MEMORY AND CONTROL ORGANIZATIONS OF STREAM PROCESSORS

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

Jung Ho Ahn
March 2007

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

William J. Dally Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mendel Rosenblum

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christoforos Kozyrakis

Approved for the University Committee on Graduate Studies.

Abstract

The increasing importance of numerical applications and the properties of modern VLSI processes have led to a resurgence in the development of architectures with a large number of ALUs, multiple memory channels, and extensive support for parallelism. In particular, stream processors achieve area- and energy-efficient high performance by relying on the abundant parallelism, multiple levels of locality, and predictability of data accesses common to media, signal processing, and scientific application domains. This thesis explores the memory and control organizations of stream processors.

We first study the design space of streaming memory systems in light of the trends of modern DRAMs – increasing concurrency, latency, and sensitivity to access patterns. From a detailed performance analysis using benchmarks with various DRAM parameters and memory-system configurations, we identify read/write turnaround penalties and internal bank conflicts in memory-access threads as the most critical factors affecting performance. Then we present hardware techniques developed to maximize the sustained memory system throughput.

Since stream processors heavily rely on parallelism for high performance, certain operations requiring serialization can significantly hurt performance. This can be observed in superposition type updates and histogram computation, which suffer from the memory collision problem. We introduce and detail scatter-add, the data-parallel form of the scalar fetch-and-op, which solves this problem by guaranteeing the atomicity of data accumulation with a memory system.

Then we explore the scalability of the stream processor architecture along the instruction, data, and thread level parallelism dimensions. We develop VLSI cost and performance models for a multi-threaded processor in order to study the tradeoffs in functionality and

cost of mechanisms that exploit the different types of parallelism. We evaluate the specific effects on performance of scaling along the different parallelism dimensions and explain the limitations of the ILP, DLP, and TLP hardware mechanisms.

Acknowledgements

It is my great pleasure to write down the precious memories I have been sharing with talented people at Stanford. First and foremost, I would like to thank Professor Bill Dally, my advisor, providing me with the opportunity to join his vision of 'stream' world through the Imagine and Merrimac projects. His passion for research and in-depth knowledge on broad area always let my research career full of inspiration and idea. Through his vision and leadership, Bill has always been my role model. There is no doubt that his patience and continuous guidance formulate much of the research presented in this thesis.

I would also like to thank the other members of my reading committee, Professor Mendel Rosenblum and Professor Christos Kozyrakis, as well as my orals chair Professor Eric Darve for their kin insight of my research from different aspects, valuable feedback regarding the work described in this thesis, and pleasant interactions over my years at Stanford. I am also grateful to Professor Beom Hee Lee at Seoul National University for giving me enough motivation and extensive guidance to initiate a journey as a researcher.

I am indebted to my amazing colleagues on the Imagine and Merrimac projects with whom I collaborated. Brucek Khailany, Ujval Kapasi, John Owens, and Ben Serebrin in the Imagine project helped me a lot to learn much about streaming and to evaluate the Imagine stream processor. Abhishek Das has been in the same projects with me and exchanged innumerable amount of intelligence ranged from random thoughts to infrastructure efforts. I want to especially thank Mattan Erez. I was blessed in interacting with Mattan most closely throughout the Merrimac project and we achieved lots of fruitful research result including this dissertation. Other Merrimac members including Tim Knight, Nuwan Jayasena, Francois Labonte, Jayanth Gummaraju, and Kayvon Fatahalian provided numerous feedback and insights on my research.

I would also like to thank the members of the CVA group. Andrew Chang, Arjun Singh, Kelly Shaw, Sarah Harris, Brian Towles, Jinyung Namkoong, Patrick Chiang, and Amit Gupta gave me a cordial reception from the first days and delightfully spent time with me on a variety of research topics as well as other issues. Recently, I had the pleasure to interact with other group members including David Black-Schaffer, Paul Hartke, Oren Kerem, John Kim, Jiyoung Park, and Manman Ren not just in the offices but also on the ski slope.

I am grateful to the assistance I got over the years from Pamela Elliot, Wanda Washington, and Jane Klickman, and to the funding agencies that supported me during my graduate career, including Stanford Graduate Fellowships, Department of Energy (under the ASCI Alliances Program, contract LLNL-B523583), and Northrup Grummon Corporation.

Finally, I cannot say enough about the warm support provided by my friends and family. My parents, Hong Joon Ahn and Keum Sook Seo, devoted themselves to my education and have been my biggest supporters. My brother, Sung Ho, not just helped me as an advisor of my life but also has been my closest friend. I would like to express my sincere gratitude to all of my friends and family members who have helped me in one way or another over the years.

Contents

Abstract **v**

Acknowledgements **vii**

1 Introduction **1**

1.1 Research Contributions 3

1.2 Outline 4

2 Stream Architecture **5**

2.1 Stream Programming Model 5

2.2 Stream Processor Architecture 9

2.3 Merrimac Streaming Scientific Processor 11

3 Streaming Applications **17**

3.1 Media and Scientific Applications 17

3.2 Experimental Setup 26

3.3 Performance Analysis 27

4 Memory System Design Space **30**

4.1 DRAM Architecture 32

4.2 Streaming Memory Systems 36

4.2.1 Address Generators 38

4.2.2 Memory Access Scheduling 39

4.2.3 Memory Channels 40

4.2.4 Channel-Split Configuration 41
4.3 Experimental setup 42
4.3.1 Test Applications 42
4.3.2 Simulated Machine Parameters 44
4.4 Application Results 44
4.5 Sensitivity to Memory System Design Parameters 48
4.5.1 Number of AGs and Access Interference 48
4.5.2 Aggregate AG Bandwidth 49
4.5.3 Memory Access Scheduling 51
4.5.4 Memory Channel Buffers 51
4.5.5 Address Interleaving Method 53
4.6 Sensitivity to DRAM Technology Trends 54
4.6.1 Read-Write Turnaround Latency 54
4.6.2 DRAM Burst Length and Indexed Accesses 54
4.7 Related Work 58

5 Scatter-Add in Stream Architectures 60
5.1 Related Work 63
5.1.1 Software Methods 63
5.1.2 Hardware Methods 64
5.2 Scatter-Add Architecture 65
5.2.1 Scatter-add Micro-architecture 66
5.2.2 Scatter-add Usage and Implications 71
5.3 Evaluation 72
5.3.1 Test Applications 72
5.3.2 Experimental Setup 75
5.3.3 Scatter-add Comparison 75
5.3.4 Sensitivity Analysis 78
5.3.5 Multi-Node Results 80

6 Control Organizations of Stream Processors 84
6.1 Related Work 86

6.2 ALU Organization of Stream Architecture 87
6.2.1 DLP 87
6.2.2 ILP 89
6.2.3 TLP 89
6.3 Impact of Applications on ALU Control 90
6.3.1 Throughput vs. Real-Time 90
6.3.2 Scaling of Parallelism 92
6.3.3 Control Regularity 92
6.4 VLSI Area Models 93
6.4.1 Cost Model 93
6.4.2 Cost Analysis 97
6.5 Performance Evaluation 100
6.5.1 Experimental Setup 102
6.5.2 Performance Overview 102
6.5.3 ILP Scaling 104
6.5.4 TLP Scaling 105

7 Conclusion **110**
7.1 Future Work 112

Bibliography **114**

List of Tables

3.1 Summary of computation, memory, and control aspects 26
3.2 Machine parameters 27
3.3 Performance summary of multimedia and scientific applications 28

4.1 DRAM timing parameters 35
4.2 Streaming memory system nomenclature 36
4.3 Baseline machine parameters 43
4.4 Application characteristics 44

5.1 Baseline machine parameters 75

6.1 Summary of VLSI area parameteres 94
6.2 Summary of VLSI area cost models 96
6.3 Scheduler results of software-pipeline initiation interval for critical kernels. 105

List of Figures

2.1 An example regular code taken from a matrix multiply application 6
2.2 An example stream code taken from a matrix multiply application 7
2.3 The synchronous data flow representation of the example stream code. . . 8
2.4 Canonical stream processor architecture 10
2.5 Floor plan of a Merrimac processor designed using $90nm$ technology . . . 12
2.6 Merrimac arithmetic cluster 13
2.7 Merrimac memory system 14
2.8 Merrimac system diagram 15

3.1 Data allocation of convolution filters and SAD kernel in clusters for DEPTH 18
3.2 A primary access pattern of the data in memory for DEPTH 19
3.3 Allocation of a QRD input matrix in the SRF and memory 20
3.4 Data allocation for row vector and matrix multiplication in QRD 21
3.5 Matrix partitioning and data mapping from memory to the SRF 22
3.6 Hierarchical data partitioning of matrices in matrix-matrix multiply 24

4.1 A typical modern DRAM organization 33
4.2 Visualization of DRAM timing parameters during DRAM operations 34
4.3 Canonical streaming memory system 37
4.4 *channel-split* configuration 41
4.5 Memory system performance for representative configurations. 45
4.6 Memory system performance with realistic arithmetic timing 47
4.7 Throughput vs. AG configuration of micro-benchmarks and applications . . 49
4.8 Throughput vs. total AG bandwidth of micro-benchmarks and applications . 50

4.9 Throughput vs. memory access scheduling policy 51
4.10 Throughput vs. total MC buffer depth of micro-benchmarks and applications 52
4.11 Throughput vs. address interleaving method 53
4.12 Throughput vs. DRAM $tdWR$ of micro-benchmarks and applications . . . 55
4.13 Throughput vs. DRAM burst length of micro-benchmarks and applications 56
4.14 Analytical model and measured throughput of random indexed access patterns 57

5.1 Memory collision in parallel histogram computation 61
5.2 Scatter-add unit in the memory channels 66
5.3 Scatter-add unit in a stream cache bank 68
5.4 Flow diagram of a scatter-add request 70
5.5 Two algorithms implemented for the sparse matrix-vector multiply 73
5.6 Performance of histogram computation for inputs of varying lengths 76
5.7 Performance of histogram computation for inputs of varying index ranges . 77
5.8 Performance of histogram computation with privatization 77
5.9 Performance of sparse matrix-vector multiplication. 78
5.10 Performance of MOLE with and without scatter-add 79
5.11 Histogram runtime sensitivity to combining store size and varying latencies. 80
5.12 Histogram runtime sensitivity to combining store size and varying throughput. 81
5.13 Multi-node scalability of scatter-add 82

6.1 ALU organization along the DLP, ILP, and TLP axes 88
6.2 Relative area per ALU with the baseline configuration 97
6.3 Relative area per ALU for configurations without inter-cluster switch 99
6.4 Relative increase of area per ALU varying cluster and storage size 101
6.5 Relative run time of the 6 applications on 6 ALU organizations 103
6.6 Relative run time of the 6 applications on two different SRF sizes 104
6.7 Relative run time of MATMUL and MOLE with and without inter-cluster switch 106
6.8 Array access patterns of FFT3D 108

Chapter 1

Introduction

The relative importance of multimedia and scientific computing applications continues to increase. This leads to a demand for improvement in performance, cost and power efficiency, and programmability of processing systems serving these applications. For example, people desire higher data rates and better coverage on wireless devices, higher resolution and more enhanced realism on video games, and bigger datasets and more complex equation solvers on scientific simulations. Additionally, consumers require these products to provide conveniences such as longer battery life, cooler and quieter CPUs, and energy efficiency. Furthermore, they want the computing solution to support multiple standards and personalized services, as well as a wide range of applications. These application domains exhibit large amounts of parallelism, specifically data parallelism, which allows concurrent computation on many data elements by data-parallel processors.

Modern VLSI implementation technology used to build these data-parallel processors makes computation cheap and bandwidth expensive. Moore's law provides myriads of transistors enabling a large and increasing number of ALUs per chip, and the energy to perform an ALU operation decreases rapidly. However the latency and energy to deliver instructions and data over the global communication path improves much more slowly. This dominates power consumption and becomes a performance bottleneck (for example in $0.13\mu m$ technology, the energy to deliver a 64-bit data over global on-chip wires is 20 times the energy to perform a 64-bit floating point operation [DHE+03]). As a result, it is critical to place most of the data necessary for computation close to the computation units.

The increasing importance of numerical applications and the properties of modern VLSI processes have led to a resurgence in the development of data-parallel architectures with a large number of ALUs as well as extensive support for parallelism (e.g., [KDR+01, KPP+97, WTS+97, SNL+03, MPJ+00, DHE+03, KBH+04, PAB+05]). In particular, *stream processors* exemplified by Imagine [KDR+01] and Merrimac [DHE+03] are designed to achieve three goals: high performance (relying on parallelism which enables the use of 10s to 100s of functional units and to overlap computation and communication), high efficiency (relying on locality which enables matching the high bandwidth requirements of functional units to limited global bandwidth and using available global bandwidth efficiently), and programmability (using high level languages designed to allow automatic compiler optimization).

This dissertation focuses on the memory and control organization of stream processors. Memory systems of stream processors must maintain a large number of outstanding memory references to fully use increasing DRAM bandwidth in the presence of rising latencies. Additionally, throughput is increasingly sensitive to the reference patterns due to the rising latency of issuing DRAM commands, switching between reads and writes, and precharging or activating internal DRAM banks. We study the design space of memory systems in light of these trends of increasing concurrency, latency, and sensitivity to access patterns. We identify the interference between concurrent read and write memory-access threads, and bank conflicts, both within a single thread and across multiple threads, as the most critical factors affecting performance.

Then we introduce a novel memory system architecture extension called *scatter-add*, the data-parallel form of the well-known scalar fetch-and-op. The scatter-add mechanism scatters a set of data values to a set of memory addresses and adds each data value to the corresponding memory location instead of overwriting it, allowing us to efficiently support data-parallel atomic update computations such as superposition type updates in scientific computing and histogram computations in media processing.

We also explore the computation control organization of the stream processor architecture along the instruction, data, and thread level parallelism (*ILP*, *DLP*, and *TLP* respectively) dimensions. We develop detailed VLSI-cost and processor-performance models for

a multi-thread stream processor and evaluate the tradeoffs, in both functionality and hardware costs, of mechanisms that exploit the different types of parallelism. We argue that for stream applications with scalable parallel algorithms the performance is not sensitive to the control structures used within a large range of area-efficient architectural choices. We evaluate the specific effects on performance of scaling along the different parallelism dimensions and explain the limitations of the ILP, DLP, and TLP hardware mechanisms.

1.1 Research Contributions

These are the main contributions of this dissertation to the field of computer architecture and stream processing:

- We provide an analysis of the performance trends of DRAM and their effect on memory system architecture. We establish that the sensitivity of DRAM throughput to application access patterns is increasing rapidly as a result of the growing ratio between DRAM bandwidth and latency, and show the importance of maintaining locality due to the high access-pattern sensitivity. We identify the trends of large granularity in DRAM bursts, and long-latency of DRAM commands as a potential problem affecting the throughput of applications that perform random accesses, and develop an accurate analytical model for the expected effective bandwidth.

- We give a detailed design space study of stream processor memory systems. We explore the performance behavior of memory systems in light of DRAM technology trends. We suggest a memory system design that focuses on locality, and introduce the novel channel-split memory system configuration that achieves locality without sacrificing channel load balance.

- We prescribe a novel architecture extension for memory system that supports binning and superposition on streaming architectures. We design an innovative hardware mechanism called scatter-add, and evaluate and contrast the data-parallel performance of previous software only techniques with hardware scatter-add. We also show the feasibility of scatter-add by detailing its micro-architecture, estimating its

small die area requirements, and analyzing its sensitivity to key system parameters and multi-node scalability.

- We evaluate the tradeoff between DLP, ILP, and TLP in stream processors. We develop detailed hardware-cost and processor-performance models for a multi-threaded stream processor. We study, for the first time, the hardware tradeoff of organizing the number of ALUs along the ILP, DLP, and TLP dimensions combined. We evaluate the performance benefits of the additional flexibility allowed by multiple threads of control, and the performance overheads associated with the various hardware mechanisms that exploit parallelism.

1.2 Outline

The remainder of this thesis is organized as follows: Chapter 2 describes the stream execution model and stream architecture as an overview of background material and provides more details of the Merrimac streaming supercomputer as an example stream architecture. Chapter 3 enumerates the multimedia and scientific applications commonly used in this thesis, experimental setup, and performance summary of the applications on baseline processor configurations.

