: Bandwidth hierarchy mapping of FEM example. Solid arrows represent stream transfers, dotted arrows represent index transfers

Local Register Files

The total register capacity of a stream processor is divided between the SRF and LRFs. This allows the SRF to be optimized for high capacity to capture the working set while the local registers are optimized for high bandwidth, albeit with smaller capacity. In order to sustain the necessary high bandwidth, the local registers are implemented in a distributed manner, consisting of several register files in each compute cluster. Each LRF has a small number of read ports that directly feed arithmetic units, and one or more write ports that are connected to the intra-cluster network. The high bandwidth for sustaining parallel computation is provided by the aggregate of all LRFs.

The implementation described above partitions the register set of a stream processor along multiple axes – between the SRF and LRFs, among the compute clusters, and the LRFs within each cluster. [RDK$^+$00b] provides a detailed analysis of partitioned register implementations for stream processing, and shows that such architectures suffer only minimal performance loss compared to unified register architectures while achieving area and power savings of orders of magnitude. In addition, the distributed LRFs can be implemented in close physical proximity to the arithmetic units that they feed, further reducing the energy spent on operand accesses.

Stream Register File

The SRF is partitioned in to N_{cl} banks, where N_{cl} is the number of compute clusters. As describe in [RDK$^+$00b], each compute cluster is closely tied to a single SRF bank. A compute cluster and the associated bank of the SRF is referred to as a *lane* of the stream processor. A compute cluster can only access the bank of the SRF in its own lane.

The banked implementation reduces the number of SRF ports needed. Since each bank only needs to support one cluster, the SRF can be implemented using single-ported memories. In order to increase the SRF bandwidth of such an implementation, the single port of each SRF bank is several words wide. Therefore, each access reads or writes a contiguous block of words in the SRF. However, a compute cluster typically consumes or generates streams a single words at a time, but may require multiple streams to be accessed simultaneously for some kernels. This difference in SRF and compute cluster stream access characteristics is resolved by *stream buffers* as introduced in [RDK$^+$00b] (not to be confused with the cache prefetch structure of the same name described in [Jou98]). Stream buffers mediate the communication between the SRF and compute clusters as shown in figure 2.7 and provide a rate matching function between the two units. On an SRF read of stream i, a wide block of several data words is read and placed in to stream buffer i. On cluster reads of stream i, the block of data is funneled into the compute cluster a single word at a time. The process is reversed for write streams, collecting the individual words written by the clusters in to the stream buffer until sufficient data is available to perform a block write in the SRF. The stream buffers also provide the abstraction of sustaining multiple concurrent streams by time-multiplexing the single, wide SRF port among several active streams, each mapped to a different stream buffer. Access to the single SRF port among multiple active streams is managed through dynamic arbitration. Data streams are distributed among SRF banks at the granularity of records such that record r of a stream maps to SRF bank $r\ mod\ N_{cl}$. The SIMD nature of the execution applies to the SRF as well. During an SRF access, the same block is accessed within every bank.

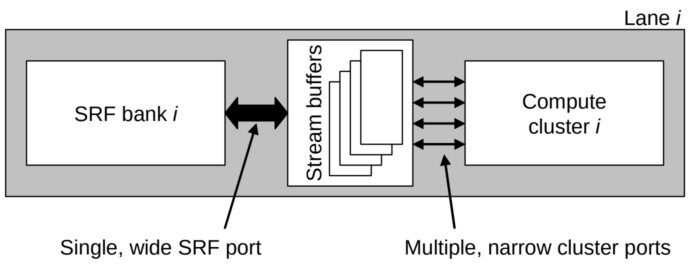


Figure 2.7: Stream buffers match the access characteristics of the SRF and compute clusters (only a single lane is shown for simplicity)

The SRF also acts as a staging area of stream transfers to and from memory. Input streams are loaded in to the SRF and held there until the kernel that consumes them is executed. Similarly, application results generated by kernels are held in the SRF until written to memory. Therefore, the SRF is a critical component in enabling the overlap of memory accesses with computation since kernels can continue to operate on data in the SRF while memory transfers take place in the background.

Streaming Memory System

The streaming memory system (SMS) manages the transfer of streams between off-chip memory and the SRF. A block diagram of the key components of the SMS is shown in figure 2.8, which includes one or more stream load/store (L/S) units, the memory switch, DRAM interfaces, and optionally, an on-chip cache.

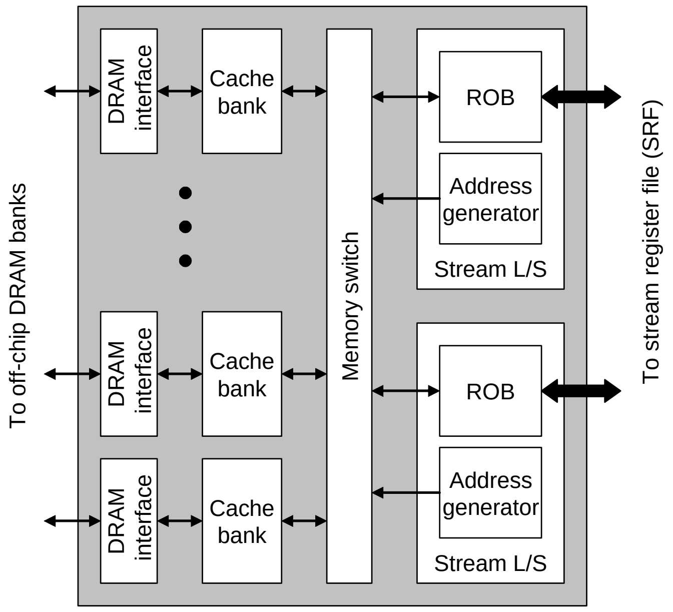


Figure 2.8: Block diagram of the streaming memory system shown with two stream load/store units and an optional on-chip data cache

Once a stream L/S units is setup for a stream transfer by the host processor, the address generator produces a sequence of memory addresses that correspond to the locations to be accessed in off-chip memory. In the case of a read stream, those locations are read, and the data is transferred to the reorder buffer (ROB) in the L/S unit. The accesses may return from the memory out of sequence, requiring the ROB to recreate the intended stream order. The ROB also acts as a stream buffer, interfacing the memory system to the SRF in much the same way the stream buffers for the compute clusters do. In the case of write streams, the data is read from the SRF via the ROB, and is written to memory. The number of ROB entries is determined by the number of memory requests necessary to be outstanding at a time in order to achieve high bandwidth utilization as described in section 2.1.

An optional cache may also be integrated in to the memory system. However, unlike caches in scalar processor architectures, the main objective of the cache is not to reduce the average memory access latency (i.e. sufficient outstanding accesses to fill the memory pipeline to off-chip memory must still be supported, even in the presence of a cache). The main purpose of the cache is to provide bandwidth filtering by capturing potentially reused data, reducing the the bandwidth that must be sustained from off-chip memory. However, since not all stream accesses may have data reuse that can be exploited through caching, the programmer or compiler may selectively specify which accesses are to be cached.

The off-chip memory of stream processors consist of commodity DRAM. The memory controllers integrate the DRAM controllers on chip, and several DRAM channels are supported to increase bandwidth. The memory interfaces also implement *memory access scheduling* to optimize the performance of the off-chip DRAMs [RDK$^+$00a].

Despite the presence of an efficient bandwidth hierarchy, performance of applications (or segments of applications) may still be constrained by bandwidth if the computation performed per unit of data accessed at any level of the hierarchy is less than what can be sustained by the processor architecture. This is a particular concern for off-chip memory, which forms the lowest bandwidth level of the hierarchy. Therefore, a key focus of this thesis is to identify and evaluate enhancements to on-chip levels of the bandwidth hierarchy that allow more efficient utilization of scarce off-chip memory bandwidth by eliminating non-essential accesses.

2.4 Alternative Data Parallel Architectures

The following sections briefly introduce vector and multi-context architectures and CPU media extensions – three commonly used alternatives to stream processors for exploiting data parallelism. A more comprehensive discussion of the similarities and relative strengths of these architectures from a bandwidth hierarchy perspective will be presented in chapter 3.

2.4.1 Vector Processors

Vector processing expresses an application as sequence of *vector operations* operating on *vectors* of data. A vector is an array of one-word data elements of a primitive type, and a vector operation is a single arithmetic operation such as an add or a multiply that is applied to all elements of its input vectors. Figure 2.9 revisits the simple computation of figure 2.1, but expressed as a sequence of vector operations. Conceptually, a single operations is applied to all elements of a vector before any operations are applied to the resulting output. This differs from stream processing where, conceptually, all operations on a single input stream element are applied before any operations are applied to the subsequent input elements. Therefore, vector processing differs from stream processing on the granularity at which the data-parallel computation is expressed.

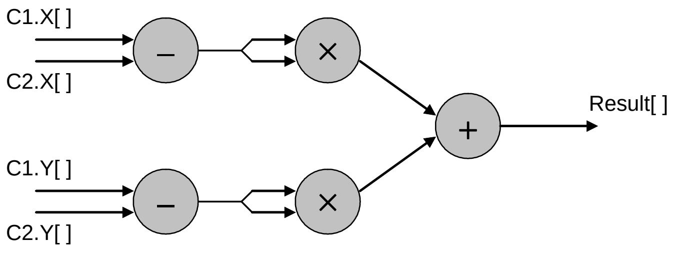


Figure 2.9: Vector processing representation of squared distance computation

Vector processing was originally proposed in the context of high-performance supercomputers [Rus78]. Modern implementations such as Cray X-1 [Cra02] and NEC SX family [KTHK03] continue to target scientific computing. However, the applicability of vector processing to other data parallel application domains, such as media processing, has been demonstrated [Koz02].

Modern vector processors share many implementation similarities with stream processors. Arithmetic units within a vector processor are grouped in to *vector pipes*. Multiple vector pipes may operate in parallel using SIMD execution, operating on independent vector element similarly to the compute clusters of a stream processor. Vectors are stored in a *vector register file* (VRF) that captures the intermediate results between vector operations. The VRF is multi-banked with independent banks associated with vector pipes, each bank providing high bandwidth access to its associated pipe only. Vector processors also exploit ILP, executing multiple vector instructions in parallel. A technique known as *chaining*, first introduced in the Cray-1 [Rus78], is used to forward results between arithmetic units within the same pipe that are executing different vector instructions in parallel. Simple chaining requires that the chained instructions be carefully synchronized, but requires no significant buffering of intermediate results or additional ports to the VRF. A more general extension of the idea, *flexible chaining*, relaxes the synchronization constraints at the cost of additional buffering and/or VRF ports for chained intermediate results.

Most vector architectures rely on the VRF to both provide the bandwidth necessary to sustain computation and to capture the working set. However, some vector architectures share a further commonality with stream processors in that they employ a 2-level register hierarchy similar to the SRF and LRFs. For example, the NEC SX family of vector processors partition the register space in to a small number of *vector arithmetic registers* (8 256-element vectors in the case of the SX-6) and a larger number of *vector data registers* (64 256-element vectors in the SX-6) [SX-02]. The vector arithmetic registers provide the operand bandwidth for the arithmetic units. The vector data registers cannot be directly operated on, and must be explicitly copied to an arithmetic register before being operated on. However, results of vector operations may be directly written to both types of registers.

A detailed discussion of vector processors and their design tradeoffs can be found in [Asa98].

2.4.2 Multi-threaded Processors

Multi-threaded architectures can also be used to exploit data parallelism. Conceptually, every iteration of a data parallel loop can be viewed as a separate thread or context, enabling stream programs to be mapped easily to such architectures. Therefore, long memory latencies can be overlapped with computation on other contexts which operate on independent data elements. Parallel execution among data elements may be achieved in multi-threaded processors using techniques such as *simultaneous multi-threading* [TEL98], which executes instructions from multiple threads concurrently on a single processor, or by integrating multiple processor cores on a single chip.

2.4.3 Media Extensions to General-purpose CPUs

Virtually all major general-purpose CPU families have incorporated some form of support for exploiting data parallelism targeted at media applications [PW96; TONH96; DDHS00; Lee96]. These techniques enable a small number of SIMD computations to be specified using a single instruction to be executed on data read from a wide register file. However, these extensions do not exploit data parallelism to the degree that specialized data parallel architectures such as stream or vector processors do. Media extensions provide neither special support for overlapping computation with memory accesses in order to tolerate memory latency (beyond what support exists for scalar computation in the CPU), nor do they provide a specialized bandwidth hierarchy to sustain computation on very large numbers of arithmetic units. Further, each instruction, which specifies only a small number of operations, must be issued via the out-of-order scalar issue mechanism present in most modern processors, incurring high energy overheads.

2.5 DRAM Characteristics

Most modern data parallel architectures use commodity DRAM for main memory due to cost, density, and power considerations. This section provides a brief introduction to DRAM access timing and characteristics in order to understand the impact on the performance of stream accesses from off-chip memory.

Modern DRAM memories, such as DDR2 SDRAM and DRDRAM, contain multiple internal banks, and the cells within each bank are arranged as a 2D array with sense amplifiers along one edge (or two parallel edges) as shown in figure 2.10. Addressing for DRAM access is done in two stages. During *row access* (also referred to as *activate*), an entire row of the array is read via the sense amplifiers and stored in latches called the *row buffer*. Data in the row buffer is referred to as the *active row*, and portions of it are read or written using *column accesses*. Therefore, typical DRAM access consists of a row access followed by one or more column accesses. When data from a different row of the DRAM is needed, the currently active row is *closed*, i.e. written back to the DRAM cells, and the array bitlines precharged to prepare for the next row access. Typically, new row activations take significantly longer than a column access to an already active row. Some critical latencies for a current DDR2 SDRAM part [Mic03] are summarized in table 2.2. Figure 2.11 shows a simple example of a sequence of accesses to a single DRAM bank and the ensuing operations. As can be seen from this simple example, the bandwidth and the distribution of access latencies of DRAMs depend heavily on the access pattern, which varies the mix of row and column accesses.

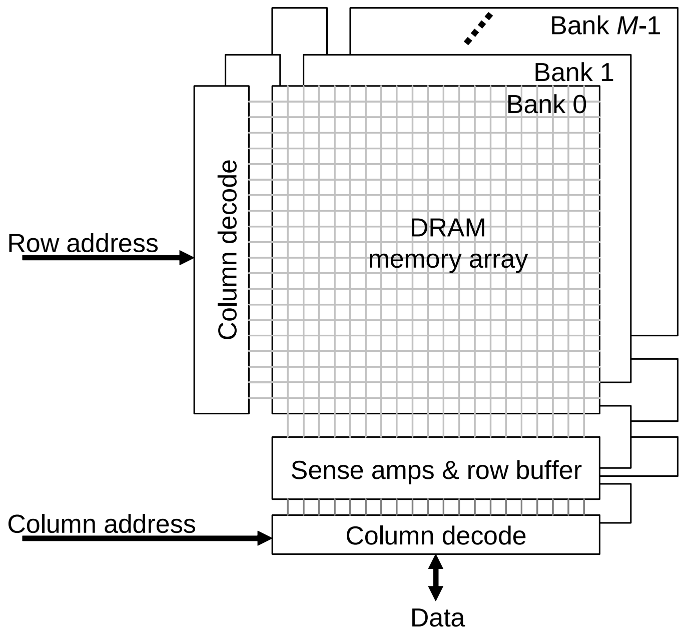


Figure 2.10: Internal structure of a modern DRAM

DRAMs support a few options to reduce the impact of long precharge and row access latencies:

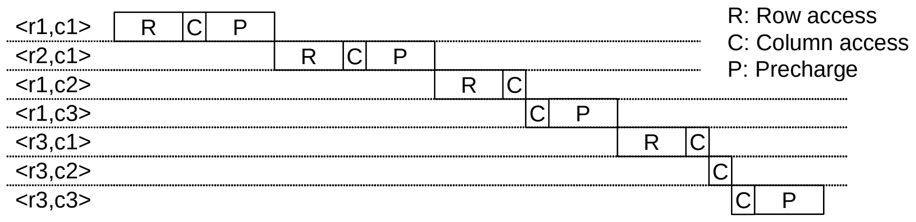


Figure 2.11: Example DRAM access operations schedule

Operation	Latency
Minimum time from row access to column access (t$_{RCD}$)	20ns
Column access to data read time (CL)	20ns
Minimum time between successive column accesses (t$_{CCD}$)	10ns
Minimum time from precharge to next activate (t$_{RP}$)	20ns
Minimum time between row accesses to same bank (t$_{RC}$)	65ns

Table 2.2: Example SDRAM access latencies (Micron 256Mb DDR2, speed grade -5)

- **Burst mode** rapidly accesses multiple consecutive data elements from the active row following a single column access command. Such accesses provide high throughput for sequential access patterns.

- **Multiple banks** within a DRAM chip allow overlapping of commands to different banks. While the address pins of the DRAM are occupied during the communication of each command, this approach enables some fraction of the long latency precharge and row activations to be overlapped with operations in other banks.

- **Open row policies** maintain the active row after pending access(es) complete, anticipating subsequent accesses to map to the same row. If this is indeed the case, row accesses are avoided. If subsequent accesses map to other rows, both precharges and row accesses are incurred. Alternatively, **closed row policies** proactively close the active row and precharge the array when pending accesses to the active row complete. This avoids the precharges but incurs row accesses for subsequent operations regardless of the row they map to.

2.6 Emerging Trends

Several trends in applications and implementation technology make memory system design even more crucial for stream processing.

While stream processors were originally proposed to handle media and signal processing applications with straightforward data access patterns, there have been recent efforts to extend stream processing to more complex applications in signal processing as well as other application domains such as scientific computing [Raj04; DHE$^+$03]. These applications require the memory system to support a wider range of access patterns. Therefore, extensions to the streaming bandwidth hierarchy that efficiently support a wider range of accesses enables the high performance potential of stream processors to be realized over a larger class of applications.

At the same time, implementation technology scaling continues to exponentially increase the number of devices available on a single chip, increasing the potential compute capability of processors. However, off-chip memory bandwidth and latency are scaling at a much lower rate [Pat04], increasing the demands on the bandwidth hierarchy to make ever more efficient use of the scarce off-chip bandwidth. A number of tradeoffs need to be evaluated in order to determine how best to use a storage hierarchy to achieve the best gains in performance, energy efficiency, and/or off-chip memory bandwidth reductions for stream processors.

The increasing number of devices also makes it possible to integrate stream processors or other data parallel processors along with general purpose CPUs on the same die in order to accelerate media processing and other similar tasks [EAE$^+$02; Bor04]. In such implementations, the the data parallel processor must share portions of the memory hierarchy designed for the general-purpose processor. Insights drawn from dedicated stream processor memory hierarchies can be used to develop techniques to adapt the general-purpose memory hierarchies to better suit stream processing in these implementations.

2.7 Summary

This chapter was intended to provide an overview of concepts central to stream processing, and a brief introduction to stream processing itself. We explored the characteristics of data parallel application classes that make them ideal candidates for exploiting the raw potential of today's VLSI circuit technology. The high degree of parallelism available in these applications were identified as the critical aspect that enables parallel computation on many arithmetic units while tolerating the long off-chip memory access latencies of modern processors.

The stream programming model was introduced as a way of expressing data parallel applications in a manner that exposes the inherent parallelism as well as the available data locality. This programming model enables filtering of the high operand bandwidth needed to sustain computation by capturing the exposed locality at multiple levels of a bandwidth hierarchy. Stream processors, a class of architectures optimized for executing applications expressed using the stream programming model was also introduced in this chapter, along with a discussion of the specific implementation of the bandwidth hierarchy in these processors.

This chapter also provided a brief overview of alternative programmable data parallel architectures, such as vector and multi-threaded processors. We will expand on this introduction in terms of the relative strengths of these architectures from the perspective of bandwidth hierarchy design in chapter 3. A short introduction was also presented to modern DRAMs, a crucial component in the memory systems of most data parallel architectures. Finally emerging trends in applications and implementation technology that encourage further research on memory systems for stream computing was presented.

The next several chapters of this thesis will build on the background provided in this chapter to identify opportunities to broaden the applicability of stream processing, and to further improve the performance and implementation efficiency of bandwidth hierarchies for data parallel architectures.

Chapter 3

Granularity of Data Parallelism

An important aspect of data parallel computing is the granularity at which the parallelism is expressed. In the case of a stream processing, data parallelism is expressed at the granularity of stream elements, or entire records. An alternative is to express the parallelism at the granularity of individual words, as is done in vector processing. This distinction leads to two important implications for memory hierarchy design. First, since the order of operations differs in these two cases as described in section 2.4.1, the amount of intermediate state generated, and hence the amount of register space needed to store that state, is different. Second, the order in which the data words are accessed from memory is different in the two cases, leading to differences in the access patterns seen by the memory hierarchy. This chapter explores the impact that expressing data parallelism at record and word granularity has on both these aspects.

3.1 Register Capacity Requirements

The differences in the amount of register capacity required by stream (i.e. record granularity) and vector (i.e. word granularity) computation can be analyzed using a simple analytical model. In both cases, the register capacity demands must satisfy two requirements in order to achieve high performance. First, the intermediate state of the application loop body being executed must be held in the registers. Second, the registers must provide sufficient space to stage data transfers to and from memory. This enables computation to take place on data already in the registers while inputs for future computations are being prefetched, effectively overlapping memory latency with computation.

In the case of vector execution, the intermediate results generated within the loop body are expanded out by the vector length since they are computed for the entire vector(s). Therefore, the register capacity required for vector computation is given by C_v of equation (3.1) where S_{max} is the maximum concurrent intermediate state required at any point during a single iteration of the loop body, L_{vec} is the vector length, B_M is the off-chip memory bandwidth, and T_M is the off-chip memory access latency. Note that L_{vec} is typically determined by the ratio of vector data bandwidth to instruction issue rate. The number of operations specified by each vector instruction, which is equivalent to L_{vec}, must be sufficiently large to fully consume the available vector register bandwidth at the vector instruction issue rate of the processor.

$$C_v = S_{max} \times L_{vec} + B_M \times T_M \tag{3.1}$$

Stream computation requires intermediate state to be maintained in LRFs for only the current iteration being executed. However, further state is required in the SRF for the input and output streams. Therefore, total register capacity in LRFs and SRF for stream order computation is given by C_s of equation (3.2) where S_{IO} is the input and output data elements per iteration of the loop, L_{str} is stream length, and S_{max}, B_M, and T_M are the same as in the vector case above.

$$C_s = S_{max} + S_{IO} \times L_{str} + B_M \times T_M \tag{3.2}$$

3.1.1 Impact of Parallel Execution

The above expressions assume no exploitation of parallelism in hardware. However, modern implementations of stream and vector architectures exploit data parallelism and ILP as described in chapter 2. SIMD execution of multiple parallel clusters increases LRF capacity requirements for streaming by a factor of N_{cl}, the number of compute clusters. Total intermediate state requirements for stream-order computation with the additional requirements for parallel execution is given by equation (3.3).

$$C_s = S_{max} \times N_{cl} + S_{IO} \times L_{str} + B_M \times T_M \tag{3.3}$$

Finally, loop unrolling and software pipelining is often used when scheduling kernels to achieve high functional unit utilization. In the case of vector architectures, unrolling and pipelining is done to the extent necessary to utilize multiple units within each vector pipe. Subsequent operations issued to the same functional unit do not often require independence for long vectors since a single vector instruction occupies the same unit for several cycles. For example, with a vector length of 256 and 8 parallel vector pipes, a single instruction occupies the functional units for $L_{vec}/N_{pipes} = 32$ cycles, which is sufficient for even a heavily pipelined functional unit to start generating results.

In the case of stream computing, intermediate results are computed only on a single element, and the results may be needed immediately by other dependent operations scheduled on the same compute cluster. In addition, multiple instructions may be issued in parallel to different arithmetic units within each cluster, similar to vector architectures. Therefore, the amount of ILP needed for high utilization of a stream processor is proportional to the product of the number of units within a cluster and the pipeline latency of the units. As a result, stream computing is likely to require a greater degree of loop unrolling and software pipelining compared to vector processing, leading to increased register requirements. Equations (3.4) and (3.5) reflect the impact of loop unrolling and software pipelining. P_{vec} and P_{str} denote the increase in register capacity requirements due to software pipelining and loop unrolling for the vector and stream cases respectively.

$$C_v = P_{vec} \times S_{max} \times L_{vec} + B_M \times T_M \tag{3.4}$$

$$C_s = P_{str} \times S_{max} \times N_{cl} + S_{IO} \times L_{str} + B_M \times T_M \tag{3.5}$$

Figure 3.1 shows C_v normalized to C_s over a range of values for S_{max} and P_{str}. The remaining parameters are fixed as listed in the figure caption, with hardware parameters such as L_{vec} based on current state-of-the-art implementations. Vector processing clearly requires more register storage due to the expansion of intermediate state by vector length except at very high values of P_{str}/P_{vec}. In reality, however, the degree of loop unrolling and software pipelining necessary to achieve the desired level of parallelism is often small, as can be seen from table 3.1 which lists the degrees of loop unrolling and software pipeline depths encountered in representative stream implementations of a few applications. The factor of register capacity increase resulting from software pipelining is often lower than the depth of the pipeline itself. Therefore, exploiting data parallelism at record granularity requires significantly less intermediate register state than doing so at word granularity.

Application	**Degree of Loop Unrolling**	**Software Pipeline Depth**
2D FFT	1	2 to 3
Rijndael encryption	1	2
MPEG Encode	1	2 to 3
2D Finite element method	1	2 to 6
3D Finite element method	1	2 to 5
Molecular dynamics	1	3

Table 3.1: Loop unrolling and software pipeline depths of application kernels for stream processing

3.1.2 Discussion

The above analysis compares the intermediate state requirements of record and word granularity exploitation of data parallelism, which closely parallel the current implementations of stream and vector processors respectively. However, the above analytical models also indicate some similarities between practical implementations of

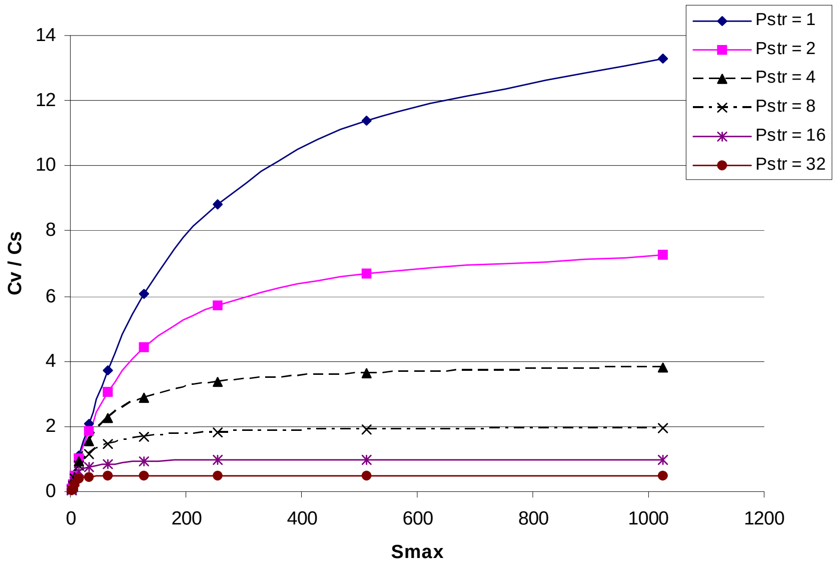


Figure 3.1: Vector register capacity requirements relative to stream registers using realistic parameters: $P_{vec} = 1$, $F = 4$, $L_{vec} = L_{str} = 256$ elements, $B_M = 8$ words/cycle, $T_M = 200$ cycles, $N_{cl} = 8$, $S_{IO} = 10$
.

stream and vector processors. In particular, N_{cl} of equations (3.3) and (3.5) closely corresponds to L_{vec} of equations (3.1) and (3.4) in that they are both multipliers of intermediate state. This reflects that fact that the SIMD execution across clusters of a stream processor, which provides high performance by exploiting data parallelism, results in word granularity computation over N_{cl} data elements. Therefore, the lower intermediate register requirements of stream processors is achieved by reducing the degree of word granularity parallelism relative to typical vector processors. N_{cl} of current stream processors is typically 8 to 16, while L_{vec} of vector architectures is typically 64 to 256 elements.

The above reasoning also implies that intermediate state requirements of vector processors may also be reduced by shortening the vector length during computation. An inner-loop would perform the computation on short vectors of length L_{svec} ($<< L_{vec}$) while an outer loop will sequence the short vector operations over long vectors of length L_{vec} fetched from memory. Such a vector processor has intermediate state requirements identical to a stream processor with *svec* number of SIMD clusters. However, operating on short vectors increases vector instruction issue requirements by a factor of L_{vec}/L_{svec}, increasing the complexity of the issue mechanism. Limited vector instruction issue bandwidth is often a key constraint dictating the long vectors used in current vector processor implementations. A potential solution may be to incorporate a dedicated sequencer similar to the microcontroller of a stream processor to issue vector instructions.

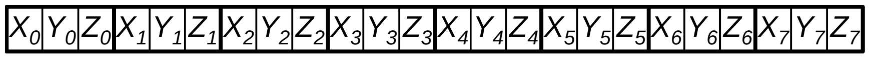
3.2 DRAM Performance

The second memory hierarchy difference between record and word granularity expression of data parallel computation is the performance of the memory system under the ensuing access patterns. As discussed in section 2.5, the performance of modern DRAMs vary based on the ordering of the access sequence. Applications group related data items into records, and streams access entire records at a time, exposing the application-level locality among fields of a record to the memory system. However, vector accesses obscure this record spatial locality by accessing sequences of individual words, typically picking out the same field of each record within a collection.

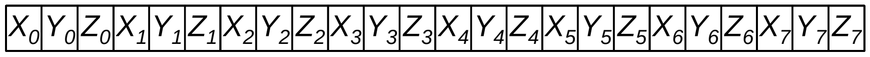
Vector and stream memory accesses that appear in applications can broadly be classified in to three categories. *Unit strides* sequentially access every vector or stream element within a contiguous range. *Non-unit strides* access every S^{th} element, where S (> 1) is referred to as the stride. *Indirect* accesses are to arbitrarily ordered sequences of elements within a finite range, with the order of accesses usually specified by an explicit sequence of indices. DRAMs perform well under unit stride accesses since the resulting contiguous memory access can benefit from bursts and repeated accesses to a row once it is activated. Non-unit strides and indirect accesses perform less efficiently as bursts are sub-optimal for non-contiguous accesses, especially in the case of vectors. In addition, as the stride or the sparseness of the index pattern increases, the number of column accesses per row activation decreases, reducing effective bandwidth.

Vectors and streams lead to identical accesses for single-word data types, but yield different access patterns for multi-word records. Figure 3.2(a) shows a simple example of an array of 3-word records stored in memory in *record order* (i.e. each record stored contiguously). A unit-stride stream access to this array results in a sequential memory access as shown in figure 3.2(b). Accessing the same data as vectors results in three accesses of stride 3 as shown in figure 3.2(c). Not only does this vector access lack spatial locality, but if the data structure spans over multiple DRAM rows, each row is touched by all of the vector accesses, potentially leading to increased DRAM row activations.

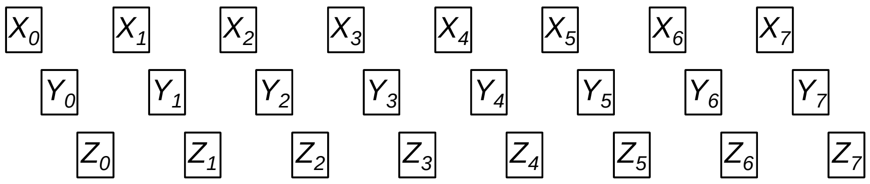
The vector access can be optimized by reordering the data in memory to be in *vector order* as shown in Figure 3.3(a), where the i^{th} word ($1 \leq i \leq$ record size) of all records are placed contiguously in memory. This results in 3 unit-stride vector accesses as in figure 3.3(b), achieving memory performance comparable to the stream access. However, rearranging data layout in memory may not be globally optimal if the same data is accessed in multiple patterns. This is a particular concern if the data is accessed by both the scalar and vector processors since scalar accesses rely heavily on cache lines to capture spatial locality. For example, a scalar processor access to a record in figure 3.3(a) could potentially cause as many cache misses as the record length.


X0 Y0 Z0 X1 Y1 Z1 X2 Y2 Z2 X3 Y3 Z3 X4 Y4 Z4 X5 Y5 Z5 X6 Y6 Z6 X7 Y7 Z7

(a) Memory data layout

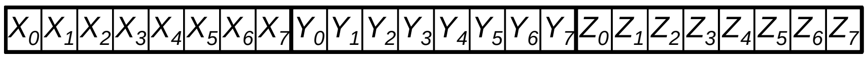

X0 Y0 Z0 X1 Y1 Z1 X2 Y2 Z2 X3 Y3 Z3 X4 Y4 Z4 X5 Y5 Z5 X6 Y6 Z6 X7 Y7 Z7

(b) Single stream access

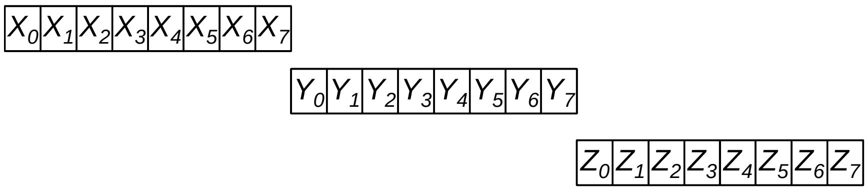


(c) Three stride 3 vector accesses

Figure 3.2: Stream and vector access to linear array of 3-word records in memory


X0 X1 X2 X3 X4 X5 X6 X7 Y0 Y1 Y2 Y3 Y4 Y5 Y6 Y7 Z0 Z1 Z2 Z3 Z4 Z5 Z6 Z7

(a) Memory data layout optimized for vector access

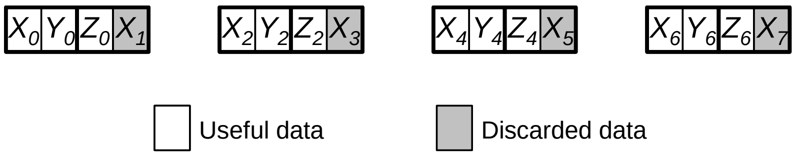


(b) Three unit stride vector accesses

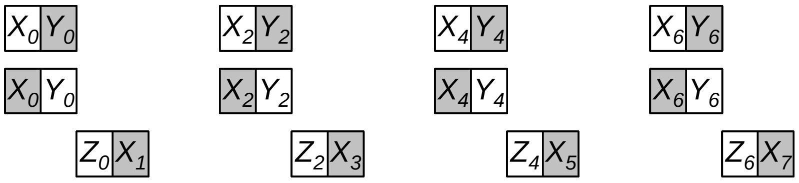
Figure 3.3: Array of 3-word records optimized for vector access

Strides and Multi-word DRAM Bursts

A stride 2 access to the array of figure 3.2(a) using a DRAM burst length of 2 words is shown in figure 3.4. The stream access uses a single burst to fetch the first two words of a record. A second burst yields one useful word and an extra word that is discarded. The vector access does not benefit from bursts, and in fact suffers since half the data returned by each burst is discarded. Note that placing the data in vector order in memory as in figure 3.3(a) does not alleviate this problem since the accesses would still be non-contiguous. A similar reasoning applies to indirect accesses as well.



(a) Stream access (burst = 2 words)



(b) Vector access (burst = 2 words)

Figure 3.4: Stride 2 access to array of 3-word records using 2-word DRAM bursts

3.2.1 Impact of Caches

Multi-word cache lines can capture spatial locality of records ignored by vector accesses such as in figure 3.4(b). However, the effectiveness of cache lines in capturing locality in non-contiguous access patterns becomes suboptimal as the record size deviates from the cache line size or a multiple of it. Figure 3.5 shows the 3-word record access example with a stride of 3 using 4-word cache lines. Since the cache line size is larger than the record size, additional data is fetched, increasing DRAM bandwidth demands. A similar increase in DRAM traffic can be expected in performing the same access in stream order when using a cache. However, a stream access does not require a cache to be used to capture spatial locality, and can be performed as an uncached access if no temporal reuse of the data is expected.