Chapter 4 summarizes the characteristics of modern DRAM architectures and explores memory system design space including application results and sensitivity studies over DRAM trends. The hardware scatter-add mechanism is studied in the context of stream processing in Chapter 5. Chapter 6 studies the ILP/DLP/TLP hardware tradeoff, discusses application properties and corresponding architectural choices, and develops the hardware cost model and scaling analysis. Finally, Chapter 7 presents conclusions and future work.

Chapter 2

Stream Architecture

The inherent parallelism and locality in media and scientific applications are exposed by a stream programming model, expressed by stream processing languages, and extracted by corresponding stream compilers. Then stream processors exploit this locality and parallelism using many ALUs with hierarchical storage enabled by modern VLSI technologies. They provide much higher performance on given area and power approaching special-purpose processors on media and scientific applications while retaining the flexibility of a programmable engine.

Section 2.1 introduces a stream programming model and explains how it exposes underlying locality and parallelism of target applications using *streams* and *kernels*. Section 2.2 explains how stream processor architecture exploits these exposed locality and parallelism by ALU control organization and storage hierarchy. Section 2.3 provides more details with an example stream processor architecture, Merrimac, targeting scientific domain applications and scaling up to thousands of processing nodes.

2.1 Stream Programming Model

The stream programming model [KRD+03, BFH+04, GR05] defines how to write a stream program and how to apply the written application to the stream architecture. A computer program is a list of instructions that explicitly implement an algorithm to process data structures.

A stream program has a native data type called *stream* and two levels of instructions manipulating these streams in addition to the normal instructions and data of the computer program. A stream program consists of a scalar processing part, which executes conventional scalar instructions, and a stream processing part. In the stream processing part, the data in memory is gathered into a stream, a collection or records which can be operated on in parallel. One or more streams are processed by one or more *kernels*. A kernel is a special type of function consisting of a list of instructions that consumes the records of input streams, processes them, and produces output streams. Finally output streams are scattered back to memory. Instructions invoking kernels, scatters, and gathers are called stream level instructions. Instructions within a kernel are called kernel level instructions.

```
for i = 1…n {
  … = dat_1[idx_a[i]];
    •
  for j = 1…m {
    … = dat_2[idx_b[i*m+j]];
    … = dat_3[idx_c[i*m+j]];
      •
  }
  dat_4[i] = …;
}

for i = 1…n {
  … = dat_4[i];
  for j = 1…m {
    … = dat_5[idx_d[i*m+j]];
    … = dat_6[idx_e[i*m+j]];
      •
  }
  dat_7[idx_f[i]] = …;
}
```

Figure 2.1: This is an example of a regular code taken from a matrix multiply application. There are two nested loops in this code. In each iteration of the first loop, the data from `dat_1` and the data arrays from `dat_2` and `dat_3` are loaded from memory whose locations are specified by index array, and then they are operated on to produce a result `dat_4[i]`, which is written back to memory. In each iteration of the second loop, similarly, the data from `dat_4`, `dat_5`, and `dat_6` are loaded from memory, and operated on to produce a result, which is written back to memory whose location in `dat_7` is specified by `idx_f`.

Data could be gathered from and scattered to sequential, strided, or random addresses

in memory. Gathering all the elements in a row of a matrix is an example of sequential access. Gathering the first element of each row in a matrix is an example of strided access with stride of the length of a row. Gathering the neighbors of a molecule in an irregular mesh is an example of random access, which needs a stream specifying memory accesses.

```
load_stream   str_1,dat_1,idx_a
load_stream   str_2,dat_2,idx_b
load_stream   str_3,dat_3,idx_c
run_kernel    krnl_1,
              str_1,str_2,str_3,str_4

load_stream   str_5,dat_5,idx_d
load_stream   str_6,dat_6,idx_e
run_kernel    krnl_2,
              str_4,str_5,str_6,str_7
store_stream  dat_7,str_7,idx_f
```

(a) Stream level instructions

```
krnl_1(str_1, str_2, str_3, str_4) {
  for i = 1…n {
    … = str_1[i];
      ⋮
    for j = 1…m {
      … = str_2[i*m+j];
      … = str_3[i*m+j];
        ⋮
    }
  str_4[i] = …;
}

krnl_2(str_4, str_5, str_6, str_7) {
  for i = 1…n {
    … = str_4[i];
    for j = 1…m {
      … = str_5[i*m+j];
      … = str_6[i*m+j];
        ⋮
    }
  str_7[i] = …;
}
```

(b) Kernel level instructions

Figure 2.2: An example stream code taken from a matrix multiply application. The data arrays `dat_1`, `dat_2`, and `dat_3` are gathered into streams `str_1`, `str_2`, and `str_3`, and then operated on within `krnl_1` producing output stream `str_4`. This `str_4` and streams gathered from the data arrays `dat_5` and `dat_6` are processed by `krnl_2` producing output stream `dat_7`, which is scattered back to memory with index stream `idx_f`.

To emphasize the dissimilarities between the conventional programming model and the stream programming model, we present two versions of an example taken from a matrix multiply application: a regular code (Figure 2.1) and a stream code (Figure 2.2). We used the stream API syntax shown in [KRD+03] to represent the stream level operations in Figure 2.2(b). The two code fragments have a number of differences. In the regular code, memory accesses are interleaved with computation while they are separated in the stream code. The stream level instructions explicitly copy the data in memory into streams and return the result streams after the kernels execute back to memory. Because these stream

transfers are performed in bulk and their data access patterns are apparently specified, the throughput of memory accesses dominates performance, not the latency.

Mapping an application into the stream programming model makes underlying locality and parallelism apparent. All the elements within a stream can be operated upon in parallel because there is ordering or dependencies implied between elements within a stream. Therefore, it exposes data-level parallelism. Indirect memory accesses within a loop of the regular code are replaced into stream accesses in a kernel of the stream code, eliminating ambiguous dependency problem. This makes it easier to extract instruction-level parallelism from a software system by using traditional compiler optimizations. Streams gathered, processed, and scattered determine dependencies between the stream level operations, which lets multiple kernels and memory transfers perform concurrently (if there is no dependency between them), and task-level parallelism can be captured. The array `dat_4` is stored into memory and read back in the regular code, but the corresponding stream `str_4` is produced and consumed between the kernels and not backed by memory, capturing long-term producer-consumer locality. Also the intermediate values of a kernel are generated and consumed within the kernel, exposing short-term producer-consumer locality.

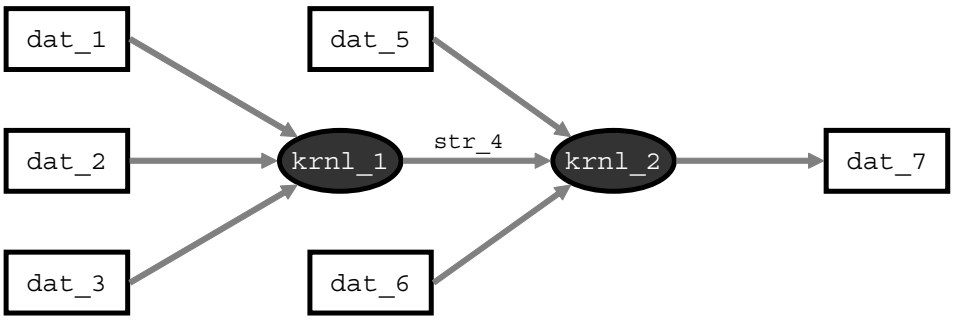


Figure 2.3: The synchronous data flow representation of the example stream code.

A restricted Synchronous Data Flow (SDF) representation [LM87, TKA02] can also be used to describe stream programs, and Figure 2.3 shows the SDF graph version of the stream code shown in Figure 2.2, where arrows, ovals, and rectangles correspond to streams, kernels, and data in memory. The SDF graph visualizes the flow of data through computation kernels.

2.2 Stream Processor Architecture

The stream architecture expands a conventional computer design model by adding intermediate storage called *stream register file* (SRF). It also adds special types of instructions called stream and kernel instructions which run the applications written under the stream programming model.

To support the stream programming model, new processor architectures are designed which dedicate hardware resources to store streams and execute kernels [KDR+01, DHE+03] or existing architectures are used [GR05]. For example, graphics processors [BFH+04] or general purpose CPUs [GR05] are used by sharing some of the existing resources for stream transfers and kernel executions. Stream virtual machine was proposed to study more details of mapping the stream programming model into various architectures [LMB+04]. In this section, we focus on the stream processor architecture and explains its components and their roles to show how it supports the stream programming model, exploits the parallelism and locality of the applications, and utilizes VLSI resources effectively.

Figure 2.4 depicts a canonical stream processor that consists of a simple general purpose control core, a throughput-oriented streaming memory system, on-chip local storage in the form of the SRF storing streams, a set of ALUs with their associated *local register files* (LRFs), and instruction sequencers running kernels. This organization forms a hierarchy of bandwidth, locality, and control, thus mitigating the detrimental effects of distance in modern VLSI processes, where bandwidth drops and latency and power rise as distance grows.

The bandwidth hierarchy [RDK+00b] consists of the DRAM interfaces that process off-chip communication, the SRF that serves as a staging area for the bulk transfers to and from DRAM and utilizes high bandwidth on-chip structures, and the LRFs that are highly partitioned and tightly connected to the ALUs in order to support their bandwidth demands. Each storage level provides an order of magnitude or more aggregate bandwidth than the one a level farther from ALUs. The same organization supports a hierarchy of locality. The LRFs exploit short term producer-consumer locality within kernels while the SRF targets producer-consumer locality between kernels. The SRF is able to capture a working set of the application. To facilitate effective utilization of the bandwidth and locality hierarchy,

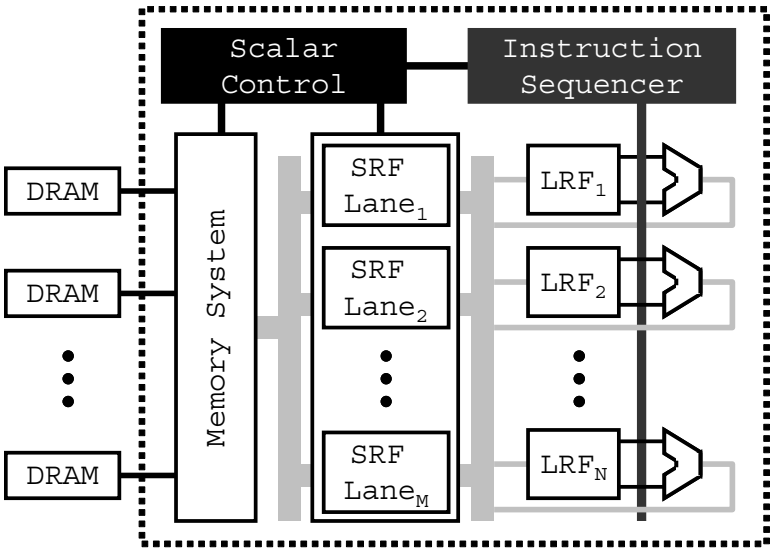


Figure 2.4: Canonical stream processor architecture

each level has a separate name space. The LRFs are explicitly addressed as registers by ALU instructions. The SRF is arbitrarily addressable on-chip memory that is banked to support high bandwidth and may provide several addressing modes – possibly performing better for certain access patterns like sequential (in address) accesses. The SRF is designed with a wide single ported SRAM for efficiency, and a set of *stream buffers* (SBs) are used to time-multiplex the SRF port [RDK+00b]. The memory system maintains the global DRAM address space.

The general purpose control core executes scalar instructions for overall program control and dispatches coarse-grained stream instructions to the memory system and instruction sequencer. Stream processors deal with more coarsely grained instructions in their control flow than both superscalar and vector processors, significantly alleviating the von Neumann bottleneck [Bac78]. The DMA engines in the memory system and the ALU instruction sequencers operate at a finer granularity and rely on decoupling for efficiency and high throughput. In a stream processor the ALUs can only access their private LRF and SRF and rely on the higher level stream program to transfer data between memory and the SRF. Therefore, the SRF decouples the ALU pipeline from the unpredictable latencies of DRAM accesses enabling aggressive and effective static scheduling and a design with lower hardware cost compared to out-of-order architectures. Similarly, the memory system

mainly handles DMA requests and can be optimized for throughput rather than latency.

The memory system uses multiple address-interleaved memory channels and makes each channel interface high-bandwidth DRAM components in order to address the demand for aggregate memory bandwidth. It relies on parallelism in order to saturate the memory bandwidth while tolerating DRAM latency and it relies on locality in order to reduce sensitivity and maintain near peak throughput. Further analysis of the factors affecting the performance of the memory system and its design space is the topic of Chapter 4.

Stream processors include a large number of ALUs partitioned into processing elements or clusters and provide mechanisms to control them through ILP, DLP, and TLP dimensions. VLIW instructions are used to control multiple functional units in a cluster exploiting ILP. A large number of clusters is operated in a SIMD fashion utilizing DLP. Multiple instruction sequencers might be used to control groups of clusters in a MIMD fashion enabling concurrent threads (TLP). More details of the scalability and trade-off of these ALU control mechanisms are covered in Chapter 6.

2.3 Merrimac Streaming Scientific Processor

In this section we explain more details of an example stream processor, the Merrimac processing node, with regard to its organization of clusters and memory system. This will be used as a baseline configuration for the design space studies in the following chapters. We also briefly introduce a Merrimac system which can scale up to thousands of processing nodes.

The Merrimac streaming supercomputer is designed to provide an order of magnitude or more efficiency than traditional supercomputers based on clusters. Figure 2.5 shows the floor plan of a Merrimac node [Ere06] whose size is estimated to be $12.0 \times 12.1mm^2$. The size of a floating-point multiply-add unit (FPU) is $0.9 \times 0.6mm^2$ and each arithmetic cluster measures $2.0 \times 2.1mm^2$. More than half of its total area is dedicated to the clusters aligned in 2-dimensional grids. Two MIPS64 20kc cores serve scalar processing role. They are mirrored to enhance reliability for a multi-node system in harmony with other fault tolerance features [EJKD05]. 16 arithmetic clusters are controlled by an instruction sequencer called a microcontroller and serves a stream computation role. The memory

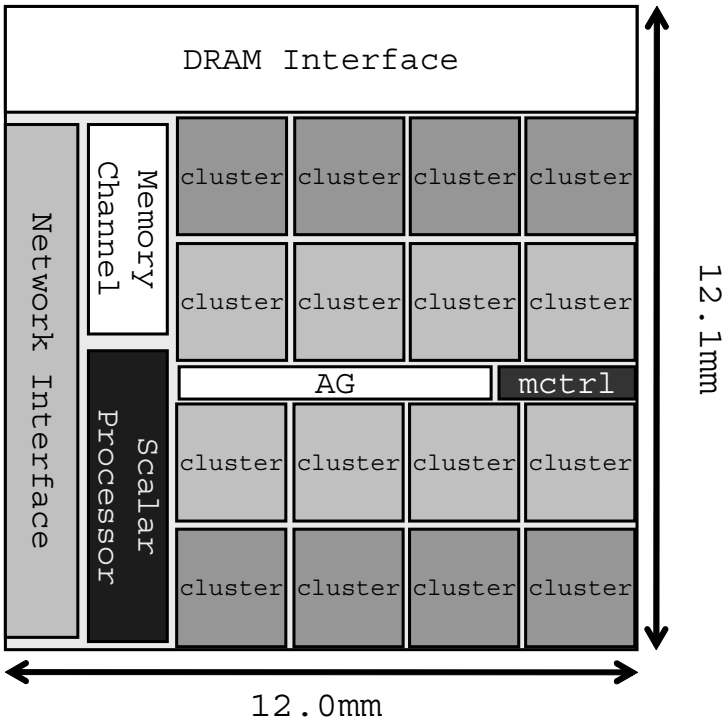


Figure 2.5: Floor plan of a Merrimac processor designed using $90nm$ technology

system consists of address generators (AGs), multiple memory channels connected to 8 XDR or future technology DRAM components supplying maximum 64 GB/s of off-chip memory bandwidth, and a network interface for interaction with other processing nodes through high-radix routers [DHE+03]. Operating at 1 GHz, a Merrimac processing node has peak performance of 128 GFLOPS and is estimated to dissipate 42 Watts, which is 10 times higher in peak performance than Pentium-4 operating at 3.2 GHz while consuming less than half of the power per chip.

An arithmetic cluster is composed of a set of ALU and non-ALU functional units, LRFs, and a lane of the SRF. As shown in Figure 2.6, each cluster has 4 FPUs, 1 iterative unit to support operations like divide and square-root, 1 juke-box unit to support conditional streams [Kap04], 1 COMM unit connected to an inter-cluster switch for data communication between clusters, and an intra-cluster switch connecting functional units, LRFs and SRF ports, together. All functional units are fully-pipelined. This means that each FPU can

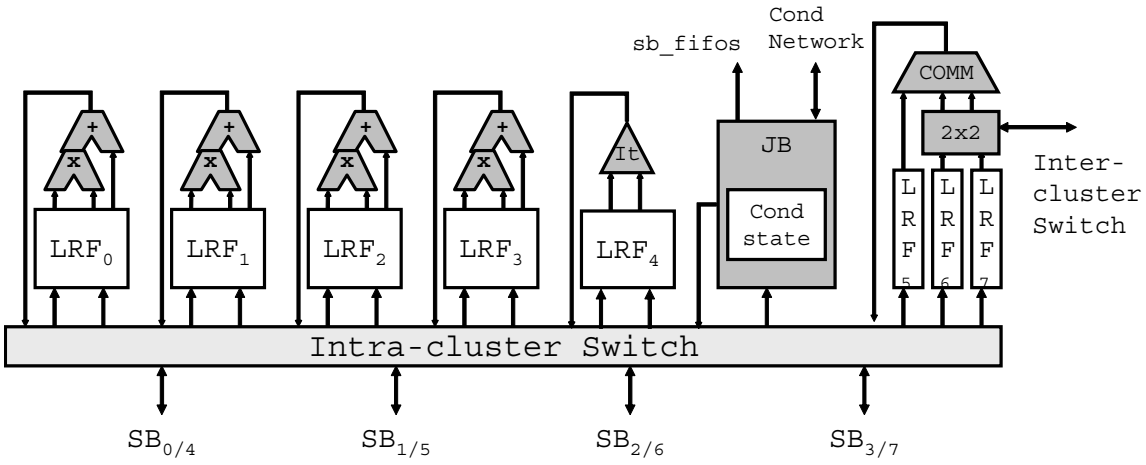


Figure 2.6: The Merrimac arithmetic cluster includes functional units, LRFs, an intra-cluster switch, a conditional support unit, and ports to inter-cluster switch and SRF.

perform one multiply-add operation per cycle making the peak arithmetic performance of 128 floating-point operations per cycle in a single node of the Merrimac processor.

Clusters are controlled by the micro-controller. When a stream instruction to invoke a kernel is received from the host scalar processor, the micro-controller starts fetching and issuing VLIW instructions from the microcode instruction storage to each functional unit in the arithmetic clusters. During kernel execution, each cluster executes an identical series of VLIW instructions on stream elements, and the 16 clusters then write output elements in parallel back to one or more output streams in the SRF. A Merrimac processing node uses 64 multiply-add floating point ALUs arranged in a (1-TLP, 16-DLP, 4-ILP) configuration by the terminology presented in Chapter 6.

The size of SRF per lane is 64 KB, so each Merrimac processor has 1 MB of SRF space. Each cluster can access data in the same lane of SRF either sequentially or arbitrarily (inlane indexing). The bandwidth of arbitrary accesses might be lower than that of sequential accesses, depending on access patterns. A cluster is also able to access SRF data in different lanes (crosslane indexing) [JEAD04] with lower performance due to the possible conflict of requests from other clusters.

Figure 2.7 shows the Merrimac memory system consisting of an address generator and

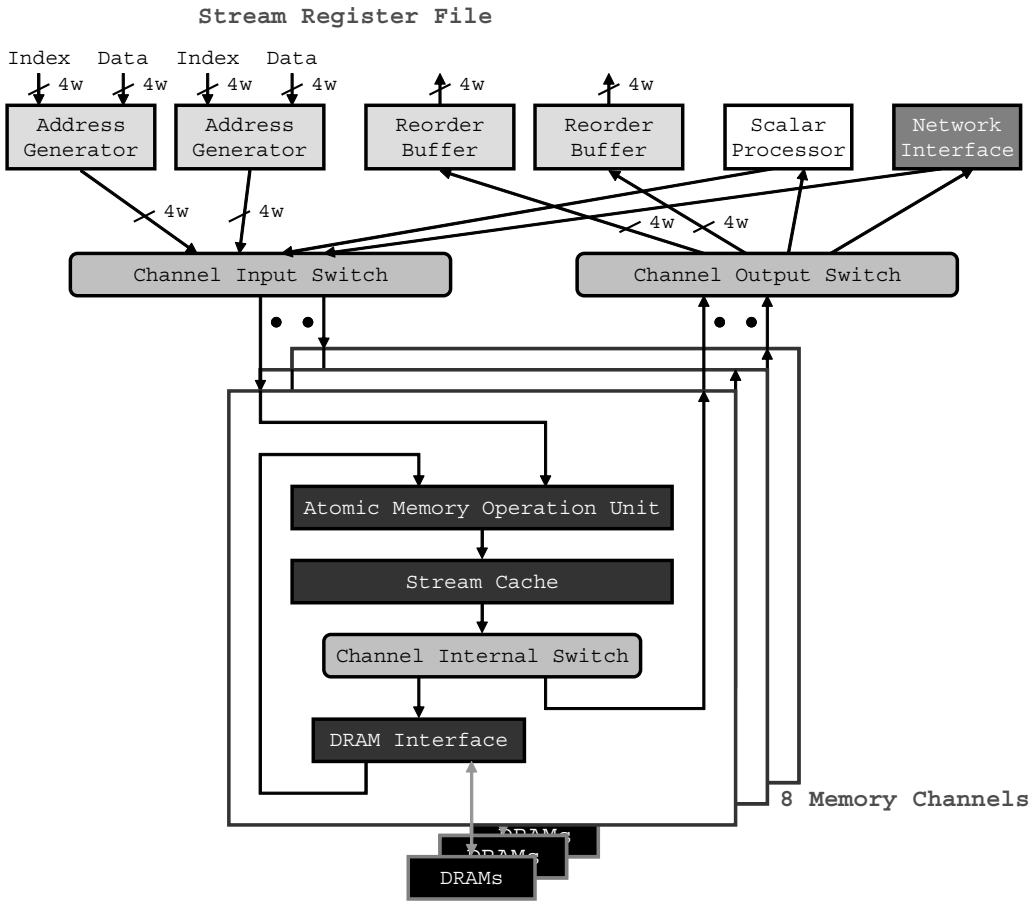


Figure 2.7: The Merrimac memory system consists of an address generator and reorder buffer pair, and memory channels connected through cross-point switches.

reorder buffer pair, as well as address-interleaved memory channels connected by cross-point switches. It also supports memory requests from the scalar processor and other Merrimac processors through network interface. The address generator receives a stream memory instruction from the scalar processor and translates a stream into a sequence of individual memory requests. It handles both strided and indexed streams where each request is a sequence of consecutive words (i.e., a record). Strided access streams have a constant distance between records, whereas indexed (gather for load and scatter for store) streams make arbitrary record references. A maximum of 8 memory requests are generated per cycle in order to feed the memory channels. Each request is moved to one of 8

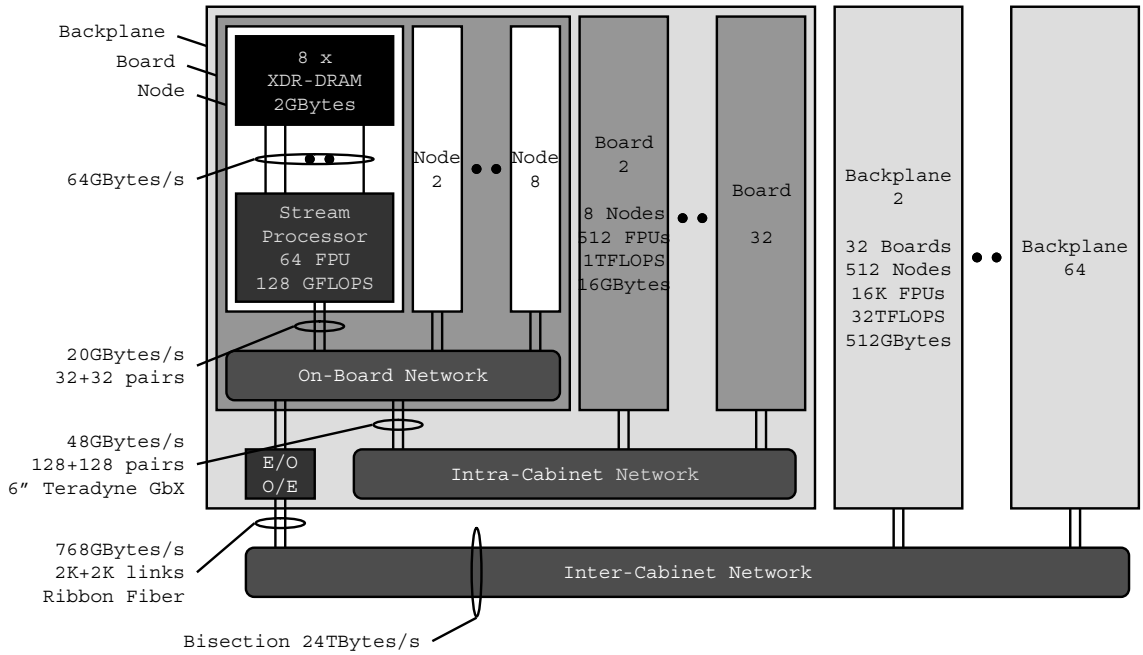


Figure 2.8: A Merrimac system with $16,384$ nodes having maximum 2 PFLOPS is connected by high-radix interconnection network.

line-interleaved memory channels through a cross-point switch.

Each memory channel includes an atomic memory operation unit, a 32 KB stream cache, and a DRAM interface. The atomic memory operation unit supports a scatter-add instruction, which acts as a regular scatter instruction, but it adds each value to the data already at each specified memory address rather than simply overwriting the data. Further discussion of this will be presented in Chapter 5. The three-dimensional structure of modern DRAMs like DDR2 and XDR DRAM causes their performance to heavily depend on the access patterns, in favor of blocked sequential accesses [AD05]. The stream cache works as a bandwidth amplifier for off-chip DRAM attached to the memory channels when the access pattern is not sequential but has spatial/temporal locality. Memory requests not served in the stream cache reach the DRAM interface and are processed out-of-order, using memory access scheduling [RDK$^+$00a], in order to maximize DRAM bandwidth utilization.

A Merrimac system can scale up to $16,384$ nodes having a maximum of 2 PFLOPS

as shown in Figure 2.8. A single Merrimac board contains 16 nodes — 16 128 GFLOPS processors each with 2 GB of DRAM — connected through an on-board network. The router chips included in an on-board network interconnect the 16 processors on the board, providing flat 20 GB/s per node. They also provide an out-going link to the intra-cabinet network by connecting 32 boards together with a memory bandwidth of one fourth of an on-board chip. The inter-cabinet network connects 32 cabinets with a bisection bandwidth totalling 40 TB/s, 2.5 GB/s per node. A five-stage folded-Clos [Clo53] network is used for the whole system, and high-radix routers [KDTG05], which effectively utilize chip pin bandwidth, compose each level of the interconnection network. Optical links are used between cabinets to cover the long latencies required at the top level of the system.

Chapter 3

Streaming Applications

This chapter explains the media and scientific applications that were used to analyze the performance of stream processors. The experimental setup which is common throughout the thesis is introduced, and then the performance summary of the applications with baseline configuration follows.

3.1 Media and Scientific Applications

In this section each application is briefly explained focusing on how the application is mapped onto a stream processor. We will show the data allocation in the SRF and DRAM as well as data access patterns in clusters and memory. The characteristics explained in this portion will be used to explain the results from experiments of the following chapters.

DEPTH is a stereo depth extractor application explained in Section 2.1 which takes 2 320×240 grey-scale images. It uses 16-bit data and integer operations for computation. Execution time of the application is dominated by a 2-dimensional convolution filter and SAD kernels, all of which have the same type of data access patterns. Figure 3.1 shows how each cluster takes data and performs the computation for 2×2 convolution filtering. Pixels are stored in the SRF in row-major order and specified as (row, column). Each cluster has an assigned pixel and takes pixels in its right, bottom, and bottom right to perform computation. For the first row of an image, consecutive pixels are allocated into clusters, each one having a pixel. Then each cluster takes a pixel data from the next cluster (a). Now

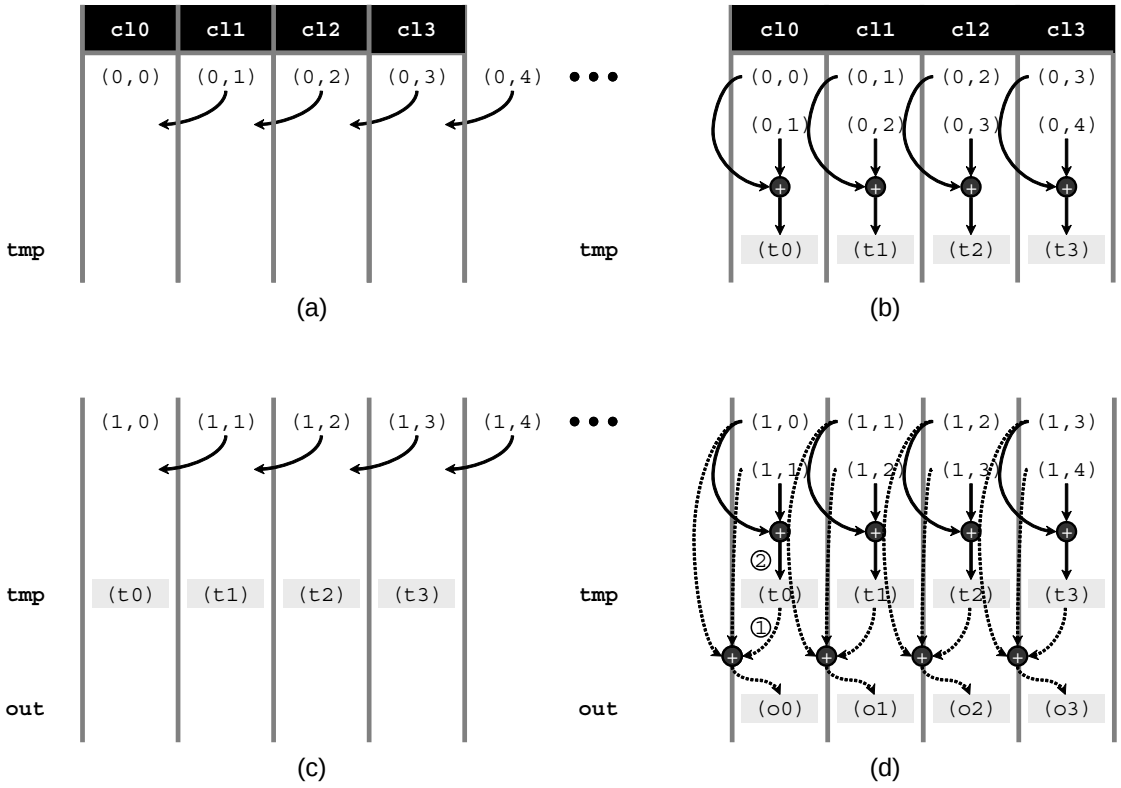


Figure 3.1: Data allocation of convolution filters and SAD kernel in clusters for DEPTH – (a) for the first row, data in the next clusters are moved, (b) two values are weighted and summed per cluster then stored into temporary space, (c) for the second and following rows, data in the next clusters are moved, (d) two values are weighted then summed with temporary value stored in (b) per cluster becoming output value, then two values are weighted, summed and stored into the temporary space

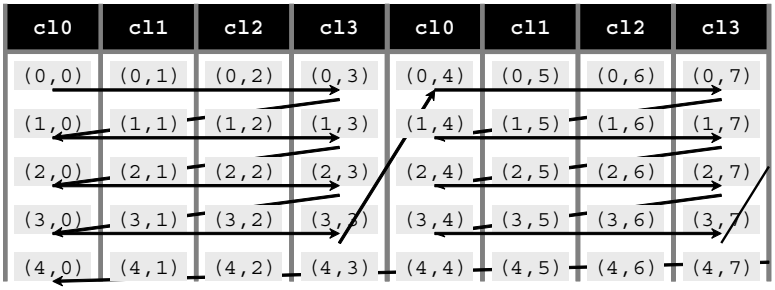


Figure 3.2: A primary access pattern of the data in memory for DEPTH – hierarchical 2-dimensional accesses

each cluster has two pixels of data, which are weighted, summed and stored into temporary space in the SRF (b). For the following rows, pixels are first allocated and moved into clusters like in a (c). These two pixels per cluster are weighted and summed with temporary value stored in b becoming output values (d(1)), then two values are weighted, summed and stored into the temporary space of b (d(2)). Note that the coefficients in b and d(2) are the same while they are different from the ones in d(1).