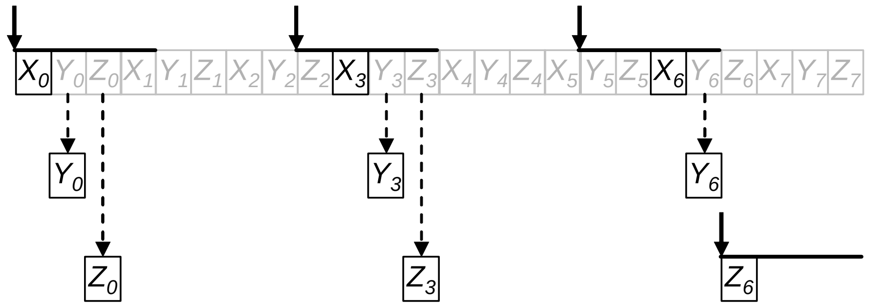
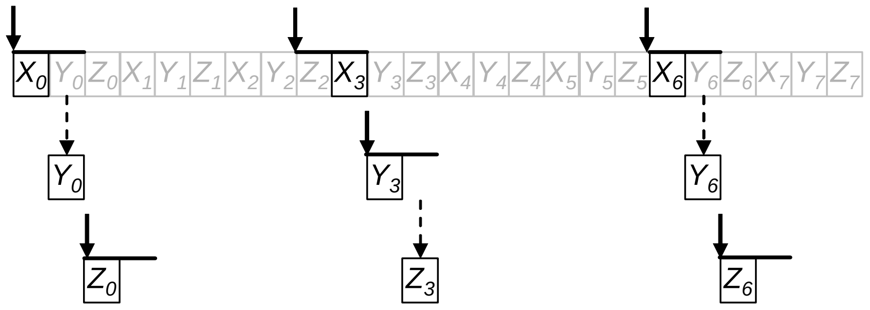

X_0 Y_0 Z_0 X_1 Y_1 Z_1 X_2 Y_2 Z_2 X_3 Y_3 Z_3 X_4 Y_4 Z_4 X_5 Y_5 Z_5 X_6 Y_6 Z_6 X_7 Y_7 Z_7
Y_0
Y_3
Y_6
Z_0
Z_3
Z_6

Figure 3.5: Strided vector access with larger cache lines than data record size. Solid arrows indicate cache line fetch initiations, dotted arrows represent cache hits due to previous fetches, and horizontal dark lines indicate the spans of cache lines

Figure 3.6 illustrates the impact of a cache line size that is shorter than the record size. While the cache lines do capture a subset of the spatial locality available in this case, the additional row activations due to multiple vector accesses are not entirely eliminated as each record incurs multiple cache misses on different vector accesses.

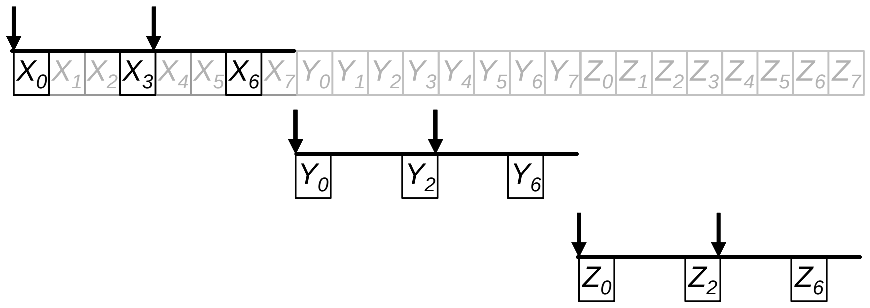
Figure 3.7 illustrates the impact of vector-optimized data layout on cache line access efficiency using the same 3-word record access with stride 3. Note that the optimized layout for this access results in 6 cache misses compared to 4 misses for the unoptimized layout of figure 3.5. This increase in cache misses as a result of vector order data layout occurs in cases where $(cache\_line\_size)/stride$ is less than



X_0 Y_0 Z_0 X_1 Y_1 Z_1 X_2 Y_2 Z_2 X_3 Y_3 Z_3 X_4 Y_4 Z_4 X_5 Y_5 Z_5 X_6 Y_6 Z_6 X_7 Y_7 Z_7
Y_0
Y_3
Y_6
Z_0
Z_3
Z_6

Figure 3.6: Strided vector access with shorter cache lines than data record size. Solid arrows indicate cache line fetch initiations, dotted arrows represent cache hits due to previous fetches, and horizontal dark lines indicate the spans of cache lines

the record size since each cache line yields fewer useful words.



X_0 X_1 X_2 X_3 X_4 X_5 X_6 X_7 Y_0 Y_1 Y_2 Y_3 Y_4 Y_5 Y_6 Y_7 Z_0 Z_1 Z_2 Z_3 Z_4 Z_5 Z_6 Z_7
Y_0
Y_2
Y_6
Z_0
Z_2
Z_6

Figure 3.7: Strided vector access with cache and vector-optimized data layout. Arrows indicate cache line fetch initiations and horizontal dark lines indicate the spans of cache lines

In summary, as illustrated by the above examples, cache lines capture all or a subset of the spatial locality ignored by vector accesses to records. However, a number of potential inefficiencies remain due to mismatches between record size and cache line size. In addition, vector-optimized data layouts, which do not retain the record locality of the data, are often sub-optimal for cached accesses.

3.2.2 Evaluation Methodology

Latency of DRAM accesses are highly dependent on the sequence of addresses being accessed and on a number of complex timing parameters as described in section 2.5. The access characteristics are further complicated by the reordering of accesses performed for memory access scheduling to optimize DRAM throughput [RDK+00a]. Therefore, a set of microbenchmark simulations are used instead of an analytical model to analyze the performance impact of vector and stream order accesses. Subsequently, a set of application benchmarks will also be evaluated.

The memory systems used for this evaluation are summarized in table 3.2. The *Full* configuration is based on the memory system proposed for the Merrimac streaming super computer [DHE+03] and is used for scientific benchmarks. The scaled down *Lite* version is used for media and signal processing benchmarks. Both memory systems are evaluated with on-chip caches, as may be the case with high-performance implementations, and without caches, as may be the case with low-cost or embedded implementations. A word size of 64 bits is used in all configurations.

	Full	**Lite**
Physical DRAM banks	16	8
Internal banks per DRAM	16	
DRAM page size	2KB	
DRAM transfer width	128 bits (2 64-bit words)	
DRAM burst size	128 bits (2 64-bit words)	
Memory bank interleaving factor	Same as burst size	
Peak memory bandwidth (GB/s)	38	19
DRAM row policy	Closed row	
Cache size (KB)	512	256
Cache banks	8	4
Cache line size	128 bits (2 words)	
Cache associativity	8	4
Peak cache bandwidth (GB/s)	64	32

Table 3.2: Memory system parameters for stream and vector access comparison

The results were obtained using cycle accurate simulations of the memory systems. Data-parallel execution is simulated (cycle-accurate) using the stream processors summarized in table 3.3. The stream access results presented are based on the actual accesses generated by the processors. Two types of vector accesses were considered for each benchmark. The first, *Vector*, is generated by rearranging the accesses from the stream processors to be in vector order (i.e. access i^{th} word of all records before accessing $(i+1)^{th}$ word of any record) with the same data layout in memory as the stream case. The second set of vector data, *Optimized Vector* (or *OptVec*), is generated by optimizing the data layout in memory for vector accesses by placing the i^{th} word of all records contiguously in memory as in the example of figure 3.3. In addition, memory access scheduling with a closed row policy is used to optimize the ordering of pending accesses in the memory controllers to improve DRAM throughput for all configurations. These results do not include the effects of accesses from scalar processors for both stream and vector accesses.

	Full	**Lite**
Clock frequency	1GHz	
Compute clusters (lanes)	16	8
Peak compute (GFLOPs)	128	64
Peak memory request (address) bandwidth	8 requests per cycle from 2 independent sequences	4 requests per cycle from 2 independent sequences

Table 3.3: Processor parameters for stream and vector access comparison

The above methodology does not reflect the differences in programming model, maturity of compiler tools, and programmer effort involved in targeting vector and stream processors. We also assume that the VRF is of comparable capacity and bandwidth to the SRF/LRF of the stream counterpart despite the implementation inefficiencies of such a VRF as mentioned in chapter 2. By factoring out these differences, this study provides a clear comparison of memory system performance for a given set of accesses when performed as streams and vectors.

3.2.3 Microbenchmarks

The microbenchmarks used for this evaluation are summarized in table 3.4. Results were obtained on the *Full* memory system using only a single address sequence in order to highlight memory behaviors without inter-sequence interference (note, however, that this limits the peak address bandwidth to 4 requests per cycle). Each benchmark performs a 16K word access, and all data in this subsection are normalized to the throughput of the *Seq* benchmark with a record size of 1 word.

Benchmark	**Access Pattern**	**Record Size (words)**	**Stride (records)**	**Cached**
Seq	Sequential sweep over array	1 to 64	1	No
SeqC	Sequential sweep over array	1 to 64	1	Yes
StrideRLn	Strided sweep over array	n	1 to 32	No
StrideCRLn	Strided sweep over array	n	1 to 32	Yes
IndirectRLn	Pseudo-random indirect access	n	-	No
IndirectCRLn	Pseudo-random indirect access	n	-	Yes

Table 3.4: Microbenchmarks used to evaluate memory system performance of stream and vector accesses

Figures 3.8 shows the performance of *Seq*. Stream access achieves high bandwidth for all record sizes except around multiples of 32. This is due to the fact that stream access multiplexes 16 (i.e. the number of arithmetic clusters) record accesses in round-robin fashion so that all clusters receive the same amount of data. With data interleaved at 2-word granularity among 16 memory controllers, a record size of 32 or a multiple causes the worst case bank conflicts among these parallel record accesses. *Vector* performs poorly for all record sizes other than 1 since the data layout in memory is in record order leading to strided accesses. Record sizes that are multiples of four suffer further since the striding causes bank conflicts. *OptVec* performs well for all record sizes as the data is laid out to enable unit stride accesses. Figure 3.9 shows the performance of *SeqC*. *Stream* and *OptVec* performance is largely unchanged with the addition of caching as the access pattern has no temporal locality, providing no additional benefit for these accesses that already exploit the available spatial locality.

The slight degradation in *Stream* access performance with a cache is due to the added contention at the cache bank inputs as a result of the 16 concurrent record accesses described above. *Vector* performance improves as multi-word cache lines capture the spatial locality otherwise lost by vector accesses.

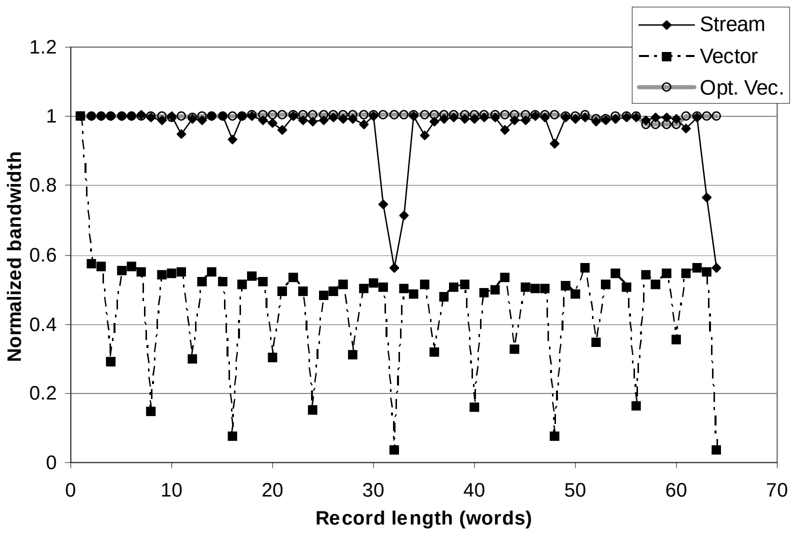


Figure 3.8: Uncached sequential access (*Seq* microbenchmark) performance under *Stream*, *Vector*, and *Optimized Vector* accesses as record size varies (normalized to *Seq* with 1-word records)

Figure 3.10 shows the performance of *StrideRLn* for $n = 2$ and 5. As can be expected, all access types suffer from bank conflicts when the *effective stride* is an even multiple of the burst size (note that addresses are distributed among DRAM banks at the granularity of the burst size). The effective stride is the product of the stride in records and the contiguous block of data per record as laid out in memory. Therefore, for *Stream* and *Vector* cases, the effective stride is $stride\_in\_records \times record\_size\_in\_words$. For the *OptVec* case, the effective stride is simply the stride in records since the records are not laid out contiguously. Therefore, *OptVec* suffers from bank conflicts less frequently than the other two cases. However, *Stream* outperforms *OptVec* for odd record sizes since the contiguous record access makes use of the 2-word burst while *OptVec* only uses 50% of each burst for all strides > 1. For even

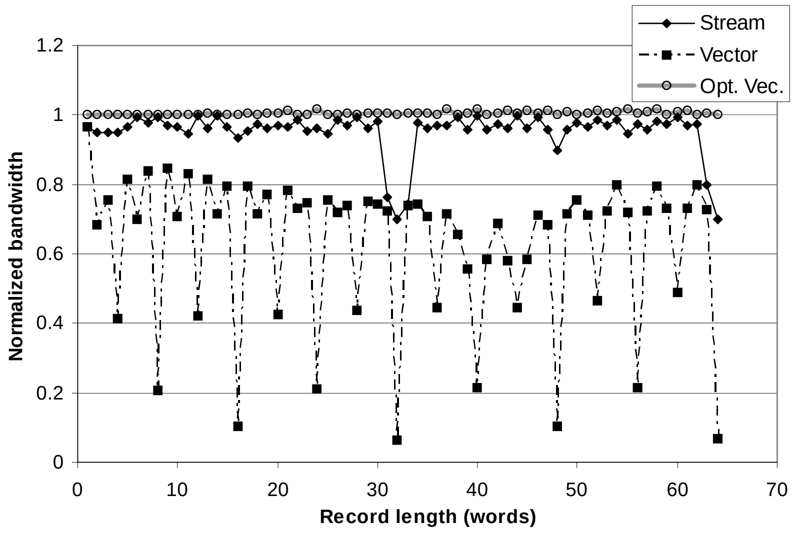
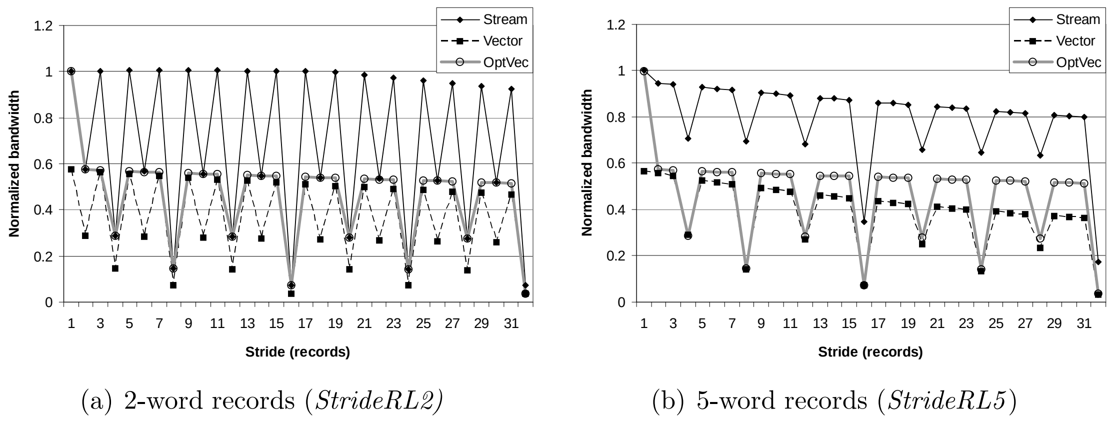


Figure 3.9: Cached sequential access (*SeqC* microbenchmark) performance under *Stream*, *Vector*, and *Optimized Vector* accesses as record size varies (normalized to *Seq* with 1-word records)

record sizes, the increased bank conflicts of *Stream* compensate for the reduced burst utilization of *OptVec*, resulting in similar performance. *Vector* is affected by both increased bank conflicts and 50% burst utilization resulting in poor performance.

Unutilized burst bandwidth shows up as a static offset from 100% normalized bandwidth. For example, *Vector* and *OptVec* lose nearly 50% of the bandwidth due to this reason for both cases shown in figure 3.10. *Stream* loses slightly less than 10% of peak due to unutilized burst bandwidth in the case of *StrideRL5*. Note that the bandwidth loss is slightly less than the fraction of unutilized burst bandwidth. This is due to the fact that the memory system is not 100% utilized by *Seq* with a record size of 1, to which these results are normalized, because only one of the two address sequencers are used in these microbenchmarks.

The gradual reduction in bandwidth with increasing stride results from fewer column accesses per row activation. This effect is not visible for *Stream* in *StrideRL2* at short strides since the memory system bandwidth is sufficient to saturate the address bandwidth for these cases. Note that this drop-off is proportional to the effective stride, and stabilizes when each row activation yields only a single data element (the figures do not extend to large enough strides to show stabilization). Strided *OptVec* performance is insensitive to record size since it does not affect the effective stride.

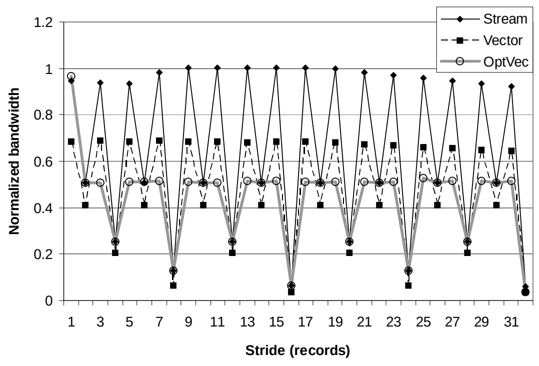


(a) 2-word records (*StrideRL2)* (b) 5-word records (*StrideRL5*)

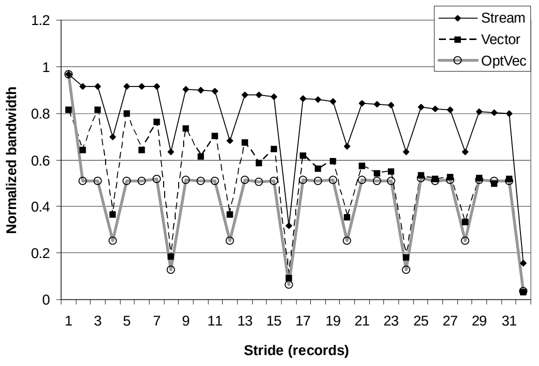
Figure 3.10: Uncached strided access performance under *Stream*, *Vector*, and *Optimized Vector* accesses as stride varies (normalized to *Seq* with 1-word records)

Figure 3.11 shows the performance of *StrideCRLn* for $n = 2$ and 5. Similar to the sequential access microbenchmarks, the main difference in these results compared to the uncached case is the significant improvement in *Vector* performance as cache lines capture spatial locality. With the addition of caching, *Vector* often outperforms *OptVec*. This demonstrates the drawback of vector-optimized data layout for strided accesses in the presence of a cache as discussed in section 3.2.1.

Figure 3.12 shows the performance of *IndirectRLn* for $n = 2$ and 5 for access patterns of varying sparseness. The sparseness of the access pattern is varied by changing the size of the address range within which the pseudo-random accesses fall. Achieved bandwidth declines as the sparseness of accesses increase causing more frequent row activations. *Stream* and *Vector* have no strong dependences on record length since the access pattern is essentially (pseudo)random over the entire address range. *OptVec* performance, however, improves slightly with larger record sizes. This results from the fact that each of the constituent vectors of the *OptVec* access falls within an increasingly narrower range of addresses as record size increases (i.e. $1/r$



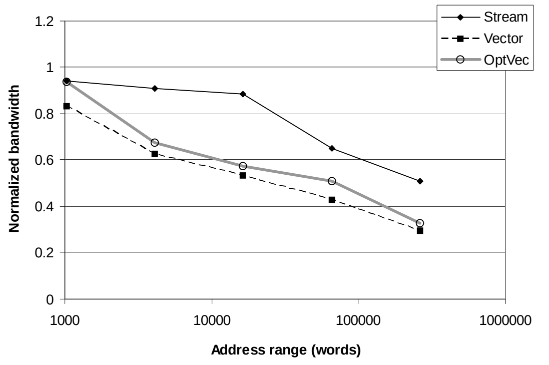
(a) 2-word records (*StrideCRL2*)



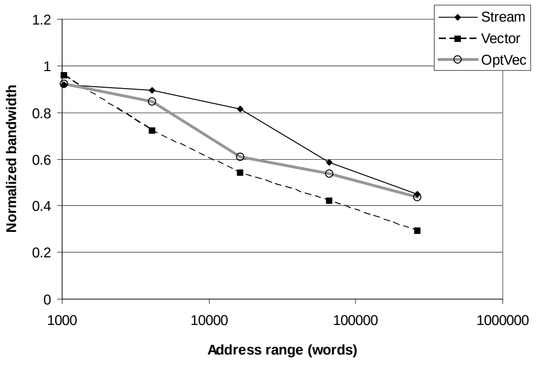
(b) 5-word records (*StrideCRL5*)

Figure 3.11: Cached strided access performance under *Stream*, *Vector*, and *Optimized Vector* accesses as stride varies (normalized to *Seq* with 1-word records)

of the overall address range, where r is the record length), reducing the sparseness of the access pattern.



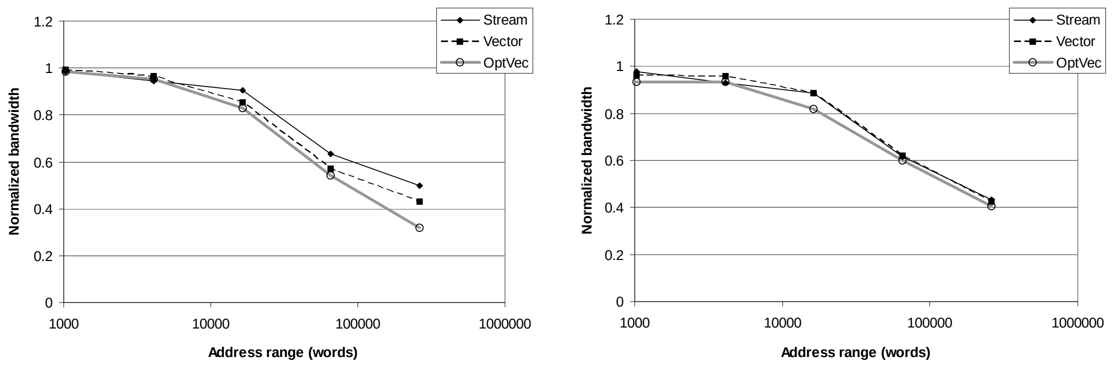
(a) 2-word records (*IndirectRL2*)



(b) 5-word records (*IndirectRL5*)

Figure 3.12: Uncached indirect access performance under *Stream*, *Vector*, and *Optimized Vector* accesses as the sparseness of the access pattern varies (normalized to *Seq* with 1-word records)

Figure 3.13 shows the performance of *IndirectCRLn* for $n = 2$ and 5. The differences between *Stream*, *Vector*, and *OptVec* are much narrower in this case relative to the uncached case of figure 3.12. The differences are particularly small when the accesses are to small ranges of addresses which lead to higher cache hit rates. The higher bandwidth of the cache obscures differences in DRAM performance among *Stream*, *Vector*, and *OptVec* on cache hits. Bandwidth declines and performance differences emerge between the three access types with increasing sparseness in the case of *IndirectCRL2* as cache hit rates drop. *Stream* and *Vector* outperform *OptVec* for sparse accesses by exploiting record locality captured in cache lines. However, as with the uncached case, *OptVec* performance improves with longer record sizes as each vector access falls within a narrower address range.



(a) 2-word records (*IndirectCRL2*) (b) 5-word records (*IndirectCRL5*)

Figure 3.13: Cached indirect access performance under *Stream*, *Vector*, and *Optimized Vector* accesses as the sparseness of the access pattern varies (normalized to *Seq* with 1-word records)

3.2.4 Application Performance

The overall performance of several application benchmarks under *Stream*, *Vector*, and *OptVec* accesses are studied in this section. Table 3.5 describes the benchmarks and table 3.6 summarizes the distribution of memory accesses performed by each benchmark. Figure 3.14 shows execution time for the application benchmarks under *Stream*, *Vector*, and *OptVec* accesses with and without caching, normalized to the uncached *Stream* case. For FFT 2D, FEM 3D, MD, and IGraph, caching is used selectively to exploit temporal locality present in their accesses. FFT 1024 and Depth do not exhibit temporal locality and caching is used for capturing spatial locality of record accesses.

Media/Signal Processing Benchmarks	
FFT 1024	1024-point FFT. Memory accesses are dominated by sequential accesses to short (2-word) records
FFT 2D	64x64 2D FFT. Performs a mix of sequential, non-unit stride, and indirect accesses to short (2-word) records. Array padding is used to reduce memory bank conflicts
Depth	Stereo depth extraction. Memory accesses are dominated by sequential accesses to 1-word data elements
Scientific Computing Benchmarks	
FEM 3D	Finite element application designed for solving systems of first order conservation laws on general 3D unstructured meshes. Most accesses are to medium (5-word) or long (20-word) records
MD	Molecular dynamics simulation based on GROMACS [vdSvBA$^+$01]. Memory accesses are dominated by indirect accesses to long (9-word) records
IGraph_S/D	Parameterized synthetic benchmark that models inter-node interactions in 2D and 3D irregular graphs. The _S version simulates a sparse graph (average graph degree 4). The _D version simulates a dense graph (average graph degree 16). Most memory accesses are sequential or indirect to medium-sized (4-word) records

Table 3.5: Application benchmarks used to evaluate memory system performance of stream and vector accesses

Uncached Results

As expected, applications dominated by 1-word data types, such as *Depth*, show no appreciable performance variation between stream and vector accesses. *Vector* performs poorly for all other benchmarks due to its inability to exploit the spatial locality in multi-word record accesses. *OptVec* performs significantly better than *Vector*, but underperforms *Stream* for all benchmarks. In the absence of caching, *Stream* accesses lead to speedups averaging 34% and 81% over *Vector* accesses for media and scientific benchmarks respectively. *Stream* accesses also lead to speedups averaging 9% and 27% over *OptVec* for the same application classes. The above results

	Stream & OptVec			***Vector***		
	Unit Stride	**Non-unit Stride**	**Indirect**	**Unit Stride**	**Non-unit Stride**	**Indirect**
FFT 1024	89	0	11	5	84	11
FFT 2D	50	25	25	0	75	25
Depth	83	0	17	83	0	17
FEM 3D	44	0	56	3	41	56
MD	10	0	90	10	0	90
IGraph_S	46	0	54	14	32	54
IGraph_D	28	0	72	18	10	72

Table 3.6: Memory access pattern distribution of benchmarks (percentages)

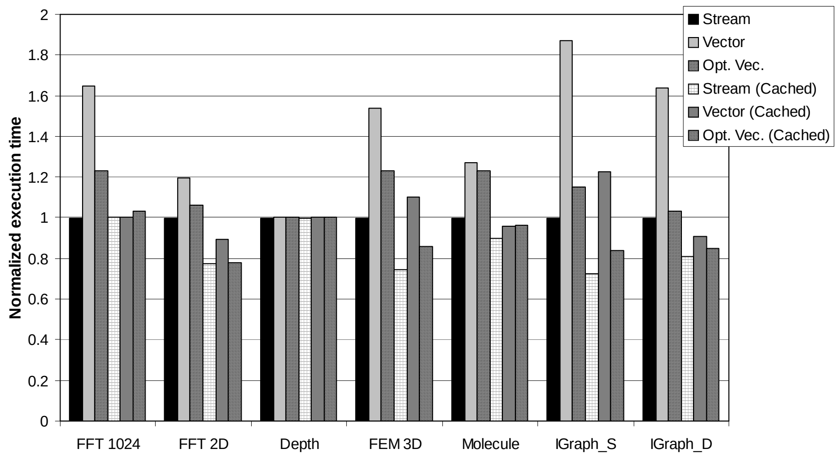


Figure 3.14: Application performance under *Stream*, *Vector*, and *Optimized Vector* accesses (normalized to the uncached *Stream* case)

also demonstrate the potential of data layout optimizations in improving vector access performance as *OptVec* significantly outperforms *Vector*. However, these results were generated assuming that data can be liberally reordered to optimize for vector accesses. In reality, optimizations for vector accesses maybe at odds with scalar accesses to the same data in non-data-parallel sections of applications. Such scalar accesses typically rely heavily on cache lines capturing spatial locality for good performance. In addition, vector data reordering flexibility may also be constrained by

API limitations [BKGA04]. Negative effects on scalar accesses and other limitations to reordering are not reflected in these results as only the data parallel portions of the codes were simulated for these results. Therefore, in realistic implementations, practically achievable vector performance will fall between *Vector* and *OptVec* performance shown above. Stream accesses, on the other hand, achieve high performance without the need for data reordering. It is conceivable, however, that some computations may not consume many fields of each record, reducing the effectiveness of stream accesses. Although this case was not encountered in the benchmarks studied, in extreme such scenarios, stream accesses can be degenerated to vector accesses, and hence perform no worse.

Cached Results

Again, *Depth* is insensitive to the ordering of accesses, as before, and does not benefit from caching since there is no temporal locality in the memory accesses. *FFT 1024* also has no temporal locality in the memory accesses, but the addition of the cache helps improve *Vector* and *OptVec* performance. All other benchmarks contain some amount of temporal locality in the memory accesses that lead to improved performance for all access patterns with the addition of a cache.

Stream accesses provide only small performance benefits over *Vector* and *OptVec* for the media benchmarks (average speedups of 5.5% and 1.5% respectively) when caching is introduced. However, for the scientific applications with longer record sizes and more complex access patterns, *Stream* achieves speedups averaging 34% and 11% over *Vector* and *OptVec* respectively, even with cached accesses.

Impact of Technology Scaling

Some of the differences in memory system performance are hidden by computation overlapped with memory accesses. However, based on VLSI trends, compute resources have been scaling more rapidly than memory bandwidth. In order to understand the implications as the sustainable memory bandwidth to computation ratio decreases, the limiting case where computation takes zero time was simulated, causing all applications to be memory-bound. These results are shown in figure 3.15. The memory and cache parameters used for this study are the same as those listed in table 3.2.

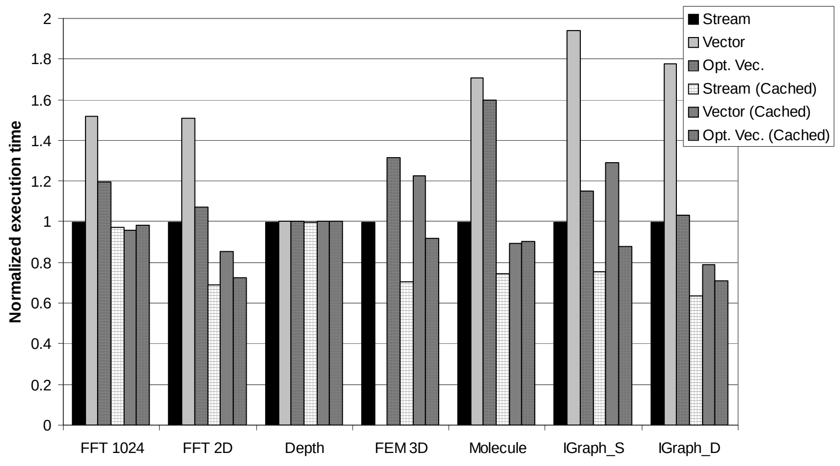


Figure 3.15: Application performance under *Stream*, *Vector*, and *Optimized Vector* accesses in the limiting case where computation is instantaneous (normalized to the uncached *Stream* case)

Relative performance of stream and vector accesses for the media and signal processing benchmarks show only minor variations from the results in figure 3.14. However, the scientific benchmarks show greater variations due to increased reliance on memory performance. For the scientific benchmarks, *Stream* outperforms *Vector* by 70% to 94% without caching and by 20% to 74% with caching[1]. *Stream* also outperforms *OptVec* by 3% to 60% without caching and by 12% to 30% with caching.

In summary, record granularity data access, as is done with stream accesses, leads to more efficient utilization of the available memory bandwidth compared to word granularity accesses. This advantage is particularly significant in the presence of large data records which lend some spatial locality to the stream accesses even though the application-level access patterns may be strided or irregular. This advantage becomes even more prominent as technology scaling makes memory system bandwidth

[1]We were unable to generate *Vector* results for FEM 3D due to an incompatibility with the technique used to emulate instantaneous execution of computations.

increasingly scarce relative to on-chip compute capabilities.

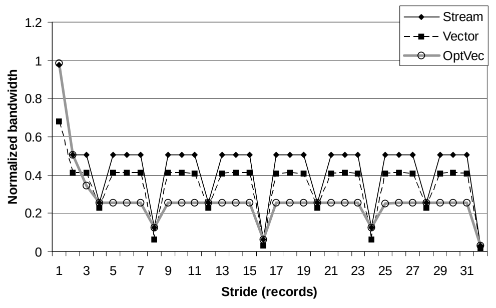
3.2.5 Sensitivity to System Parameters

Several microbenchmark studies were used to determine sensitivity to DRAM burst length and cache line size, and an interesting subset is presented and discussed below. The benchmarks are the same as in section 3.2.3 unless otherwise noted. Throughout this subsection the notation $< b, t, c >$ will be used to denote the combination of memory burst size b, memory transfer width t, and cache line size c (all in terms of 64-bit words). All other system parameters are held the same as those listed in table 3.2 except where noted otherwise. Only cached results will be presented for brevity since the relationship between cached and uncached results can be inferred from the discussion in section 3.2.3.

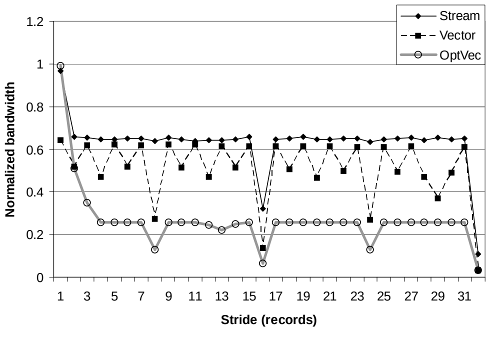
Impact of Long DRAM Bursts and Cache Lines

Figure 3.16(a) shows the performance of *StrideCRL2* for the $< 4, 2, 4 >$ case. The main difference compared to the $< 2, 2, 2 >$ case of figure 3.11(a) is the approximately 50% reduction in effective bandwidth for non-bank-conflict cases of stride > 1 for all three access types. This is due to the additional memory traffic generated for accessing 4-word bursts while only 2 words in each burst are useful for all non-unit strides. The second key difference relative to $< 2, 2, 2 >$ is the reduced frequency of bank conflicts. Since addresses are distributed over the memory banks at a granularity equal to the burst size, bank conflicts, which occur when the effective stride is an even multiple of the burst size, occur less frequently.

Figure 3.16(b) shows the performance of *StrideCRL5* for $< 4, 2, 4 >$. The effects described for *StrideCRL2* are visible here as well. Effective bandwidth is reduced due to unused memory traffic that result from the long bursts and cache lines. However, *Stream* and *Vector* bandwidth is improved relative to *StrideCRL2* since a higher percentage of the accesses contain useful data (*StrideCRL5* achieves 62.5% burst utilization since 5 words out of 2 bursts are useful, compared to 50% utilization for *StrideCRL2*). *OptVec* performance is independent of record size, and achieves only minimum utilization of one word per burst or cache line for all strides equal to or larger than the burst or cache line size.