Figure 3.2 shows a primary pattern accessing data in memory for DEPTH. As it can be seen, it first follows small 2-dimensional square in row-major order. The length of each edge of the square is the number of clusters. Then it accesses the pixels of the next square in row-major order. These hierarchical 2-dimensional gather accesses are described using indexed streams.

MPEG encodes three frames of 320×240 24-bit video images according to the MPEG-2 standard [KRD+03]. Similar to DEPTH, it uses 16-bit data and integer operations. In MPEG, input images are divided into 8×8 macroblocks and processed in this macroblock basis. Out of three frames, the first one is intra-coded (I-frame) with 2D DCT, zig-zag and run-level encoding. Variable length coding kernels are applied on each luminance and chrominance macroblock. The next two frames are inter-coded (P-frame) and in addition to the kernels in the I-frame, a sequence of motion estimation kernels are used to find the reference macroblock which is most similar to the one currently encoded. This motion estimation phase gathers pixels using indexed streams. Pixels in macroblocks are aligned in the way same as Figure 3.2 where each cluster has a column of a macroblock. As a

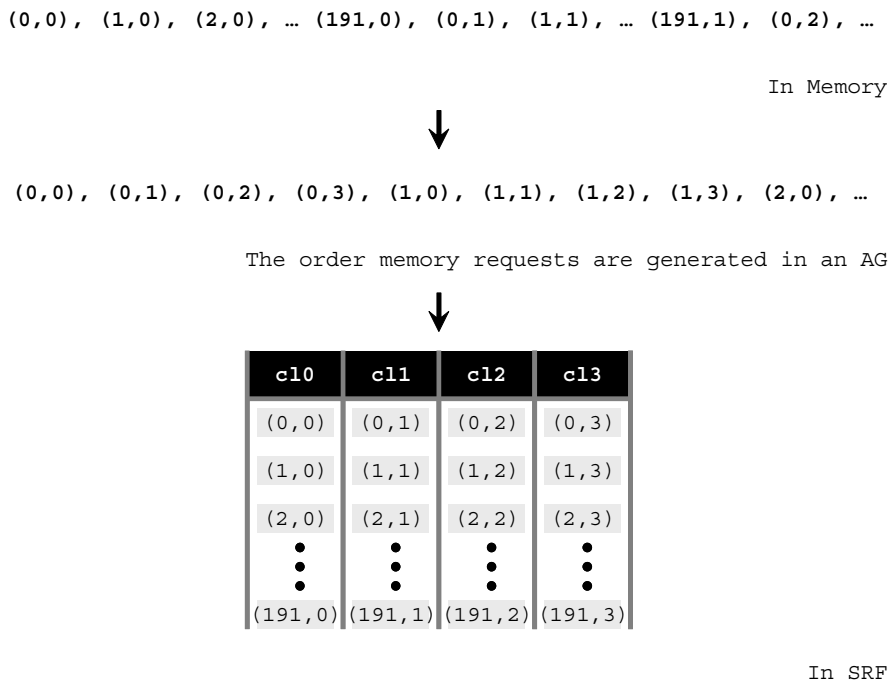


Figure 3.3: Allocation of a QRD input matrix in the SRF and memory — the matrix is stored in column-major order in memory, and each cluster takes each column of the matrix in the SRF

result, inter-cluster communication occurs in the cases like row-direction DCT and zig-zag encoding.

QRD converts a 192×96 complex matrix into an upper triangular and an orthogonal matrix, and is a core component of space-time adaptive processing [CTW97]. A blocked householder transformation method is used. Per column of input matrix A, householder matrix Q is computed and then multiplied with A to generate a new matrix A' which is upper triangular up to the column used. This application uses floating-point operations. Because its input matrix is stored in column-major order in memory and each cluster takes each column of the matrix in the SRF as shown in Figure 3.3, strided accesses with a stride of 192 and a large record size are used for this partial-transpose. Figure 3.3 also shows how an AG generates memory requests to load or store a stream consisting of large records. It generates the first words of C records (here $C = 4 =$ number of clusters), the second words

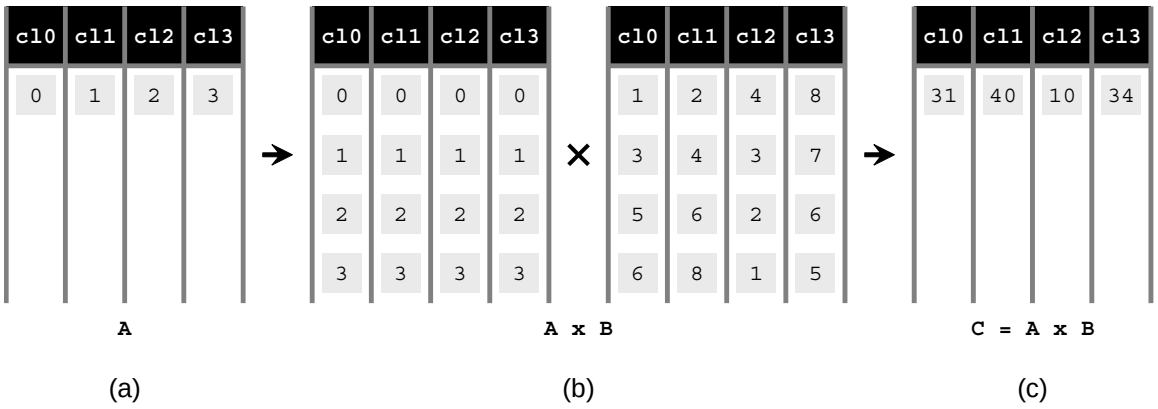


Figure 3.4: Data allocation for row vector and matrix multiplication in QRD — (a) Elements in a row vector q are distributed into clusters. (b) q is duplicated in clusters becoming column vectors in clusters. (c) The inner-products of column vectors of A and duplicated-q are computed in clusters becoming output vector B.

of the records and so on.

Figure 3.4 shows how matrix multiplication is performed in QRD. Matrix-matrix multiplication can be decomposed into a sequence of vector-matrix multiplications. In this example, elements of a row vector q are distributed into clusters (a). The elements are duplicated into clusters through an inter-cluster switch, so that each cluster has a complete row vector q. Since the elements of the matrix A are distributed over clusters so that a cluster takes a whole column of the matrix, inner-products of column vectors of A and duplicated-q are computed, becoming output vector B (b, c).

RTSL renders the first frame of the SPECviewperf 6.1.1 advanced visualizer benchmark with lighting and blending disabled and all textures point-sampled from a 512×512 texture map using the Stanford Real-Time Shading Language [PMTH01, Owe02]. It uses both integer and floating-point operations. The rendering pipeline of the application consists of four steps: per-primitive group work, the geometry, rasterization and composition steps. Each step consists of one or more kernels. The fragment program in the rasterization step needs to load textures while a z-compare kernel in composition step requires depth and color buffer accesses all performed using indexed streams. During the geometry stage, conditional streams [Kap04] are often used to remove unused vertices out of the working

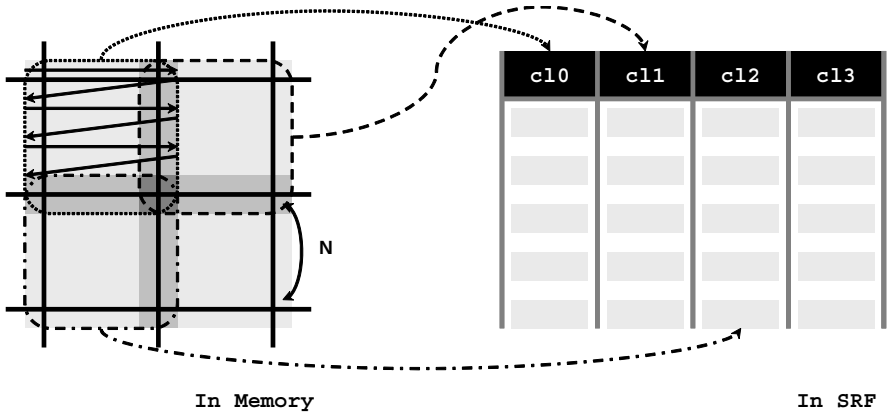


Figure 3.5: Matrix partitioning and data mapping from memory to the SRF — a matrix to be computed is divided into sub-matrices, here square matrices with each edge sized $(N + 4)$. Then each $(N + 4) \times (N + 4)$ sub-matrix is flattened into a 1-dimensional array in row-major order and loaded into each lane of the SRF.

dataset in clipping and backface culling kernels. The rasterization kernel generates variable length streams whose size depends on the geometry of triangles in the screen space. Sorting fragments and discarding ones farther than the fragments in the frame buffer during the composition step require conditional streams which use inter-cluster communications.

In the following scientific applications, arrays are stored in row-major order in memory and floating-point operations are used.

CONV2D performs 2-dimensional 5×5 convolution on a 2048×2048 dataset [JEAD04]. The input matrix is first divided into smaller 2-dimensional sub-matrices, possibly square matrices. Boundaries of adjacent sub-matrices are overlapped by $5 - 1 = 4$ elements per side. So in order to compute the convolution of an $N \times N$ sub-matrix, an $(N+4)\times(N+4)$ matrix is used. Then each sub-matrix is flattened into a 1-dimensional array or stream block in row-major order and loaded into each lane of the SRF as shown in Figure 3.5. The kernel performing the convolution computation uses temporary space in the SRF like in Figure 3.1 of DEPTH, but doesn't need inter-cluster communication since each cluster has a 2-dimensional sub-matrix necessary to produce an output sub-matrix.

FFT3D computes $128 \times 128 \times 128$ 3D complex FFT performed as three 1D FFTs by dividing the data into collections of *pencils*. First, rows are gathered into clusters computing

the X dimension of the data and scattered back to the original location; then columns are loaded, transformed, and stored in the Y dimension; again FFTs are computed over the last Z dimension after data is gathered. The decimation-in-time algorithm is used for 1D FFTs and bit-reversal of input data before butterfly stages are combined with gather operations using indexed streams.

MATMUL executes blocked 512×512 dense matrix-matrix multiply. When the size of a matrix is $N \times N$, the matrix requires $O(N^3)$ floating point operations and $O(N^2)$ words of data. So the bigger the size of the matrix, the higher the computation to communication ratio of the matrix-matrix multiply. The computation is blocked into three levels: matrix, sub-matrix, and inner-most matrix levels to take advantage of the high computation to communication ratio of the application using bandwidth hierarchy of LRFs, SRF and DRAM.

Figure 3.6 shows how 32×32 matrices are partitioned for multiplication utilizing bandwidth hierarchy of stream processors. Input matrices A and B are stored in DRAM and partitioned into 8×8 sub-matrices (a). The first row of A ($= A_0 + A_1 + A_2 + A_3$) and the second column of B ($= B_0 + B_1 + B_2 + B_3$) are multiplied producing a sub-matrix C_0 (b). Sub-matrices for the multiplication are gathered into the SRF using indexed streams, e.g. A_1 and B_1 are gathered to produce partial sum of C_0 (c). These sub-matrices are partitioned into 2×2 inner-most matrices, each row of sub-matrix A_1 belongs to each cluster, and each column of B_1 is distributed across all the clusters through inter-cluster switch (d). In the middle of computing C_0' called C_{010} (e), all the elements of the inner-most matrices A_{112} and B_{120} are loaded into LRFs of cluster 1 through in-lane indexing [JEAD04], multiplied, and stored back to the SRF (f).

The following unstructured mesh and graph applications have irregular structures where nodes have a possibly variable number of neighbors, ending up with irregular control and memory access patterns. These applications can be represented using a framework [Ere06] whose parameter space includes mesh connectivity degree and variability, arithmetic intensity, and runtime properties. Because the SRF and memory have separate address space, the node and neighbor information are *localized* into the SRF from memory in the following possible ways: values for nodes and neighbors are copied to the SRF without duplicate removal, or pointers for nodes and neighbors are loaded directing the values copied in the

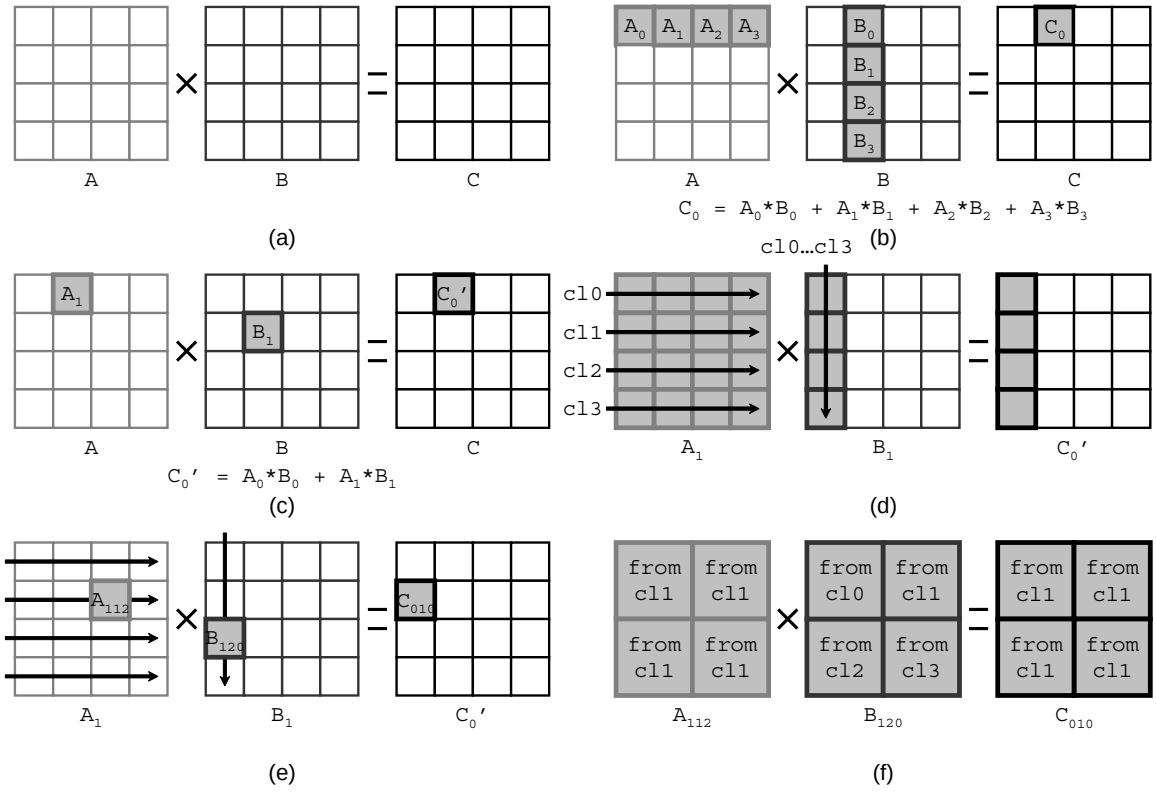


Figure 3.6: Hierarchical data partitioning of matrices — (a) input matrices A and B with size 32×32 are partitioned into 8×8 sub-matrices, (b) first row of A ($= A_0 + A_1 + A_2 + A_3$) and second column of B ($= B_0 + B_1 + B_2 + B_3$) are multiplied producing a sub-matrix C_0, (c) in the middle of computing C_0 called C_0', A_1 and B_1 are multiplied and accumulated, (d) each sub-matrix is partitioned into 2×2 inner-most matrices, each row of sub-matrix A_1 belongs to each cluster, and each column of sub-matrix B_1 is distributed into all the clusters, (e) in the middle of computing C_0' called C_{010}, (f) multiplication of inner-most matrices A_{112} and B_{120} is performed in cluster 1.

SRF with duplicate removal. The latter method can remove duplicates either within a lane of the SRF or across the lanes that a cluster can reach using the inter-cluster switch. The SIMD nature of the stream processors forces computation to be parallelized by sorting nodes based on their connectivity degree, padding dummy neighbors into nodes, or using conditional accesses. These parallelization schemes are combined with all the localization methods letting each application have a multiple number of implementations. Among them, the algorithm with highest performance is selected for evaluation.

MOLE simulates molecular dynamics of a $4,114$ molecule water system [EAG+04]. It solves Newton's equations of motion to update the location of each molecule surrounded by neighbor molecules for a single time-step. An efficient cut-off by distance approximation is used to determine the neighbors interacting with the center molecules. As a result, water molecules become both nodes and neighbors while edges in the graph mean interaction.

FEM is a 3D blast wave solver using a Discrete Galerkin finite element method for systems of nonlinear conservation laws using magnetohydrodynamics flow equations operating on a $9,664$ element mesh [Ere06]. The computation is composed of a face loop processing faces and the elements they share, and an element loop processing elements and their faces. During the face loop, faces are vertices or nodes of the graph and elements are edges or neighbors. Roles are exchanged during the element loop.

CDP calculates finite volume large eddy flow simulation (LES) of a $29,096$ element mesh solving the transport advective equation [Ere06]. The second-order WENO (Weighted Essentially Non-Oscillatory) component of the solver is used for evaluation containing two main loops, one over control volumes and the other over faces. During the control volume loop, control volumes become nodes while faces become neighbors, and vice versa during the face loop.

Table 3.1 shows the summary of computation, memory, and control aspects of media and scientific applications. 3 out of 4 multimedia applications and 4 out of 6 scientific applications have regular control and memory access patterns. Note that FEM has regular control within a kernel but includes irregular constructs in the stream-level program. All the scientific applications perform floating-point operations while some of the media applications like DEPTH and MPEG perform integer operations. The ratio of computation to

Applications	Regularity	Operation type	Comp-to-comm ratio	Scatter-add
DEPTH	regular	integer	high	unused
MPEG	regular	integer	high	unused
QRD	regular	floating-point	high	unused
RTSL	irregular	int + fp	high	unused
CONV2D	regular	floating-point	high	unused
FFT3D	regular	floating-point	middle	unused
MATMUL	regular	floating-point	high	unused
MOLE	irregular	floating-point	middle	used
FEM	(ir)regular	floating-point	high	used
CDP	irregular	floating-point	low	used

Table 3.1: Summary of computation, memory, and control aspects of multimedia and scientific applications

communication is different from application to application, and Table 3.3 contains quantitative values measured from a single chip stream processor. Scatter-add operation is used in MOLE, FEM, and CDP.

3.2 Experimental Setup

The applications in this thesis are written using a two-level programming model: kernels, which perform the computation referring to data resident in the SRF, are written in KernelC and are compiled by an aggressive VLIW kernel scheduler [MDR+00], and stream level programs, which control kernel invocation and bulk stream memory transfers, are coded using one of two kinds of APIs – StreamC or macrocode. A stream compiler [DDM06] takes StreamC code and generates stream instructions and generic C++ codes executed in the scalar control processor. Then the stream dispatcher feeds a stream instruction with dependency information when the stream core is ready to take one. A runtime scheduler running in the scalar core takes the application using macrocode API and interprets API functions – performing dependency analysis and turning them into one or more stream

Parameter	Multimedia applications	Scientific applications
Operating frequency	1 GHz	
Number of instruction sequencer per chip	1	
Number of clusters per instruction sequencer	8	16
Number of ALUs per cluster	4	
Number of memory channels	4	8
Peak DRAM bandwidth	4 GW/s	8 GW/s
DRAM type	XDR DRAM	
Number of scatter-add units per channel	1	
SRF bandwidth	32 GW/s	64 GW/s
SRF size	128 KWords	
Stream cache size	32 KWords	

Table 3.2: Machine parameters

instructions – then dispatching these stream instructions to the memory system and instruction sequencers.

We use the Imagine and Merrimac cycle accurate simulators to get the performance of the multimedia and scientific applications. The baseline configurations of the simulated Imagine and Merrimac are detailed in Table 3.2, and are used for all experiments unless noted otherwise.

3.3 Performance Analysis

Table 3.3 shows the performance summary of multimedia (first 4 rows) and scientific (remaining rows) applications in a single chip stream processor using the baseline configurations reported in Table 3.2.

Under the baseline configurations, applications are compute limited, memory limited, or balanced between the two. Many of the tested applications have their kernels busy over 90% of the execution cycles except RTSL and CDP. In RTSL, stream memory and kernel operations are not perfectly overlapped because the length of the streams out of a kernel

Applications	Computation		Memory BW		LRF/SRF/MEM	Kernel busy
DEPTH	30.8	GOPS	0.33	GW/s	638/11/1	97.7%
MPEG	30.4	GOPS	0.16	GW/s	768/9/1	97.0%
QRD	25.7	GFLOPS	0.12	GW/s	1136/22/1	95.4%
RTSL	10.5	GOPS	0.26	GW/s	369/13/1	87.2%
MAX	64	GFLOPS	4	GW/s	N/A	100%
CONV2D	78.6	GFLOPS	3.56	GW/s	84/6/1	99.4%
FFT3D	47.4	GFLOPS	5.44	GW/s	43/5/1	92.4%
MATMUL	117.3	GFLOPS	1.95	GW/s	178/11/1	99.8%
MOLE	45.9	GFLOPS	4.43	GW/s	67/3/1	90.9%
FEM	37.5	GFLOPS	2.84	GW/s	92/6/1	95.1%
CDP	7.38	GFLOPS	4.73	GW/s	12/2/1	32.8%
MAX	128	GFLOPS	8	GW/s	N/A	100%

Table 3.3: Performance summary of multimedia and scientific applications

invoked is determined at runtime requiring scalar core interaction for the subsequent stream operations to effectively compose dependencies. CDP is limited by the memory system. It issues irregular access patterns to memory with little spatial locality and performs only a small number of arithmetic operations for every word transferred from memory, which is evidenced by high memory bandwidth and low LRF-to-MEM ratio.

Multimedia applications, whose performance is less than half of the peak, are mainly limited by insufficient ILP, short stream effects, and transient SRF bandwidth shortages, which are studied in detail at [ADK+04]. Among the scientific applications, both CONV2D and MATMUL are regular control applications that are compute bound and achieve a high fraction of the 128 GFLOPS peak performance, at 78.6 and 117.3 GFLOPS respectively. FFT3D is also characterized by regular control and regular access patterns but places higher demands on the memory system. During the third FFT stage of computing the Z dimension of the data, the algorithm generates long strides (2^{14} words) that limit the throughput of DRAM and reduce the overall application performance, which is a topic of Chapter 4. FEM has regular control within a kernel but includes irregular constructs in the stream-level program. This irregularity limits the degree to which memory transfer latency can

be tolerated with software control. FEM also presents irregular data access patterns while processing the finite element mesh structure. This irregular access limits the performance of the memory system and leads to bursty throughput, but the application has high arithmetic intensity and achieves over half of peak performance. MOLE has highly irregular control and data access as well as a numerically complex kernel with many divide and square root operations. It achieves a significant fraction of peak performance at 45.9 GFLOPS.

Both classes of applications exhibit bandwidth hierarchy of LRFs, SRF and memory well, while multimedia applications use memory less than scientific applications per LRF access. Also the SRF is less utilized for scientific applications.

Chapter 4

Memory System Design Space

Modern VLSI technology enables a single processor chip to take 100s of ALUs demanding high off-chip memory bandwidth. It also enables a single DRAM chip to have giga-bits of data. Improvement in signaling technology and I/O pin integrity provides 100s of Gb/s bandwidth between two components exemplified by XDR DRAM interface [ELP05].

In stream processors, this supply and demand for aggregate memory bandwidth are met by building memory systems using multiple address-interleaved *memory channels* and implementing each channel using high-bandwidth DRAM components [AED06, ATI05]. Such memory systems, however, are very sensitive to access patterns. This sensitivity is compounded by the fact that the latency of modern DRAMs has been improving very slowly. Therefore, to achieve high performance on a streaming memory system, an access pattern must have sufficient parallelism to saturate the memory bandwidth while tolerating memory latency – i.e., the number of outstanding memory references must equal at least the product of bandwidth and latency [Lit61]. Additionally, to reduce sensitivity and maintain near-peak throughput, the access pattern must also exhibit locality to minimize activate/precharge cycles, avoid bank conflicts, and minimize read-write turnaround penalties.

To achieve the required levels of concurrency, streaming memory system designs have turned to the inherent parallelism of memory references generated in the granularity of *streams* or *threads*, each containing a large number of individual memory accesses. A

*This chapter is based on [AED06] –

streaming memory system exploits this parallelism by generating multiple references per cycle within a stream, by generating concurrent accesses across streams, or both. Accesses from the same stream tend to display locality, yet this locality may lead to load imbalance between the different channels and lower the effective bandwidth. Load balancing is achieved by interleaving accesses from multiple streams at the expense of locality and potentially reduced performance due to access-pattern sensitivity. Based on current technology trends, our experiments show that this loss of locality is particularly problematic when bank conflicts occur within a single DRAM, and when streams performing reads and writes are interleaved. The reason is the long delays associated with switching between read and write accesses, and between rows within a single bank, on a modern DRAM.

A second memory-system architecture trend is to rely on long cache lines and long DRAM bursts. Long DRAM bursts are used to maintain high bandwidth from a single DRAM in face of rapidly increasing access latencies. Since DRAM commands must be separated by several cycles, long bursts are used to utilize internal DRAM bandwidth. However, this large granularity introduced by the memory system does not always match application access granularity. For example, if an application performs random references, only a fraction of each long burst contains useful data and effective performance is significantly reduced.

A final important trend is the use of memory access scheduling [RDK+00a]. Memory access scheduling strives to increase performance and reduce access-pattern sensitivity by reordering references to enhance locality. We analyze several memory access scheduling policies and their interplay with other high-bandwidth techniques.

We explore the detailed design space of streaming memory systems [AED06] in light of these new trends in modern DRAM architectures and application characteristics. We experiment with several scientific and multimedia benchmarks as well as micro-benchmarks, and show that the most critical performance factors are high read-write turnaround penalties and internal DRAM bank conflicts. A second important aspect is the detrimental effect of increasing burst lengths on applications that display random-access memory reference patterns. To better characterize this trend, we develop an accurate analytical model for the effective random-access bandwidth given DRAM technology parameters and the

burst-length. A sensitivity study of the amount of hardware buffering shows that a reasonable 16-entry channel-buffer achieves near-maximal performance. We also experiment with varying DRAM timing and measure a significant bandwidth drop as latencies are increased when multiple streams are accessed concurrently and when the applications display random-access patterns.

Based on our experimental results, we develop techniques for improving the effective bandwidth of modern memory systems. Our designs are based on the observation that locality and interference are the most critical factors affecting modern DRAM performance. We investigate using a single, wide address generator, which sacrifices load-balancing in favor of locality, and suggest a novel *channel-split* architecture that regains load-balance through additional hardware.

The rest of this chapter is organized as follows: Section 4.1 summarizes the characteristics of modern DRAM architectures; Section 4.2 explores the design-space of streaming memory systems; Section 4.3 details the experimental setup; Section 4.4 presents results for our application benchmarks; Sections 4.5 and 4.6 discuss sensitivity studies and trends; and Section 4.7 presents related work.

4.1 DRAM Architecture

In this section we summarize the pertinent details of modern DRAM architectures. More complete descriptions can be found in related publications [Dav01, Sam06c, ELP05].

Modern DRAMs, such as SDR, DDR, DDR2, GDDR3, RDRAM, and XDR, are composed of a number of memory banks where each bank is a two-dimensional array as shown in Figure 4.1. A location in the DRAM is accessed by an address that consists of bank, row, and column fields. For efficiency, row and column addresses are delivered through a single set of chip pins[1], and read and write data share a bi-directional data path from the pins to the sense amplifiers. Additionally, all the banks within a DRAM share the request and data paths. As a result of this minimalistic hardware, accessing data in a modern DRAM must adhere to the rules and timing constraints detailed below.

[1]There are DRAMs like RLDRAM [Mic06] not sharing row and column addresses, but they are designed for limited applications with higher price target.

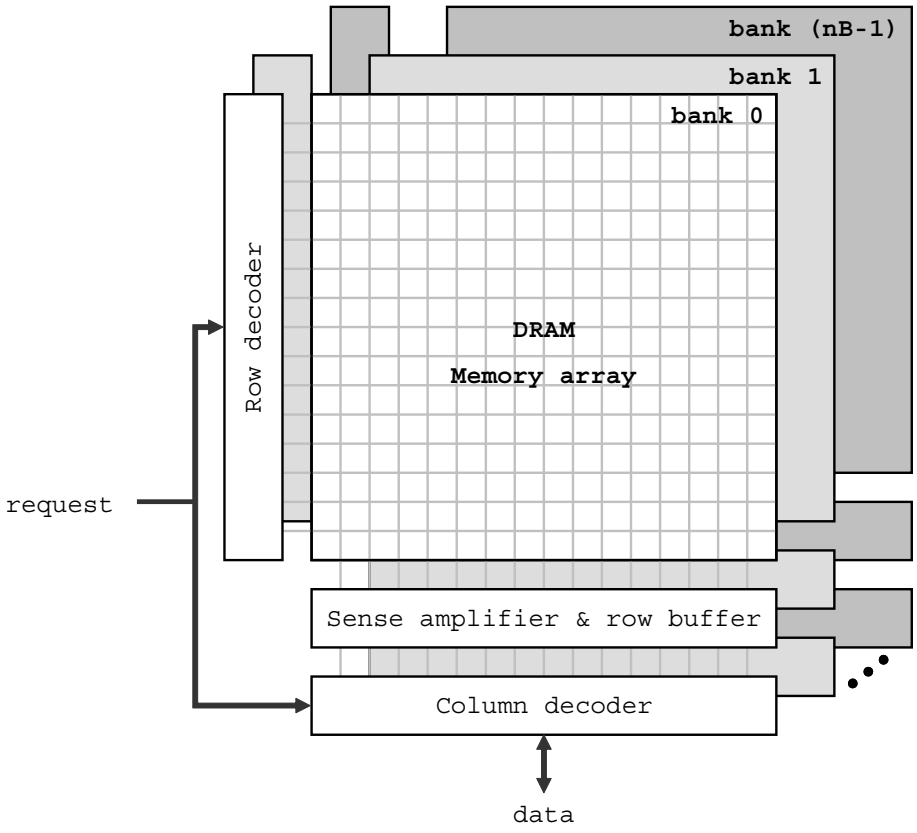


Figure 4.1: A DRAM component is composed of a number of memory banks where each bank is a 2-dimensional array.