(a) 2-word records (*StrideCRL2)*

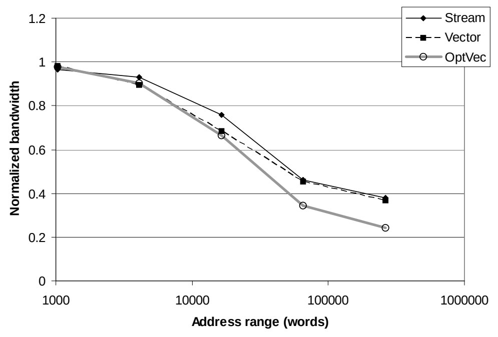


(b) 5-word records (*StrideCRL5*)

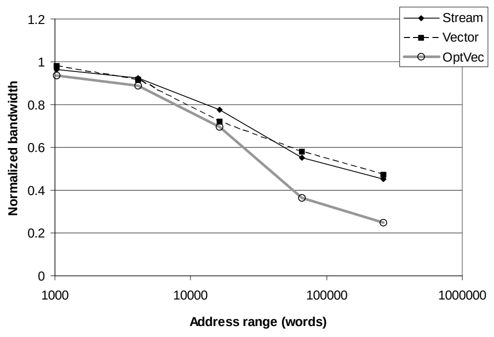
Figure 3.16: Cached strided access performance *StrideCRLn* $< 4, 2, 4 >$ (normalized to *Seq* $< 2, 2, 2 >$ with 1-word records)

Figure 3.17 shows the performance of *IndirectCRL2* and *IndirectCRL5* for $< 4, 2, 4 >$. *IndirectCRL2* performance is degraded at sparse access patterns for *Stream*, *Vector*, and *OptVec* compared to $< 2, 2, 2 >$ due to the added bandwidth imposed by longer DRAM bursts which are not fully utilized by the short records. In the case of *IndirectCRL5*, *Stream* and *Vector* performance is improved relative to $< 2, 2, 2 >$ due to the added spatial locality captured by the longer cache lines and reduced probability of bank conflicts. However, *OptVec* performance is degraded due to the low utilization since a cache line access may contribute only a single useful word at sparse access patterns.

In summary, longer DRAM bursts and/or cache lines result in low utilization of the available bandwidth during accesses to data elements that are short or are not a multiple of the burst/cache line size. This impact is visible in all three access types considered. This result is not unexpected, and agrees with prior studies on vector caches such as [KSF$^+$94]. However, in the presence of long records, *Stream* is able to achieve higher utilization of long DRAM bursts and cache lines. *Vector* requires cached access to achieve similar benefits, and *OptVec* cannot leverage the long record sizes to improve utilization even in the presence of a cache.



(a) 2-word records (*IndirectCRL2*)



(b) 5-word records (*IndirectCRL5*)

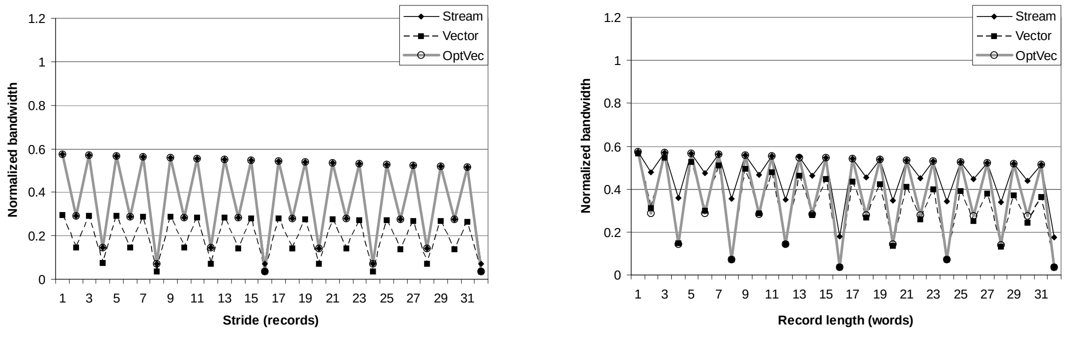
Figure 3.17: Cached indirect access performance of *IndirectCRLn* $< 4, 2, 4 >$ (normalized to *Seq* $< 2, 2, 2 >$ with 1-word records)

The general trend in commodity DRAMs over time have been towards longer bursts. For example, while DDR SDRAM supported a minimum burst length of 2 words, DDR2 SDRAM only support a minimum burst of 4 words. Therefore, the advantages of stream order memory accesses are likely to increase in future generations of commodity DRAMS.

Impact of Short DRAM Bursts

While short DRAM bursts and cache line sizes lead to better utilization of bandwidth, burst accesses are a critical element of achieving high bandwidth in modern memory systems. Figure 3.18(a) shows the performance of *StrideCRL2* for $< 1, 1, 1 >$ (i.e. without using multi-word burst accesses from DRAM). The achieved bandwidth of this case is lowered due to the fact that the peak bandwidth available from the 16 DRAM banks is reduced. In addition, due to data interleaving among banks at the granularity of a burst, bank conflicts occur at all even strides for *Stream* and *OptVec*. *Vector* incurs bank conflicts for all strides > 1 due to the effective stride always being an even multiple of the interleave factor due to the 2-word record size. Similar inefficiencies are observed for *StrideCRL5* for $< 1, 1, 1 >$ in figure 3.18(b). Therefore, while short burst and cache line sizes are preferred from a bandwidth utilization perspective, memory system designers must balance that with the high

DRAM bandwidth enabled by longer burst sizes.



(a) 2-word records (*StrideCRL2*) (b) 5-word records (*StrideCRL5*)

Figure 3.18: Cached strided access performance of *StrideCRLn* $< 1,1,1 >$ (normalized to *Seq* $< 2,2,2 >$ with 1-word records)

3.2.6 Discussion

Results presented in this section indicate that record granularity access of data streams improve memory system performance relative to word granularity accesses. In addition, stream programming explicitly decouples memory accesses from computation, allowing the programmer and/or compilation system to structure memory accesses independently of computation. As a result, stream processor implementations also expose memory system control to the software subsystem through the instruction set architecture.

On traditional vector architectures, decoupling of memory transfers from computation is only done for latency tolerance and not for optimizing memory system performance. With the addition of more explicit decoupling of memory transfers from computation, however, vector architectures can be adapted to perform memory transfers at record granularity, achieving the benefits seen in stream processors. Some recent research in vector architectures have evolved in this direction. For example, [BKGA04] describes the use of two-dimensional prefetches in to on-chip caches, which mimics a subset of the record-granularity memory access capabilities present in stream processors.

3.3 Multi-threaded Architectures

The granularity at which the parallel computation is expressed in multi-threaded architectures depends on the frequency of context switches. A continuum of possibilities exist for implementable context switch frequencies, and in order to simplify the analysis, we will consider the two opposing extreme design points to establish bounds on the intervening design points.

One extreme design point is a single multi-threaded processor that switches contexts every cycle. Such a policy leads to memory access patterns identical to a vector architecture with vector length N_t, where N_t is the number of available thread contexts. Current implementations of architectures that support multiple thread contexts provide a fixed number of physically distinct registers for each context. Therefore, the intermediate register requirements of such an architecture are also identical to a vector architecture of vector length N_t since intermediate state for all active threads must be maintained. Note that N_t is determined to provide sufficient latency tolerance to cover off-chip memory accesses.

The opposite extreme design point is a multi-core processor where each core executes a single thread to completion before switching to another thread. Such an architecture's memory accesses and register requirements are identical to a stream processor with N_c compute clusters, where N_c is the number of cores in the multi-threaded processor. Note, however, that such an implementation cannot rely on context switches to effectively overlap long latency memory accesses with computation, and therefore requires additional mechanisms to ensure long latency memory accesses are minimized through aggressive prefetching.

More complex implementations of multi-context processors, such as those that switch contexts on a subset of memory accesses (e.g. on cache misses), simultaneous multi-threading [TEL98] etc., lead to memory access patterns that fall between the two bounds expressed by the two simple extremes discussed above. However, the unpredictable nature of context switches in such architectures implies that register capacity must be provisioned to support the intermediate state of all live contexts, approximating those of a vector architecture with vector length N_t. As N_t approaches the vector lengths of traditional vector architectures (e.g. 64 to 256), the capacity requirements approach that of vector architectures while proving a large numbers of contexts to cover memory access latencies. Alternatively, as N_t approaches the number of compute clusters in stream processors (e.g. 8 to 16), register capacity requirements approximate those of stream processors, but require aggressive prefetching as the number of contexts alone is insufficient to cover long memory latencies.

3.4 Summary

This chapter evaluated the efficiency of expressing data parallelism at record granularity as is done in stream processing relative to expressing the data parallelism at word granularity as is done in vector processing. The evaluation consisted of two main parts - register capacity requirements and memory system performance.

Register capacity requirements were evaluated using an analytical model. The record granularity expression of data parallelism was shown to require significantly fewer registers to hold intermediate state compared to the word granularity case. This was largely due to the fact that in the case of word granularity, all intermediate results are expanded by the length of the input vector(s). In the record granularity case, intermediate results are only expanded by the number of parallel execution units, leading to lower register requirements.

Memory system performance was evaluated using a set of micro- and application-benchmarks. Record granularity expression of data parallelism leads to contiguous access to data records, leading to better utilization of DRAM burst accesses and cache lines. Word granularity data parallelism results in vector accesses that do not exploit the spatial locality of record-order data layout in memory, leading to inefficient use of DRAM capabilities. One technique for reducing this inefficiency is the use of an on-chip cache with multi-word cache lines to capture record locality. However, this requires caching vector accesses that may not exhibit temporal locality, potentially leading to cache pollution (i.e. evicting data that does have temporal reuse). Alternatively, vector performance can be improved by reordering data layout in memory to optimize for vector accesses. However, such layouts can be inefficient for sparse access patterns with multi-word DRAM bursts or cache lines, as well as for data that requires both vector and scalar accesses.

Across a set of media and scientific benchmarks, execution using stream order accesses was found to outperform the same computation with vector order accesses by an average of 45% when not using an on-chip cache. With an on-chip cache, stream accesses were found to outperform vector order accesses by 22% on average. Even with the data layout in memory reordered to optimize for vector accesses, stream access lead to speedups of 13% and 7% on average without and with an on-chip cache over the same set of benchmarks.

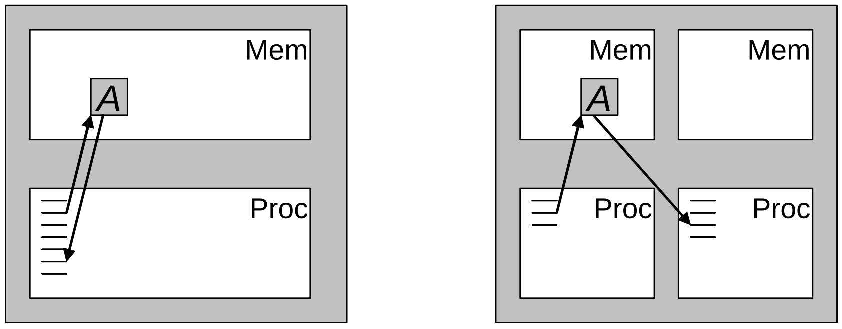
The impact of two scaling trends on memory system performance was also considered. First, as on-chip computation resources continue to scale more rapidly than off-chip bandwidth, the performance benefits of record granularity access over word granularity access was found to improve, especially for applications with long record sizes and irregular access patterns. Second, stream performance was also found desirable as DRAM burst lengths increase, which has been the recent trend in commodity DRAMs.

Chapter 4

Communication Hierarchy

The objective of the bandwidth hierarchy of a data parallel processor is to capture as much data reuse at the high-bandwidth levels, closer to the compute units, as possible. However, as a result of mapping applications to parallel execution hardware, a subset of the available data reuse is converted to spatial communication. A trivial example of this effect is shown in figure 4.1. In 4.1(a), the value A is generated in one part of the application and is consumed by a later part, resulting in producer-consumer reuse over time. However, after parallelization of the application over two processors as shown in 4.1(b), the locality is transformed in to spatial communication. Supporting this communication, therefore, becomes critical to capturing the data reuse in the parallelized execution of applications.

This chapter analyzes the temporal and spatial communication requirements of stream processors. The spatial communication resources in current stream processors are limited to the LRF and memory levels of the bandwidth hierarchy. These resources are shown to be inefficient for capturing stream-level data reuse present in complex access patterns found in some data parallel application classes. This chapter also explores techniques for enabling a complete communication hierarchy with appropriate resources at each level of the bandwidth hierarchy. Specifically, an indexed SRF architecture is proposed that enables communication at the stream level. This novel SRF is demonstrated to achieve significant reductions in memory bandwidth



(a) Uniprocessor execution (b) Parallelized execution

Figure 4.1: Parallelization transforms a subset of temporal reuse to spatial communication

demands as well as performance gains for a variety of complex stream access patterns. As a result, this design enables a broader class of data parallel applications to be executed efficiently on stream processors.

4.1 Communication in Time: Data Reuse

Temporal reuse occurs at both the kernel and stream levels for applications expressed in the stream programming model. At the kernel level, all intermediate data are represented as scalar variables, and hence the reuse occurs at the granularity of single words. The reuse available at this level can be classified into two types:

- *Temporal locality*: stream data values that are read from the SRF, and are referenced multiple times within the execution of a single kernel.
- *Producer-consumer locality*: intermediate results that are generated during a kernel's execution, and are consumed later during the same kernel.

At the higher, application level, data are represented as streams composed of tens to thousands of words. Therefore, the data reuse at this level can be categorized based on both the type and the granularity as follows:

- *Stream temporal reuse*: Streams that are read in from memory that are consumed multiple times at the granularity of the entire stream, in the same sequence each time.

- *Irregular temporal reuse*: Stream data that are read from memory and are reused multiple times, but the reuse occurs in a different sequence from the order of the original access. Further, the reuse may only apply to a subset of the data in a stream. This reuse may be *intra-stream*, where the same data is referenced multiple times within a single stream, or *inter-stream*, where the reuse occurs in multiple streams.

- *Stream producer-consumer locality*: Streams that are generated during the execution of a kernel, and are consumed by one or more later kernels of the same application. The consumption of the stream occurs in the same sequence it was generated in.

- *Irregular producer-consumer locality*: Stream data that are generated during the execution of a kernel, and are consumed by one or more later kernels of the same application. The consumption of the stream occurs in a different sequence than the one it was generated in.

4.2 Communication in Space: Resources and Constraints in Stream Processors

Kernel level reuse is captured in the LRFs within compute clusters as described in section 2.3.1. The spatial communication requirements at this level arise due to the two levels of parallel computation – multiple units within each compute cluster and multiple clusters. Communication among the units of a cluster is supported via the intra-cluster network, and communication among compute clusters is supported via the inter-cluster network as described in section 2.3.

Stream level communication is expected to be captured at the SRF as described in section 2.3.1. The communication requirements at this level arise due to three sources:

the presence of multiple independent data elements within a stream, the partitioning of the SRF into multiple banks, and the strip-mining of data sets. Therefore, stream locality, when mapped to the hardware implementation, requires three degrees of communication freedom to be fully captured:

- *In-lane*: communication among stream elements (possibly from multiple streams) that are mapped to the same SRF bank and are within the same strip. This often corresponds to irregular reuse of stream elements within a single lane.
- *Cross-lane*: communication among stream elements (possibly from multiple streams) within the same strip mapped to different SRF banks. This often corresponds to irregular reuse within a strip of an application.
- *Cross-strip*: communication among stream elements from different strips. This often corresponds to irregular reuse over the global data set of an application.

The SRF implementation of current stream processors severely restricts all three degrees of communication freedom at the stream level. In order to sustain high bandwidth from the SRF, a block of b contiguous words are read or written on each access to an SRF bank. To ensure these block accesses yield useful bandwidth, all stream accesses to the SRF are constrained to be performed in the same sequence in which the stream is stored in the SRF. This sequential access restriction prevents in-lane communication. In addition, as can be seen from figure 2.3, no cross-lane communication network exists at the SRF level. Similar constraints apply to vectors in the VRF of vector architectures as well. While a few vector architectures support a very small number of fixed communication patterns on vectors in the VRF (e.g. permute operations in V-IRAM for reductions and FFT butterfly patterns [Koz02]), arbitrary communication among vector elements is not supported.

As a result of the sequential access restriction, current stream processors only capture temporal or producer-consumer reuse at stream granularity at the SRF level of the bandwidth hierarchy. Capturing irregular reuse among data within one bank of the SRF requires *in-lane indexed access* – arbitrarily ordered access within each lane of the processor. Capturing irregular reuse among data mapped to different banks of the SRF requires *cross-lane indexed access* – arbitrarily ordered access to any bank of the SRF from any cluster.

Cross-strip data reuse occurs due to partial overlaps between the stream state of different strips of an application's data set. Given that stripmining is done due to data sets being too large to fit in the LRF/SRF register hierarchy, this reuse can often be captured only at the memory system level of the bandwidth hierarchy.

4.2.1 Alternatives to SRF-level Communication

Irregular stream-level reuse that cannot be captured in the SRF can be transformed to use other levels of the bandwidth hierarchy. One option, therefore, is to convert that reuse to kernel level communication. For in-lane communication, the stream data is read in-order from the SRF into the LRFs within the lane, and is permuted over the intra-cluster network. For cross-lane communication, the stream data is read in-order from the SRF in to the LRFs, and communicated via the intra- and inter-cluster networks. However, converting stream level reuse to the kernel level extends the live range of data in the LRFs, artificially increasing LRF capacity requirements. Since LRFs are optimized for high bandwidth with low capacity, this approach reduces the effectiveness of the two-level register hierarchy. Further, in realistic implementations, converting irregular stream locality to kernel locality is often not an option since the LRF capacity is insufficient to hold any significant amount of stream data along with the intermediate state of kernel execution(s).

A second option is to convert irregular stream-level reuse to memory-level reuse. In this case, any stream data requiring in-lane or cross-lane communication are written to the memory system, and read back in the desired new permutation. Exposing stream locality to the memory system is always possible, and is often the only realistic option in current stream processors. However, the bandwidth of the memory system is significantly lower than that of the SRF, and exposing locality that can be captured at the SRF to the memory system reduces the effectiveness of the bandwidth hierarchy. Further, in cases where intra-stream irregular reuse exists, multiply referenced data elements are replicated in the SRF in order to conform to the sequential access pattern. This data replication in the SRF exacerbates capacity limitations for applications with large data sets, increasing the number of strips required compared to a solution that avoids replication. For such applications, overheads of starting and finishing a strip, such as priming and draining of software-pipelined loops and initialization code, are incurred a greater number of times in the presence of data replication in the SRF, increasing execution times.

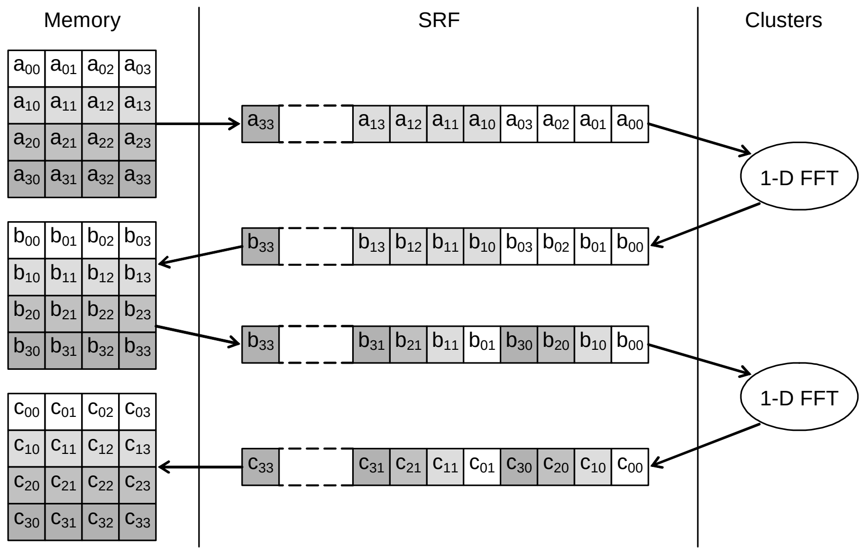
A third option is to introduce an additional memory structure that supports flexible communication patterns. For example, the Imagine processor contains a 256-word scratchpad memory in each lane [RDK+98]. However, these scratchpads can only accommodate a small amount of data, and provide lower bandwidth than the SRF. Further, such memories often require data to be transferred from off-chip memory to the SRF, and then from the SRF to the scratchpads before arbitrary accesses can be performed, incurring the bandwidth overhead of additional data transfers.

4.2.2 Application Examples

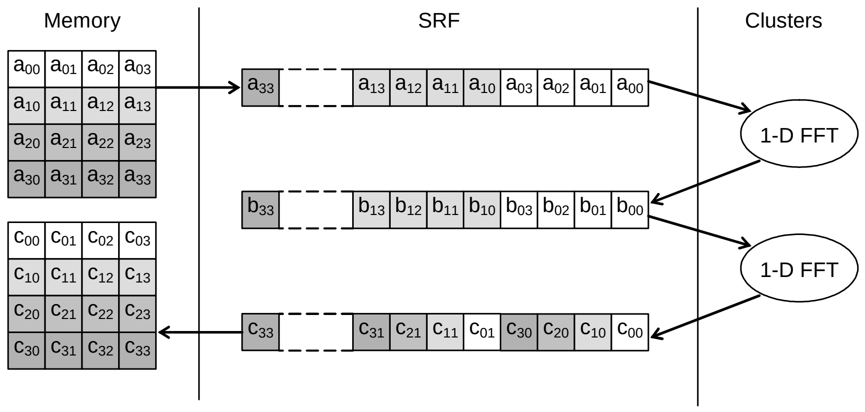
In order to illustrate the inefficiency of converting stream level locality to other levels of the bandwidth hierarchy, this section explores a few simple application examples.

Multi-dimensional Array Accesses

Accesses along different dimensions of a multi-dimensional array can be supported efficiently by capturing irregular stream locality in the SRF. Many signal processing applications with two- or higher-dimensional data sets require such operations. For example, consider the simplified 2D FFT shown in figure 4.2. The first invocation of the 1D FFT kernel generates an intermediate result array in row-major order. The second kernel invocation needs to apply the algorithm along the columns, requiring a reordering of the array. With sequential SRF access, this requires writing the entire array to memory and re-reading it in column-major order as shown in figure 4.2(a). However, indexed SRF access allows the column-major access to take place directly from the SRF as illustrated by figure 4.2(b).


Memory
SRF
Clusters
1-D FFT
1-D FFT
a33 a13 a12 a11 a10 a03 a02 a01 a00
b33 b13 b12 b11 b10 b03 b02 b01 b00
b33 b31 b21 b11 b01 b30 b20 b10 b00
c33 c31 c21 c11 c01 c30 c20 c10 c00

(a) Irregular stream locality converted to reuse through memory


Memory
SRF
Clusters
1-D FFT
1-D FFT
a33 a13 a12 a11 a10 a03 a02 a01 a00
b33 b13 b12 b11 b10 b03 b02 b01 b00
c33 c31 c21 c11 c01 c30 c20 c10 c00

(b) Irregular producer-consumer locality captured at the SRF

Figure 4.2: Irregular stream access in 2D FFT: capturing irregular producer-consumer locality in the SRF reduces memory bandwidth demands

Neighborhood Accesses in Multi-Dimensional Arrays

Accessing data neighboring a given element in a multi-dimensional data structure using sequential accesses requires adjacent values in all dimensions to be contiguous within the stream. A simple 2D example of such a case is shown in figure 4.3, where a 3x3 filter kernel is applied to the row shown in gray. Applying the filter to subsequent rows requires reordering the data set. Avoiding this reordering with only sequential SRF access requires retaining 3 rows of the image in LRFs, converting stream level communication to the kernel level[1]. Arbitrary SRF access makes it trivial to perform the neighbor accesses by capturing the reuse in the SRF. Operations such as these are common in some classes of scientific simulations with regular grid structures and in various filters and solvers.

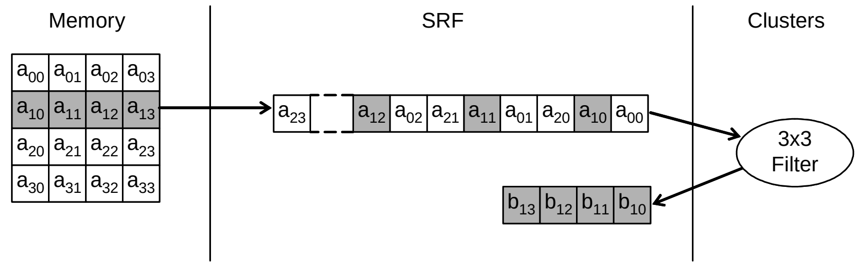
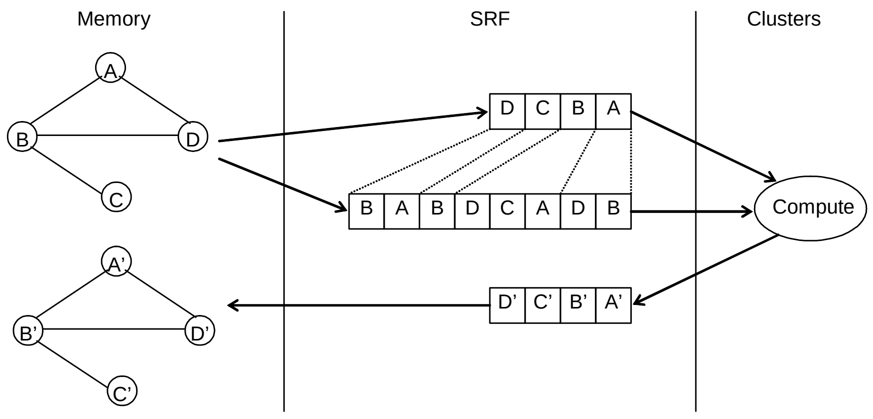

Memory
SRF
Clusters
a_{00} a_{01} a_{02} a_{03}
a_{10} a_{11} a_{12} a_{13}
a_{20} a_{21} a_{22} a_{23}
a_{30} a_{31} a_{32} a_{33}
a_{23} a_{12} a_{02} a_{21} a_{11} a_{01} a_{20} a_{10} a_{00}
3x3 Filter
b_{13} b_{12} b_{11} b_{10}

Figure 4.3: Neighborhood access in 2D array: communication at the SRF eliminates the need to hold large amounts of array data in local registers

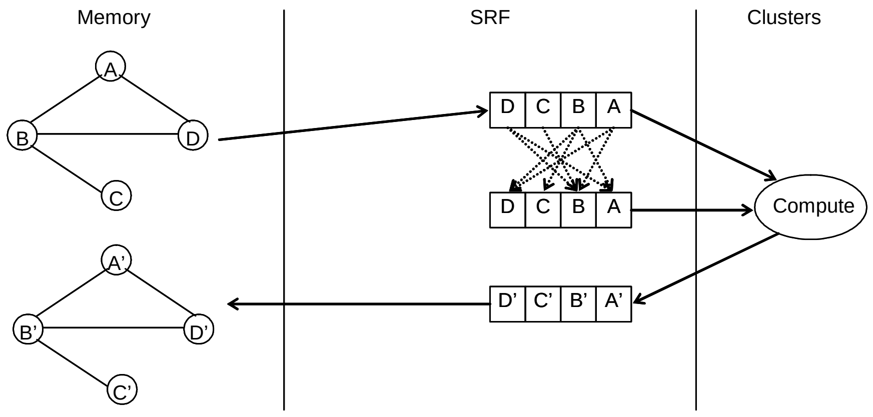
Neighbor Access in Irregular Graphs

Accessing neighbors of nodes in irregular graph structures using sequential streams result in nodes that neighbor multiple other nodes being replicated as shown in figure 4.4. Exploiting this intra-stream irregular temporal locality at the SRF reduces memory traffic by eliminating redundant fetches and reduces the space occupied in the SRF by eliminating replicated copies. Application classes with such accesses include scientific applications with irregular structures and some graph traversal algorithms.

[1]The simple 3x3 filter case discussed in this example can be efficiently implemented even using a sequentially accessed SRF by maintaining each row as a separate stream. However, that implementation does not scale to larger filters or higher dimensions as each 1-D row requires a separate, concurrently active stream, and the number of concurrent streams are often limited by hardware constraints in stream processors.



(a) Irregular temporal locality captured at the memory level



(b) Irregular temporal locality captured at the SRF level

Figure 4.4: Neighbor access in irregular graphs: capturing irregular temporal locality at the SRF reduces memory bandwidth and data replication in the SRF

Table Lookups

Data-dependent table lookups are used in a variety of applications for algorithmic reasons as well as for storing pre-computed values as an optimization. With a sequentially accessed SRF, these accesses require indexed gather operations from memory as described in section 2.2. With indexed SRF access, lookups can be performed in the SRF if the table or a useful partition of it can fit in the SRF.

Conditional Accesses

Conditional control flow is an expensive operation in stream and other SIMD processors, and is often translated into conditional data accesses in order to improve efficiency [SFS00; KDR$^+$00]. Conditional stream accesses without SRF indexing require communication among lanes as data statically mapped to SRF banks must be accessed sequentially and distributed among clusters based on dynamically evaluated conditions. In some cases, however, conditional computation of SRF indices provides an alternative for expressing conditionals with greatly reduced cross-lane communication requirements, as in the sort benchmark that will be described in section 4.3.5.

Other Uses

Indexed access to the SRF also enables a number of resource constraints to be relaxed through virtualization. One example is the ability to spill long-lived intra-kernel LRF state to the SRF. With a sequentially accessed SRF, LRF spilling is severely restricted since the data must be read back in the same order it was spilled. Therefore, kernels that require more intermediate state than available physical LRF space can only be supported via splitting to multiple kernels. While kernel splitting is an elegant solution in such cases, compilation technology for performing this optimally has only been demonstrated for linear filters [LTA03] – a small subset of the streaming application space. Indexed SRF access enables automated compilers to support kernels with large amounts of intermediate state through flexible spilling of long-lived LRF state to the SRF for any streaming application.

A second set of resources that may be virtualized via indexed SRF access are the stream buffers. With sequential SRF access, each SB is statically allocated to a single stream. This is required since the SB internally maintains the stream state (i.e. next location in the SRF to access), and the block accesses from the SRF prohibit fine-grain interleaving of data from multiple streams over a single SB. However, this also limits the number of streams that a single kernel can consume and generate to be no greater than the number of hardware SBs. With indexed access, multiple streams can be multiplexed over a smaller number of SBs, again enabling the compiler to support kernels that exceed physical resource constraints (a detailed discussion of how indexed SRF access uses SBs and other hardware resources will be presented in section 4.3).

4.2.3 SRF-Level Bandwidth Requirements

The bandwidth requirements and performance potential of enabling SRF-level communication can be analyzed by a simple model that expresses an application's execution time as limited by some level of the bandwidth hierarchy. Consider an application whose execution time is limited by the memory system bandwidth or the SRF bandwidth. The necessary condition for SRF-level communication to provide a speedup for such an application is expressed in equation 4.1. w_S and w_I are the number of words of sequential and indexed accesses performed by the compute kernels at the SRF level respectively. p is the fraction of w_S that must be read from or written to the memory system for reasons other than SRF-level communication. M, S_S and S_I are the sustained bandwidths of the memory system, sequential SRF access, and SRF-level communication respectively. The left side of the inequality, therefore, is the time spent in memory system accesses if no SRF-level communication is supported and that communication is exposed to the memory system. n is 2 for irregular producer-consumer reuse since the data must be written to memory from the SRF, and than read back in a different order, and is 1 for irregular temporal reuse. The right hand side of the inequality is the time spent on SRF accesses with support for indexed accesses at that level. Note that this simple model does not characterize applications that are constrained by compute rather than bandwidth requirements, but such applications can tolerate inefficient use of the bandwidth hierarchy and hence are not relevant to this analysis.

$$\frac{pw_S + nw_I}{M} > \frac{(1+p)w_S}{S_S} + \frac{w_I}{S_I} \tag{4.1}$$

Let $f = \frac{w_I}{w_S + w_I}$ be the fraction of SRF accesses that are indexed. Also, let the ratio of sequential SRF bandwidth to memory system bandwidth (S_S/M) be W. Therefore, the necessary sustained bandwidth of indexed SRF accesses relative to the memory system bandwidth to achieve an application speedup is shown by the expression in

4.2.

$$\frac{S_I}{M} > \frac{Wf}{W(p + (n-p)f) - (1+p)(1-f)} \tag{4.2}$$

Figure 4.5 shows a plot of the necessary indexed SRF to memory bandwidth ratio in order to achieve a speedup as a function of f, with $W = 10$ and $n = 1$. The value of 10 for W is based on the approximately order of magnitude increase in bandwidth at each level of a streaming memory hierarchy. Note that the inequality of expression 4.2 is undefined at very low values of p and f where the sequential SRF access time exceeds the memory access time.

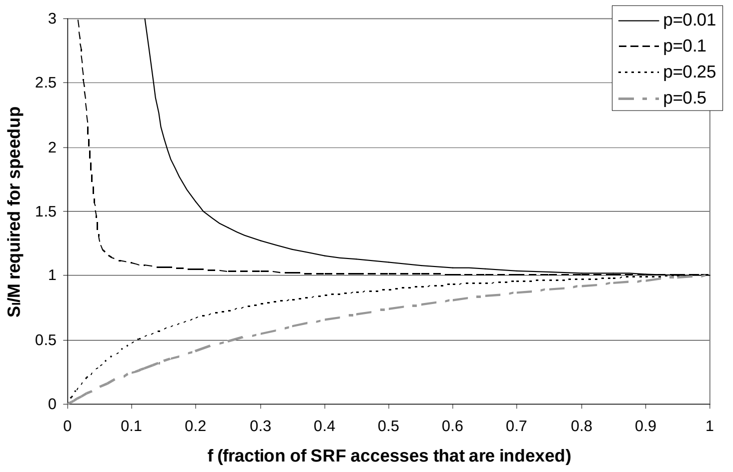


Figure 4.5: Required ratio of indexed SRF to memory system bandwidth in order to achieve application speedup based on model of application performance as being limited by memory or SRF bandwidth

Figure 4.5 shows that little or no bandwidth multiplication over the memory system is needed for improved performance through indexed SRF access for most values of p and f. As can be expected, at very low memory system utilizations (i.e. very low p), a higher indexed SRF bandwidth is needed in order to achieve a speedup. However, even with p and f as low as 10% each, less than 10% bandwidth increase in indexed SRF accesses over the memory system is needed to achieve a speedup. When the memory system is heavily utilized (e.g. p of 25% and 50% lines of the plot), indexed SRF access can provide a speedup by reducing the load on the memory system even if the indexed SRF bandwidth is lower than the memory bandwidth.

4.3 A Stream Register File with Indexed Access

Supporting flexible communication at the SRF level can lead to a significant reduction in the inefficiencies that arise in the presence of irregular stream communication as described in section 4.2.2. This section presents and evaluates an SRF architecture that supports such flexible communication by allowing compute clusters to explicitly specify the locations to be accessed. We start with a description of a sequentially accessed SRF (base) implementation, and extend the design to support flexible communication patterns.

4.3.1 Standard SRF Implementation

Figure 4.6 shows a high-level view of an 8-lane sequentially accessed SRF. Each SRF bank is a single-ported SRAM and is accessed only by the cluster within the same lane as described in section 2.3.1. On each access, active streams arbitrate for access to the single SRF port. The sequential access restriction enables important simplifications in such an implementation. First, high bandwidth can be achieved by using a single wide port into the SRF banks to read or write a block of b words on every access. Second, coupled with the fact that the compute clusters also run in lock-step across all lanes, the SRF access requirements are identical across all lanes. Therefore, a single arbiter can be used for selecting which stream is granted access to the SRF port in all lanes on a given cycle. Additionally, since all banks of the SRF access the same location in this case, a single row address decoder can be shared among all lanes, reducing hardware overheads.

Each bank of the SRF has a capacity of 16KB (4K 32-bit words) and 64KB (8K 64-bit words) respectively in the Imagine and Merrimac architectures [RDK$^+$98; DHE$^+$03]. Such large capacity SRAMs are typically not implemented as a monolithic array of memory cells. Instead, they are composed of several smaller SRAM arrays

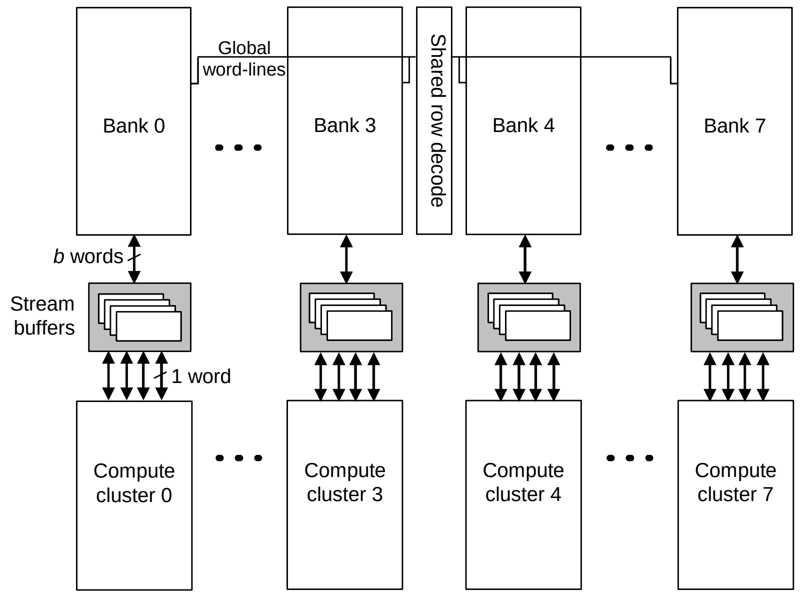


Figure 4.6: Block diagram of sequentially accessed SRF with 8 banks

(sub-arrays) that result in improved access latency and reduced access energy [AH00]. An implementation for a 16KB SRF bank composed of four sub-arrays is shown in figure 4.7. The example shown in the figure assumes a 128-bit block access in each bank and 2:1 column multiplexing at the sub-array outputs. The outputs of sub-arrays drive 128 shared "global" bitlines in each bank. During each access, a single sub-array is activated in each bank, which provides the entire 128-bit block for that access. Since only one sub-array is active within a bank at any one time, no additional arbitration is necessary for access to the global bitlines. Finally, since the row decoding is shared among all SRF banks, only a driver is needed within each bank for activating the local word lines of the memory arrays.

4.3.2 In-lane Indexed Access

The next address to be accessed for a given stream is tracked internally by the stream buffer (SB) associated with the stream in the base SRF implementation since the access pattern is predetermined (i.e. sequential). In order to extend that design to support in-lane indexed access, application kernels executing on the compute clusters

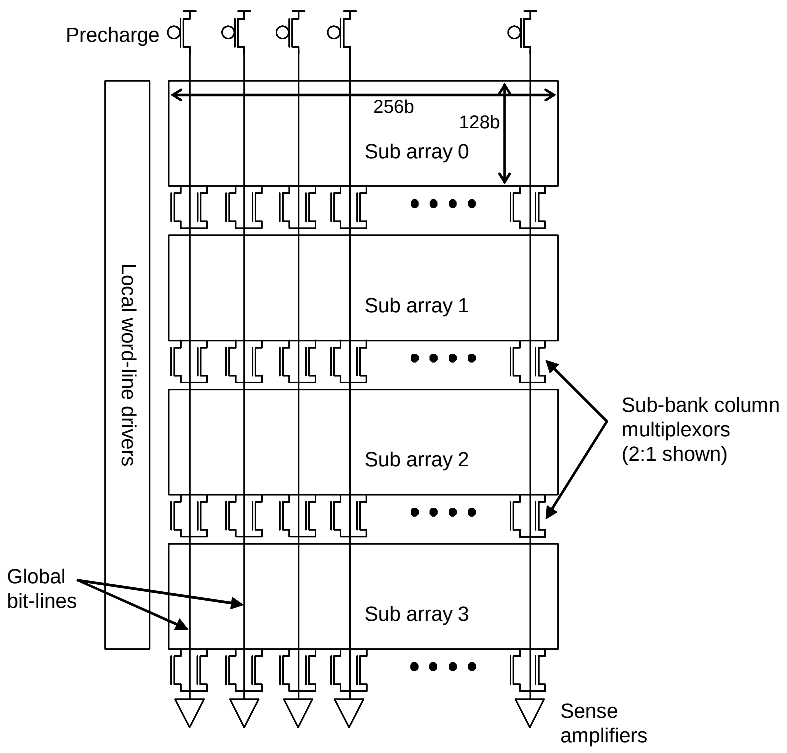


Figure 4.7: Block diagram of a single bank in a sequentially accessed SRF. Bank capacity = 16KB, block access = 128b, and array width = 256b as shown

must be allowed to explicitly specify the index of the next stream element to be accessed. Therefore, a key requirement for enabling in-lane indexed SRF access is an address path from the compute clusters to the SRF bank. Further, since access to the SRF port is dynamically arbitrated, the address path must tolerate variable latencies from address issue to access completion.

Figure 4.8 shows a high level view of the modifications to the SRF in order to support in-lane indexed access. A set of address FIFOs are added to the SRF banks. The corresponding address FIFOs across all lanes form a single logical address FIFO associated with a single stream. Indices for indexed access to stream i are placed in address FIFO i by the compute clusters. Address FIFOs are unused during sequentially accessed streams, which operate similar to the base implementation. Stream buffer i mediates data transfers between the clusters and the SRF for stream i during both indexed and sequential accesses. The address FIFO and the SB provide tolerance for the variable SRF access latency. This enables multiple outstanding access requests, which allows the computation in clusters to be statically scheduled using a conservative estimate of indexed SRF access latency.

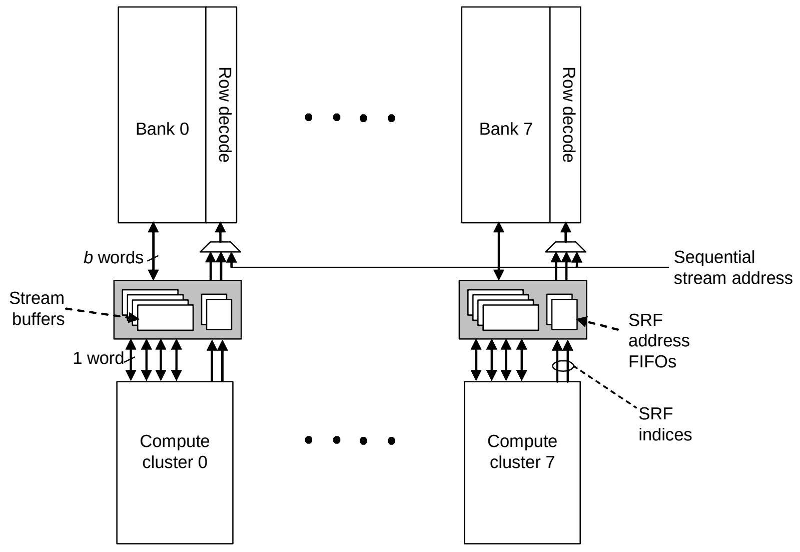


Figure 4.8: Block diagram of an 8-bank SRF with in-lane indexed access

For indexed reads, once the addresses are placed in to the address FIFOs by the compute clusters, active streams arbitrate for SRF access. Once access is granted for indexed stream i, the earliest pending index for that stream is accessed, and the data is placed in stream buffer i. For SRF arbitration, a stream is considered active if it is sequentially accessed or there are pending indices, and its stream buffer has sufficient space to hold the data after the SRF access completes.

For indexed writes, clusters place the indices in the address FIFO and write the associated data in to the corresponding stream buffer. Active write streams participate in SRF arbitration along with the read streams. A write stream is considered active if there is sufficient data in the stream buffer to perform an SRF access.

Since each cluster issues an independent index on indexed accesses, each bank of the SRF may be required to access a different row of the memory array. Therefore, the single shared row decoder is replaced by an independent row decoder in each bank as shown in figure 4.8.

Improving Indexed Access Bandwidth

For irregular stream-level reuse, accesses at the granularity of individual words or short records occur frequently in several application classes. Even in the case of long record accesses, they are not guaranteed to be aligned with the b-word blocks that are accessed using the single wide port of the SRF banks. Therefore, each indexed access from the SRF, in the worst case, yields one useful data word compared to the b words accessed during sequential accesses, leading to a potential bandwidth reduction by a factor of b for indexed accesses.

The internal structure of the SRF banks, which consist of multiple sub-arrays, can be leveraged to improve indexed access bandwidth. Instead of performing a single, b-word-wide block access from one sub-array per cycle, the same bandwidth can be achieved by performing a one-word access from b sub-arrays each cycle. Barring bank conflicts, this enables high bandwidth for both sequential and indexed accesses. The key modifications necessary to support such accesses are as follows:

- *A row decoder per sub-array*: Since each sub-array must perform an independent access simultaneously, a dedicated row decoder must be implemented for each sub-array.

- *Multiple addresses per SRF bank*: b addresses must be issued to each bank of the SRF in order to perform b concurrent accesses from each bank. Each address requires an independent address bus in to the SRF bank from the address FIFOs (for indexed accesses) and/or centralized address generators (for sequential accesses).

- *Per-sub-array arbitration*: In order to avoid bank conflicts among accesses from multiple streams, an independent access arbitration must be performed for each sub-array in each lane.

Retaining Energy Efficiency of Sequential Accesses

While the method of performing a single word access from multiple sub-arrays every cycle achieves the desired bandwidth, it significantly increases the energy consumed by the SRF for sequential accesses. Where b words are accessed for each sub-array activation in the base design, b separate sub-arrays must be activated to access the same number of words in the modified design. Therefore, the energy consumed by sequential accesses is increased by almost a factor of b in the modified design. In order to alleviate this drawback, the SRF bank implementation can be modified to support either a single block accesses for sequential streams or multiple simultaneous sub-array accesses for indexed accesses every cycle. Figure 4.9 shows a block diagram of a single bank of an SRF that supports both these access types. The key modification is the presence of two column multiplexers at the output of the sub-arrays, either of which may drive the global bit-lines.

During a block access, a single sub-array is activated, and a b-word block of data is accessed via the small degree column multiplexers. In the specific example shown in figure 4.9, a 128-bit block is read via the 2:1 column multiplexers. In this case, all 128 global bit-lines are driven by the single activated sub-array.

During indexed accesses, up to b sub-arrays are activated subject to the number of available indexed access streams and bank conflicts. Each sub-array drives a single word of data on to a subset of the global bit-lines. The global bit-lines are statically allocated to sub-arrays to avoid conflicts. In the specific example of figure 4.9, up to 4 sub-arrays drive 32 bits of data each on to disjoint subsets of the global bitlines. On accesses where less than 4 subarrays are activated, only the bitlines with valid data need to be driven, reducing energy consumed.

In order to support irregular temporal reuse, both indexed and sequential accesses must sometimes be supported to the same data stream, potentially during different stages of an application. Therefore, data distribution among the sub-arrays must be managed in a manner that facilitates both forms of accesses. Figure 4.10 shows the distribution of a stream of 2-word records among the banks and sub-arrays of the SRF from the example organizations shown in figures 4.8 and 4.9 (i.e. 8 banks, 4 sub-arrays per bank, and 4-word block accesses). Note that data records are interleaved

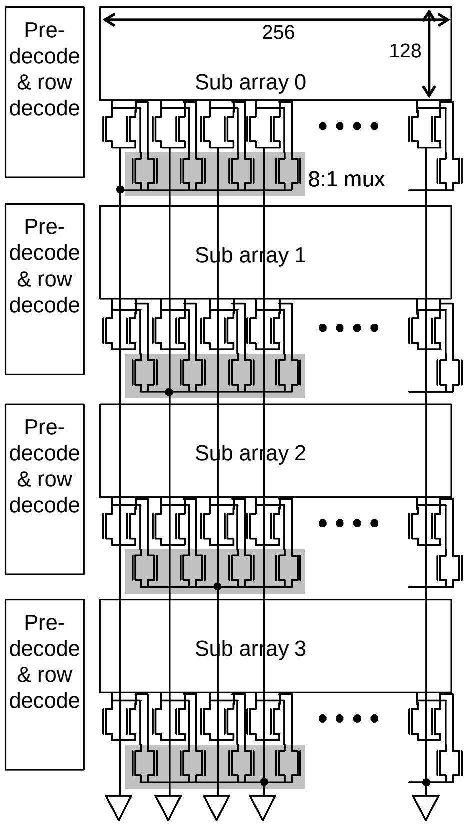


Figure 4.9: Block diagram of a single bank in an SRF with in-lane indexed access. Bank capacity = 16KB, block access = 128b, and array width = 256b as shown

among banks of the SRF, and address bits determine the distribution of data among sub-banks within a bank. Figure 4.11 shows the mapping of SRF address bits to the different aspects of the data layout.

Arbitration for SRF Access

In the indexed SRF implementation described above, arbitration for the SRF must be performed at two levels. Since sequential streams are controlled by a unified set of resources common to all lanes, any SRF access grants for such streams must be uniformly enforced in all lanes. However, for indexed access, an independent arbitration must be performed within each lane to achieve high utilization of the

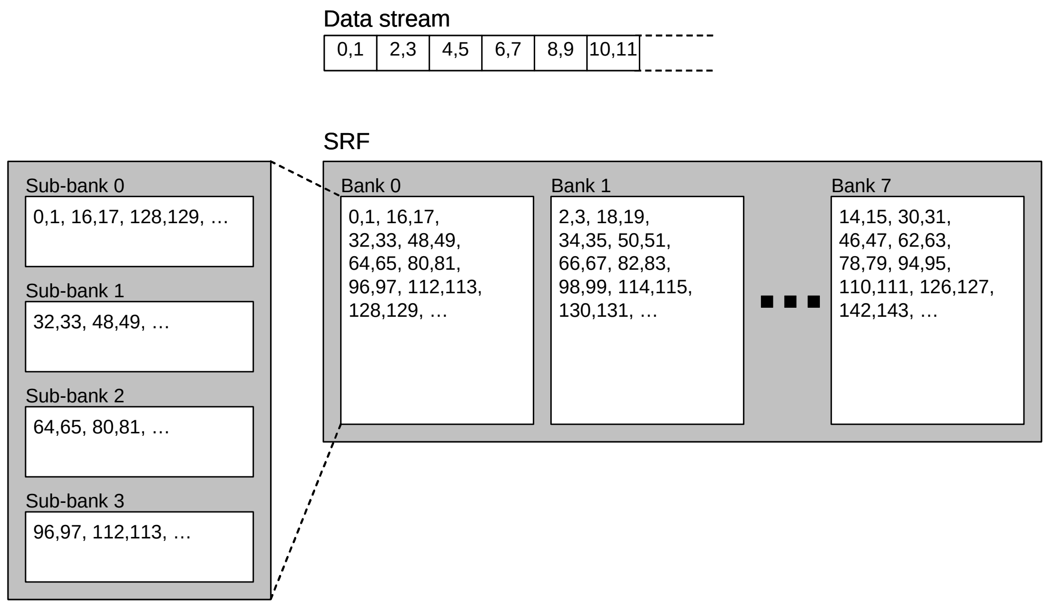


Figure 4.10: Data distribution among banks and sub-banks of the SRF. Example shown for 8 banks, 4 sub-banks per bank, and 128-bit block size (32-bit words)

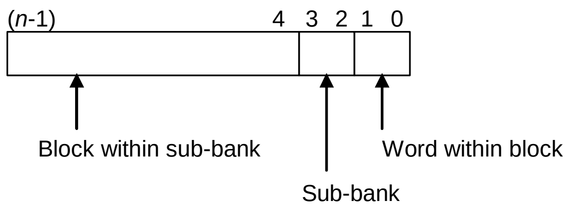


Figure 4.11: In-lane SRF address bit mapping. Example shown for 4 sub-banks per bank, and 128-bit block size (32-bit words)

available bandwidth. Therefore, the two-level arbitration is performed as follows:

- *Global arbitration*: Select one active sequential stream or all in-lane indexed access streams.
- *Local arbitration*: Select among active in-lane indexed streams for each sub-bank independently in each lane. This arbitration may be performed speculatively in parallel with the global arbitration to shorten the access pipeline, but is

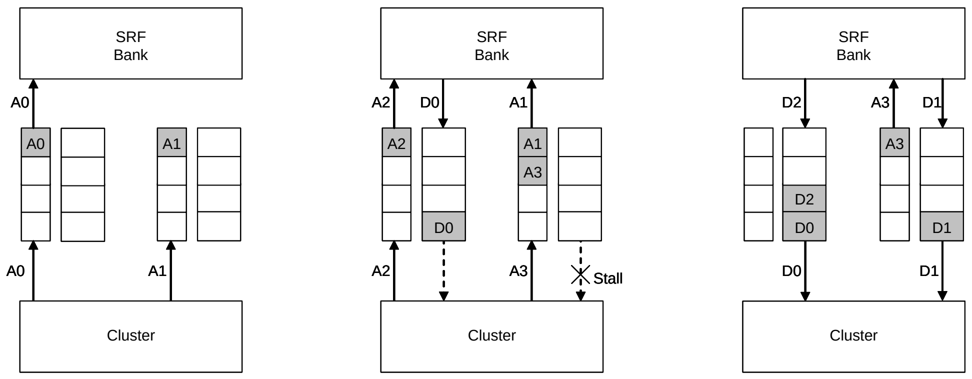
overridden if a sequential stream is selected at the global level.

The dynamic arbitration for the SRF results in variable latency for the indexed SRF accesses. Therefore, it is possible for accesses to not complete at the rate expected by the application, especially in the presence of high SRF bandwidth requirements or many sub-bank conflicts. While the buffering in the address FIFOs and stream buffers tolerate some mismatch between the application's access issue rate and SRF access completion rate, extreme mismatches lead to stalling compute clusters. Since all clusters execute in lock step, a stall condition in one cluster leads to stalling all clusters. A cluster stall can occur due to one of the following cases:

- *Stream buffer empty*: A cluster read to stream i stalls if SB i is empty due to previously issued reads not having completed.
- *Stream buffer full*: A cluster write to stream i stalls if SB i is full due to previous writes not having drained to the SRF.
- *Address FIFO full*: A cluster read or write to in-lane indexed stream i stalls if address FIFO i is full due to previously issued indexed accesses not having been issued to the SRF.

Figure 4.12 shows an example of in-lane indexed SRF access. Only a single lane and two streams are shown for simplicity. Figure 4.12(a) shows that in cycle i, the cluster issues one access each to the two streams by placing addresses in the address FIFOs. During local arbitration, both accesses $A0$ and $A1$ are determined to be to the same sub-array and are serialized. The delayed $A1$ access is performed in cycle $i+1$ along with other new or pending non-conflicting accesses as shown in figure 4.12(b). The cluster stalls when it tries to read the data for the accesses issued in cycle i as the data for access $A1$ are not available yet due to the conflict with $A0$. Figure 4.12(c) shows the cluster data access succeeding after the data for both streams are available. In-lane indexed SRF access is shown as a single cycle operation in this example for brevity. In reality, it is a pipelined multi-cycle operation.

The evaluations performed later in this chapter implement both the global and local SRF arbitrations using round-robin allocation among active streams. More



(a) Cycle (i) (b) Cycle (i+1) (c) Cycle (i+2)

Figure 4.12: In-lane indexed SRF access example. Hardware and timing simplified for clarity

elaborate schemes were studied (e.g. prioritizing streams likely to cause stalls due to the SBs or address FIFOs being full or empty beyond a threshold) and was found to provide only marginal improvements over round-robin arbitration despite significantly increased complexity.

4.3.3 Cross-lane Indexed Access

Cross-lane indexed SRF access requires inter-lane communication resources in addition to the in-lane indexing capabilities described above. Figure 4.13 shows a block diagram of the modifications necessary for supporting cross-lane indexed access. The inter-cluster network that was used for LRF-level communication in the base architecture is conceptually relocated to the interface between the compute clusters and the SRF. Therefore, both LRF-level and SRF-level cross-lane communications are multiplexed over this single network. In order to communicate the indices during cross-lane indexed SRF access, a dedicated address network is added as shown in figure 4.13. In order to reduce the number of ports in to the address network, all cross-lane indexed stream addresses from any one cluster are issued over a single address FIFO.

The cross-lane data traffic resulting from this implementation is comparable to the conversion of stream-level communication to kernel-level as discussed in section

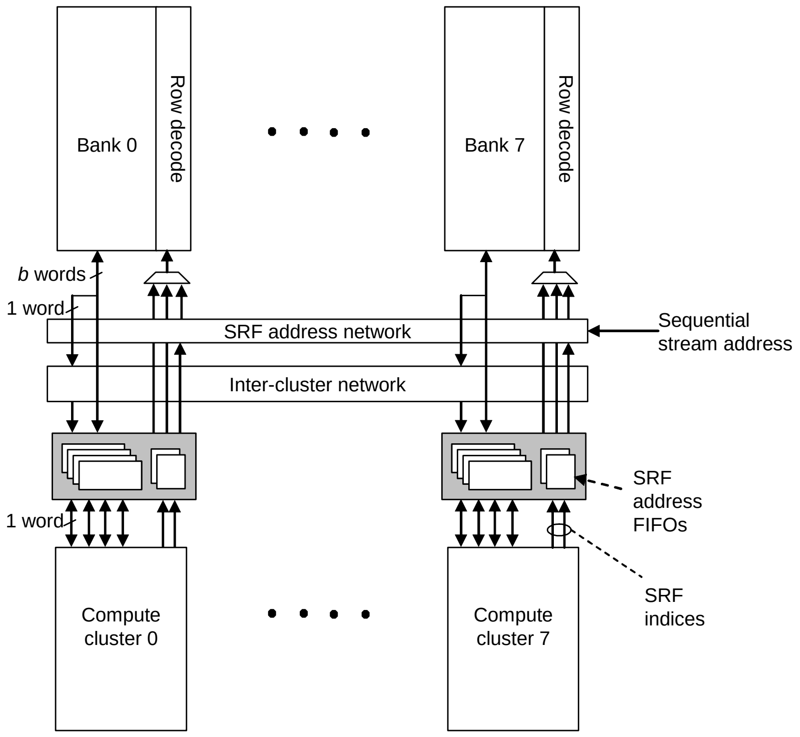


Figure 4.13: Block diagram of an 8-bank SRF with cross-lane indexed access

4.2.1. However, this implementation directly performs the stream-level communication from the SRF into the consuming cluster(s) without intermediate storage in the LRFs. Therefore, the excessive LRF occupancy that was inefficient when such communication was converted to the kernel level is avoided.

The bandwidth bottleneck for cross-lane communication lies in the inter- cluster data network that sustains a single word per lane per cycle bandwidth. Multiplexing LRF-level communication and cross-lane indexed SRF accesses over the same network could potential exacerbate this bottleneck. However, these two types of accesses are largely mutually exclusive in the benchmarks studied to date. Qualitatively, this behavior can be reasoned as follows. Kernel-level inter-cluster communication is used predominantly by applications that operate on regularly structured data where each lane is “aware” of the data being operated on in the neighboring lanes at compile time.

However, cross-lane indexed access is useful in applications with data-dependent irregular stream-level reuse where the data being operated on neighboring clusters are dynamically determined, rendering statically scheduled register-to-register communication difficult.

An exception to the above reasoning on mutually exclusive communication types is the presence of cross-lane indexed accesses along with conditional streams in some applications with irregular data structures. Conditional stream accesses also generate inter-lane data traffic. A quantitative evaluation of cross-lane indexed SRF performance in the presence of other inter-lane traffic is presented in section 4.3.8.

A potential complication of cross-lane indexed SRF access is the ordering among multiple writes to the same location from different clusters. One option is to require the software (application, compiler, and/or run-time system) to guarantee all writes on cross-lane indexed streams to be non-conflicting. However, it is worth noting that indexed SRF accesses, when used for capturing irregular temporal locality, do not require any writes since these are always read streams. Indexed accesses for irregular producer-consumer reuse do require writes, but such transactions can often be restructured to write sequentially and perform the reordering during subsequent reads. Therefore, the implementation evaluated in this chapter assumes cross-lane indexed SRF access is used for reads only. However, the design can be trivially extended to support writes, albeit without ordering guarantees.

Arbitrating Cross-lane Accesses

Cross-lane indexed access can be broken in to three phases. First, the indices must be communicated from issuing clusters to the intended SRF bank. Second, the indexed read must be performed at the intended SRF bank. Third, the data must be communicated back to the cluster the request originated from. Each of these phases introduces a potential new point of contention that must be dynamically arbitrated:

- Conflicts in the address network may arise due to multiple clusters issuing addresses simultaneously for the same target SRF bank
- Conflicts at the SRF banks with pending in-lane accesses

- Conflicts on the data return due to multiple outstanding data requests from the same originating cluster completing simultaneously in different SRF banks.

Arbitrating each of these phases separately improves resource utilization as each is independently optimized. However, in order to simplify the implementation, the cross-lane indexed implementation evaluated in this chapter assume a more limited degree of flexibility. All arbitration is constrained to the address traversal and the remainder of the cross-lane access is kept in lock-step across all lanes.

During index issue, bank conflicts due to multiple clusters targeting the same SRF bank are resolved dynamically in the address network using round-robin arbitration at the input to each SRF bank. Once communicated over the address network, the cross-lane indices are stored in a single-entry buffer (*remote address buffer* or RAB) at the intended SRF bank. Lanes that do not receive a valid request during this stage enter a null entry in the RAB in order to maintain lock-step access across all lanes. This level of conflict resolution and address network traversal can be overlapped with global and local SRF arbitration in order to reduce cross-lane access latency.

Arbitration for the SRF bank port for cross-lane accesses is integrated in to the existing arbitration among sequential and indexed streams. In order to ensure lock-step execution, cross-lane accesses are considered during global arbitration along with sequential streams. Therefore, when cross-lane accesses are granted the SRF port, all lanes perform the indexed access specified by the entry in the RAB, and the resulting data is written to the *cross-lane return buffer* (CLRB). Null entries in the RAB result in corresponding null entries in the CLRB.

Once the data is placed in the CLRBs, they can be returned to the original requesting clusters on any cycle that does not have a LRF-level inter-cluster communication scheduled. The return transfer is guaranteed to be conflict-free since each lane could have issued only one request per cycle, and all operations since the requests are performed in lock-step across all lanes.

Data returns of these accesses may be delayed due to the presence of LRF-level communication operations. A straight-forward policy that avoids stalling the in-lane SRF access pipeline due to cross-lane data return conflicts is for the CLRB in each lane to have at least as many entries as the SRF access latency, and for cross-lane indexed accesses to not be granted SRF access on cycles that initiate LRF-level communications.

4.3.4 Programmer Interface

The SRF-level communication may be exposed to the programmer via kernel-level languages such as KernelC. One potential representation that uses a syntax similar to C arrays for specifying explicit indices is shown in line 8 of the simplified table lookup kernel of figure 4.14. The index is specified at the source level in terms of records, similar to the access of C arrays. During compilation, operations are automatically inserted to compute the corresponding SRF location to be accessed based on the base address of the stream in the SRF, record size, and the specified index.

```
1   kernel lookup(
2         instream<int> in,        // sequential input stream
3         idx_instream<int> LUT,   // indexed in stream
4         outstream<int> out) {    // sequential out stream
5     int a, b, c;
6     while(!eos(in)) {
7       in >> a;        // sequential stream input
8       LUT[a] >> b;    // indexed stream input
9       c = foo(a, b);
10      out << c;       // sequential stream output
11     }
12   }
```

Figure 4.14: Indexed SRF access syntax example

An alternative programmer interface is via high-level or domain-specific language constructs. For example, the Brook high-level streaming language provides semantics for expressing neighborhoods and local groupings of data in multi-dimensional streams [BFH$^+$04]. Similar language constructs can be envisioned for explicit neighbor lists in irregular data structures and other forms of non-sequential stream-level communications. During compilation, such high-level constructs may be mapped to hardware utilizing indexed SRF accesses.

Reducing Address Computation Overheads

The SRF index computation is performed by the compute clusters using the existing arithmetic resources. Therefore, the index computation increases the number of operations that must be performed within the kernel body. For each indexed record accessed, a multiply and an add are incurred for computing the address for the first word of the record (multiplying the record index by record size and adding to the base of the stream; the multiply is avoided when record size is one word). Each subsequent word of the record incurs an address increment. For kernels that have high arithmetic intensity, these extra operations lead to a non-trivial performance overhead. Further, since SRF addresses are relatively short (10 to 20 bits in current stream architectures), performing the address computations using the 32- or 64-bit data paths in the clusters is inefficient.

Providing dedicated address computation units decreases the addressing overheads. However, adding such units in a manner that enables them access to the register state of clusters, which contain the record indices, increases the cost of the intra-cluster switch. However, the address increment for accessing subsequent words of a record can easily be implemented via dedicated units isolated from the intra-cluster network. Therefore, the results presented in this section assume the presence of dedicated address incrementers associated with the address FIFOs. Note that the number of address incrementers needed is independent of the number of address FIFOs, and is equal to the maximum number of indexed accesses that may be issued per cycle.

Tolerating SRF Access Latency

In order to tolerate the multi-cycle latency of indexed reads, the SRF index issue and data access are broken into two operations and are scheduled several cycles apart. The separation between the index issue and data access must be sufficient to allow the SRF access to complete, including a high probability of any bank and/or sub-bank conflicts being resolved. Ideally, this separation should be extended as much as possible when the operations are not on the critical path in order to minimize the probability of stalls on data access. However, the maximum number of outstanding addresses per stream before the corresponding data accesses are performed is bound by the number of entries in the address FIFO and the stream buffer.

4.3.5 Evaluation Methodology

The performance of a collection of application and synthetic benchmarks on four machine configurations are evaluated in order to assess the impact of indexed SRF access. Table 4.1 summarizes the different machine configurations, and table 4.2 lists their key parameters. The evaluations presented in this section were performed using modified versions of the Imagine simulation infrastructure. As a result, the machine configurations used here resemble the Imagine processor in many ways, and differ in terms of specific parameter values from the machine configurations used for evaluations in other chapters of this thesis.

Config.	**Description**
Base	Sequential SRF backed by off-chip DRAM
ISRF1	Indexed SRF with 1 word per cycle per lane in-lane and cross-lane indexed bandwidth (no multiple concurrent sub-bank accesses), backed by off-chip DRAM
ISRF4	Indexed SRF with up to 4 words per cycle per lane in-lane (4 concurrent sub-bank accesses per lane) and up to 1 word per cycle per lane cross-lane indexed bandwidth, backed by off-chip DRAM
Cache	Sequential SRF backed by on-chip cache and off-chip DRAM

Table 4.1: Machine configurations for indexed SRF evaluation

Base, *ISRF1*, and *ISRF4* configurations are similar to the designs described in this chapter so far. The *Cache* configuration is presented for comparison purposes since integrating an on-chip cache is a straight-forward approach for capturing stream-level communication exposed to the memory system when using a sequentially accessed SRF without generating off-chip accesses. Such caches have been integrated in to a number of vector architectures [BB90; BBC+00; Cra02], as well as the Merrimac streaming architecture [DHE+03]. However, the cache stores redundant copies of data in the SRF and incurs a high area overhead (100%-150% if capacity of the cache is

Parameter	Base	ISRF1	ISRF4	Cache
Lanes	8			
System clock	1 GHz			
Peak compute	32 GFLOPs			
Peak DRAM bandwidth	9.14 GB/s			
SRF capacity	128 KB			
Peak sequential SRF bandwidth	32 words/cycle (128 GB/s)			
Sequential SRF latency	3 cycles			
Stream buffer size (per lane per stream)	8 words			
Address FIFO size (per lane per stream)	-	8	8	-
Peak in-lane indexed SRF bandwidth (words/cycle/cluster)	-	1	4	-
Peak cross-lane indexed SRF bandwidth (words/cycle/cluster)	-	1	1	-
In-lane indexed SRF latency (cycles)	-	4	4	-
Cross-lane indexed SRF latency (cycles)	-	6	6	-
Cache size (KB)	-	-	-	128
Cache associativity	-	-	-	4
Cache banks	-	-	-	4
Peak cache bandwidth (GB/s)	-	-	-	16
Cache line size (words)	-	-	-	2
Cache replacement policy	-	-	-	LRU

Table 4.2: Machine parameters for indexed SRF evaluation (SRF access latencies shown for the case with no arbitration failures or bank conflicts)

comparable to that of the SRF) over a sequentially accessed SRF. Cache parameters such as the short line size are based on previously published vector cache studies [Asa98; KSF$^+$94]. Caching is only performed for streams with potential for temporal locality in order to minimize cache pollution. All machine configurations assume 4 fully pipelined functional units per lane, each of which supports integer and floating-point operations. SRF and cache accesses are fully pipelined.

Simulation Methodology

All benchmark simulations were performed using a cycle accurate simulator. Benchmark kernels were written in KernelC and scheduled using an automated scheduler based on [Mat02]. Ideally, indexed SRF reads would be scheduled such that addresses are issued as early as possible and data read as late as possible (subject to critical path and buffering constraints) in order to minimize stalls due to bank and sub-bank conflicts. However, the scheduler used for these evaluations does not support variable latency operations, and therefore all benchmarks were scheduled with a fixed address and data separation of 6 cycles for in-lane accesses and 20 cycles for cross-lane accesses. The sensitivity of performance to address and data separation is explored in section 4.3.7.

Benchmarks

To evaluate the performance impact of indexed SRF access, we simulated a set of application benchmarks and a synthetic benchmark representative of data parallel applications with complex access patterns. The synthetic benchmark was parameterized to rapidly explore a wide range of the application space. Note that indexed SRF access does not benefit all applications - particularly those that only require sequential streams. However, as stream programming extends to application classes with complex data access patterns, such as scientific computing and sophisticated signal and media processing algorithms, indexed SRF access will benefit a significant portion of stream applications. The benchmarks are described below.

2D FFT: 2-dimensional FFT on 64x64 array. The entire array fits in the SRF. Both the base and indexed SRF implementations perform each of the 1D FFTs along the first dimension across all lanes. The base version then performs a 90° rotation of the data array through memory and applies the same computation along the second dimension. In the indexed SRF version, each cluster performs the second dimension FFT for the data in its local bank of the SRF using in-lane indexed accesses.

Rijndael: A block encryption algorithm that was recently adopted as the Advanced Encryption Standard [DR02]. While the algorithm can be implemented in many ways, we consider an optimized implementation that relies on large numbers of lookups into pre-computed tables [FD01]. In the base case, the table lookups generate gathers from memory while the indexed case performs the lookups in the SRF. In order to facilitate high lookup bandwidth, the tables are replicated in each lane, enabling in-lane indexed SRF access. Both versions implement the *cipher block chaining* mode of the algorithm, with each cluster encrypting an independent data stream. Such an implementation is suitable for encrypting network traffic or other applications with many independent data streams.

Sort: Merge sort of 4096 values. Each iteration of the algorithm requires conditional merging of two input streams, which in a sequential SRF requires the use of conditional streams [KDR+00], resulting in cross-lane communication on every iteration. With SRF indexing, the conditional inputs are formulated as conditional address computations, and no cross-lane communication is necessary until all data in each lane are internally sorted.