Since all the row and column fields of a DRAM address share the same data-path, multiple DRAM commands are required for accessing a particular location (Figure 4.2). First, an entire row within a specific DRAM bank is *activated* using a row-level command (act in Figure 4.2a). Then, after the activation latency has passed, any location within that row can be accessed by issuing a column read or write command (rd in Figure 4.2a and wr in Figure 4.2b). *Internal bank conflicts* occur when two successive requests hit different rows of the same bank. In such an event, the bank must first be *precharged* before the new row can be activated. This second access requires two row-level commands (precharge and activate) followed by one column-level command (column read or write). Hence, requests with internal bank conflicts ((2) in Figure 4.2b) take much longer than the ones within the

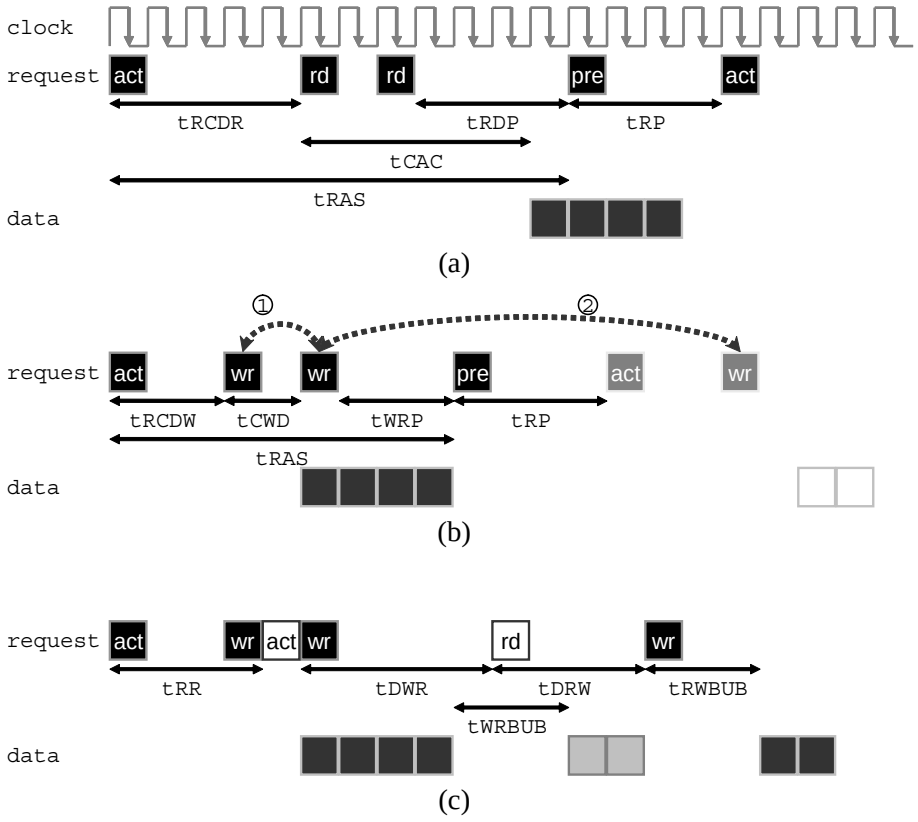


Figure 4.2: Visualization of DRAM timing parameters during DRAM operations — (a) read operations (b) write operations (c) operations involving multiple banks and read-write switches

same row ((1) in Figure 4.2a) or between different banks.

Table 4.1 shows key timing parameters of several DRAM generations, which are illustrated in Figure 4.2. The table shows that as DRAMs evolve, latencies (in cycles) and access granularity are increasing. At the same time, DRAMs are becoming more sensitive to access patterns as follows. Data pin bandwidth has been increasing rapidly while command bandwidth, specified by the number of cycles between two DRAM commands (tCK), has been improving more slowly while the burst time (tCC) in the unit of tCK nearly stays constant. This results in increasing access granularity, since each command must refer to a growing number of DRAM locations. Additionally we observe higher timing sensitivity to the actual access pattern. The time to read or write data on an idle

	Unit	SDRAM	DDR	DDR2	GDDR3	XDR
Pin bandwidth	Mbps	133	400	800	1600	4000
tCK	ns	7.5	5	2.5	1.25	2
$tRCDW/tRCDR$	tCK	3/3	3/3	5/5	8/12	3/7
$tCWD/tCAC$	tCK	0/3	1/3	4/5	6/11	3/7
tRC	tCK	9	11	23	35	20
tRR	tCK	2	2	3	8	4
$tDWR/tDRW$	tCK	1/4	5/4	9/2	14/9	10/9
$tWRBUB/tRWBUB$	tCK	0/1	0/1	0/1	2/2	3/3
Minimum burst (tCC)	tCK	1	1	2	2	2

Table 4.1: DRAM timing parameters based on [Sam05, Sam06a, Sam06b, Sam06c, ELP05]

bank ($tRCDR + tCAC$ for reads and $tRCDW + tCWD$ for writes[2]) rises, requiring the memory system to be more deeply pipelined. Also, the interval between successive row activations to the same bank ($tRC(= tRAS + tRP)$) has been increasing rapidly, making internal bank conflicts more costly. Even with no bank conflicts, the latency of consecutive row activations to different banks (tRR) has been growing, widening the gap between sequential-access and random-access performance. Finally, because internal data paths are shared between reads and writes, a write command followed by a read must be separated by a penalty of $max(tDWR, tWRBUB + tCC + tCWD - tCAC)$ cycles ($max(tDRW, tRWBUB + tCC + tCAC - tCWD)$ cycles for a read followed by a write). This bus turnaround time has also been growing over time, making DRAM performance more sensitive to interleaved reads and writes (we call this *read-write turnaround penalty*). Taken together, these trends show that DRAM performance is now very sensitive to the access pattern applied, leading to varied issues and tradeoffs in streaming memory system design, which we discuss in the next section.

[2] $tRCDR$ and $tRCDW$ are the cycles between row level commands and column level commands, and $tCAC$ and $tCWD$ are the cycles between column commands and the actual data transfers

Symbol	Definition
AG	address generator
CB	channel-buffer
lB	number of words in a DRAM burst
lR	number of consecutive words requested by application (record-length)
MAS	memory access scheduler
MC	memory channel
mC	number of DRAM commands for accessing an inactive row
nAG	number of address generators
nB	number of internal DRAM banks
nCB	number of channel-buffer entries
nMC	number of memory channels
RB	memory reorder buffer
tCK	minimum time between two DRAM commands
$tDWR$	write to read turnaround cycle latency
tRC	activate-to-activate cycle latency within a single internal DRAM bank
tRR	activate-to-activate cycle latency across two internal DRAM banks
W	peak DRAM bandwidth (words per DRAM cycle)

Table 4.2: Streaming memory system nomenclature

4.2 Streaming Memory Systems

Streaming memory systems extract maximum throughput from modern DRAMs by exploiting parallelism and locality. Parallelism is utilized by pipelining memory requests to high-bandwidth DRAM components and also by interleaving accessing over multiple memory channels. Streaming memory systems use *memory access scheduling* [RDK+00a] to enhance locality by reordering memory accesses. This reordering improves performance by reusing open rows and by minimizing internal bank conflicts and read-write turnaround penalties.

Figure 4.3 shows a canonical streaming memory system that moves data between on-chip storage and one or more channels of off-chip DRAM. Table 4.2 summarizes the

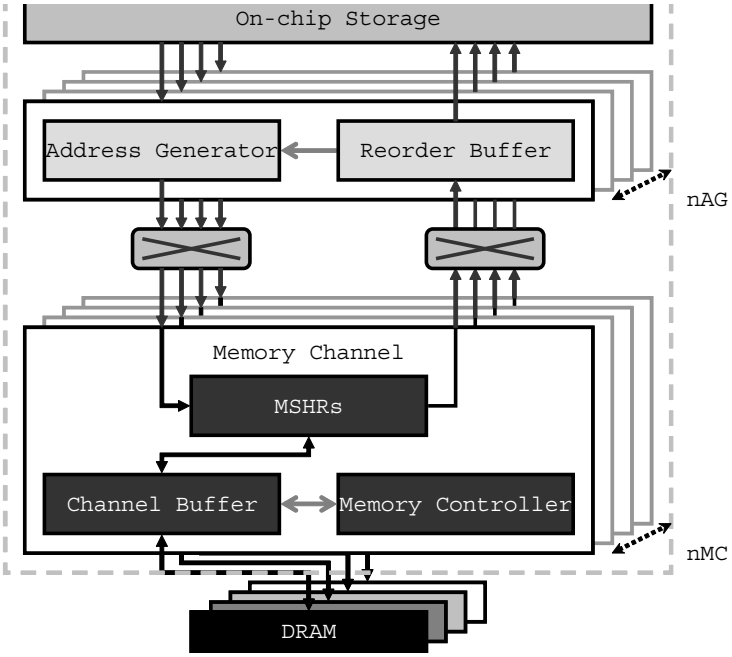


Figure 4.3: Canonical streaming memory system

nomenclature used throughout this thesis. Requests from the processor to perform memory operations are expanded into individual memory addresses by one or more address generators (AGs). To supply adequate parallelism, each AG is capable of generating multiple addresses per cycle, and several AGs may operate in parallel, with each generating accesses from different streams. Each access is routed to the appropriate memory channel (MC) via a crosspoint switch. The memory channel performs the requested accesses and returns read data to a reorder buffer (RB) associated with the originating AG. The optional RB collects and reorders replies from the MCs so they can be presented to the processor and on-chip storage in order.[3]

Within each MC, miss status holding registers (MSHRs) [Kro81] act as a small non-blocking cache, keeping track of in-flight references and performing read and write coalescing of the requested accesses. The width of each MSHR entry is equal to the DRAM burst length. Accesses that cannot be satisfied by the MSHRs are forwarded to the *channel-buffer*

[3]If a RB is not used, the on-chip storage must be capable of handling out of order replies from the memory system.

to be scheduled. In every DRAM command cycle, the memory controller finds a proper DRAM command per pending access based on the status of each internal DRAM bank, selecting one access based on the resource and timing constraints of the external DRAM and the priority of the scheduling policy, sending the appropriate command, and updating the state of the pending accesses. Note that a single memory access may require as many as three DRAM commands to complete (maximum two commands are required if a DRAM supports auto-precharge command which combines read or write command and precharge command into a single command). The number of channel-buffer entries determines the size of the scheduling window used by the memory controller.

In the following paragraphs we examine the design options for each streaming memory system component, and identify critical issues that arise from DRAM technology trends.

4.2.1 Address Generators

The design space of AGs consists of the number of AGs, the number of addresses each AG can generate each cycle (AG width), and the expressiveness allowed in the request streams.

Number of AGs

The number of AGs in a streaming memory system can be varied to trade off inter-stream parallelism for intra-stream parallelism. This tradeoff affects load balance across MCs and the locality of the resulting merged access sequences. Several recently reported streaming memory systems [ADK+04, FAD+05] use multiple AGs that can each generate a small number of references per cycle. This approach exploits parallelism mostly across memory reference streams, and improves load balance across MCs by combining references from multiple streams to even-out an unbalanced pattern from a single stream. Such access interleaving, however, reduces locality leading to decreased throughput as evidenced by the results in Sections 4.4 and Subsection 4.5.1. To avoid the loss of throughput due to such inter-stream interference, we suggest using a single wide AG, that generates many accesses per cycle, in place of multiple narrow AGs. This approach uses intra-stream parallelism in place of inter-stream parallelism to keep the memory pipeline full without sacrificing locality. However, it can suffer from MC load imbalance. In Subsection 4.2.4 we introduce

a *channel-split* memory system organization that maintains both load balance and locality.

AG Width

The AG width (the number of accesses each AG can generate per cycle) determines the aggregate address bandwidth. When using multiple AGs, performance may be increased by providing more total AG bandwidth than the MCs are capable of handling. This allows the memory system to maintain throughput even when the available stream parallelism is low. Providing excess AG bandwidth allows the memory system to adapt — exploiting intra-stream parallelism when it is available and falling back on inter-stream parallelism at other times. There is little advantage to increasing the bandwidth of a single wide AG beyond the available MC bandwidth.

AG Functionality

The simplest AGs handle only unit-stride access streams. To handle a wider range of applications, more sophisticated AGs handle both *strided* and *indexed* streams where each request is a record not just a single word.

4.2.2 Memory Access Scheduling

Memory access scheduling is used to order the DRAM commands associated with pending memory accesses to enhance locality and optimize throughput. We briefly explain MAS scheduling policies below and a more detailed description is available in [RDK+00a].

The `inorder` policy makes no attempt to increase locality, issuing necessary row and column commands to process pending accesses in the order they arrive. When the access pattern causes interference in DRAM, this policy results in very poor performance.

The `inorderla` policy issues column commands in order the associated accesses are arrived but looks ahead and issues row commands early, when they do not interfere with active rows or earlier pending column commands. This policy requires simple hardware and noticeably improves performance as shown in Subsection 4.5.3.

With the `firstready` policy, the oldest ready reference that does not violate DRAM timing constraints is issued to the DRAM. This reference may bypass older requests that

are not ready to issue due to interference.

The `opcol`, `oprow`, `clcol`, and `clrow` policies all perform out-of-order scheduling to improve DRAM command locality. *Open* policies (`op*`) issue a row precharge command when there are no pending accesses to the row and there is at least one pending access to a different row in the same bank. *Closed* policies (`cl*`), on the other hand, precharge a row when the last available reference to that row is performed even if no other row within the bank is requested. Closed policies can take advantage of DRAM *auto-precharge* commands, which combine a precharge with a column read or write. When all else is equal, `*col` policies choose a column command, while `*row` give priority to row commands.

In this thesis we show that while reordering references can lead to significantly improved DRAM throughput for indexed accesses, the specific scheduling policy chosen for reordering has only a second-order affect once both row and column commands are allowed to be reordered.

In addition to the scheduling policy, the channel-buffer size also affects performance. The deeper the buffer the greater the opportunity for the scheduler to discover and exploit locality. This is particularly important in the case of indexed reference patterns where successive references contain little spatial locality.

4.2.3 Memory Channels

The design space of the memory channels includes the total number of MCs, the method of interleaving accesses across MCs, the number of MSHRs, and the channel-buffer depth.

Number of MCs

The number of MCs and the with of each MC set the peak memory bandwidth. More and wider MCs provide more throughput at the expense of higher system cost. However, as mentioned above, care must be taken to ensure that the memory requests can be balanced among the MCs.

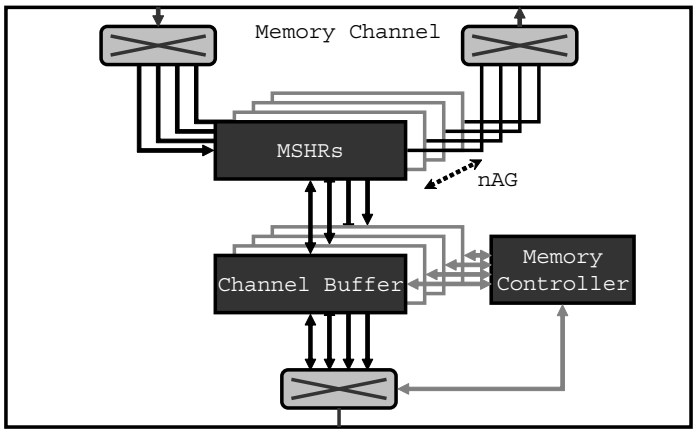


Figure 4.4: *channel-split* configuration

Address Interleaving Method

With multiple MCs, an interleaving policy is used to perform a mapping from a memory address to a MC. With direct interleaving, the MC is selected based on the low bits of the memory address. This simplistic distribution scheme can lead to load imbalance between channels when stride width is multiple of the number of channel buffers, and transient load imbalance when record length is large (Subsection 4.5.5). Pseudo-random interleaving, suggested in [Rau91], alleviates these problems by basing the mapping on a hash of multiple address bits. In either case, the interleaving is done at the granularity of a DRAM burst.

Internal MC Buffers

The channel-buffers and MSHRs must be deep enough to allow the scheduler to effectively deal with indexed accesses that display little inherent locality. We explore the performance sensitivity to this parameter in Subsection 4.5.4.

4.2.4 Channel-Split Configuration

We introduce a *channel-split* memory channel organization, shown in Figure 4.4, to exploit inter-stream parallelism to load balance across multiple MCs without reducing locality due to interference. In this organization, each AG has its own set of MSHRs and channel buffer

entries in each MC. Dividing the resources by AG enables the scheduler to issue references from a single AG to maintain locality as long as the preferred AG has references to issue. If, due to load imbalance, the preferred AG has no references for the MC, the scheduler can select a reference from another AG to balance load. The channel-split organization improves performance, at the cost of increased channel-buffer entries and MSHRs. We discuss this further and show sensitivity results in Subsection 4.5.4.

This arrangement works well to avoid read-write turnaround penalties and internal bank conflicts. For example, if one stream is performing read operations and a second stream is performing write operations, the scheduler will prefer scheduling reads from the first stream as long as they are available. The scheduler will only switch to performing writes when no further reads are available in the channel buffer. Note that with resources shared by all AGs, the second stream would eventually occupy all available buffers and the scheduler would be forced to issue a write, even if many reads were still available.

4.3 Experimental setup

To evaluate the streaming memory system design space and quantify the effects of DRAM technology trends, we measure the performance of several multimedia and scientific applications as well as targeted micro-benchmarks.