Filter: Application of a 5x5 convolution filter to a 256x256 2D image. The base implementation temporarily stores neighborhood data in scratchpad memories to avoid replication in the SRF while the indexed SRF version simply reads the neighborhood data directly from the SRF.

Irregular Graph Simulation (IG): Synthetic benchmark that simulates neighbor interactions in a static irregular graph. For each node in the graph, all of its neighbors are accessed, and the node value is updated based on the neighbors' values. The graph is assumed to be much larger than the available SRF space, requiring the data set to be partitioned into several strips. No data is replicated across lanes, and therefore, all indexed SRF accesses are cross-lane. Amount of computation per neighbor access, average graph degree, and strip length are parameterized to explore the application space. Table 4.3 summarizes the parameter values for a cross section of the data sets explored. The three letter suffix at the end of the data set names are as follows: the first letter indicates a "Sparse" or "Dense" graph – an indication of the average number of neighbors per node; the second letter specifies whether the data set is "Compute" or "Memory" limited on the base architecture; and the third letter indicates "Short" or "Long" strip size. The strip sizes for the base and indexed SRF implementations of each data set were set to occupy approximately the same storage space in the SRF.

Data set	**FP ops per neighbor**	**Avg. graph degree**	**Avg. strip size**	
			Base SRF	**Indexed SRF**
IG_SML	16	4	1163	2316
IG_SCL	51	4	1163	2316
IG_DMS	16	16	265	528
IG_DCS	16	16	265	528

Table 4.3: Parameters for IG benchmark datasets (strip size is the average number of neighbor records processed per kernel invocation)

4.3.6 Results and Discussion

All results presented in this section assume the benchmarks are executed multiple times in software pipelined loops. This assumption is representative of most applications of interest since they are typically applied repeatedly to long input streams such as sequences of images or multiple strips of large, partitioned data sets.

Figure 4.15 shows the breakdown of execution time of the benchmarks for *Base*, *ISRF4*, and *Cache* (performance of *ISRF1* relative to *ISRF4* will be discussed later in this section). In the graph, the *kernel loop body* component corresponds to the time spent executing the main loops of the kernels where much of the useful computation is performed. *Memory stall* corresponds to time spent waiting for memory or cache accesses to complete. *SRF stall* is time spent stalling for SRF accesses to complete. *Kernel overheads* include time spent executing kernel code before and after the main loop body, including initializations, software pipeline fills and drains, and the impact of load imbalances among lanes.

As figure 4.15 shows, *ISRF4* provides speedups over the *Base* configuration for all benchmarks. *FFT 2D* and *Rijndael* on the *Base* machine are constrained by memory bandwidth, and reducing memory traffic via indexed SRF access provides speedups of 2.24x and 4.11x respectively.

Improvements in the *Sort* benchmark on *ISRF4* are a result of efficient support for conditional SRF access reducing kernel loop execution time. The speedup of the *Filter* benchmark is due to efficient access of neighbor values directly from the SRF, also reducing kernel loop execution time.

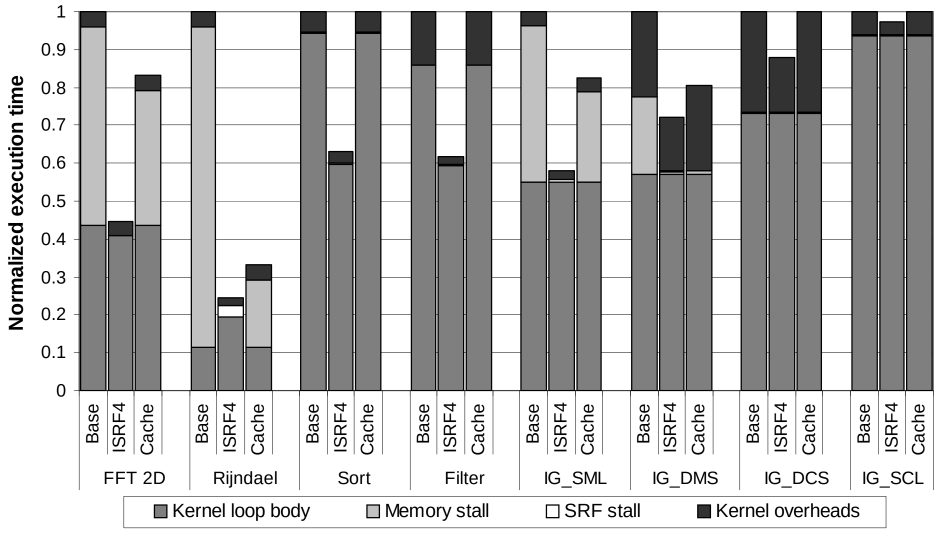


Figure 4.15: Benchmark performance impact of indexed SRF access

The *IG* benchmarks span a wide range of application characteristics. *IG_SML* and *IG_DMS* have low compute density and are constrained by memory bandwidth. Capturing intra-strip locality in these data sets in the SRF improves performance of both. Another factor contributing to improved performance on *ISRF4* is the increased strip sizes that can be accommodated in the SRF as a result of eliminating replication, which amortizes kernel start and end overheads over larger batches of useful computation. The two dominant components of these overheads for this benchmark are software pipeline overhead and load imbalance between lanes. While we do implement dynamic load balancing using the technique presented in [KDR$^+$00], some imbalance still exists at the end of each strip as all lanes remain occupied until the last lane has completed processing its final input. These overheads represent a significant portion of the run time for *IG_DMS* and *IG_DCS* which have shorter strip sizes. Other overheads such as application initialization operations are minimal in this benchmark but may be significant in real applications, which would further benefit *ISRF4*.

IG_SCL represents a scenario where SRF indexing provides little performance benefit since it is compute limited and has long strips even on the *Base* configuration.

ISRF4 also outperforms the *Cache* configuration for all benchmarks. The data set reordering of *FFT 2D* is fully captured in the cache, but unlike *ISRF4*, an explicit reordering operation must still be performed on the data in the SRF. Therefore, the software pipelined loop in the *Cache* configuration is longer than in *ISRF4*, requiring more data sets to be retained in the SRF concurrently. In this case, the SRF capacity limits the initiation interval, and hence is not able to achieve the best possible overlap of memory transfers and computation. For *Rijndael* and *IG_SML*, *Cache* captures at least as much locality as *ISRF4* but does not have adequate bandwidth to eliminate all memory stalls. The cache does not provide the conditional and complex SRF accesses enabled by *ISRF4* that benefit *Sort* and *Filter*, and consequently, does not provide any speedup for these benchmarks. The cache also does not eliminate data replication in the SRF, and therefore, does not provide the strip size increases that improve the performance of *IG_DMS* and *IG_DCS* on *ISRF4*.

Figure 4.16 shows the sustained SRF bandwidth demands in the main loops of the benchmarks executing on *ISRF4*. While the sustained bandwidths are relatively low, the access patterns are bursty, and indexed SRF bandwidth can become a bottleneck for some benchmarks. For example, figure 4.17 shows the execution breakdown for *Rijndael* and *Filter* benchmarks on *ISRF1* and *ISRF4*. While none of the benchmarks suffer significantly from a lack of indexed SRF bandwidth on *ISRF4*, *Rijndael* and *Filter* spend 42% and 18% of the execution time in SRF stalls on *ISRF1*, demonstrating the need for high indexed SRF bandwidth. The indexed SRF implementation described in this section limits each indexed stream to issuing a single indexed SRF access per cycle for simplicity. Therefore, *ISRF1* and *ISRF4* differ only for benchmarks with more than one indexed stream. In our benchmark set, only *Rijndael* and *Filter* require multiple indexed streams, and therefore, all other benchmarks perform identically on *ISRF1* and *ISRF4*.

Figure 4.18 shows off-chip memory bandwidth requirements of the benchmarks for *ISRF* and *Cache* configurations, normalized to the *Base* case (note that *ISRF1* and *ISRF4* have identical memory bandwidth requirements). Indexed SRF access provides significant bandwidth savings for all benchmarks except *Sort* and *Filter*.

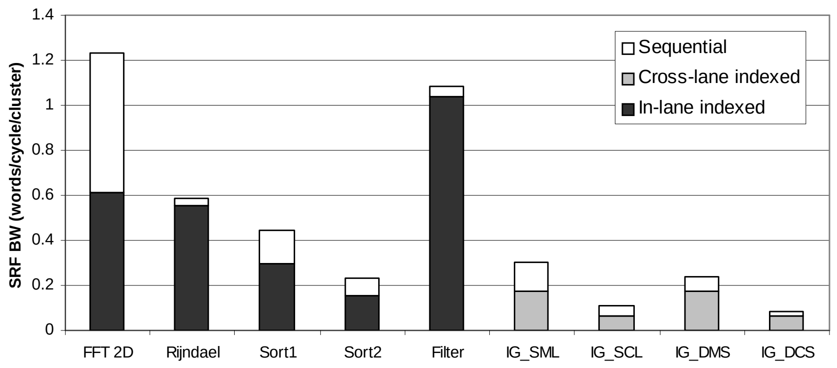


Figure 4.16: Sustained SRF bandwidth (both *Sort1* and *Sort2* kernels are used by the *Sort* benchmark)

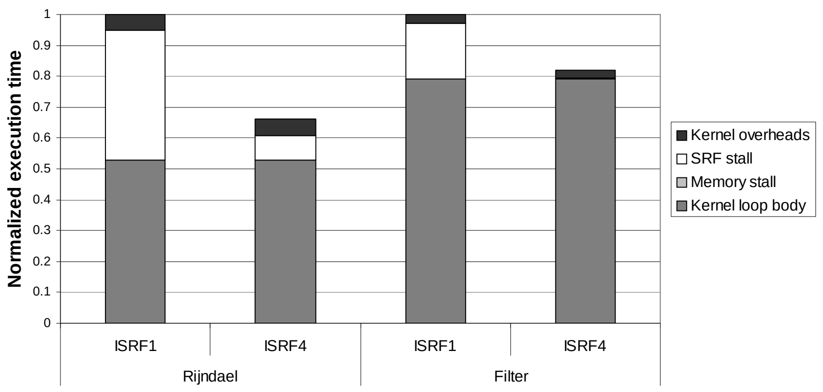


Figure 4.17: Impact of indexed SRF bandwidth on benchmark performance

FFT 2D benefits by eliminating a dataset reordering through memory, and *Rijndael* benefits by eliminating large numbers of table lookups from memory as the tables fit in the SRF. Reductions in the *IG* benchmarks for ISRF are due to eliminating intra-strip node and neighbor record replication, partially offset by the overhead of indices (pointers) into the condensed neighbor data array. *Sort* and *Filter* do not gain any bandwidth reduction as all available locality is captured by the base configuration as well.

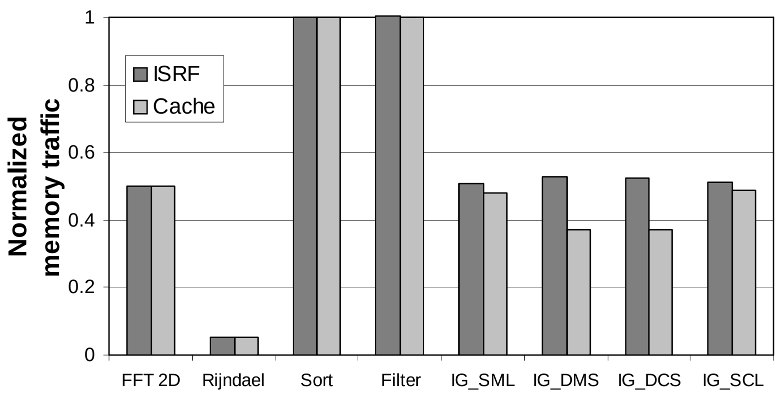


Figure 4.18: Off-chip memory traffic generated by indexed SRF and *Cache* machine organizations, normalized to *Base*

Cache also completely captures the *FFT 2D* data reordering and *Rijndael* table lookups, achieving bandwidth reductions similar to ISRF for those benchmarks. *Cache* outperforms *ISRF* in terms of locality capture for the irregular (*IG*) benchmarks as it is also able to capture inter-strip irregular temporal locality when partial overlaps exist between strips of the data set. An in-depth comparison of the locality capture provided by a cache compared to a SRF can be found in chapter 5.

4.3.7 Sensitivity to Access Latency

As discussed in section 4.3.4, issuing SRF addresses for read operations early enough in order for the read to complete before the data is needed is critical for reducing SRF stalls. However, increasing the separation between address and data can extend the static schedule length of kernels leading to a loss of performance. Figure 4.19 shows the variation of static schedule lengths of the inner loops of benchmark kernels as address and data separation increases. This separation is varied from 2 to 10 cycles for in-lane indexed access and from 2 to 24 cycles for cross-lane indexed access. The *IGraph1* kernel is used in *IG_SML* and *IG_DMS* benchmarks, and *IGraph2* is used in *IG_DCS* and *IG_SCL*.

Rijndael, *Sort1*, and *Sort2* kernels have loop-carried dependencies that affect index

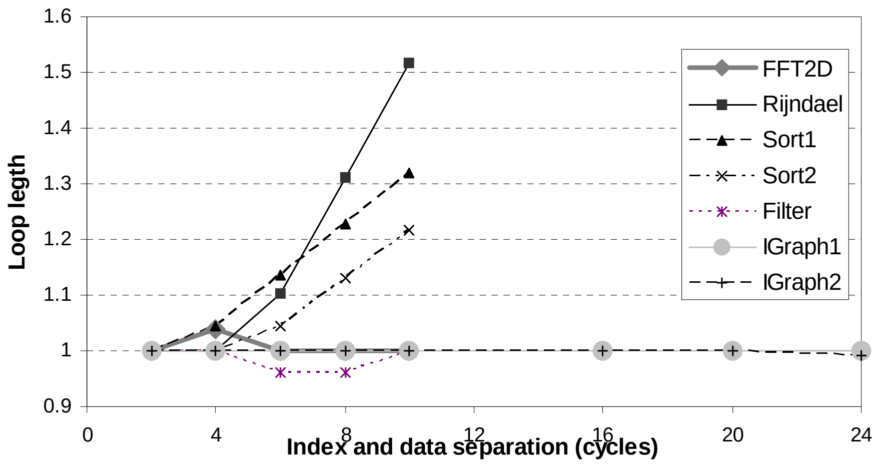


Figure 4.19: Static schedule length variation of loop bodies with address and data separation of indexed SRF accesses

computation of adjacent iterations, which causes schedule length to increase rapidly with address and data separation. *FFT 2D*, *Filter*, and the *IGraph* kernels, in contrast, are able to use software pipelining to tolerate very long separations with no increase in static schedule length. The minor fluctuations in schedule lengths are due to randomized algorithms used in the scheduler.

Figure 4.20 shows the variation in execution time of kernels as address and data separation increases for the in-lane indexed kernels. Performance initially improves for all benchmarks with increasing separation as SRF stalls reduce, and then degrades as static schedule length increases dominate. In the case of *FFT 2D*, which shows no schedule length increase, performance degradation occurs as a result of increased overheads due to deeper software pipelining.

Figure 4.21 shows the execution time variation for the cross-lane indexed kernels. These kernels are able to tolerate long address and data separations due to their high compute density and lack of loop-carried dependencies. The irregularity in the curve for *IGraph1* at 20 cycles corresponds to an increase in the software pipeline length.

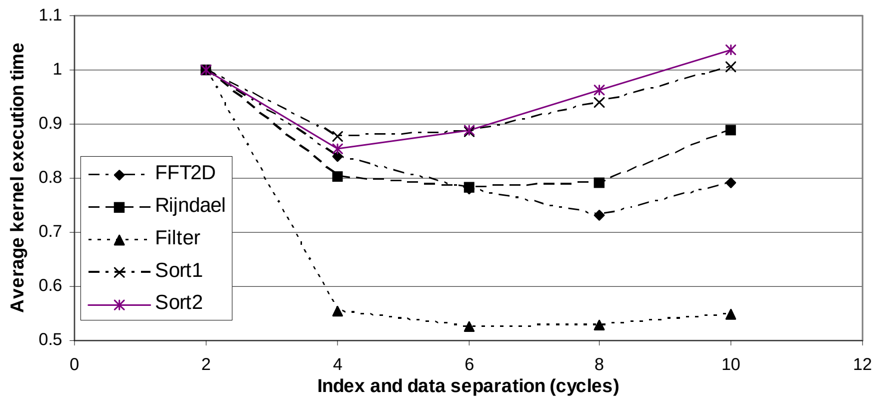


Figure 4.20: Execution time variation of benchmarks with address and data separation of in-lane indexed SRF accesses

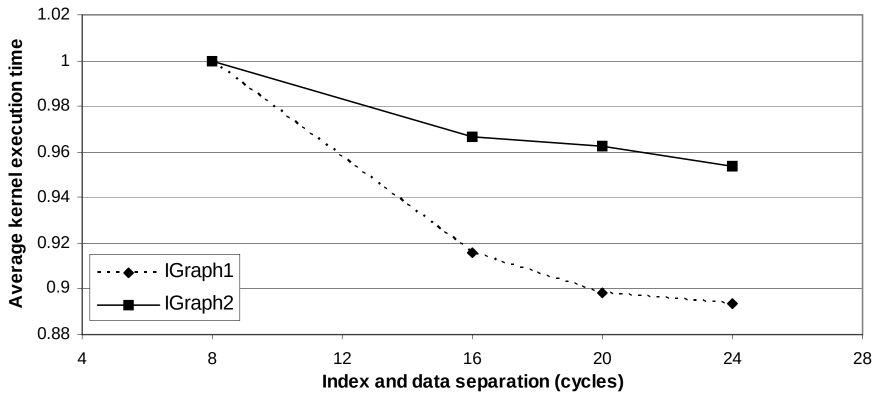


Figure 4.21: Execution time variation of benchmarks with address and data separation of cross-lane indexed SRF accesses

4.3.8 Cross-lane Communication Resource Utilization

Figure 4.22 shows cross-lane indexed access throughput as a function of the peak number of cross-lane accesses permitted in a bank of the SRF each cycle and the percentage of cycles in the static kernel schedule that contain inter-cluster communications unrelated to cross-lane SRF access. The results in figure 4.22 were obtained by issuing 1 pseudo-random cross-cluster read and 3 sequential stream accesses per cycle per cluster.

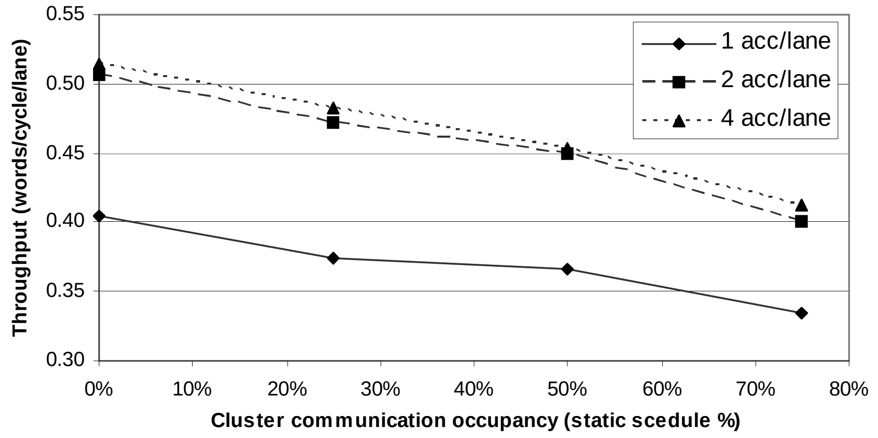


Figure 4.22: Cross-lane indexed throughput variation with number of SRF accesses per cycle and inter-cluster traffic

Increasing the number of cross-lane network ports per SRF bank from 1 to 2 provides a significant improvement in throughput, while increasing this number beyond 2 provides only marginal improvements. Adding an additional network port on the SRF side alone in linear lane arrangements such as the one in figure 4.13 does not require an increase in the bisection bandwidth and thus can be implemented with relatively low hardware overhead. However, this is not the case for more complex layouts like the 2D grid of lanes proposed in [Kha03]. Consequently, the results presented in section 4.3.6 were obtained using one port per lane on both the SRF side and the cluster side into the cross-lane network.

Figure 4.22 also shows that the reduction in cross-lane SRF throughput is 20% or less for a wide range of inter-cluster communication traffic loads. This shows that in the presence of both in-lane and cross-lane SRF traffic, the dominant factor in reducing cross-lane access throughput is contention for SRF access rather than inter-cluster traffic. Therefore, multiplexing both types of inter-lane traffic over a single network instead of two dedicated networks is the preferred design option, particularly given the high area cost of the networks.

4.3.9 Area and Energy Impact

Area overheads for the SRF designs with indexed access were estimated using models extracted from CACTI 3.0 [SJ01]. *ISRF1* and *ISRF4* configurations incur 11% and 18% area overheads over a sequential-only SRF of equal capacity. The area increase in *ISRF1* is largely due to the addition of a dedicated row decoder per SRF bank and the address FIFOs. Much of the extra overhead of *ISRF4* over *ISRF1* is in the multiple address busses that need to be communicated to the SRF, and the addition of a predecoder for each sub-bank. While the predecoders themselves do not consume much area, they dictate the spacing between sub-banks within SRF banks leading to a significant area increase. The additional column multiplexors needed for *ISRF4* and their control logic fit within the spacing between subarrays as dictated by the predecoders.

Area overhead of cross-lane indexing is 22% over a sequential SRF, including the 18% required for in-lane *ISRF4* implementation. Much of the incremental overhead over *ISRF4* is associated with the address network. Note, however, that SRF addresses are short (10 to 20 bits based on current sizes), and therefore the address switch consumes less area than the 32- or 64-bit data networks.

The above overheads are expressed as a fraction of SRF area only. When computed as a percentage of overall die area, they represent increases of 1.5% to 3% based on the Imagine processor statistics reported in [KDC$^+$02]. Therefore, fully flexible communication at the SRF level can be supported with only a modest increase in the overall area of a stream processor.

SRF energy consumption for sequential stream accesses is comparable for both the indexed and sequential-only designs. Indexed accesses in the design presented in this chapter consume approximately 4x the energy per word in the SRAM array compared to sequential stream accesses due to increased column multiplexing. However, the estimated energy consumed by an in-lane indexed SRF access is approximately 0.1nJ per word in a 0.13μ technology based on the energy models of CACTI 3.0. This access energy is still an order of magnitude lower than the approximately 5nJ required for an off-chip DRAM access. In addition, as seen from the application examples of section 4.2.2, a singe indexed SRF access often replaces multiple memory and sequential SRF accesses leading to significant energy savings.

Design Scalability

Stream processors may be scaled along multiple axes. The number of arithmetic units per cluster could be scaled, requiring corresponding changes in SRF bandwidth per lane and intra-cluster network bandwidth. Alternatively, the number of lanes could be scaled, keeping the number of resources per lane constant. In this case, cross-lane communication resources must be scaled accordingly. [Kha03] presents a detailed analysis of stream processor scaling along these axes, and shows that from a hardware scalability perspective, the preferred option is to scale the number of lanes.

A key consideration in implementing an indexable SRF across a large number of clusters is the scalability of the cross-cluster data and address networks. The study by [Kha03] shows, however, that scaling the inter-cluster data network up to 256 lanes can be achieved efficiently by placing the lanes as a 2D grid. Such a layout of lanes results in comparably efficient scaling of the address network as well. Fully pipelined implementations of these networks enables the throughput to be sustained despite increased latencies. From an application perspective, those with cross-lane indexed SRF accesses typically have few or no loop carried data dependencies, enabling deep software pipelining, and thus high degrees of latency tolerance. This behavior for the irregular graph benchmark was demonstrated in section 4.3.7. Therefore, the communication resource scaling can be managed efficiently, and the added latency tolerated by applications for up to hundreds of lanes.

A second consideration in scaling to large numbers of clusters is the added latency incurred in control signals. In particular, the stall signals that must be broadcast among all lanes when an indexed SRF access fails in any one cluster must be designed to tolerate multi-cycle wire delays. A simple implementation would be to extend the indexed access pipeline (both in-lane and cross-lane) to allow additional stages for stall broadcast before data is used. However, this extends the address to data separation of indexed reads, which can be detrimental to some applications with in-lane indexed SRF access and loop-carried data dependencies. A more complex but efficient implementation is to allow some amount of slip between the clusters.

If cluster c stalls on an SRF access in cycle i, that cluster stalls immediately, but other clusters continue to execute until the global stall signal is communicated to all clusters in cycle $i + n$, where n is the global broadcast latency. When the stalling condition is relieved in cycle j, cluster c resumes immediately, but the other clusters do not resume until cycle $j + n$ when the global stall de-assert broadcast reaches all clusters. This technique requires the ability to buffer up to n instructions and n cross-lane communication data words at each cluster during stalls since instruction broadcast continues for n cycles after the initial stall. Further, global stall signals received by clusters must be correlated with the cycle they are received in order to handle multiple overlapping stalls occurring in separate clusters. This correlation can be implemented by buffering stall signals in the n-deep instruction buffer, along with the instruction that was received in the same cycle as the stall signal.

A third axis of stream processor scaling is to place multiple stream processors on a single processor die. In such an implementation, the SRF is treated as a local memory within each processor incurring no additional complexity in terms of scaling. However, such a multi-processor may incorporate other memory structures that are shared by all processors on the die, which may introduce additional levels to the bandwidth hierarchy, and corresponding communication resources may need to be provisioned.

4.3.10 Further Optimizations

The indexed SRF implementation presented in this section makes several tradeoffs that are sub-optimal for performance in order to reduce implementation complexity. Alternative design decisions that relieve a few of these performance drawbacks are discussed here. However, it should be noted that as seen by the results in figure 4.15, the implementation as proposed provides sufficient bandwidth to meet the demands of the benchmarks explored to date.

In the implementation evaluated here, only the head entry of each address FIFO is considered in arbitration for the SRF. This results in two constraints. First, an indexed SRF stream may perform at most one access on each SRF cycle, even if there are no other pending requests from other streams. Second, no reordering among accesses for the same stream is possible within a lane. However, these constraints enable important implementation simplifications. The arbiters for determining access to each sub-bank only need to consider a single entry per address FIFO, reducing their complexity and latency over the case where multiple entries must be considered from each FIFO. In addition, by avoiding reorderings, the existing FIFO interface of the stream buffers is retained.

The limitation of a single access per cycle per indexed stream could potentially lead to low utilization of the available indexed SRF bandwidth. However, this can easily be worked around at the application level. For example, multiple streams can be defined at the application level, all of which access the same data set. Then, the indexed accesses can be interleaved or otherwise distributed over these multiple streams, providing the hardware view of multiple simultaneous indexed streams.

The inability to reorder accesses within a stream could potentially lead to *head-of-line blocking*, where subsequent entries in the FIFO are held back due to a bank conflict for the head entry. This effect is reflected in results of figure 4.23, which were obtained using a micro-benchmark that issues 4 pseudo-random reads per cluster on every cycle. As the number of sub-banks per banks increases, the achieved bandwidth increase is lower than the anticipated peak due to head-of-line blocking. Note that while increasing the address FIFO size helps by reducing cluster stalls due to address FIFOs filling up, the gains saturate before the anticipated peak. One potential solutions is to allow reordered accesses from the address FIFOs. Requests may be sorted by destination sub-array, enabling a request for each sub-bank to be issued every cycle. Resorting the data returns into access access order may be performed in the stream buffers. Such an implementation, while not impractical, adds complexity to the arbitration mechanism and stream buffers, and was not used in the results presented in this chapter since in-order bandwidth was sufficient to satisfy application requirements.

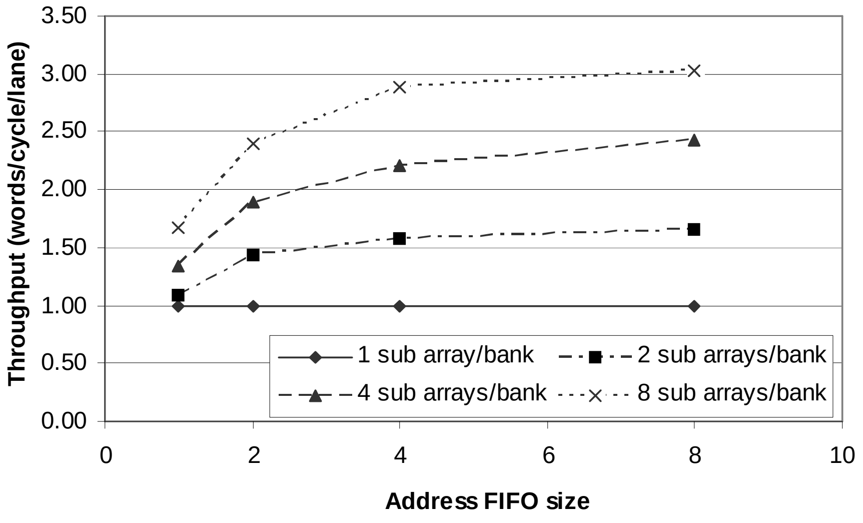


Figure 4.23: In-lane indexed SRF throughput scaling with the number of sub-banks per SRF bank and address FIFO size

4.4 Summary

This chapter presented a classification of the types of data reuse found in streaming applications, and the levels of the bandwidth hierarchy these reuse types correspond to. However, due to parallel execution and the multi-element nature of streams, a subset of the data reuse in time manifests as communication in space at execution time. Current stream (and vector) processor architectures do not provide a complete communication hierarchy that enables irregular data reuse to be captured at the SRF (or VRF) level of the bandwidth hierarchy. Irregular data reuse at the stream (or vector) level occurs frequently in applications with complex data access patterns, such as signal processing applications with multi-dimensional data structures and scientific applications.

This chapter presented an SRF design that enables full communication freedom at the stream level, enabling a broad range of data reuse types at that level to be captured. This design was evaluated using a set of benchmarks that require complex access patterns, and was found to provide speedups ranging from 3% to 411%.

Further, by capturing stream level communication at the SRF, memory system bandwidth requirements were reduced by up to 50% for the benchmarks considered.

A microarchitecture for implementing the proposed SRF design was also presented. By exploiting the existing circuit structure within the SRF and reusing the inter-cluster network, the implementation was shown to be possible with modest area overheads. The area increase for supporting indexed SRF access was estimated at approximately 3% of the die area in the case of the Imagine parocessor. In addition, stream level communication at the SRF was shown to replace multiple memory and SRF accesses with a single indexed access in some cases, leading to reduced energy consumption in the bandwidth hierarchy.

Chapter 5

Software- vs. Hardware-Managed Memories

Architectures specialized for data parallel computing, such as stream and vector processors, have typically implemented on-chip memory in the form of register files, such as SRF/LRFs and VRFs, especially in single-chip implementations of such processors. These register resources are allocated and managed explicitly by compile-time and run-time software. However several trends in applications and technology have lead to many data parallel computations being performed on memory systems that include standard cache memories which are allocated and managed by hardware with no direct software control.

On the applications front, the popularity of media and signal processing applications have lead to many such applications being executed on a wide range of architectures. A large fraction of these highly data-parallel applications are therefore executed on general-purpose processors with cache-based memory hierarchies. Similarly, commodity-processor-based supercomputers, clusters, and workstations are used for executing many data-parallel scientific applications.

On the technology front, the increasing number of devices available on a single chip enables heterogeneous multi-processors that integrate both general-purpose and data-parallel processor cores on a single die [PAB+05; EAE+02; Bor04]. In many such designs, the data parallel execution units share part of the on-chip memory hierarchy, which often consists of caches, with the general-purpose cores. On the other hand, the increasing device counts also enable dedicated data parallel architectures to incorporate hardware-managed caches on-chip along with the traditional local and stream or vector register files [DHE$^+$03; Cra02].

This chapter compares the performance characteristics of stream processors with cache-based and SRF-based memory hierarchies. We correlate benchmark simulation results with an application space characterization based on stream accesses, enabling conclusions to be drawn about the advantages and disadvantages of each type of memory based on an application's requirements. Further, the inefficiencies of cache systems for stream computing are highlighted, establishing a set of desired cache characteristics in order to minimize these inefficiencies. This chapter also introduces *epoch based invalidation*, a hardware-software technique for reducing off-chip memory traffic generated by cache-based systems through active invalidation of cached data at the end of their live ranges. Throughout this chapter, the terms *hardware-managed memory* and *cache* are used interchangeably. Similarly, the terms *software-managed memory* and *SRF* are used interchangeably.

The evaluations in this chapter are limited to systems with either a cache or a SRF only. Characteristics of hybrid bandwidth hierarchies that incorporate both a cache and a SRF will be explored in chapter 6.

5.1 Sources of Performance Differences

Several differences exist between software- and hardware-managed memories that affect their performance for streaming applications. The most important of these are described below.

Name translation: hardware-managed memories, such as caches, are typically addressed using main memory addresses. Therefore, a mapping function is required for translating main memory addresses to on-chip locations. Such mapping functions are implemented in hardware (e.g. tag lookup in caches), and hence require little execution time, enabling fine-grain (word-granularity) address translation on every access. However, in order to reduce the implementation area cost, access latency, and access energy, these hardware mapping functions often restrict data placement freedom in on-chip memory. For example, associativity of caches, which determines the mapping freedom of an off-chip address to an on-chip location, is often limited in real systems. On the other hand, software-managed memories explicitly maintain a separate address space for the on-chip memory. Name translation between the address spaces is done in software, and therefore, there is significantly more freedom to map any off-chip address to any location in on-chip memory. However, as the name translation incurs run-time software overheads, it can only be done efficiently for large blocks of data that allows the overheads to be amortized over many data elements. As a result, software name translation in stream processors is done at the granularity of entire streams, requiring each stream to be contiguous within the on-chip memory.

Replacement policy: When a new data item is allocated in on-chip memory, an existing data item may need to be replaced if no free location exists within the set of locations the new item can map to. In a hardware-managed memory, replacement of data items is done using an application-independent fixed policy such as least-recently used (LRU). This could potentially lead to replacing data items that the application may still need to access, resulting in added off-chip memory traffic to re-fetch that data. In addition, when replacing data items that have been modified by the processor, hardware policies must always preserve these changes by writing them to off-chip memory since the replacement policy is unaware of the live ranges of data in the applications. However, due to efficient hardware implementations, the replacement decisions require little time, and hence can be made at a relatively fine cache-line granularity. Software-managed replacement schemes can factor in application-specific characteristics (e.g. live ranges of data items) using complex analyses to make more accurate replacement decisions. Given this application-specific knowledge, software schemes can also simply overwrite modified data in on-chip memory that are no longer live, without writing that data to off-chip memory. However, software schemes are again limited to dealing with large data blocks in order to amortize run-time software overheads.

Data fetch policy: When a data item is to be read by the compute units, and the data is not already known to be present in on-chip memory, it must be fetched from off-chip memory. In a straight-forward cache implementation, the absence of such data in the on-chip memory is detected only once the access is attempted and misses in the cache. Therefore, execution of operations that depend on the missed data must stall until the data access from off-chip memory completes. In a software-managed memory, since application data use is statically analyzed, the available data parallelism in streaming applications is exploited to schedule the data to be fetched sufficiently in advance to avoid stalling. Similar analyses, however, can be used in caches to prefetch the data in to the on-chip memory to reduce cache misses and resulting stalls. However, since such prefetches must first check the presence or absence of the data in the cache, the bandwidth requirements of the name translation mechanism is increased.

Implementation constraints: hardware-managed memories often incur additional area overheads for the implementation of the name-translation functions mentioned above (e.g. storage for cache tags). More importantly, the name translation function can often be a bandwidth bottleneck as well. For example, in an n-way set-associative cache, n tag lookups must be performed for address translation on each single data memory access. Therefore, in order to sustain a given b data bandwidth, $b \times n$ bandwidth must be sustained for tag lookups. This manifests in realistic implementations as either increased overheads due to the high tag bandwidth required, or reduced name translation flexibility as n is made smaller to reduce implementation cost.

It should be noted that the software overheads incurred by software-managed memories for name translation and replacement policy are divided between compile-time and run-time. Much of the complex analyses can be done at compile-time, possibly using profiling, incurring only a one-time penalty. However, high run-time overheads are incurred, in terms of time and/or register capacity, in the presence of large numbers of fine-grain data items due to the need for maintaining and manipulating base, bounds, and dependency information associated with each. Aggregating all data in to a small number of coarse-grain data items, such as streams consisting of up to thousands of words that are placed contiguously within the memory, significantly reduces this overhead.

Programming and Compilation

Another source of significant difference between software- and hardware-managed memories is the programming and compilation effort required. Hardware-managed memories require little programmer or compiler intervention beyond the partitioning of data sets in order to fit the working set in on-chip memory. Software-managed memories, in addition to data set partitioning, require explicit instructions from the programmer or compiler to orchestrate transfers to/from off-chip memory and to manage the name translation and replacement policy tasks. As a result, achieving high performance on software-managed memories typically require more sophisticated analyses on the part of the programmer and compiler, while hardware-managed memories may provide performance improvements, albeit suboptimal, for naive programming and compilation efforts. For the evaluations in this chapter, we do not factor in programming effort for three reasons. First, programming effort and compiler sophistication are difficult to quantify, particularly given the recency of stream compilation, which lacks the decades-long development history that underpin conventional programming and compilation technologies. Second, application-domain-expert programmers in fields such as media, signal processing, and scientific computing have historically implemented applications in their respective domains that are highly tuned to the capabilities of available technologies, often leveraging as much control of the underlying hardware as afforded by the programming system. For such programmers, the added control provided by software-managed memories could potentially provide opportunities for added optimizations. Finally, the benchmarks used in these evaluations were implemented using the stream programming model, which already requires applications to be implemented in a manner amenable to software-controlled memories. Further, the additional locality information exposed by this programming model can be of use under both software- and hardware-managed memories.

5.2 Memory Use Characteristics of Applications

The manner in which an application uses on-chip memory determines its performance sensitivity to the sources of differences between software and hardware-managed memories. As discussed in chapter 2, on-chip memory is used by streaming applications for two primary purposes. First, the temporal and producer-consumer locality in intermediate stream state of applications is captured in the on-chip memory. This reuse can be classified based on the type and granularity as discussed in section 4.1. Second, off-chip accesses are staged via the on-chip memory, enabling computation on data already available on-chip to be overlapped with the long-latency transfers of data to and from off-chip memory.

Table 5.1 lists a set of stream application benchmarks used for analyzing the performance of software and hardware-managed memories in this chapter. These benchmarks were selected to cover some of the most important data parallel application classes and a varied mix of the data reuse characteristics discussed in section 4.1. Figure 5.1 shows the distribution of on-chip memory accesses performed by these benchmarks at the stream level of the bandwidth hierarchy, assuming an ideal on-chip memory with unlimited capacity. *Prefetch* traffic shows the accesses performed for reading application inputs which require loads from off-chip memory. *Post-store* accesses are those performed for writing application results to off-chip memory. The remaining categories correspond to the classification of stream locality types discussed in section 4.1.

The media applications exhibit large amounts of stream granularity producer-consumer reuse, which results from the prevalence of sequential intermediate streams in these applications. On the other hand, the scientific applications exhibit significant amounts of temporal locality as well (especially *MD*). This corresponds to the frequent gather and scatter accesses to large, irregular data structures found in these applications, which often result in intra-stream temporal reuse as well as partial overlaps between strips, leading to inter-stream reuse. The irregular temporal reuse seen in *Rijndael* result from large numbers of accesses to small lookup tables.

Irregular reuse, particularly when present in the form of inter-stream reuse, is not

Media and Encryption Benchmarks	
FFT 2D	64x64 2D FFT. On-chip memory accesses are dominated by stream-granularity and irregular producer-consumer reuse. Array padding is used to reduce memory bank conflicts
MPEG	MPEG 2 encoder. On-chip memory accesses are dominated by stream-granularity producer-consumer reuse.
Rijndael	Optimized implementation of AES encryption standard. On-chip memory accesses are dominated by irregular temporal reuse due to table lookups. However, the working set (mostly lookup tables) is small enough to fit in on-chip memory
Scientific Computing Benchmarks	
FEM 3D	Finite element application designed for solving systems of first order conservation laws on general 3D unstructured meshes. On-chip memory accesses are largely a mix of temporal producer-consumer reuse
FEM 2D	Finite element application on 2D unstructured meshes. On-chip memory accesses are again a mix of temporal producer-consumer reuse, but with a higher percentage of irregular temporal reuse compared to the 3D case
MD	Molecular dynamics simulation based on GROMACS [vdSvBA+01]. On-chip memory accesses are dominated by irregular temporal locality

Table 5.1: Application benchmarks used to compare stream computing performance on software- and hardware-managed on-chip memories

efficiently captured by software-managed memories due to the coarse granularity at which name translation and replacement are performed. The simple example in figure 5.2 illustrates this inefficiency. In this example, two streams are derived from an array of eight records in memory. Each of the two streams consist of a subset of the records. In a software-managed memory, name translation is performed at the granularity of entire streams, and therefore, all records of a stream must be stored contiguously in the on-chip memory as shown in figure 5.2(a). As a result, the data reuse due to partial overlap between the streams (e.g. records 3 and 6 that are common to both streams in this example) are not captured in the software-managed on-chip memory. In a hardware-managed memory, address translation is performed at a fine granularity, and therefore, each data item in memory creates only a single copy in the on-chip

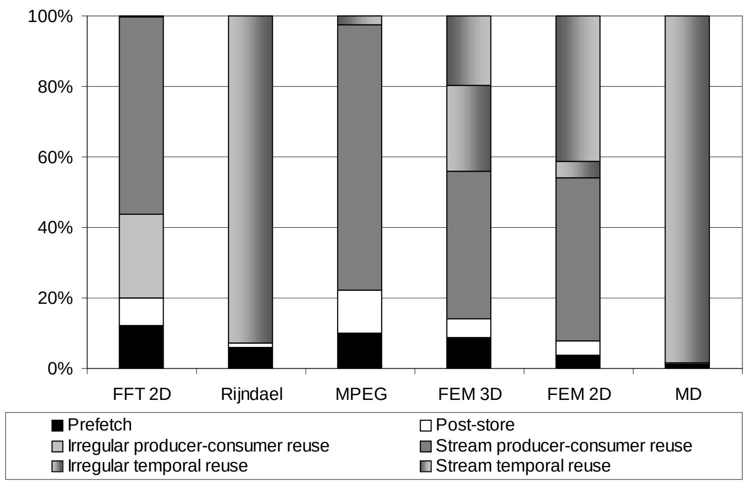
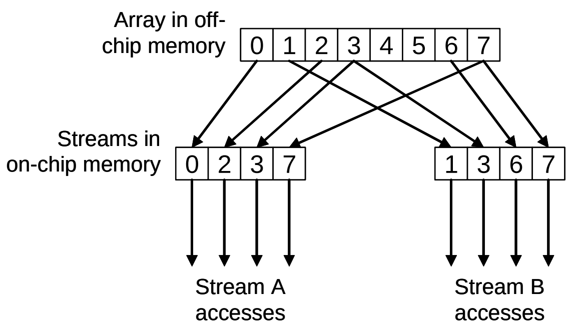
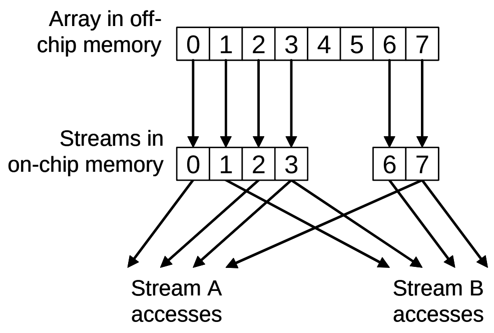


Figure 5.1: On-chip memory access classification of benchmarks used in the comparison of software- and hardware-managed memories for stream computing

memory as shown in figure 5.2(b). As a result, hardware-managed memories can more efficiently capture this form of inter-stream irregular locality unless the access pattern leads to conflict or capacity misses. However, software-managed memories can efficiently exploit inter-stream irregular reuse is cases where the source array that streams are derived from entirely fits in the on-chip memory (as is the case with the lookup tables in *Rijndael*), or when one of the streams is a subset of the other.



(a) Software-managed on-chip memory



(b) Hardware-managed on-chip memory

Figure 5.2: Example of inter-stream irregular reuse

5.3 Performance Evaluation

The performance of the benchmarks listed in section 5.2 were analyzed using software- and hardware-managed memory systems. The following sub-sections discuss the methodology, results, and implications for cache design for stream computing.

5.3.1 Evaluation Methodology

The evaluations in this section compare the performance of the benchmarks across two processor organizations. The first implements the on-chip memory as software-managed SRF and LRFs. This organization is similar to the stream processors that have been described in this thesis thus far. The second organization implements most of the on-chip memory as a hardware-managed cache. This organization is similar to a stream processor with its SRF replaced by a cache. The compute resources, which include N_{cl} compute clusters along with the associated LRFs, and the off-chip memory resources are identical for both processor organizations. Since the focus of this study is to understand the memory behavior of data parallel stream computation, the host processor is assumed to have an independent L1 cache, and its accesses are not modeled in both the software- and hardware-managed cases.

Key properties of the execution resources and off-chip memory system used for the evaluations are listed in table 5.2. Similar to chapter 3 the *Full* configuration is used for the scientific benchmarks, and its machine parameters were chosen to approximate a single node of the Merrimac architecture [DHE+03]. The *Lite* configuration is intended to approximate a media processor such as Imagine [RDK+98] in terms of the compute resources, and is used for the media and encryption benchmarks.

Benchmark simulations for the software-managed memory model were performed using a cycle-accurate simulator of the stream processor and the bandwidth hierarchy. Access traces from these simulations were used to generate corresponding cache access traces, which were simulated on a cycle-accurate cache and memory system model to evaluate hardware-managed memory performance.

In the hardware-managed memory model, intermediate streams are allocated in

	Full	**Lite**
Clock frequency	1GHz	
Compute clusters (lanes)	16	8
Peak compute (GFLOPs)	128	64
LRF capacity per lane	768 words	
Word size	64 bits	
Peak DRAM bandwidth (GB/s)	38	19

Table 5.2: Processor parameters for software- and hardware-managed on-chip memory comparison

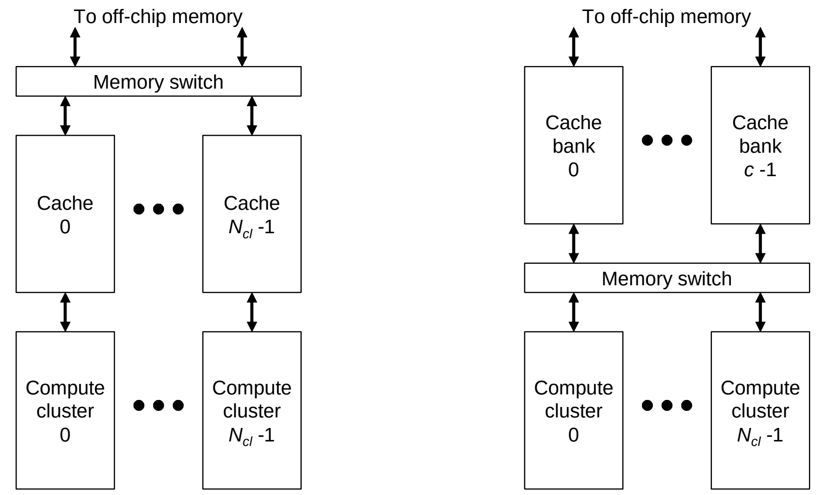
a contiguous region of memory of size equal to the SRF. Any streams that are allocated only in the SRF in the software-managed case are allocated in this region. This enables the same address range to be reused as new intermediate streams are allocated, reducing the likelihood of conflict and compulsory cache misses. In addition, allocated temporary streams are often reused where possible to further reduce cache misses.

The software-managed memory used for these evaluations is an SRF with N_{cl} banks. Key SRF parameters for the *Full* and *Lite* configurations are listed in table 5.3. Software management of the SRF is performed by an automated tool based on the *StreamC Compiler* described in [Mat02]. This tool uses profile-based analysis of stream flows between kernels to efficiently allocate SRF storage as well as to schedule stream post-stores and prefetches to and from off-chip memory.

	Full	**Lite**
SRF capacity (KB)	1024	256
Peak sequential SRF bandwidth	4 words per lane per cycle	
Peak in-lane indexed SRF bandwidth	4 words per lane per cycle	
Peak cross-lane indexed SRF bandwidth	1 word per lane per cycle	
Peak stream load/store address bandwidth	8 per cycle	4 per cycle

Table 5.3: SRF parameters for software- and hardware-managed on-chip memory comparison

The design options available in replacing the SRF of a stream processor with a hardware-managed cache fall into two classes. One is to replace each bank of the SRF with an independent cache. In such an organization, each cache is only accessed by the compute cluster directly associated with it in the same lane. Such an organization is shown in figure 5.3(a). The other option is to replace the entire SRF with a single, multi-banked cache that is shared by all compute clusters as shown in figure 5.3(b). In this arrangement, date elements are mapped to cache banks based on their addresses independently of the cluster in which the accesses originated.



(a) Independent cache per lane

(b) Single cache shared by all clusters

Figure 5.3: Cache implementation alternatives for stream processing

Individual caches in each lane, as shown in figure 5.3(a), enables localized communication between a compute cluster and its associated cache, much like the in-lane access of an SRF bank. However, achievable bandwidth is limited by the tag lookup requirements as discussed in section 5.1, and therefore the proximity does not provide bandwidth benefits similar to in-lane access in an SRF. A key drawback of independent caches per lane is the need to keep them coherent due to the possibility of replicated copies of the same data in multiple lanes. Hardware coherence mechanisms greatly increase the demands on tag bandwidth due to the high access rates sustained at the stream level. For example, the *FFT 2D* benchmark sustains over 1.2

words per cycle per lane of stream access bandwidth on average over the entire kernel execution as shown in figure 4.16, with approximately 50% of those accesses being writes. Even in applications with lower bandwidth requirements, stream accesses are often bursty, requiring high sustained bandwidth over certain parts of kernel execution. Therefore, coupled with SIMD execution of the clusters, sustaining the necessary coherence bandwidth requires each tag memory to sustain close to N_{cl} accesses per cycle, where N_{cl} is the number of compute clusters. Such high bandwidth clearly leads to impractical implementations for architectures with 8 to 16 clusters as is discussed here. Alternatively, software techniques may be used to maintain coherence among the caches. However, such techniques often require inter-lane communication to take place through memory, resulting in stream level cross-lane communication being exposed to the memory system, resulting in inefficient use of the bandwidth hierarchy.

A single cache shared by all clusters, as shown in figure 5.3(b) leads to all stream level accesses traversing the memory switch. This results in increased bandwidth requirements on the memory switch as well as increased energy consumption on stream accesses due to the switch traversal. However, since any given memory address can only be present in a single bank of the cache, no additional coherence support is required. In addition, for implementations where N_b, the number of cache banks, is comparable to N_{cl}, the bandwidth requirements of the memory switch are comparable to those of the inter-cluster switch, which has been shown to scale efficiently up to at least 256 clusters [Kha03]. Therefore, for the evaluations in this chapter, a single cache shared by all clusters is used as the hardware-managed memory configuration.

Four variants of the shared cache configuration were evaluated in order to ascertain the sources of performance differences between software- and hardware-managed memories:

- *Unlimited bandwidth cache (UBW)*: A cache implementation with each bank sustaining unlimited bandwidth. While not a realistic implementation, this configuration eliminates any performance bottlenecks due to bandwidth limitations. Stream reads are prefetched in advance to minimize cache misses.

- *High bandwidth cache (HBW)*: A cache implementation providing the same peak aggregate bandwidth as the SRF used for comparisons. As discussed in section 5.1, such an implementation requires several times higher sustained tag lookup bandwidth relative data bandwidth for a set associative implementation, increasing implementation overheads. The particular results presented for this configuration assumes true multi-ported tag and data memories, allowing each cache bank to sustain n accesses per cycle, where n is the peak accesses per bank per cycle of the SRF designs considered in the evaluations. While this is not a practical implementation, it is intended to normalize bandwidth differences between the hardware and software-managed configurations for evaluation purposes. Stream reads are prefetched in advance to minimize cache misses.

- *Realistic bandwidth cache (RBW)*: A cache implementation providing a peak bandwidth of a single access per bank per cycle. The degree of banking in this cache is held to the same as in the corresponding SRF implementation. This configuration corresponds to a realistic, implementable cache. Stream reads are prefetched in advance to minimize cache misses.

- *Realistic with no prefetch (RNP)*: Similar to the *RBW*, but with no prefetching for read accesses.

UBW, *HBW* and *RBW* cache configurations perform data prefetching in order to reduce load misses. The prefetches are scheduled based on analysis similar to that used for performing stream prefetches for the software-managed SRF. Prefetches for cache accesses are scheduled a fixed l number of cycles prior to the anticipated data use, where l is determined based on the expected memory access latency. One exception however, is for stream accesses within the first l cycles of a kernel. Kernels are compiled independently of each other, and the same compiled kernel may be reused in different parts of an application with different input and output streams. Therefore, it is not possible to schedule prefetches across kernel boundaries. As a result, for stream reads in the first l cycles of kernels, the prefetches are scheduled to be issued at the beginning of the kernel execution, subject to bandwidth limitations. All prefetches are issued at the granularity of a single prefetch per cache line.

Prefetch accesses are treated with lower priority than non-prefetch accesses throughout the on-chip memory hierarchy. At both the compute clusters and the cache, prefetch requests are maintained in separate queues from the other accesses. Prefetches are issued to subsequent levels of the memory hierarchy only after any pending non-prefetch accesses are issued. However, once accesses are issued to the DRAM controllers, the distinction between prefetch and non-prefetch accesses is lost.

Table 5.4 lists the key parameters for the *HBW*, *RBW* and *RNP* configurations[1] *UBW* parameters are identical to *HBW*, but with unlimited bandwidth. Note that the number of cache banks are identical to the number of compute clusters in each configuration. This design point was chosen to maintain the same degree of banking for both the software- and hardware-managed memories although the number of cache banks does not necessarily need to match the number of compute clusters.

	Full			**Lite**		
	HBW	**RBW**	**RNP**	**HBW**	**RBW**	**RNP**
Cache capacity (KB)	1024			256		
Cache banks	16			8		
Peak bandwidth (words per cycle)	64	16		32	8	
Line size	2 words (128 bits)					
Associativity	4-way set associative					
Replacement policy	Least recently used (LRU)					
Prefetch	Yes		No	Yes		No

Table 5.4: Cache parameters for software- and hardware-managed on-chip memory comparison

All cache configurations used for the evaluations in this section use short cache line sizes, which were shown to be desirable in chapter 3 of this thesis as well as prior literature on vector caches such as [KSF+94]. In addition, a separate valid bit is maintained for each word within cache lines, enabling write misses to be allocated in the

[1]Bandwidth of *RBW* may be increased for sequential accesses by allowing contiguous block accesses from the cache. While these evaluations do not support these accesses, *HBW* provides an upper bound for the benefit that can be achieved via increased bandwidth. In addition, results presented later in this chapter show that, for the benchmarks considered, those with contiguous access patterns are not bandwidth-constrained in the *RNP* configuration.

cache without fetching the line from off-chip memory. Since stream writes often update up to thousands of contiguous words, this eliminates loading values from memory that are immediately overwritten, reducing memory bandwidth requirements.

Buffers are added to enable bursty stream accesses to be distributed in time for *RBW* and *RNP* organizations, reducing the cache bandwidth bottleneck. The buffers at the compute clusters are not capacity bounded for the purposes of these evaluations, but the execution stalls if a cache access does not complete within 20 cycles. The cache hit latency for all configurations is 6 cycles when the system is unloaded.

5.3.2 Results and Discussion

Execution times of the benchmarks using the SRF and various cache organizations are shown in figure 5.4, normalized to the SRF case. In the graph, *Compute* indicates the time spent in kernel execution. *Memory stalls* indicate time spent stalling on memory accesses. *Resource stalls* correspond to stalls arising due to buffers in the memory system filling up as a result of insufficient bandwidth, exerting back-pressure which stalls kernel execution. Performance on *UBW* is identical to that on *HBW* for all benchmarks evaluated, and therefore is not shown.

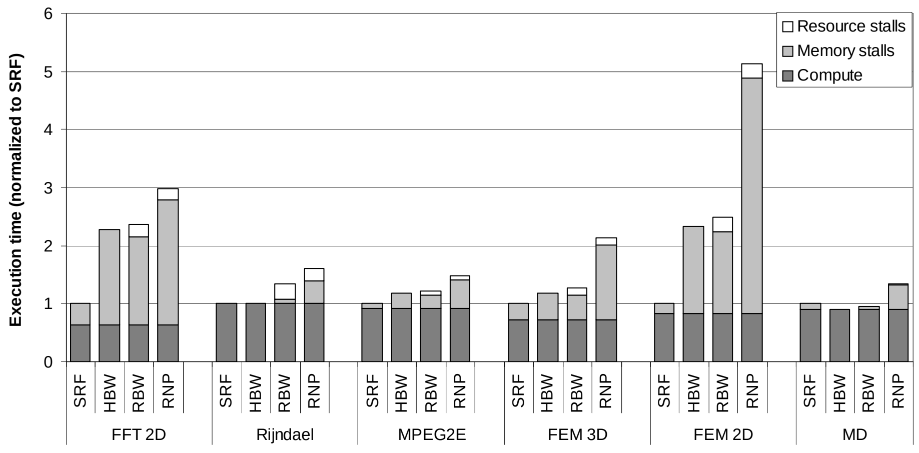


Figure 5.4: Benchmark execution times on software-managed SRF and hardware-managed cache configurations

For the *MD* benchmark, *SRF* underperforms all cache configurations except *RNP*. The access types classification of figure 5.1 shows that *MD* is dominated by irregular temporal reuse, and as was seen in section 4.3.6 as well, hardware-managed caches efficiently capture the available irregular temporal locality. This is a key advantage of the fine-grain name translation of hardware-managed memories over software-managed ones. Since the reuse is at the granularity of individual records, the software-managed memory cannot efficiently capture that locality, especially when the reuse is inter-stream, arising from partial overlaps between multiple streams. In addition, the particular implementation of *MD* used in this study does not utilize cross-lane indexed access to capture intra-stream irregular locality in the *SRF* configuration.

For all benchmarks other than *MD*, *SRF* performs as well or better than all cache configurations. *Rijndael* has a small data set, which is effectively captured by both the SRF and caches. However, it is bandwidth intensive. Therefore, the majority of the performance loss is seen between *HBW* and *RBW*, indicating the main source of performance loss is the limited bandwidth of a realistic cache implementation.

In the case of the *FFT 2D* and *MPEG* benchmarks, the majority of the performance loss is observed between *SRF* and *HBW* cases. The stream accesses of these benchmarks are largely to sequential producer-consumer streams, as can be seen from the classification in figure 5.1. In addition, the intermediate streams are laid out in a manner that fits without spills in the SRF. Therefore, the majority of the performance loss in these benchmarks are due to mismatches between the LRU replacement policy used by the caches and the application behavior.

UBW significantly underperforms the *SRF* case for both *FEM 3D* and *FEM 2D* as well. These benchmarks have non-trivial amounts of irregular temporal reuse as a result of gathers from large data sets. Therefore, the performance loss on *UBW* for these benchmarks is largely due to conflict misses resulting from limited cache associativity.

Among the benchmarks explored here, *Rijndael* is the only one where the performance loss of a realistic cache relative to an SRF is dominated by limited bandwidth. For all other benchmarks, while the reduced bandwidth of *RBW* does contribute to performance losses, the dominant source of performance loss in caches are replacement policy inefficiencies and limited associativity.

The benchmarks studied here were scheduled targeting the bandwidth of the SRF, which is the same as that of the *HBW* cache organization. However the caches incur higher bandwidth demands than the SRF due to prefetches. None of the benchmarks benefit from the higher bandwidth of *UBW* over *HBW*, indicating the presence of bandwidth headroom in the SRF configuration.

As can be expected, prefetching is a significant factor in improving performance for all benchmarks on caches. The cache miss behavior of the benchmarks over the different cache configurations are shown in figure 5.5, which further highlights the importance of prefetching. The statistics for loads are classified in to two categories. *Temp* loads are for intermediate results that are only allocated in on-chip memory in the case of an SRF and are never written to off-chip memory. *Non-temp* loads are for data allocated in off-chip memory, including application inputs.

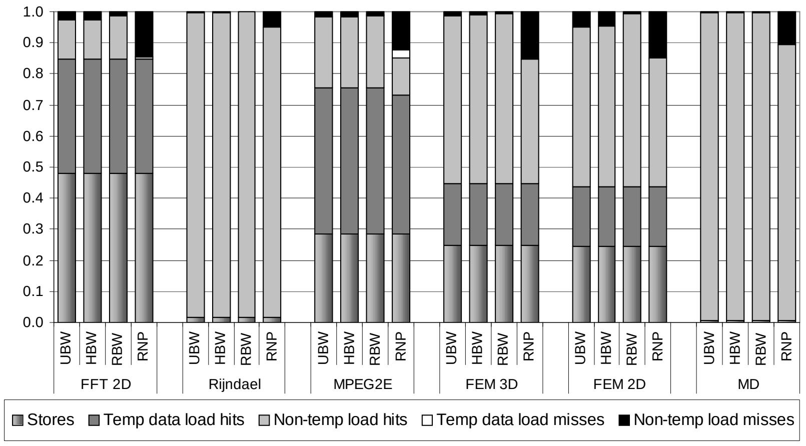


Figure 5.5: Cache miss behavior for benchmarks used in software- and hardware-managed memory configurations

As can be seen in figure 5.5, prefetching reduces miss rates of all benchmarks to 4.6% or less on *HBW* and 1.3% or less on *RBW*. However, even these relatively low miss rates result in very significant increases in execution time over *SRF* ranging from 18% to 131% for *HBW*, excluding *Rijndael* and *MD*. This indicates the high sensitivity of SIMD architectures to cache misses as a large number of issue slots are wasted during each stall cycle due to the wide issue width of these processors.

5.3.3 Off-chip Accesses on Cache-based Systems

Figure 5.6 shows the off-chip memory traffic generated by the benchmarks for the *SRF* and *RBW* configurations, normalized to the *SRF* case. These accesses are classified as follows:

- *Non-temp stores*: Stores for application results and other data that are allocated in the off-chip address space in both configurations.
- *Live temp stores*: Stores for producer-consumer data that are only allocated in on-chip memory in the software-managed case. In the hardware-managed case, however, cache lines containing this data may be evicted due to limited associativity and replacement policy mismatches to application behavior. Since these data were written by the processor during the producer kernel's execution, the cache lines are marked dirty, and therefore generate memory stores when evicted from the cache. These stores are for such temporary data that are live (i.e. will be used later in the application) at the time the stores are generated.
- *Dead temp stores*: Stores for producer-consumer data similar to *live temp stores*, except that this data is dead (i.e. no longer needed by the application) at the time the stores are generated. Since the cache lines are marked dirty, they are written to memory when evicted since the cache is unaware of application data liveness.
- *Non-temp loads*: Loads for application inputs and other data that are allocated in the off-chip address space in both configurations.
- *Temp loads*: Loads to producer-consumer data that were evicted under *live temp stores*, and must be read back on subsequent use(s).

For all benchmarks with producer-consumer reuse (i.e. *FFT 2D*, *MPEG*, *FEM 3D* and *FEM 2D*), *dead temp stores* represent 16% to 35% of the overall off-chip memory

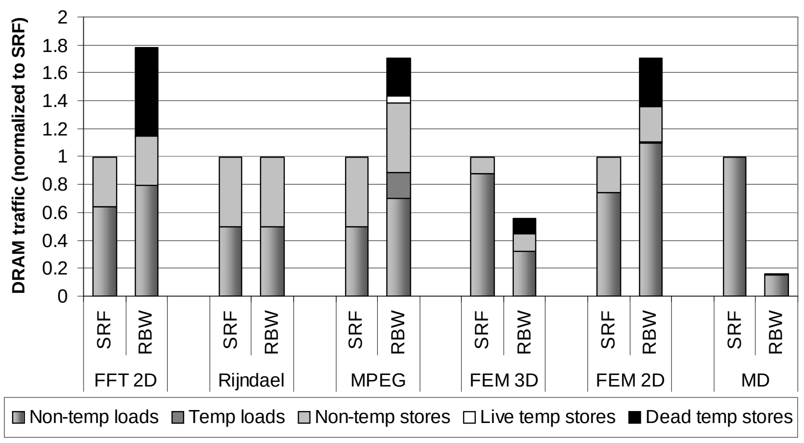


Figure 5.6: Off-chip memory traffic generated by benchmarks under software- and hardware-managed on-chip configurations

traffic generated on the *RBW* configuration. These useless memory transfers also contribute to the significant overall memory traffic increase in the *FFT 2D*, *MPEG* and *FEM 2D* benchmarks on*RBW* over *SRF*. Another source of increased off-chip memory traffic in these benchmarks is increased *non-temp loads*, which corresponds to uncaptured temporal locality due to data being evicted before all uses complete. In the case of *MPEG*, memory traffic is also increased due to evictions of live temporary data and subsequent loads of that data. Note that the memory traffic for *temp loads* can be greater than that for *live temp stores* due to repeated evictions and references to the same intermediate results.

MD and *FEM 3D* show significant reductions in overall off-chip memory traffic in the *RBW* configuration over the SRF case. Both these applications have large amounts of irregular temporal reuse which is efficiently captured by hardware-managed caches due to the fine-grain name translation and replacement, but is not efficiently exploited by a software-managed memory. A subset of this reuse is intra-stream, and may be exploited through the use of cross-lane indexed accesses as discussed in section 4.3.3, but the current implementations of these benchmarks do not use this feature. However, a significant fraction of this reuse is inter-stream, which cannot be captured efficiently in a software-managed memory even with indexed access as the reused records are distributed over multiple streams in disjoint regions of the SRF.

5.3.4 Performance Sensitivity to Cache Parameters

In order to determine the sensitivity of the above performance results to important cache parameters, the variations listed in table 5.5 were evaluated. These configurations are identical to *RBW* except for the parameter variations listed in the table. Figure 5.7 shows the normalized execution times of the benchmarks for these configurations alongside original *RBW* performance.

Configuration	**Cache line**	**Associativity**
CL4	4 words	4-way
CL8	8 words	4-way
AS8	2 words	8-way
AS32	2 words	32-way

Table 5.5: Variations of *RBW* cache configuration for evaluating performance sensitivity to cache parameters

Longer cache lines reduce prefetch overhead since the software prefetch policy used in these studies issues a single prefetch per cache line[2]. Therefore, some benchmarks benefit from longer cache lines of the *CL4* configuration as a result of the reduced cache bandwidth demands due to fewer prefetches. However, as cache line sizes increase, the overhead of unused data in strided or indirect accesses dominate, leading to lower performance for all benchmarks. Note that the added spatial locality captured by long cache lines does not benefit stream computing beyond reduced prefetches since spatial locality is explicitly specified in stream access patterns.

Increased associativity leads to reduced conflict misses, resulting in improved memory system performance. This is especially true for applications with many

[2]One prefetch is scheduled for each cache line at compile time for all sequential and strided accesses. In the presence of data dependent indexed loads, prefetches are scheduled for the first word of each record and every c^{th} subsequent word of the record where c is the cache line size.

conflict misses, such as *FEM 3D* and *FEM 2D*. Applications with mismatches to replacement policy, such as *MPEG* and *FFT 2D* also benefit from increased associativity since reduced contention leads to lesser reliance on accurate replacement.

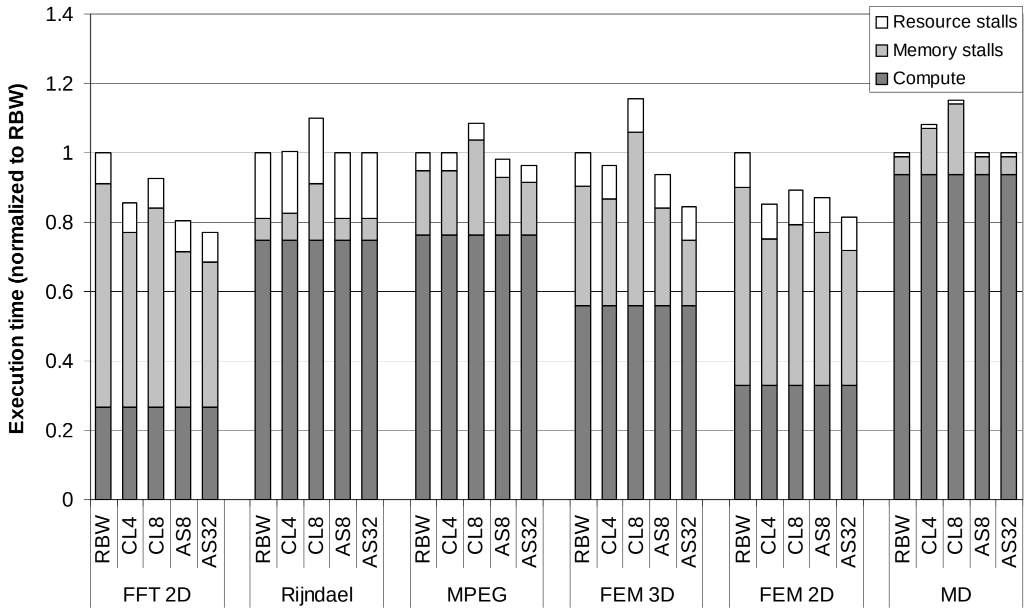


Figure 5.7: Sensitivity of benchmark execution time to cache parameters

5.3.5 Desired Cache Characteristics

Based on the evaluations in this section, the relative strengths of the software- and hardware-managed memories can be correlated to the types of locality they best exploit. As was also seen in chapter 4, the hardware-managed cache configurations were efficient at capturing irregular temporal reuse, particularly when the reuse is inter-stream.

The software-managed memories were shown to be more efficient at capturing stream-granularity temporal reuse and producer-consumer reuse. The application-oblivious nature of the replacement policies and the limited placement freedom due to name translation in hardware-managed memories were shown to lead to significant inefficiencies in hardware-managed memories. Tag lookup bandwidth constraints were also shown to contribute to lower performance in caches. In addition, the evaluations in this section highlighted the off-chip bandwidth overhead imposed due to stores of temporary data that are no longer live in the application, which arise from the replacement policy's inability to take in to account the live ranges of data in applications.

Based on the insights developed in this section, the following set of characteristics are desired in caches for stream computing:

- High associativity, particularly for scientific applications that perform gathers from large data sets
- Ability to adapt replacement policy to applications, perhaps via selecting from a few predefined policies, based on compile-time analyses
- Support for invalidating cached data that are no longer live before they are written to off-chip memory
- High bandwidth

High associativity and high bandwidth require well-understood but brute-force solutions. Prior work such as [LRYT99; WMRW02] have studied techniques for providing software hints to improve cache replacement policies. However, these approaches do not mitigate dead data writes to off-chip memory when such data are evicted from the cache. The remainder of this chapter introduces and evaluates *epoch based invalidation*, a hardware-software hybrid technique for identifying and invalidating dead intermediate state in caches. Invalidation of dead data is also shown to improve replacement decisions.

5.4 Epoch based Invalidation

The objective of epoch based invalidation is to identify and proactively invalidate dead data in the on-chip caches during the execution of streaming applications. Doing so is primarily intended to minimize off-chip memory traffic generated for cache line write-backs of dead data. However, replacement policy decisions also benefit from explicit invalidation of dead data soon after the end of their live ranges instead of allowing that data to be replaced over time in LRU order.