4.3.1 Test Applications

We use six applications from the multimedia and scientific domains introduced in Chapter 3 — DEPTH, MPEG, RTSL, MOLE, FEM, and QRD. Here MOLE and FEM are slightly modified. In MOLE, hardware scatter-add is not used, and molecules are allocated in SRF without duplicate removal and with dummy values padded to be processed in SIMD fashion. FEM takes 2-dimensional data and solves Euler equation.

Our micro-benchmarks are designed to stress particular aspects of memory system performance. Benchmarks starting with a AxB format have strided accesses where the record length is A and the stride width is B. Benchmarks with a rA or crA prefix have indexed accesses where the index range is either 4M words (rA), or constrained to 64K words (crA),

Parameter	
Number of clusters	8
Operating frequency	1 GHz
Number of words in a DRAM burst (lB)	4
Peak DRAM bandwidth	4 GW/s
Number of words per DRAM row	1,024
Number of internal DRAM banks (nB)	8
Number of memory channels (nMC)	4
Number of addresses an AG can generate per cycle	4
Number of channel-buffer entries (nCB)	16
nCB for channel-split configurations	16 per AG
Number of words an MC can process per cycle	1
Memory access scheduling policy	`opcol`
DRAM timing parameters (Table 4.1)	XDR
Peak DRAM bandwidth(W)	2
Number of DRAM commands for accessing an inactive row (mC)	3

Table 4.3: Baseline machine parameters

and A denotes the record length. Postfixes of the micro-benchmarks stand for the type of memory accesses and if consecutive streams can cause conflicts in DRAM when processed concurrently. Benchmarks named with rd contain only stream loads while rw benchmarks have stream loads and stores interleaved such that DRAM read-write turnaround penalties can be observed.[4] Benchmarks containing cf have stream loads and stores interleaved and these consecutive access streams hit different rows in the same DRAM bank to expose internal bank conflicts. Each micro-benchmark contains 8 access streams, and each stream accesses 4,096 words, except for the 48x48rd benchmark, which accesses 3,840 words per stream.

[4]Note that stream, not record or word, loads and stores are interleaved. Each stream still contains either all loads or all stores.

Appli-cation	Average strided record size (W)	Average strided stream length (W)	Average strided stride/ record	Average indexed (W) record size	Average indexed (W) stream length	Average indexed (W) index range	Strided access	Read access
DEPTH	1.96	1,802	1.95	1	1,107	1,180	46.6%	63.0%
MPEG	1	1,515	1	1	1,280	2,309	90.1%	70.2%
RTSL	4	1,170	4	1	264	216,494	65.1%	83.5%
MOLE	1	480	1	9	3,252	7,190	9.9%	99.5%
FEM	12.4	1,896	12.4	24	3,853	203,342	48.8%	74.0%
QRD	115	1,053	350	N/A	N/A	N/A	100%	69.0%

Table 4.4: Application characteristics

4.3.2 Simulated Machine Parameters

Simulations were performed under the environment explained in Section 3.2, however the parameters for multimedia applications in Table 3.2 are used regardless of the application domains. The baseline configuration of the simulated stream processor and its memory system is detailed in Table 4.3, and is used for all experiments unless noted otherwise.

In order to study DRAM trends and design space parameters in isolation of specific arithmetic execution characteristics, we treated all execution kernels as single-cycle operations. The memory access patterns and data dependencies in stream processors do not depend on operation timing, which allowed us to obey all dependencies and maintain memory ordering in our applications. We chose to minimize arithmetic execution time in order to amplify the effect of memory system performance trends. We demonstrate the amplification effect on memory system trends and verify the validity of our zero-cost computation evaluation methodology by also presenting results with realistic arithmetic timing.

4.4 Application Results

Figure 4.5 shows the throughput of the six applications on five simulated memory system configurations (five bars per graph associated with left axis). With four MCs operating at 1 GW/s each, peak bandwidth is 4 GW/s. The figure also shows the fraction of memory

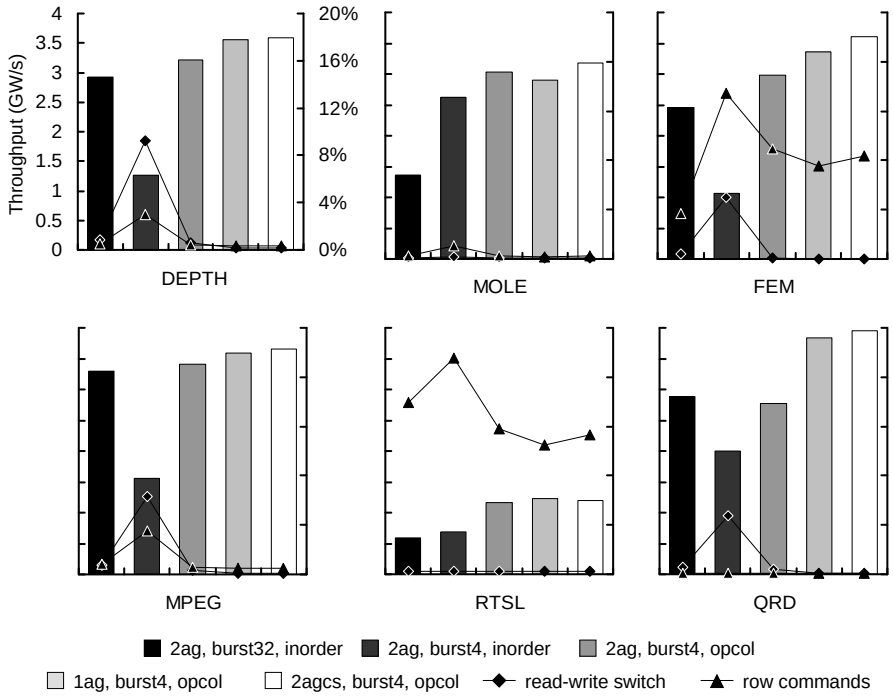


Figure 4.5: Memory system performance for representative configurations. Bars show throughput against left axis. Points show fraction of references of a particular type against the right axis.

references that result in read-write switches and row commands (triangles and diamonds associated with right axis). To explain the results of Figure 4.5 we refer to Table 4.4 which summarizes the memory access characteristics of the six applications.

Figure 4.5 shows that, except for the RTSL program, overall bandwidth is quite high — between 3.2 and 3.9 GW/s, 80% to 96% of the peak bandwidth. Table 4.4 shows that 35% of RTSL references are single-word indexed references over a large (200K word) address range. These essentially random references have little locality and hence result in low bandwidth — 1.2 GW/s. This lack of locality is also reflected in the high percentage of row commands for RTSL. The figure also shows that when the fraction of read-write switches and row commands increases, performance drops. Each read-write switch makes the DRAM idle during the turn-around interval, and row commands often result in bank conflicts as explained in Section 4.1.

The left-most bar in each graph shows the performance of a memory system with a

long (32-word) DRAM burst length, 2 AGs, and in-order scheduling. DEPTH, MPEG, FEM, and QRD perform quite well — more than 60% of the peak bandwidth. This is because these applications have small stride or large record size on memory accesses exhibiting high spatial locality, which makes large of the long burst useful. The large lB gives poor performance on RTSL and MOLE because they make many indexed accesses to short records over a wide range. Hence, very little of the 32-word burst is used to transfer useful data. The next bar shows that reducing lB to 4 words, while improving performance on MOLE, reduces performance on most of the other applications. The smaller lB increases the number of row and column commands, since each command transfers less data, increasing interference and contention for command bandwidth. The center bar shows that this interference can be greatly reduced by using memory access scheduling (`opcol` policy) to enhance locality. This reordering significantly reduces the number of read-write switches and row commands, reducing idle time and bank conflicts respectively.

Even with `opcol` scheduling, DRAM bandwidth is limited by interference between the interleaved accesses from two AGs. The fourth bar shows how this interference can be reduced by using a single, wide AG. This reduction in read-write switches is most pronounced in QRD, which has a high fraction of writes (30%) and a large lR. This combination leads to competition over scarce buffering resources, limiting scheduler effectiveness and forcing read-write switches. Throughput can be improved by providing more buffering as discussed in Subsection 4.5.4.

MOLE has a small amount of MC load imbalance in each of its access threads, which results in a 4.2% reduction in performance when changing to a single AG. The right-most bar shows how the channel-split organization overcomes this load imbalance (second to the right-most bar) without sacrificing locality (center bar) by dedicating MC resources to each AG. The channel-split configuration has the highest throughput for all of the applications except for RTSL for which it has 2.0% less throughput than the single-AG case. This anomaly is a result of the large number of relatively short stream-level memory operations of RTSL as discussed in Subsection 4.5.4.

Figure 4.6 presents memory system performance results with realistic arithmetic timing using eight ALUs per clusters and shows the same basic trends as with zero-cost arithmetic,

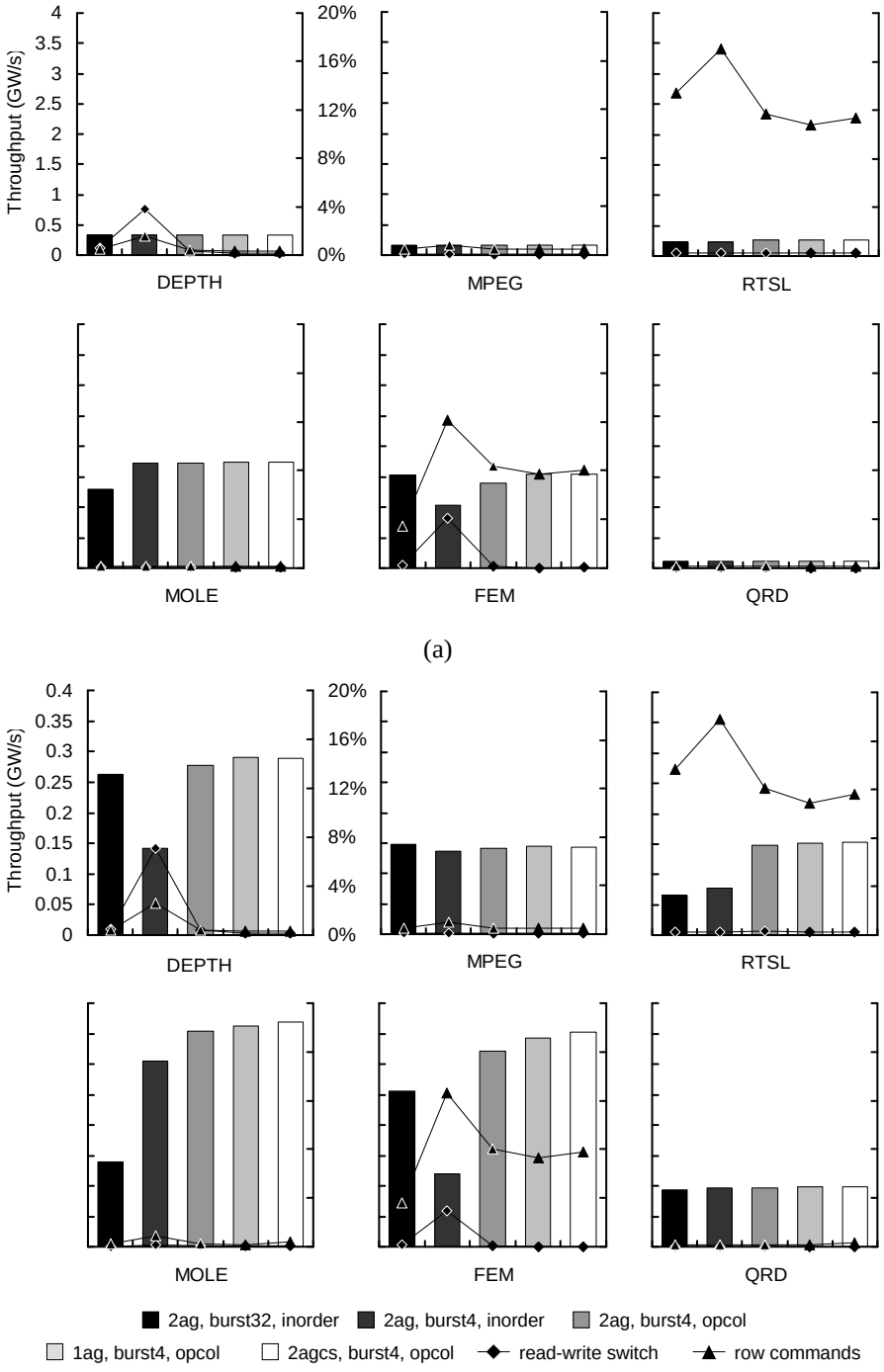


Figure 4.6: Memory system performance for representative configurations with realistic arithmetic timing and (a) 4 GW/s peak aggregate DRAM bandwidth, and (b) 0.4 GW/s peak DRAM bandwidth. Bars show throughput against left axis. Points show fraction of references of a particular type against the right axis.

although much less pronounced. When using the baseline DRAM configuration with realistic arithmetic, as shown in Figure 4.6a, only MOLE and FEM require a significant portion of the 4 GW/s peak DRAM bandwidth. Therefore, there is virtually no observable performance difference for the other applications as the memory system configurations is varied. With a peak DRAM bandwidth of 0.4 GW/s (Figure 4.6b), which better matches application requirements, the trends demonstrated by DEPTH and RTSL also match the results of zero-cost arithmetic.

4.5 Sensitivity to Memory System Design Parameters

In this section we explore the sensitivity of throughput to streaming memory system design parameters, including the number of AGs, AG bandwidth, MC buffer depth, and scheduling policy. For each experiment, we take the baseline configuration described in Section 4.3 and vary one parameter while holding the others constant.

4.5.1 Number of AGs and Access Interference

Figure 4.7a shows throughput of the micro-benchmarks as the number of AGs is varied, for both conventional and channel-split organizations, while the number of channel-buffers is fixed to 64 per MC.

The channel-split configurations consistently give the highest throughput because they are able to achieve locality without sacrificing load balance. For benchmarks where no locality is present (cr1rd and r1rd), or when no interference exists (1x1rd), the number of AGs does not affect performance. Sequential access patterns with interference (1x1rw, 1x1rdcf, and 1x1rwcf) have poor performance with multiple AGs without channel splitting due to excessive read-write switches and row conflicts. Because of load imbalance, multiple AG configurations outperform the single-AG configuration on 48x48rd and r4rw.

Figure 4.7b shows application throughput as the number of AGs is varied. The results in the figure can be explained by matching the application characteristics (Table 4.4) with the micro-benchmarks. DEPTH and MPEG are dominated by sequential or small range of read and write accesses and therefore behave similarly to 1x1rw, 1x1rdcf, and 1x1rwcf.

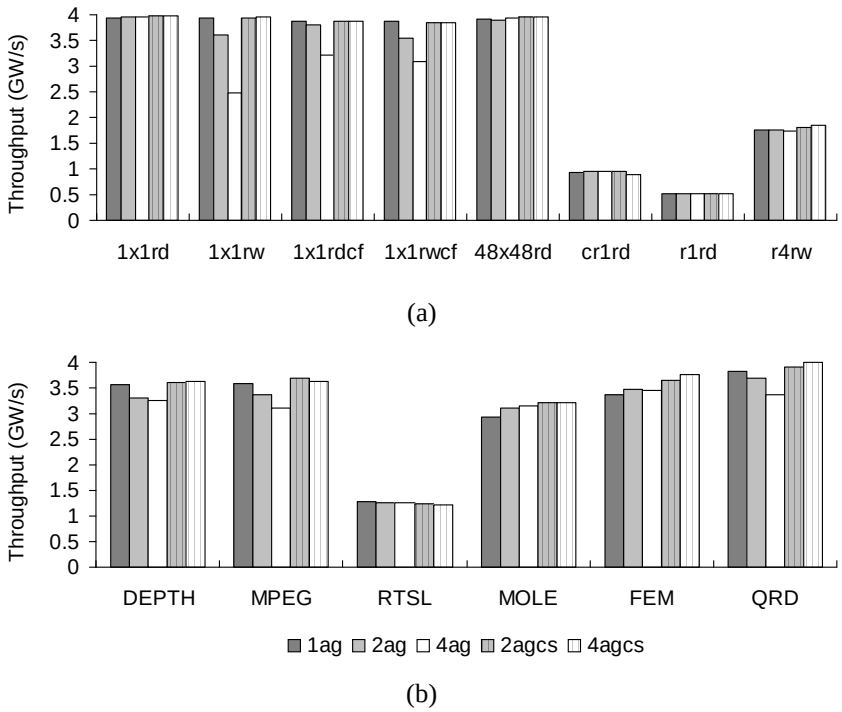


Figure 4.7: Throughput vs. AG configuration of micro-benchmarks (a) and applications (b)

RTSL contains mostly reads with little locality and hence matches the behavior of r1rd. MOLE and FEM display similarities to r4rw as all three are dominated by indexed accesses to multi-word records, however the load imbalance across memory channels in the real applications gives greater advantage to configurations with multiple AGs. QRD appears as a combination of 1x1rwcf and 48x48rd, since it has both large records and conflicting reads and writes characteristics.