We partition the *stream graphs* of applications into a sequence of overlapping *epochs* – regions contiguous in time. The stream graph of an application is simply the graph whose nodes correspond to kernels and the edges correspond to stream data flows. Any streams whose entire live range falls within an epoch can then be invalidated at the end of that epoch. In order to simplify the necessary analysis and to reduce implementation complexity, we define epoch boundaries at the granularity of entire kernels.

A simple example of partitioning an application in to epochs is shown in figure 5.8. This example shows one possible assignment of epochs over the stream graph of the simplified finite element example from chapter 2. In the example, epoch *E0* covers from the beginning of the execution to the end of kernel *K2*, entirely encapsulating streams *s1* and *s2*. Therefore, those two streams can be invalidated at the end of *E0* (i.e. end of kernel *K2*'s execution). Note that while *s1* is not directly consumed by any later kernels, it is an index stream, and hence its live range stretches until the last use of all gathers and scatters based on it, which in this case is the use of *s2* in *K2*. Epoch *E1* stretches from the beginning of execution to the end of kernel *K3*, and encapsulates *s0* and *s3*. While *s1* and *s2* also fall within *E1*, they are not considered to belong to *E1* since they already belong to *E0*, which ends earlier. Similarly, epoch *E2* captures stream *s4*. Stream *s5* is an application output which should not be invalidated in the cache, and hence does not belong to any epoch.

By allowing overlap between epochs, all intermediate streams in the above example were captured for invalidation. However, if no overlap between epochs is permitted, streams *s0* and *s3* would not be captured for invalidation since their live ranges do not fall within any single epoch. Alternatively *E0* could have been extended to cover up to the end of kernel *K3*, capturing *s0* and *s3*. However, doing so increases the time that streams *s1* and *s2* remain in the cache beyond their live ranges, increasing the probability of that data being evicted from the cache and causing dead data writes to

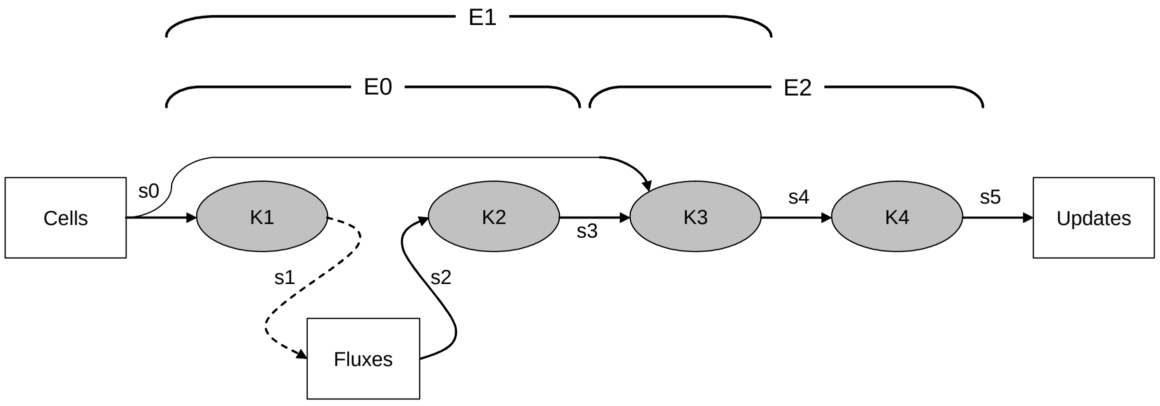


Figure 5.8: Example epoch allocation over the stream graph of simplified finite element method

off-chip memory. In addition *s4* can now no longer be captured in any epoch without allowing overlap.

5.4.1 Epoch Allocation

Allocating streams to epochs is done in two stages. First, the live ranges of streams in the stream graph is determined. The necessary analysis for this is similar to that performed for allocating streams in a software-managed SRF, and has many similarities to live range analysis for register allocation in standard compilers. Second, the stream graph must be divided in to overlapping epochs and streams that belong within each epoch assigned to it. The partitioning of the stream graph in to epochs used for these evaluations is based on a greedy algorithm that operates on the profiled stream graphs of applications. Loops with data-independent finite iteration counts are treated as fully unrolled for the purposes of our current implementation of this analysis. The key parameters of the epoch allocation algorithm are E_{max}, the maximum number of overlapping epochs allowed at any point in time, and t_{ep}, a threshold on the minimum size of an epoch defined in terms of the amount of data that must be captured within it for invalidation. Pseudocode for a simplified version of the algorithm for epoch allocation within a basic block of a stream graph is shown in figure 5.9.

```
Add all streams in basic block to pool $P$
Remove all streams live at block exit from $P$
for($i$ = 0 to ($E_{max}$-1))
  $K_s[i]$ = first kernel of block
i = 0
do {
  $K_e[i]$ = $K_s[i]$
  while(stream data w/ live range in $K_s[i]..K_e[i]$ < $t_{ep}$)
  {
    if($K_e[i]$ is last kernel in block)
      Break out of while loop
    $K_e[i]$ = NextKernel($K_e[i]$)
  }
  Assign streams w/ live range in $K_s[i]..K_e[i]$ to epoch $i$
  Insert FlushEpoch($i$) after $K_e[i]$
  Remove streams w/ live range in $K_s[i]..K_e[i]$ from $P$
  $K_s[i]$ = NextKernel($K_e[i]$)
  $i$ = ($i$ + 1) mod $E_{max}$
} while not at end of block
```

Figure 5.9: Simplified algorithm for epoch allocation within a basic block of stream graphs

E_{max} and t_{ep} impact the effectiveness of epoch based invalidation. Increasing E_{max} allows finer-grain overlapping between epochs, increasing the likelihood of capturing all available reuse. However, in order to support invalidation based on epochs, cache lines must be tagged with their epoch ID, and the hardware cost of maintaining epoch IDs increases with E_{max} (a more detailed discussion of hardware overheads of epoch invalidation follows in section 5.4.2). Increasing t_{ep} leads to longer epochs, increasing the likelihood of capturing long-lived streams. However, longer epochs increase the average time between the ends of live ranges of data and the invalidations, increasing the probability of being evicted from the cache before invalidation.

We have used $E_{max} = 2$ for this evaluation to minimize hardware overheads, and found it sufficient to capture much of the available temporary state. The threshold $t_{ep} = 2048$ words was selected based on a heuristic of the average lifespan and size of temporary streams across all benchmarks. Note that t_{ep} can be varied per benchmark or even within phases of a single benchmark since the sizes and live ranges of streams are known at compile time. However, doing so was not shown to provide significant performance benefits. The sensitivity of achieved performance to these and other system parameters will be evaluated in section 5.4.4. Epochs were not allocated to indexed memory loads due to the difficulty of address disambiguation and the potential risk of increasing cache misses on subsequent accesses in the case of inter-stream irregular temporal reuse.

Our current implementation of the epoch allocation algorithm only operates within basic blocks in the profiled stream graph. However, we do not anticipate significant gains to be had by extending this analysis beyond block boundaries or significant losses if implemented in a non-profiling framework due to the following reason. Typically, there is little data-dependent variation in the control flow within a stream graph of an applications for a given set of algorithmic parameters [Mat02]. The variations that do occur between different data sets usually result in stream size variations and changes in the number of iterations that data-dependent loops are executed (i.e. different numbers of strips in strip-mined loops). Size variations within a static graph can easily be tolerated by the epoch allocation algorithm since the reliance on t_{ep}, the only stream-size-dependent aspect of the algorithm, is weak as mentioned before. Further, limiting the analysis to a single strip of the application often does not lead to inefficiencies since most applications do not retain intermediate state across strips.

5.4.2 Hardware Support for Epochs

Epoch based invalidation requires additional state in each cache line to store the identifier of the epoch it belongs to. Note that non-overlapping epochs may use the same epoch ID in hardware without loss of generality. Therefore, the number of bits needed in each cache line for storing the epoch ID is $log_2(E_{max} + 1)$. Note that the $+1$ is needed to indicate cache lines that do not belong to any epoch, such as application outputs or long lived input or temporary data that are not encapsulated within any epoch. The $E_{max} = 2$ implementation evaluated in this section requires two additional bits per cache line. Even with the relatively short cache line size of 128 bits, this results in only modest increases in storage requirements.

Another hardware modification necessary is to enable load and store operations to set the appropriate epoch ID when allocating or writing epoch-assigned data in the cache. Careful attention must be paid to *epoch conflicts* – cases where multiple words within a cache line belong to multiple epochs. While this case can be most efficiently handled through independent epoch bits per word, a more simple implementation is to remove all epoch IDs from any cache line that contains words from multiple epochs. In other words, on writing a word that belongs to epoch *E1* in to the cache, if the corresponding line is found in the cache and belongs to epoch $E2 \neq E1$, the epoch ID of that line should be set to *null* after the write. However, this requires read-modify-write operations on the epoch ID bits to check the existing epoch prior to modifying it. The implementation of this is simplified by using a one-hot encoding of epoch IDs at the cost of increasing the number of epoch ID bits per cache line to E_{max}. In such an implementation, any epoch ID with n bits set, where $n \neq 1$, can be treated as the null epoch. All epoch bits are reset when allocating a cache line for an access that does not belong to any epoch or upon invalidating a line at the end of its epoch. Note that implementing $E_{max} = 2$ using one hot encoding does not require extra bits beyond the 2 bits per cache line discussed previously. Figure 5.10 shows a simplified version of the state transitions of the epoch ID bits for the $E_{max} = 2$ case.

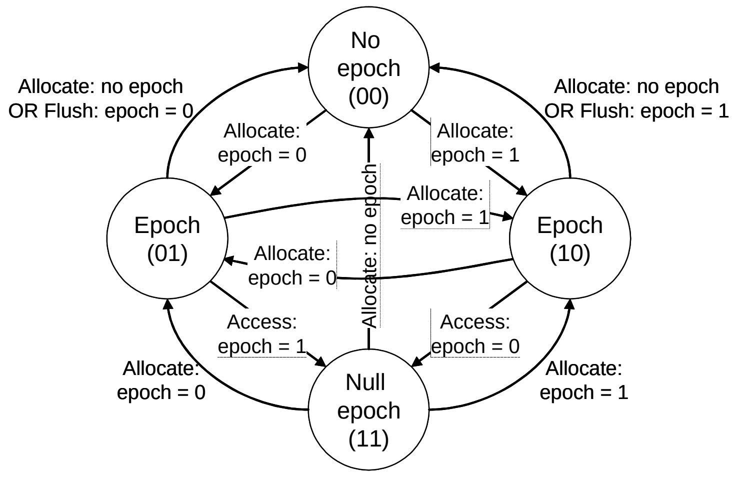


Figure 5.10: State transitions of the epoch ID bits of a single cache line for $E_{max} = 2$

Finally, hardware support is necessary to gang-invalidate all cache lines that belong to a specific epoch at the end of that epoch. The implementation evaluated here does not actually invalidate cache lines at the end of the epoch they belong to, but instead reset the *dirty* bits and demote the cache lines to the lowest LRU priority state. In our model of caches, where line fetches are not incurred on store misses, this approach provides little benefit. However, in a standard cache, this may help reduce cache line fetches on store misses in cases where the same region is repeatedly used to hold temporary data with disjoint live ranges.

5.4.3 Performance Evaluation

The performance of epoch invalidation was evaluated for the benchmarks that showed significant amounts of dead temporary data stores in the evaluations of section 5.3.3 – *FFT 2D*, *MPEG*, *FEM 3D* and *FEM 2D*. The hardware configuration used is similar to *RBW* with necessary modifications to support epoch invalidation as described in section 5.4.2, and will be referred to as *RBW-Ep*.

Figure 5.11 shows the overall off-chip traffic generated by the benchmarks on *RBW-Ep* normalized to *RBW*. The components of memory traffic is the same as in figure 5.6. As can be seen from these results, epoch invalidation effectively eliminates most dead data stores to off-chip memory in all benchmarks shown except *MPEG*. In the case of *MPEG*, the intermediate state is very long lived, and therefore that state is not captured by the limited number of epochs used in these evaluations. Further, load traffic from off-chip memory is also reduced as a result of epoch invalidation. This is due to reduced conflict misses among live data as dead data are explicitly invalidated, enabling more accurate replacement decisions. The overall reduction in off-chip memory traffic due to epoch invalidation in the benchmarks evaluated here average 21%. In addition, cache misses for *FFT 2D*, *FEM 3D* and *FEM 2D* are reduced by 9%, 16% and 10% respectively with epoch invalidation compared to the base *RBW* configuration. No reduction in cache miss rates is seen for *MPEG*.

Figure 5.12 shows the execution times of the benchmarks on *RBW-Ep* normalized to *RBW*. The reduced off-chip memory bandwidths directly lead to improved

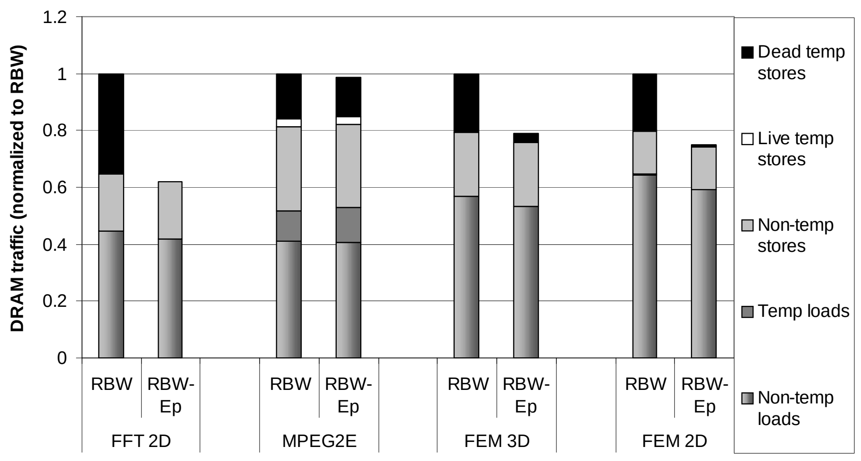


Figure 5.11: Off-chip memory traffic generated by benchmarks with epoch invalidation

performance, with speedups averaging 23% among these benchmarks.

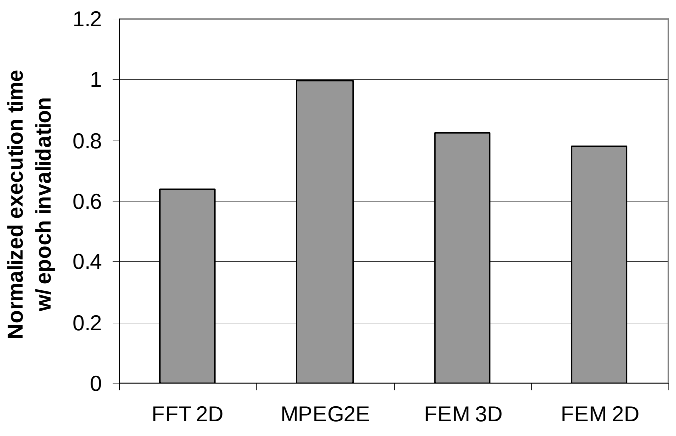


Figure 5.12: Benchmark execution times with epoch invalidation

5.4.4 Sensitivity to System Parameters

Figure 5.13 shows the sensitivity of the performance gains of epoch invalidation as system parameters vary. The first configuration shown is the same as *RBW-Ep* presented before. The next four configurations are similar to those listed in table 5.5. Finally, *CL8-M* is similar to *CL8*, but with the clock frequency of on-chip resources (compute units and cache) doubled to evaluate the impact of the rapid scaling of on-chip resources relative to off-chip bandwidth over time due to technology trends. All execution times in this graph are normalized to the corresponding system configuration without the use of epoch invalidation.

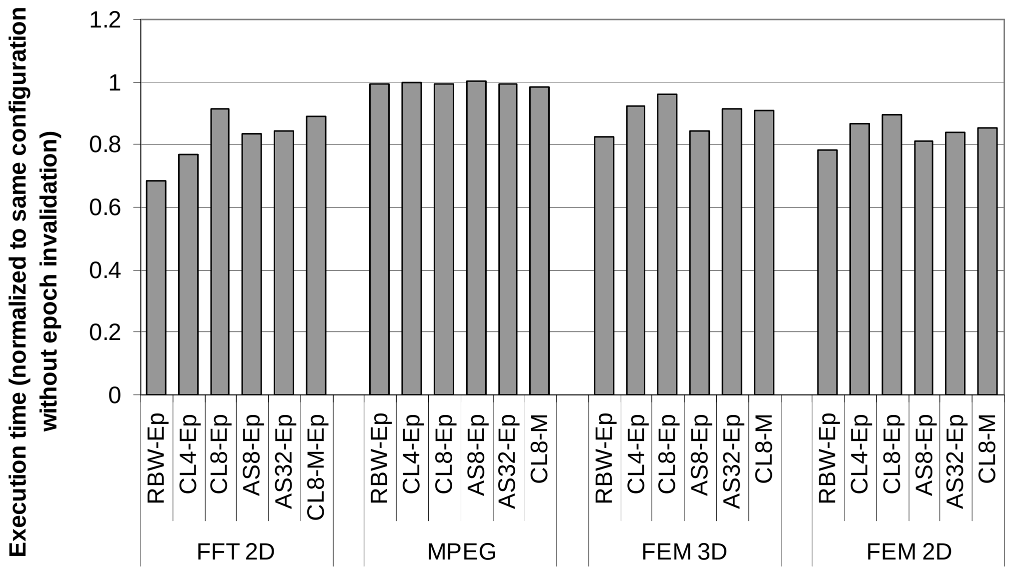


Figure 5.13: Sensitivity of execution time reduction with epoch invalidation to system parameters

As cache line sizes and associativity increase, the performance benefits of epoch invalidation degrade. As was shown in figure 5.7, applications spend less time on memory stalls on *CL4*, *AS8*, and *AS32*, and therefore has less to benefit from optimizing memory behavior. On the other hand, the performance drawbacks of long cache lines, such as in *CL8*, affect both the cases with and without epoch invalidation. Longer cache lines also increase the probability of epoch conflicts. However, this effect is rare in stream processing since each intermediate stream occupies up to thousands of contiguous words in memory. As seen by *CL8-M* relative to *CL8*, as compute capability scales faster than off-chip memory bandwidth over time, the performance benefits of epoch invalidation improves.

Figure 5.14 shows the sensitivity of off-chip memory bandwidths reductions from epoch invalidation to system parameters. The bandwidth reductions are largely insensitive to cache line size, and improves slightly with higher associativity for applications with many conflict misses. The improvement with higher associativity reflects the fact that data are less likely to be written out to off-chip memory between the end of their lifetimes and the invalidations due to the reduced conflicts in higher associativity caches.

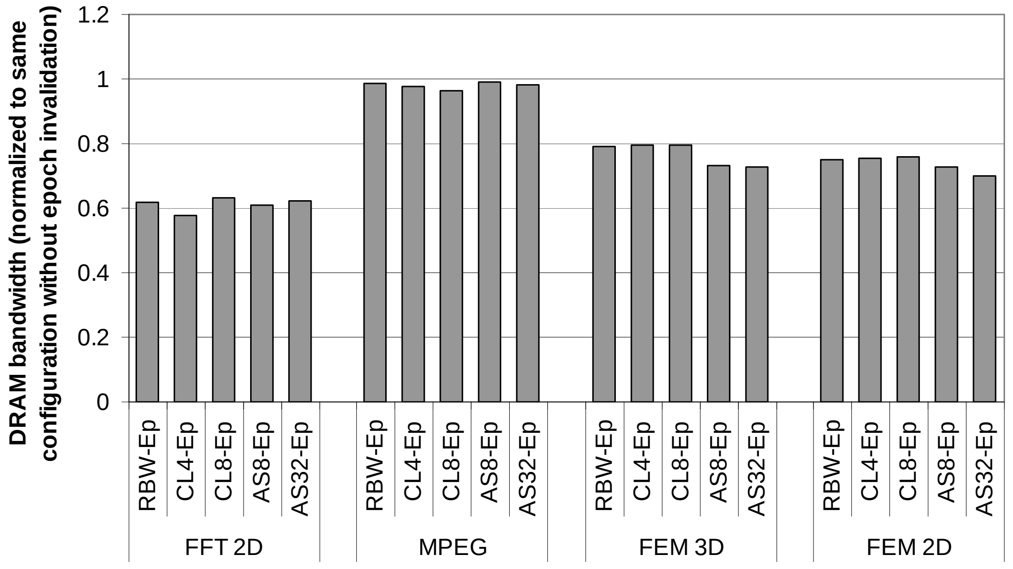


Figure 5.14: Sensitivity of off-chip memory bandwidth reduction with epoch invalidation to system parameters

5.5 Related Work

Studies such as [KSF$^+$94] and [YY92] have looked at caching for vector architectures and [Asa98] presents a thorough treatment of memory system design for vector microprocessors. [Asa98; QCEV99; EAE$^+$02] etc. have proposed specialized cache architectures and access methods for vector processing, but these deal with bandwidth and bank conflict issues and do not address name translation and replacement inefficiencies, which we show are dominant sources of performance loss for data parallel applications. However, these specialized caches can be augmented with optimizations such as epoch based invalidation proposed in this thesis.

As mentioned before, approaches such as [WMRW02; LRYT99] provide software replacement hints, but do not alleviate the lack of placement freedom due to hardware-based name translation limitations. [HR00] presented an architecture and software-management methodology for secondary caches of superscalar processors that alleviates placement limitations. None of these techniques, however, mitigate the problem of dead data writebacks. While some processor architectures allow explicit invalidation of cache lines [DM02], which provides the potential to reduce DRAM traffic due to dead data, invalidations must be issued per cache line leading to high instruction overheads. The epoch based invalidation scheme we proposed provides the capability to influence replacement decisions and invalidate dead state using only a small instruction overhead.

Timestamp and version-based cache invalidation techniques with some similarities to epoch based invalidation have been proposed for software-based multi-processor coherence [Ste90]. However, these techniques are used to keep distributed state coherent, and are not used to improve uniprocessor cache performance by invalidating the only valid copy of data items at the end of the live ranges.

5.6 Summary

This chapter analyzed the performance differences between software-managed on-chip memories such as stream register files and hardware-managed on-chip caches for stream computing. The main objectives were to understand the relative merits of each in terms of locality capture and the desired cache characteristics for stream computing.

Three main sources were identified as the causes of performance differences between software- and hardware-managed memories. Differences in name translation result in limited placement freedom in caches (i.e. limited associativity), which was shown to be an important limitation particularly for scientific applications with gathers and scatters from large data sets. Replacement policy was another source of differences. Application-independent replacement policies of caches, such as LRU, was shown to be a source of performance loss for some applications whose data access behavior differs from the history-based assumptions of the replacement policy. However, since both the name translation and replacement are handled at a fine granularity in hardware-managed memories, they were found to be efficient at capturing irregular stream-level locality, both inter-stream and intra-stream. While software-managed memories provide greater placement freedom and application-aware replacement, these are performed at a coarse granularity to amortize the associated overheads. Therefore, software-managed memories were shown to be inefficient at capturing irregular locality, especially when it exists in the form of inter-stream reuse. Therefore, software-managed memories were shown to perform best for applications with large amounts of stream granularity temporal or producer-consumer locality, while hardware-managed memories were shown to perform best for benchmarks dominated by irregular reuse.

A third source of performance differences between software- and hardware-managed memories was shown to be the available bandwidth. software-managed memories provide high bandwidth while hardware-managed caches provide lower bandwidth in realistic implementations, mainly due to name translation function (i.e. tag lookup in caches) bandwidth bottlenecks.

A side effect of inefficient replacement policies in caches is the write back of dead data (i.e. data beyond their live range in applications) to off-chip memory. Epoch invalidation was introduced in this chapter as a means of reducing these dead data writes. Epoch invalidation uses live range analysis of stream data to explicitly invalidate dead data in the cache. This technique was shown to be effective at drastically reducing dead data writes except in applications where the intermediate state is very long-lived. Explicitly invalidating dead data was also shown to improve replacement policy decisions, increasing cache hit rates and further reducing off-chip bandwidth demands. Among the benchmarks studied, those with stream-level producer-consumer locality benefited from an average off-chip memory bandwidth reduction of 21% as a result of epoch invalidation. This in turn resulted in an average 23% speedup for those benchmarks.

Chapter 6

Hybrid Bandwidth Hierarchies

The relative merits of software- and hardware-managed memories for stream computing were evaluated in chapter 5. This chapter explores hybrid bandwidth hierarchies that incorporate both these types of memories.

A straightforward implementation of a hybrid software- and hardware-managed bandwidth hierarchy for stream processing is to incorporate a hardware-managed on-chip cache between the software-managed SRF and the off-chip memory of a stream processor as shown in figure 6.1. A few data parallel architectures, such as the Merrimac stream processor [DHE$^+$03] and the Cray X-1 vector processor [Cra02], have already proposed or implemented hybrid memory hierarchies similar to this organization[1].

The advantage of hybrid bandwidth hierarchies is that they capture both regular and irregular forms of locality by exploiting the presence of both software- and hardware-managed memories on-chip, providing better filtering of off-chip memory traffic. A key drawback of such hybrid hierarchies with disjoint software- and hardware-managed memories, however, is that the sizes of the two independent memories must be determined at design time, and any such static partitioning is unlikely to be optimal across a wide variety of streaming application classes. As shown in chapter 5, applications whose memory accesses are largely regular stand to benefit little from the addition of a cache. Similarly, applications dominated by irregular

[1]The software-managed memory in the case of the Cray X-1 is the vector register file.

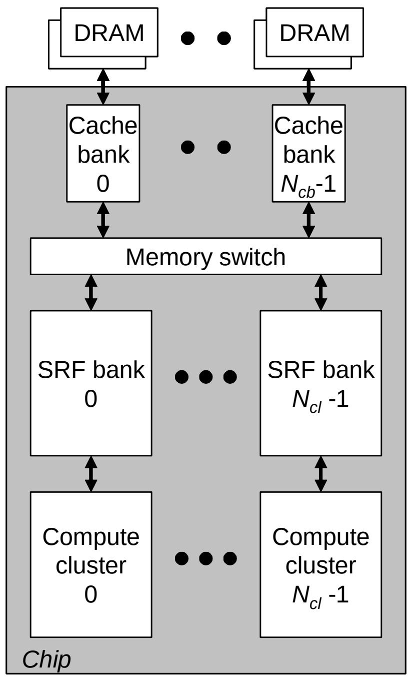


Figure 6.1: Block diagram of stream processor bandwidth hierarchy with hardware-managed on-chip cache (SRF-level cross-lane communication network not shown for simplicity)

accesses may not benefit much from a large SRF.

This chapter introduces a hybrid memory structure that allows the available on-chip memory capacity to be dynamically allocated to software or hardware-managed memories. Therefore, for applications whose accesses are largely regular or producer-consumer in nature, the majority (or all) of the on-chip memory capacity can be configured as software-managed SRF storage. Similarly, for applications dominated by irregular access patterns, the majority of the on-chip memory can be configured as hardware-managed caches.

We also evaluate the performance impact of the proposed hybrid memory. This evaluation is intended to quantify the incremental performance improvement that results from the ability to reallocate capacity between software- and hardware-managed memory within the bandwidth hierarchy of a stream processor. Several prior proposals also exist that provide varying degrees of software control over hardware-managed memories as described in section 6.3. However, these techniques are not specifically targeted for stream accesses, and evaluating their relative merits within the context of stream computing is beyond the scope of this chapter.

6.1 Organization and Microarchitecture

The hybrid memory described in this section implements a bandwidth hierarchy that is logically similar to the one shown in figure 6.1 in that the hardware-managed cache is placed between the software-managed SRF and off-chip memory. In order to enable dynamically changing the ratio of software- to hardware-managed memory capacity, however, the physical implementation incorporates both memory types within a single structure. A simplified physical overview of the proposed hybrid memory is shown in figure 6.2.

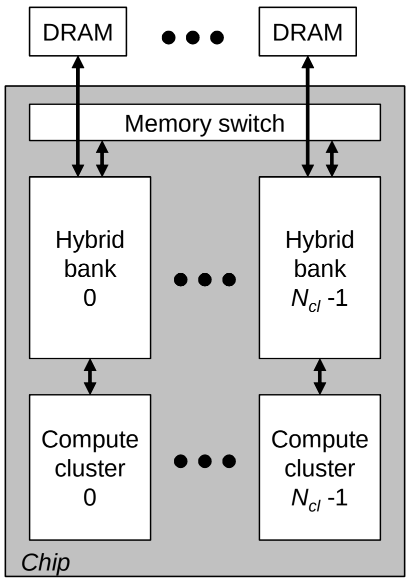


Figure 6.2: Overview of hybrid software- and hardware-managed on-chip memory for stream computing (SRF-level cross-lane communication network not shown for simplicity)

During typical operation, $s\%(0 < s <= 100)$ of the total capacity of each bank of the on-chip memory is allocated as software-managed SRF space. All stream accesses from the compute clusters are served by this SRF as in the typical stream processors described in the early chapters of this thesis. The remaining capacity of each bank is allocated to a hardware-managed cache. When transferring data between the SRF and off-chip memory, the cache is checked for hits if the particular stream transfer in progress is identified as cacheable. Whether a stream transfer is cacheable or not is determined (by the programmer or the programming system at compile time) depending on whether irregular data reuse is expected within or across stream accesses that include the current access.

The address space of the SRF portion of on-chip memory is independent from the off-chip memory space as is the case with a typical software-managed SRF. The cache portion is addressed using off-chip addresses, and cached data are distributed among the banks using a subset of the off-chip address bits. Therefore, while the cache has the same number of banks as the SRF in this implementation, the data that maps to the software-managed portion of a bank and the hardware-managed portion of the same bank are not necessarily the same. The memory switch shown in figure 6.2 is traversed on each cache access.

On a cached stream load, the request is issued via the address buses of the memory switch to the bank whose cache portion the address of the access maps to. If there is a cache hit, the data is returned to the SRF portion of the appropriate bank over the data buses of the memory switch. On a cache miss, the fill request is issued to the DRAM bank(s) associated with the cache bank. Similarly, on a cached write, the address is issued on the address buses, and the data are transferred from the SRF portion of the source bank to the cache portion of the destination bank over the data buses. On an uncached load or store, the network traversals remain the same, but the operation bypasses the cache access.

In order to support both software- and hardware-managed memories, the data storage of the hybrid memory must support the following access types:

- Contiguous block reads and writes for sequential SRF accesses
- Individual word reads and writes for indexed SRF accesses
- Individual reads and writes with tag checks for cache accesses

The data memory banks of the indexed SRF design proposed in section 4.3 is already capable of supporting block and individual word accesses. Augmenting that structure with the ability to store and check tags allows for all of the desired access types to be supported. Therefore, each bank of the hybrid memory is essentially implemented as an indexed SRF bank with associated tags. Figure 6.3 shows a block diagram of a single such bank. The data storage capacity of each bank of the memory is implemented as several sub-arrays in order to reduce access time and energy as discussed in section 4.3.1.

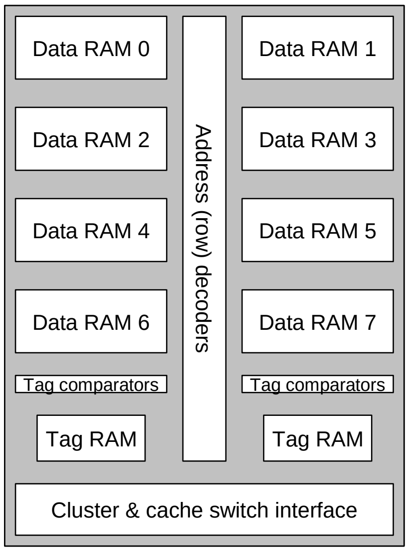


Figure 6.3: Block diagram of a single bank of the hybrid software- and hardware-managed on-chip memory. This example shows a bank consisting of 8 sub-arrays.

6.1.1 Capacity and Resource Allocation

A subset of the data sub-arrays within each bank are allocated to software-managed SRF and the remainder are allocated to data storage of the hardware-managed cache. The allocation of sub-arrays to software- and hardware-managed memory is identical in all banks. While it is not fundamentally necessary to allocate between the two memory types at the granularity of entire sub-arrays, doing so simplifies the implementation. Therefore, the discussion and results presented in this chapter use sub-array-granularity allocation only.

Varying the capacity allocated to the hardware-managed cache can manifest as a change in the associativity, line size, or the number of sets. The presence of gathers and scatters in stream accesses favor short line sizes as seen in section 3.2. High associativity is also desired for stream computing, especially in the context of scientific applications, as was shown in section 5.3. Therefore, varying line size and associativity as capacity changes is undesirable since some configurations would lead to long cache lines or low associativities, and hence inefficient cache behavior for stream accesses. The design proposed here maintains a fixed line size and associativity, and varies the number of sets as cache capacity changes.

In many implementations of scalar caches, each way of a set-associative cache bank is mapped to independent SRAM arrays. This enables the data access in all ways to proceed speculatively and in parallel with the tag check. This overlap between data access and tag check leads to performance gains in latency-sensitive scalar applications. However, as discussed in chapter 2, streaming applications are able to tolerate latencies by exploiting the data parallelism to overlap computation with memory accesses. Therefore, in a stream processor cache, tag comparison and data access can be serialized with minimal performance impact. Once the tag check completes, at most one way of the cache needs to be accessed. This allows the data storage for multiple ways of a cache set to be mapped to a single memory array, enabling high associativity caches even when as few as one sub-array within each hybrid memory bank is allocated to hardware-managed cache.

While the data bandwidth demands of a hardware-managed cache implementation can be reduced by serializing tag and data accesses, a tag check is still required in all ways of the cache on each access. Therefore, the tag memory in each bank is implemented in a manner that enables a accesses per cycle, where a is the associativity of the cache. Since each way of the cache accesses the same index within the tag memory, the tags are stored in single-ported SRAMs where the port is sufficiently wide to access a tags simultaneously. The tag capacity must be sized to hold a sufficient number of tags for the maximum sized cache anticipated during dynamic reconfiguration. Therefore, in configurations with less than the maximum possible cache capacity, only a correspondingly reduced capacity of the available tag memory is utilized. The remaining unused tag capacity can potentially be allocated to software-managed memory at the cost of additional implementation complexity. In doing so, address decoding is complicated in particular due to the capacity mismatches between the partial tag memory and the data sub-arrays that consist the software-managed memory. The focus of this chapter, however, is to evaluate the macroscopic performance potential of hybrid on-chip memories, and therefore, the evaluation presented here do not assume reuse of unused tag storage capacity.

To simplify hardware name translation from off-chip address space to on-chip address space, the hardware-managed cache capacity is constrained to be a power-of-two number of sub-arrays. For the purposes of this chapter, we will focus on a 16-lane stream processor with the on-chip memory bank in each lane consisting of 8 sub-arrays of 8KB each. Therefore, hardware-managed cache sizes supported consist of 0, 1, 2, 4 or 8 sub-arrays in each bank. Note that the software-managed memory does not require the number of sub-arrays to be a power of 2 since name translation is performed explicitly in software. Figure 6.4 shows an example configuration with two sub-arrays allocated to hardware-managed cache and 6 sub-arrays allocated to software-managed SRF.

Another key resource that must be allocated between the software- and hardware-managed memories is bandwidth to the sub-arrays. In a decoupled implementation, both the SRF and cache may be accessed on every cycle. In a unified, hybrid implementation, the bandwidth available to the memory must be shared by the two memory structures. A straightforward approach is to statically allocate the bandwidth such that each memory is accessed on a fixed percentage of cycles. Alternatively, each memory may dynamically arbitrate for access depending on bandwidth requirements. Such a dynamic arbitration may be implemented by augmenting the existing arbitration for SRF access among streams, and introducing cache accesses as yet another consumer of the available bandwidth.

A further consideration in allocating sub-arrays and bandwidth is the impact on peak indexed SRF bandwidth. As discussed in section 4.3, high indexed SRF

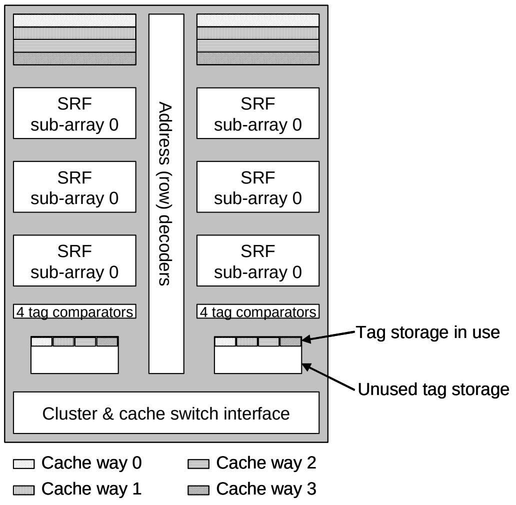


Figure 6.4: Example hybrid memory configuration with two and six sub-arrays allocated to hardware cache and software SRF respectively

bandwidth is achieved by performing multiple independent accesses in several sub-arrays simultaneously. For example, in order to achieve n words per lane per cycle of indexed SRF bandwidth, the SRF bank in each lane must consist of at least n sub-arrays. As the results in chapter 4 point out, a peak indexed SRF access rate of 4 words per cycle per lane is sufficient to meet the requirements of most streaming applications. In the particular configuration considered in this chapter where each bank of the hybrid memory consists of 8 sub-arrays, the SRF is always made up of at least 4 sub-arrays, enabling 4 words per lane per cycle of in-lane indexed SRF bandwidth in all configurations[2].

A final resource that must be provisioned is the memory switch bandwidth. Note that exploiting the strengths of each memory type, hardware-managed memory is

[2]Note that if all 8 sub-arrays are allocated to hardware-managed memory, no SRF accesses are required.

intended to capture irregular and unpredictable reuse patterns, while the software-managed memory is intended to capture statically analyzable reuse patterns. Therefore, a memory switch traversal is required only for hardware-managed memory accesses[3]. In a static allocation of memory array bandwidth to software- and hardware-managed memory, switch bandwidth is trivially determined. If the hardware-managed memory is accessed every i^{th} cycle, the required bisection bandwidth of the data portion of the memory switch is $n \times b/i$, where n is the number of banks in the hybrid memory structure and b is the data width of each access. Similarly, control bandwidth bisection required within the memory switch is $n \times c/i$, where c is the width of control and address information associated with each cache access. For example, the results presented in section 6.2 are based on a simple static bandwidth allocation scheme where software- and hardware-managed accesses to the on-chip memory are interleaved on every other cycle. In this case, the memory switch must sustain a peak of 16 accesses, one from each bank, every two cycles. Therefore, the bisection bandwidth required on the network is 8 accesses per cycle each for data and control information.

In a dynamically arbitrated system, the memory switch bandwidth requirements may be set based on the expected fraction of hybrid memory accesses that are to hardware-managed cache. For example, if $f = 20\%$ of on-chip memory accesses are expected to be to hardware-managed memory, and the expected SRF block size is $B = 4b$ (where b is the width of a cache access), the expected fraction of cycles allocated to hardware-managed accesses is $f/(f + (1 - f)/b) = 50\%$. Therefore, the required bisection bandwidth of the memory switch is again 8 in a 16-lane stream processor[4].

[3]Cross-lane indexed SRF accesses, along with software-scheduled cluster communication, utilize the separate, inter-cluster communication network.

[4]In a realistic implementation, the bandwidth allocation must factor in some speedup over the anticipated traffic in order to keep the network from saturating. This requirement is assumed to be reflected in the expected traffic percentages in this example for simplicity.

6.1.2 Addressing Considerations

As the size of the memories are varied, the address bits used for addressing into them also change. In the case of the software-managed SRF, the the key implication is that the application and/or run-time system must not issue accesses to addresses beyond its allocated size. In the hybrid memory proposed here, the allocation of capacity to the two forms of memory is directed by the programmer or the compiler. Therefore, addresses can statically be guaranteed to not exceed the available SRF capacity. The only other concern is that the SRF address range within each bank must be mapped to a contiguous region in the physical on-chip memory space, which can also be guaranteed statically.

The addressing for the hardware-managed memory is more complex since the accesses are performed using subsets of off-chip addresses. As the cache size varies, the subsets of the address bits used for indexing into the cache and tagging cache lines also varies. The variation in the bits used for selecting the sub-array to access is achieved by masking the sub-array selection bits using a mask based on the number of sub-arrays allocated to the hardware-managed cache. The masks for valid configurations are listed in table 6.1. The number of tag bits also vary with cache size, with the lowest capacity configuration dictating the maximum tag size. While larger capacity cache configurations require fewer tag bits as more bits are used in sub-array selection, the maximum number of tag bits can always be stored for implementation simplicity. A simplified view of the logic for extracting cache set index and tag bits is shown in figure 6.5.

6.1.3 Memory Reallocation

The number of sub-arrays allocated to software- and hardware-managed portions of the on-chip memory can be changed dynamically by updating the sub-array selection mask for the cache and a register indicating the base address of the SRF in on-chip memory. While this can be done at any point in execution from a hardware perspective, live data in on-chip memory must be preserved across memory reallocations.

Sub-arrays allocated to cache	**Sub-array selection mask**
0	-
1	000
2	001
4	011
8	111

Table 6.1: Bit masks for qualifying sub-array selection in hardware-managed cache access in hybrid memory

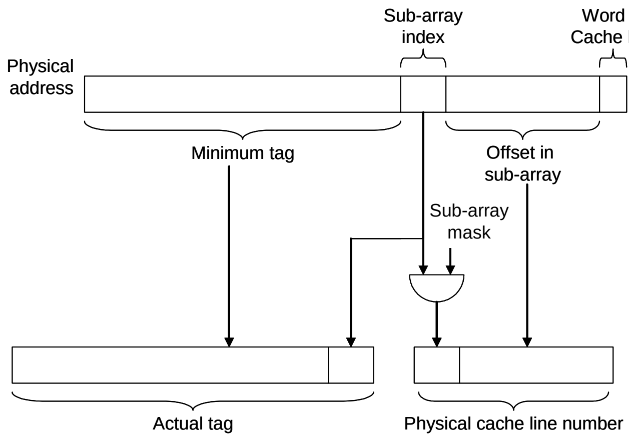


Figure 6.5: Cache index and tag extraction for hybrid memory (assuming physically indexed and tagged cache)

During any change to the allocation of sub-arrays between the two types of memories, all dirty lines in the entire cache must be written to off-chip memory and all valid bits must be cleared. This is required since updating the sub-array selection mask changes the mapping of off-chip addresses for the entire cache. Identifying and evicting all dirty cache lines is an inherently serial operation, and may take thousands of cycles depending on the cache size and the percentage of lines that are dirty at the time of reallocation.

During a reallocation that increases the number of sub-arrays allocated to software-managed memory, no change is necessary to the data in the SRF. Since reallocation is performed under the control of the software system, the run-time environment is aware of, and able to utilize, the increased capacity in the SRF after the change. When reducing the number of sub-arrays allocated to software-managed memory, any live streams that have not been written to memory and overlap any part of the sub-arrays being reallocated must be written to memory. The necessary stream stores for orchestrating this must be issued by the software system, and completed, before the reallocation is initiated. The stream liveness and occupancy information necessary for performing these stores is already available in the software system as the allocation and management of the SRF is fully under its control.

6.2 Performance Evaluation

The performance impact of the hybrid memory proposed in this chapter was evaluated using the scientific benchmarks, *FEM 3D*, *FEM 2D*, and *MD*, introduced in chapter 5. We focus on these applications since they are the ones with the most complex access patterns that mix regular and irregular accesses and hence stand to benefit the most from a hybrid bandwidth hierarchy.

The applications' performance was determined using cycle-accurate simulations on 4 machine configurations which are described in table 6.2. The configurations *C0*, *C1*, *C2*, and *C4* correspond to allocating 0, 1, 2, and 4 sub-arrays to hardware-managed caches (out of a total of 8). The benchmarks were re-scheduled for each configuration resulting in changes to strip size to minimize stream spills to off-chip memory as SRF size is varied. No reallocation of sub-banks to SRF and cache were performed during any one benchmark's execution for the results presented here. Therefore, all reallocation of sub-banks were performed with no live data in on-chip memory, avoiding the need to store live, dirty state to off-chip memory as discussed in section 6.1.3. More efficient execution may be achieved through dynamic reallocation of sub-banks during different phases of applications, enabling the memory system to be adapted to application requirements at a finer granularity.

Arbitration between SRF and cache for on-chip bandwidth was performed using a simple, static allocation scheme for the results presented in this section. Under this scheme, the full bandwidth of the on-chip memory is alternated between SRF and cache access on every other cycle. Other machine parameters are the same as those specified for the *Full* configuration evaluated in chapter 5. Note that the case with all 8 sub-arrays allocated to cache is not presented in this chapter since the all-cache on-chip memory organizations were discussed and evaluated in detail in chapter 5.

Configuration	**SRF**	**Cache**
C0	1024KB (8 sub-arrays)	0KB (0 sub-arrays)
C1	896KB (7 sub-arrays)	128KB (1 sub-array)
C2	768KB (6 sub-arrays)	256KB (2 sub-arrays)
C4	512KB (4 sub-arrays)	512KB (4 sub-arrays)

Table 6.2: Hybrid streaming memory configurations evaluated. Memory capacity shown is the total across all lanes. Number of sub-arrays shown in parentheses is per lane.

Figure 6.6 shows the execution times for the three benchmarks on the four configurations described in table 6.2, normalized to the *C0* configuration. The execution times are divided in to time spent in kernel execution, memory accesses (including memory stalls), and overhead due to sharing the total on-chip memory bandwidth between the cache and SRF. The overhead due to bandwidth sharing is computed by comparing to the execution time on a configuration with separate SRF and cache memories, enabling each to sustain 2x the peak bandwidth available on the hybrid architecture.

The kernel execution portion of the benchmark run-times increase as the amount of memory allocated to the SRF decreases. This results from smaller strip sizes in order to avoid SRF spills as the SRF capacity is decreased. With smaller strip sizes, a greater number of kernel invocations are required to process the entire data sets, which incur more kernel startup and shut-down overheads, increasing overall execution time.

As was seen in section 5.2, *FEM 3D* accesses have significant amounts of irregular temporal reuse that can be effectively captured in a hardware-managed cache. Therefore, as the cache size increases, the time spent on memory accesses drop for

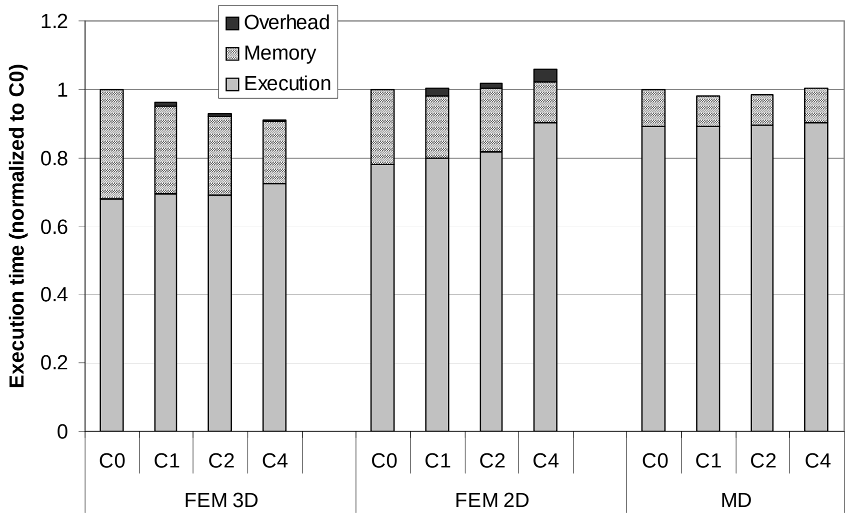


Figure 6.6: Execution times of scientific benchmarks on hybrid memory organizations (normalized to *C0*)

this benchmark, outweighing execution time increases and resulting in an overall performance improvement. *FEM 2D*, on the other hand, has fewer irregular accesses, and therefore, the reduction in memory access time is insufficient to overcome execution time increases and bandwidth overheads as SRF size decreases. The *MD* benchmark's accesses are dominated by irregular temporal locality as seen in section 5.2. However, this benchmark is largely compute-limited, and improving memory system performance leads to little overall performance improvement. The memory access time that is not overlapped by computation in this application is due to a load that occurs at the beginning of each kernel invocation, and therefore the overhead of that access increases as smaller strip sizes increase kernel invocations.

Execution time overhead due to bandwidth sharing between the SRF and cache is modest at 3.7% at most for *FEM 2D*, and less than 0.5% for *FEM 3D* and MD. Therefore, this sharing is not a performance bottleneck for the scientific streaming applications considered here, even with the simple static bandwidth allocation used in these simulations.

The overall performance impact of the hybrid SRF and cache organization is mixed for these benchmarks. While *FEM 3D* achieves a 10% speedup by allocating 50% of the available on-chip memory to a hardware-managed cache, *FEM 2D* performs best with all memory configured as a software-managed SRF. While MD performs best with a small 128KB cache, the speedup is modest at only 2% over a SRF-only configuration. On one hand, these results demonstrate the varying needs of the applications, and the importance of being able to tailor the hybrid memory organization to the specific needs of each. No one configuration provides the best performance across all benchmarks. On the other hand, the benefits over a purely software-managed memory is relatively small, indicating that an SRF alone is largely sufficient to overcome the bandwidth constraints of off-chip memory for these compute-intensive benchmarks. However, with technology scaling, on-chip computation capabilities continue to improve at a faster rate than off-chip bandwidth and latency gains. Therefore, in order to estimate the applicability of hybrid memory structures for future technology generations, we evaluated the benchmark performance with a 4x increase in the compute capability to bandwidth ratio of the above configurations. The results for these simulations are shown in figure 6.7.

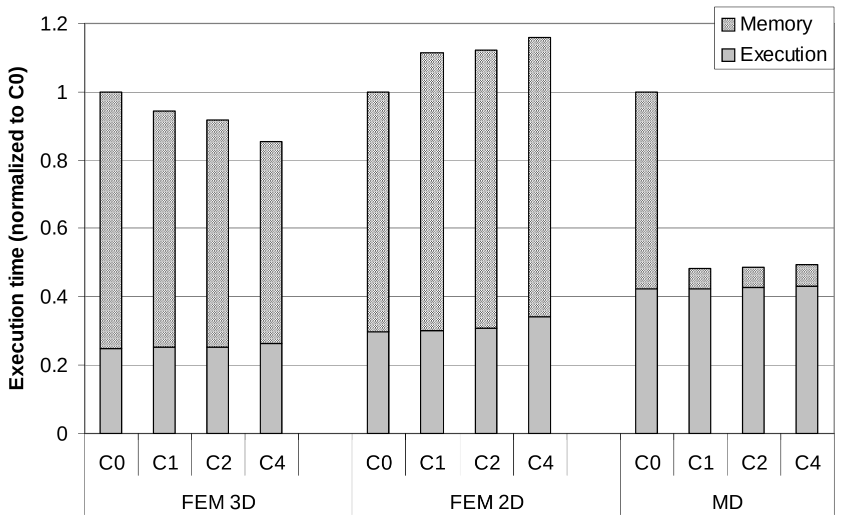


Figure 6.7: Execution times of scientific benchmarks on hybrid memory organizations (normalized to *C0*) with increased computation to off-chip bandwidth ratio

As compute capability scales more rapidly than off-chip memory bandwidth, the impact of memory system on overall performance is clearly amplified. In the case of *FEM 3D*, increasing cache sizes continue to offset increased execution time due to shorter strip sizes. The best performance is seen for the hybrid *C4* configuration with 512MB each of cache and SRF, which achieves a 17% speedup over the base *C0* configuration. In the case of *FEM 2D*, the limited irregular locality present in the application does not make caching an appealing option. In fact, as the SRF size reduces, per-kernel memory accesses that occur during kernel initialization cause the memory bottleneck to exacerbate. Therefore, the optimal configuration for this benchmark is *C0*, with no on-chip hardware-managed cache memory. In the case of MD, the access patterns are dominated by irregular temporal locality. These accesses also have a fairly high degree of locality as many of the particles interact with each other, causing multiple, repeated accesses to the same data. As a result, even a small cache is sufficient to cause this benchmark to be compute-limited, leading to a 102% speedup over the memory-limited base (*C0*) configuration. Increasing the cache further leads to small performance losses as execution times increase, adding pressure to the performance-limiting compute resources. Therefore, the optimal configuration for this benchmark is *C1*, with a small hardware-managed cache and a large SRF.

These results corroborate the trends observed in the current technology results in figure 6.6, and demonstrate the varying memory hierarchy requirements, even among applications within the scientific computing domain. Further, the increasing performance benefits of hybrid memories with technology scaling is clearly demonstrated by these results.

6.3 Related Work

As mentioned previously, Merrimac and Cray X-1 architectures employ hybrid memory hierarchies. However, the software- and hardware- managed memories in these architectures are physically disjoint, and their capacities are fixed at design-time. As seen in section 6.2, such static partitionings of capacity can be sub-optimal over a broad range of the data parallel application space, particularly as off-chip bandwidth becomes increasingly scarce.

Hybrid memories with dynamically variable allocation of resources to software- and hardware-managed memories have also been proposed [MPJ+00; RAJ00; SKMB03]. However, unlike the design presented in this chapter, many of these prior proposals require a reduction in cache associativity in order to allocate capacity to software-managed memories. In addition, the prior proposals are not geared towards exploiting the latency tolerance of stream processing, and do not attempt to identify and preserve specific memory characteristics that are important for achieving high performance for stream processing.

Locking cache lines has been proposed as a technique for gaining software control over a subset of the lines within a hardware-managed cache. While this technique is practical for a relatively small amount of software-managed storage, software control over a large capacity incurs high run-time overheads as line locking and replacement must be managed at the fine granularity of individual cache lines. Further the hardware- and software-managed regions exist within the same address space, leading to conflicts between the two types of data, reducing effectiveness. The design proposed in this chapter exposes the division between software- and hardware-managed memory, and the data that map to each, to the hardware implementation. This enables a separate address space to be maintained for software-managed memory, which does not conflict with the hardware-managed memory.

6.4 Summary

This chapter explored the performance potential of hybrid memory hierarchies that incorporate both software- and hardware-managed memories for stream computing. The general organization evaluated incorporates a hardware-managed, on-chip cache between the software-managed SRF and off-chip memory. This enables irregular data reuse present in stream loads from off-chip memory to be captured in the on-chip cache.

We presented an on-chip memory organization that allows flexible allocation of the total available on-chip memory capacity between the SRF and cache. This design was based on the indexed SRF architecture presented in section 4.3, and is augmented to store tags and perform addressing for the hardware-managed cache.

Evaluations in this chapter focused on scientific application benchmarks since they exemplify the complex access patterns that stand to benefit most from hybrid memory organizations. Our results showed that no one partitioning of storage between SRF and cache was optimal even among the scientific applications studied. Therefore, by allowing dynamic reallocation of capacity between the SRF and cache, the architecture can be optimized for best performance on a per-application basis. However, the results also showed that in today's implementation technology, there is little performance gain (10% or less for the benchmarks evaluated) to be had by supporting hybrid memory hierarchies relative to an SRF-only on-chip memory. Evaluations approximating likely future technology trends indicate that the benefit of hybrid memory hierarchies increase significantly as off-chip bandwidth becomes an increasingly scarce resource with respect to on-chip compute capability. Results of these studies indicate speedups of over 2x for some applications with the addition of even a small cache, while other applications require a larger cache to achieve appreciable performance gains, and yet others lose performance as SRF capacity is lost with the addition of a cache.

Chapter 7

Conclusions

Stream processing has been shown to outperform mainstream programmable computing solutions while consuming less power for data parallel applications. Exploiting the data- and instruction-level parallelism inherent in these applications, stream processors sustain many operations in parallel, and overlap them with memory accesses in order to improve computation throughput. Realizing the performance potential of stream processing, however, depends on the ability to manage bandwidth demands in the memory hierarchy to sustain the operands needed for highly parallel computation.

This dissertation presented and evaluated several techniques for and tradeoffs in improving bandwidth hierarchy performance for stream processing with a particular emphasis on improving off-chip bandwidth utilization. We also presented evaluations of memory hierarchy requirements of stream processing with respect to other programmable, parallel execution techniques. The results of these studies demonstrated that stream processing requires less intermediate state storage capacity, and is a better match to modern DRAM characteristics than vector and multi-context execution models.

We introduced an indexed stream register file architecture that enabled data reuse patterns found in a broad range of data parallel applications to be captured in on-chip memories of stream processors, reducing off-chip bandwidth demands by several fold in some cases. This, in effect, enables classes of data parallel applications that, due to bandwidth bottlenecks, could not previously be efficiently executed on stream processors to be supported efficiently.

Software-managed on-chip memories that are typically found on data parallel architectures were shown to lead to better performance for many streaming applications than the hardware-managed caches found in general-purpose CPU memory hierarchies. However, hardware-managed caches were shown to be better at capturing irregular data reuse patterns of some applications, particularly in the case of scientific computing. We presented a framework for identifying the suitability of software- or hardware-managed memories for a given application based on the types of data reuse found in the access patterns. We also introduced epoch-based cache invalidation, a hardware-software hybrid technique to improve cache performance for producer-consumer reuse patterns that are common in many streaming applications.

Hybrid on-chip memory hierarchies that consist of both software- and hardware-managed memories have been incorporated on a few recent designs as a means of achieving the benefits of both types of memories. Our evaluations of such memories show that no single partitioning of the total available on-chip memory capacity between the two memory types is optimal for all applications. We presented a hybrid organization where the resource allocation between software- and hardware-managed memory can be dynamically altered to suit the needs of individual applications. Evaluations of this memory organization for a set of scientific applications showed little benefit from hybrid memory hierarchies in current implementation technologies. However, models approximating future technologies showed significant performance advantages resulting from the use of hybrid memories, particularly when resource allocation between software- and hardware-managed memories can be tailored to suit application-specific requirements.

7.1 Future Research

The techniques and evaluations presented in this dissertation both improve bandwidth hierarchy performance and broaden the range of applications amenable to stream computing. However, a number of related research areas have the potential to further extend the applicability of stream computing in general and the techniques presented in this thesis. In addition, incorporating techniques that are found to be highly effective in stream processors in to other classes of architectures also has the potential to provide significant benefits for executing streaming applications on such architectures.

Multi-processor Systems

The evaluations in this thesis focused on the bandwidth hierarchy of a single stream processor. In large-scale, distributed shared-memory systems, such as the Merrimac streaming supercomputer, accesses to memory on remote nodes exacerbate the bandwidth and latency bottlenecks of off-chip accesses. The techniques discussed in this thesis are likely to yield even greater benefits in such situations.

Multi-processor systems also introduce tradeoffs in work and data distribution. Data layout across multiple nodes and the resulting inter-node accesses during computation are likely to have large impacts on performance for many applications and determine the bandwidth and complexity of the interconnect between nodes. Additional complexity is also introduced in terms of maintaining coherence among the bandwidth hierarchies of different processors. The distribution of coherence responsibilities between software and hardware components as well as the role of synchronization, which have been thoroughly studied for general-purpose CPUs, need to be reevaluated in the context of stream processors. Several of these issues are already under active investigation in the Merrimac project and other research efforts.

Scaling and Further Integration

With implementation technology scaling, stream processor architectures are likely to scale along several axes including functional units within a lane, number of lanes within a processor, and number of processors on a chip. This scaling is likely to lead to increased demands on the memory hierarchy as briefly highlighted in some of the evaluation in this thesis, increasing the utility of the techniques presented here. These trends may also lead to sparse interconnects in place of the fully-connected intra- and inter-cluster networks modeled in this thesis as the number of communicating units grow. Issues of sparse interconnects were discussed briefly in [Kha03], but need to be evaluated more thoroughly.

Further integration between memory and computing is also likely with technology scaling, resulting in computation capabilities interspersed at various levels of the bandwidth hierarchy. A simple form of this integration may be the inclusion of light-weight computing capabilities embedded in off-chip DRAM as discussed in [OCS98]. Such architectures lead to step changes in the otherwise gradual variation of the ratio of computing to memory bandwidth that results from technology scaling, and the resulting impact on bandwidth hierarchy design need to be evaluated.

Memory System APIs

A key characteristic of stream processors is the ability to specify stream transfers to and from off-chip memory in a manner entirely decoupled from the computation. Control of the memory system for such transfers is exposed to the programmer and/or compiler via streaming APIs. The potential for Incorporating this level of decoupling between memory transfers and computation, and exposing coarse grain memory transfers via APISs, in other classes of processors needs to be evaluated.

The likely benefits of such support in vector processors was already briefly discussed in section3.2.6. Similar benefits are likely to be observed in multi-context processors for streaming applications as well. Even in general-purpose CPUs, such decoupled memory APIs could lead to significantly more effective data prefetches with reduced instruction overheads and improved accuracy, including cases where indirect addressing may make traditional hardware prefetch techniques less effective.

7.1.1 Hybrid Memory Hierarchies and Epoch-based Invalidation

The effectiveness of hybrid software- and hardware-managed memories and epoch-based cache invalidation for stream computing was demonstrated in this thesis. These techniques have the potential to be useful in other programming environments and execution models as well. The degree to which these techniques provide performance improvements in the absence of the additional locality exposed by the stream programming model needs to be evaluated.

Register Storage Implementations

Chapter 3 of this thesis discussed the ability of stream processing to reduce the register capacity requirements over multi-context execution models. However, multi-context processors support more general programming styles, including the execution of different instruction traces in each parallel context. Techniques for leveraging the benefits of each of these two types of architectures within the other could lead to far more efficient yet flexible implementations than are available today.

In multi-context processors, register capacity requirements are increased due to the unpredictability of context switches, and therefore, the need to hold the entire register space of each context in hardware. Stream processors, on the other hand, manage the register space far more efficiently through software-management, aided by compile-time analysis and static scheduling of computation. Context storage structures that provide similar benefits in terms of compressed register management that do not require strict software scheduling present an interesting research direction that has the potential to yield multi-context processors with register requirements on a par with stream processors.

In stream processors, relaxing the strict software-scheduling and SIMD execution has the potential to greatly increase the range of applications amenable to stream processing. The key future research for realizing such designs lies in the development of efficient execution engines. From a bandwidth hierarchy perspective, the indexed SRF implementation proposed in chapter 4 adapts trivially to support fully independent accesses across lanes with no requirement of SIMD execution. Further, the hybrid memory structure proposed in chapter 6 provide support for irregular reuse patterns likely to arise in a loosely-coupled parallel execution environment. Therefore, the contributions of this thesis lays the groundwork for far more general execution models to be supported while retaining the benefits of a streaming bandwidth hierarchy.

References

[AH00] Bharadwaj Amrutur and Mark Horowitz. Speed and power scaling of SRAMs. *IEEE Journal of Solid-State Circuits*, 35(2):175–185, February 2000.

[Asa98] Krste Asanovic. *Vector microprocessors.* PhD thesis, University of California, Berkeley, 1998.

[BB90] Dileep Bhandarkar and Richard Brunner. VAX vector architecture. In *ISCA*, pages 204–215, 1990.

[BBC$^+$00] Maynard Brandt, Jeff Brooks, Margaret Cahir, Tom Hewitt, Enrique Lopez-Pineda, and Dick Sandness. *Benchmarker's Guide for CRAY SV1 Systems.* Cray Inc., July 2000.

[BFH$^+$04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, 2004.

[BKGA04] Christopher Batten, Ronny Krashinsky, Steve Gerding, and Krste Asanovic. Cache refill/access decoupling for vector machines. In *MICRO 37: Proceedings of the 37th annual International Symposium on Microarchitecture*, pages 331–342, 2004.

[Bor04] Shekhar Borkar. Microarchitecture and design challenges for gigascale integration. In *MICRO*, December 2004. Invited keynote address.

[Cra02] Cray Inc. *Cray X1^{TM}System Overview*, 2002.

[DD97] Keith Diefendorff and Pradeep K. Dubey. How multimedia workloads will change processor design. *Computer*, 30(9):43–45, 1997.

[DDHS00] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, and Hunter Scales. Altivec extension to PowerPC accelerates media processing. *IEEE Micro*, 20(2):85–95, 2000.

[DHE^{+}03] William J. Dally, Patrick Hanrahan, Mattan Erez, Timothy J. Knight, Franois Labont, Jung-Ho Ahn, Nuwan Jayasena, Ujval J. Kapasi, Abhishek Das, Jayanth Gummaraju, and Ian Buck. Merrimac: Supercomputing with streams. In *SC2003*, Phoenix, AZ, USA, November 2003.

[DM02] Eric Delano and Dean Mulla. Data cache design considerations for the itanium®2 processor. In *Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 356–362, September 2002.

[DR02] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.

[EAE^{+}02] Roger Espasa, Federico Ardanaz, Joel Emer, Stephen Felix, Julio Gago, Roger Gramunt, Isaac Hernandez, Toni Juan, Geoff Lowney, Matthew Mattina, and Andre Seznec. Tarantula: A vector extension to the Alpha architecture. In *Proceedings of the 29th annual international symposium on Computer architecture*, pages 281–292, 2002.

[FD01] Viktor Fischer and Milos Drutarovsky. Two methods of Rijndael implementation in reconfigurable hardware. In *CHES '01: Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*, pages 77–92. Springer-Verlag, 2001.

[HR00] Erik G. Hallnor and Steven K. Reinhardt. A fully associative software-managed cache design. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 107–116, June 2000.

[Jou98] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache prefetch buffers. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 388–397. ACM Press, 1998.

[KDC$^+$02] Brucek Khailany, William J. Dally, Andrew Chang, Ujval J. Kapasi, Jinyung Namkoong, and Brian Towles. VLSI design and verification of the Imagine processor. In *ICCD*, pages 289–294, 2002.

[KDR$^+$00] Ujval J. Kapasi, William J. Dally, Scott Rixner, Peter R. Mattson, John D. Owens, and Brucek Khailany. Efficient conditional operations for data-parallel architectures. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 159–170. ACM Press, 2000.

[Kha03] Brucek Khailany. *The VLSI Implementation and Evaluation of Area- and Energy-Efficient Streaming Media Processors.* PhD thesis, Stanford University, June 2003.

[Koz02] Christoforos Kozyrakis. *Scalable Vector Media Processors for Embedded Systems.* PhD thesis, University of California, Berkeley, 2002.

[KSF$^+$94] Leonidas I. Kontothanassis, Rabin A. Sugumar, Greg Faanes, James E. Smith, and Michael L. Scott. Cache performance in vector supercomputers. In *Supercomputing*, pages 255–264, 1994.

[KTHK03] Kenji Kitagawa, Satoru Tagaya, Yasuhiko Hagihara, and Yasushi Kanoh. A hardware overview of SX-6 and SX-7 supercomputer. *NEC Research and Development Journal*, 44(1):2–7, January 2003.

[Lee96] Ruby B. Lee. Subword parallelism with max-2. *IEEE Micro*, 16(4):51–59, 1996.

[Lov77] David B. Loveman. Program improvement by source-to-source transformation. *Journal of the ACM*, 24(1):121–145, 1977.

[LRYT99] Alvin R. Lebeck, David R. Raymond, Chia-Lin Yang, and Mithuna Thottethodi. Annotated memory references: A mechanism for informed cache management. In *European Conference on Parallel Processing*, pages 1251–1254, 1999.

[LTA03] Andrew A. Lamb, William Thies, and Saman Amarasinghe. Linear analysis and optimization of stream programs. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 12–25, 2003.

[Mat02] Peter Raymond Mattson. *A Programming System for the Imagine Media Processor.* PhD thesis, Stanford University, 2002.

[Mic03] Micron Technology Inc. *Micron 256Mb: x4, x8, x16 DDR2 SDRAM Data Sheet*, 2003.

[MPJ+00] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart memories: a modular reconfigurable architecture. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 161–171, 2000.

[OCS98] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Active pages: a computation model for intelligent memory. In *ISCA '98: Proceedings of the 25th annual international symposium on Computer architecture*, pages 192–203, 1998.

[PAB+05] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel,

T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation CELL processor. In *ISSCC*, pages 184–185, 2005.

[Pat04] David A. Patterson. Latency lags bandwith. *Communications of the ACM*, 47(10):71–75, 2004.

[PW96] Alex Peleg and Uri Weiser. MMX technology extension to the Intel architecture. *IEEE Micro*, 16(4):42–50, 1996.

[QCEV99] Francisca Quintana, Jesus Corbal, Roger Espasa, and Mateo Valero. Adding a vector unit to a superscalar processor. In *Proceedings of the 13th international conference on Supercomputing*, pages 1–10, June 1999.

[RAJ00] Parthasarathy Ranganathan, Sarita Adve, and Norman P. Jouppi. Reconfigurable caches and their application to media processing. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 214–224, 2000.

[Raj04] Sridhar Rajagopal. *Data-parallel Digital Signal Processors: Algorithm Mapping, Architecture Scaling and Workload Adaptation.* PhD thesis, Rice University, Houston, TX, May 2004.

[RDK$^+$98] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo Lopez-Lagunas, Peter R. Mattson, and John D. Owens. A bandwidth-efficient architecture for media processing. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dallas, TX, November 1998.

[RDK$^+$00a] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. In *International Symposium on Computer Architecture (ISCA)*, Vancouver, B.C., Canada, June 2000.

[RDK$^+$00b] Scott Rixner, William J. Dally, Brucek Khailany, Peter Mattson, Ujval J. Kapasi, and John D. Owens. Register organization for media processing. In *International Symposium on High Performance Computer Architecture (HPCA)*, Toulouse, France, January 2000.

[Rus78] Richard M. Russell. The Cray-1 computer system. *Commun. ACM*, 21(1):63–72, 1978.

[SFS00] J.E. Smith, Greg Faanes, and Rabin Sugumar. Vector instruction set support for conditional operations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 260–269, June 2000.

[SJ01] Premkishore Shivakumar and Norman P. Jouppi. CACTI 3.0: An integrated cache timing, power, and area model. *WRL Research Report*, August 2001.

[SKMB03] Karthikeyan Sankaralingam, Stephen W. Keckler, William R. Mark, and Doug Burger. Universal mechanisms for data-parallel architectures. In *MICRO*, pages 303–314, 2003.

[Ste90] Per Stenström. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12–24, 1990.

[SX-02] NEC Corporation. *SX-6 Series CPU Functional Description Manual*, 2002.

[TEL98] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 533–544. ACM Press, 1998.

[TKA02] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, London, UK, 2002. Springer-Verlag.

[TONH96] Marc Tremblay, J. Michael O'Connor, Venkatesh Narayanan, and Liang He. VIS speeds new media processing. *IEEE Micro*, 16(4):10–20, 1996.

[vdSvBA$^+$01] David van der Spoel, Aldert R. van Buuren, Emile Apol, Pieter J. Meulenhoff, D. Peter Tieleman, Alfons L. T. M. Sijbers, Berk Hess, K. Anton Feenstra, Erik Lindahl, Rudi van Drunen, and Herman J. C. Berendsen. *Gromacs User Manual version 3.1.* Nijenborgh 4, 9747 AG Groningen, The Netherlands. Internet: http://www.gromacs.org, 2001.

[WMRW02] Zhenlin Wang, Kathryn S. McKinley, Arnold L. Rosenberg, and Charles C. Weems. Using the compiler to improve cache replacement decisions. In *IEEE PACT*, pages 199–208, 2002.

[YY92] Qing Yang and Liping Wu Yang. A novel cache design for vector processing. In *Proceedings of the 19th annual international symposium on computer architecture*, pages 362–371, May 1992.