The 4agcs configuration has 4% lower throughput than the 2agcs configuration on cr1rd, MPEG, and RTSL partly because of high channel-buffer pressure. We discuss this issue in more detail in Subsection 4.5.4.

4.5.2 Aggregate AG Bandwidth

Figure 4.8 shows throughput for micro-benchmarks and applications as the bandwidth of the AGs is varied and the number of channel-buffers is fixed to 64 per MC.

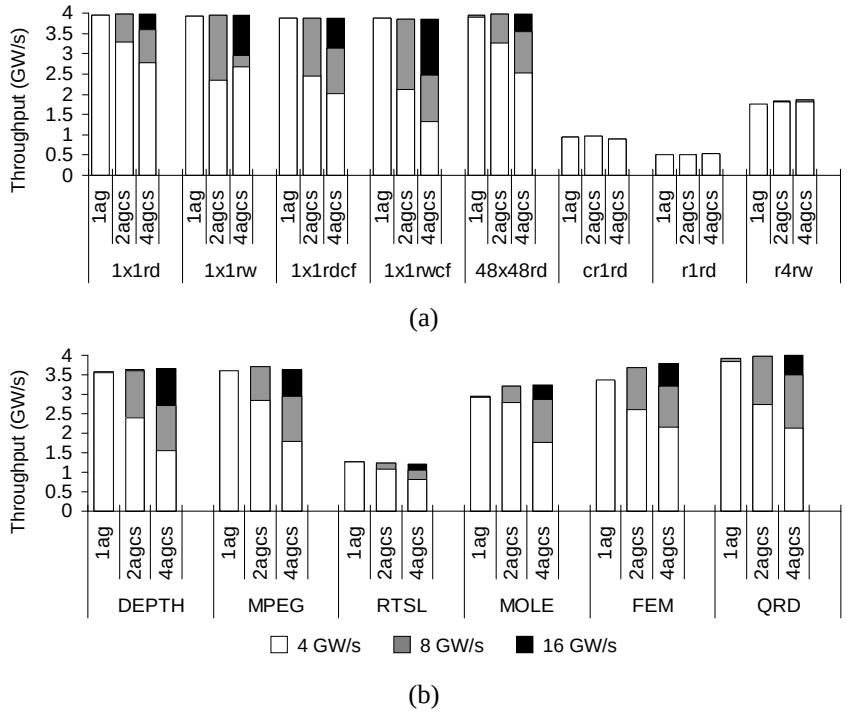


Figure 4.8: Throughput vs. total aggregate AG bandwidth of micro-benchmarks (a) and applications (b)

The figure shows that as the number of AGs is increased, higher aggregate AG bandwidth is needed to get maximum performance. A single AG achieves near-maximum performance when the AG bandwidth matches MC bandwidth (4 W/cycle). Increasing single AG bandwidth beyond this point provides at most a 2% performance advantage. Similarly, increasing the individual AG bandwidth beyond 4 W/cycle (e.g., aggregate bandwidth of 8 W/cycle on 2agcs) has no effect on effective DRAM bandwidth. For the multiple AG configurations, increasing aggregate bandwidth beyond the 4 GW/s needed to saturate the MCs increases performance because it enables the scheduler to enhance locality by issuing all references from a single AG without causing a bottleneck. In benchmarks with little locality, such as cr1rd and r1rd, increasing aggregate AG bandwidth provides little return.

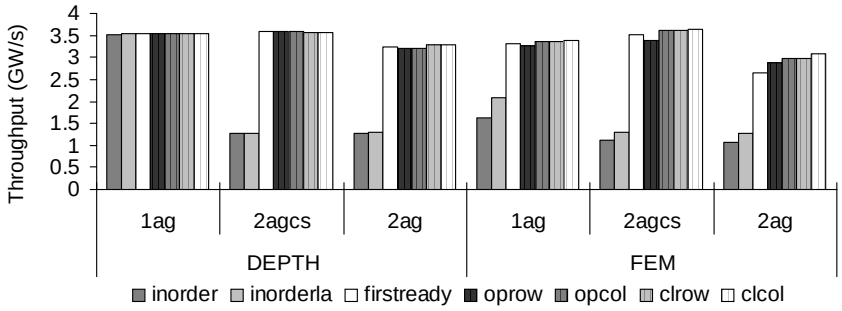


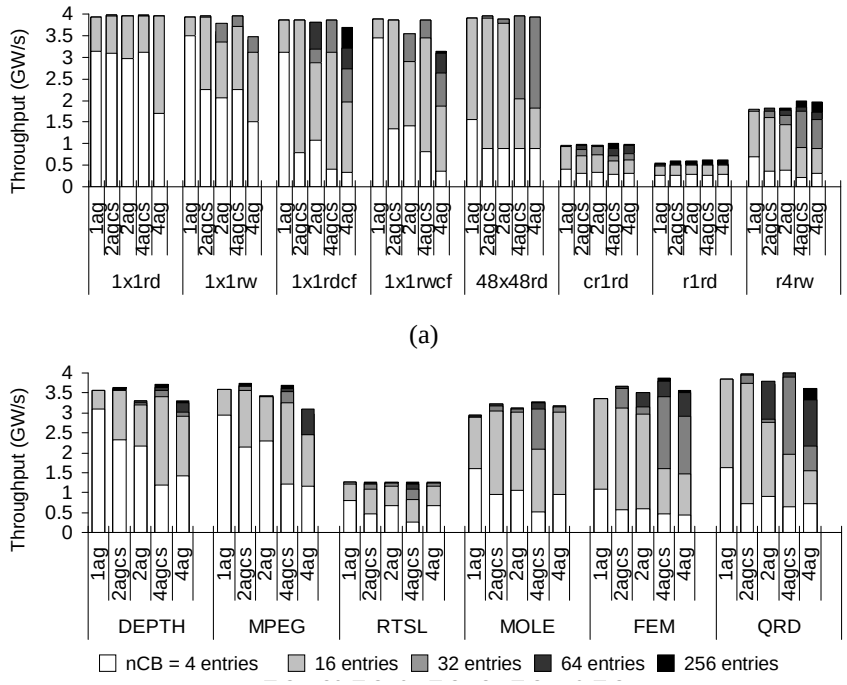
Figure 4.9: Throughput vs. memory access scheduling policy

4.5.3 Memory Access Scheduling

Figure 4.9 shows the sensitivity of throughput to scheduling policy for the DEPTH and FEM applications and three AG configurations. We present just two applications to avoid replication of data. DEPTH is representative of all strided applications and FEM is typical of all indexed applications. For strided applications, the single AG configuration performs well regardless of scheduling policy. With multiple AGs or indexed accesses, column reordering is needed to restore locality and achieve good performance. The throughput is largely insensitive to the exact column reordering policy used. `Inorder` and `inorderla` policies work better with 1 AG configuration because it eliminates almost all the inter-stream interferences.

4.5.4 Memory Channel Buffers

Figure 4.10 shows that the sensitivity of throughput to the size of MC buffers rises as the number of AGs is increased. The single AG configuration reaches near-peak performance with only 16 channel-buffers, while even 64 buffers are not sufficient to achieve peak performance on the 4agcs configuration. Because the multiple AG configurations divide buffers across multiple threads, a single AG, even though it cannot load balance amongst MCs, gives higher performance when the total number of channel-buffers is limited. Because they can more flexibly allocate a scarce resource, non-split configurations outperform the channel-split approach when we limit channel-buffer depth to under 16 entries per MC.



(a)

(b)

Figure 4.10: Throughput vs. total MC buffer depth of micro-benchmarks (a) and applications (b)

When enough buffering is provided, the locality-preserving configurations perform well, and providing load balancing through multiple AGs can improve throughput. The largest performance improvements are seen for DEPTH, FEM, and QRD where 4agcs outperforms 1ag by 4.2%, 15%, and 6.8% respectively. 1ag, however, has a 2% higher throughput than the channel-split configurations on RTSL. This minor difference is the result of the large number of stream-level memory operations of RTSL, combined with the increased latency of each operation caused by the deeper buffering required to support the channel-split mechanism.

Additionally, we observe diminishing returns from increasing channel-buffer depth beyond 16 entries per AG. In 2agcs, for example, increasing nCB from 32 to 64 entries results in at most 3.5% and 9.4% gains in throughput for the applications and micro-benchmarks respectively (RTSL and cr1rd).

Due to the potential higher cost, in terms of area and/or cycle-time pressure, of a large

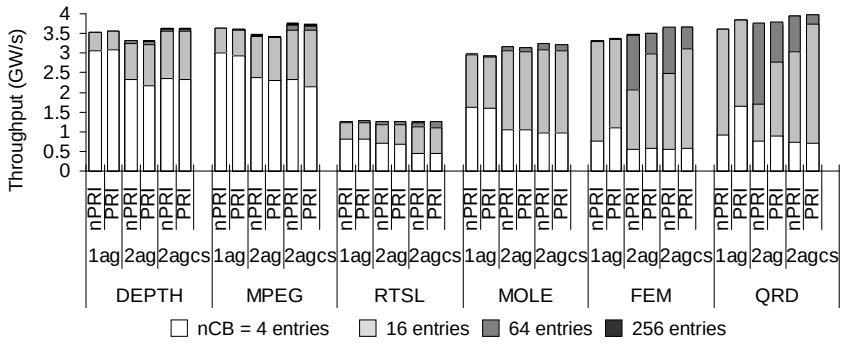


Figure 4.11: Throughput vs. address interleaving method — (n)PRI = (no) Pseudo Random Interleaving

number of channel-buffer entries and high AG width, we advocate the use of the 1ag configuration with 16 channel-buffer entries. If higher performance is desired, 2agcs with a total buffer depth of 32 can reach throughput that is within 6.3% of the maximum throughput of any configuration across all six applications.

4.5.5 Address Interleaving Method

The method used to map addresses to MCs in an interleaved DPMS can have a significant effect on the amount of channel buffering required to achieve optimal performance. When a direct method is used, transient load imbalance between MCs can occur, requiring deeper buffers to prevent stalls. As shown in Figure 4.11, even with $nCB = 16$, the effect is quite small (less than 3.3%) for DEPTH, MPEG, RTSL, and MOLE. FEM and QRD have a greater disparity unless deep channel buffers are used (19.4% with $nCB = 16$ and 1.7% with $nCB = 64$ for QRD). The reason for this sharp difference is the clustered access pattern resulting from the very long records of QRD. The simulated stream processor contains 8 SIMD clusters and each cluster processes an entire record. When the records are large, the generated access pattern is sets of 8 words separated by multiples of the stride. When the low-order address bits directly determine the MC, each group of 8 accesses is mapped to the same MC, leading to a transient imbalance. If the channel buffers are not deep enough to tolerate such transients, the AGs must stall and wait for buffers to be freed as accesses complete. A pseudo-randomly interleaved mapping decorrelates the mapping

from the number of clusters and alleviates the temporary buffer pressure.

4.6 Sensitivity to DRAM Technology Trends

In this section we explore the effects of extending the trends of increasing DRAM latencies and burst lengths. We also examine the implications of limited DRAM command issue bandwidth.

4.6.1 Read-Write Turnaround Latency

As DRAM latencies increase performance becomes even more sensitive to access patterns and locality is even more critical to maintain throughput. Figure 4.12 shows that the locality-conserving 1ag and 2agcs configurations lose only 5.2% throughput as $tdWR$ is increased from 5 to 40 tCK (DEPTH). When locality is sacrificed in the 2ag configuration, performance drops by up to 46% (r4rw). With some cases of MOLE, FEM, and QRD, performance slightly increases as latency is increased in the 2ag configuration. This anomalous behavior is a result of scheduling artifacts when switching between read and write streams. Due to the greedy nature of the scheduling algorithm, increasing $tdWR$ occasionally gives a better schedule.

4.6.2 DRAM Burst Length and Indexed Accesses

Figure 4.13a presents the results of increasing DRAM burst length for several microbenchmarks. Changing lB has no effect on 1x1rd. The 1x1rw benchmark shows that 1ag and 2agcs, which maintain locality even when both reads and writes exist, are also insensitive to the value of lB. The throughput of 2ag increases with burst length (from 2.9 to 3.8 GW/s) because the number of read-write switches is reduced (see also Section 4.4). The performance of 1x40rd drops rapidly with rising lB because the large stride of this benchmark results in only a single usable word from each burst. Thus, the throughput drops by a factor of 2 each time we increase the burst length by a factor of 2. The throughput of 48x48rd also drops as lB grows. This behavior is a result of timing sensitivities in channel buffer and MSHR management. A detailed examination of the behavior shows

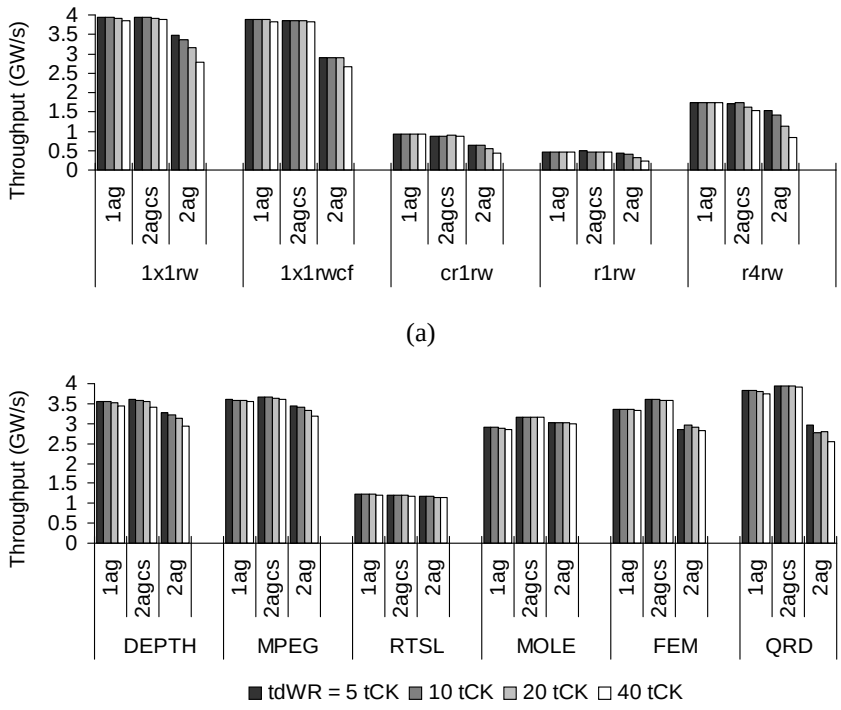
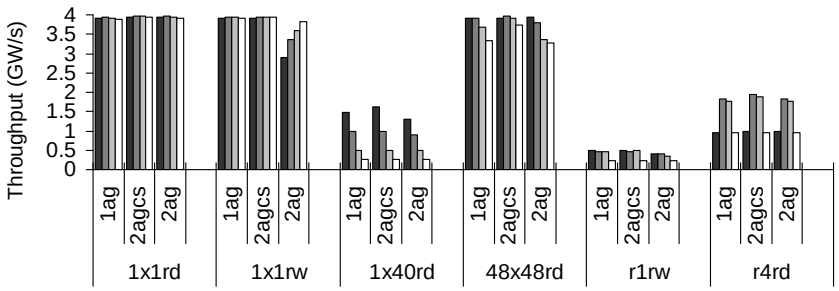


Figure 4.12: Throughput vs. DRAM $tdWR$ of micro-benchmarks (a) and applications (b)

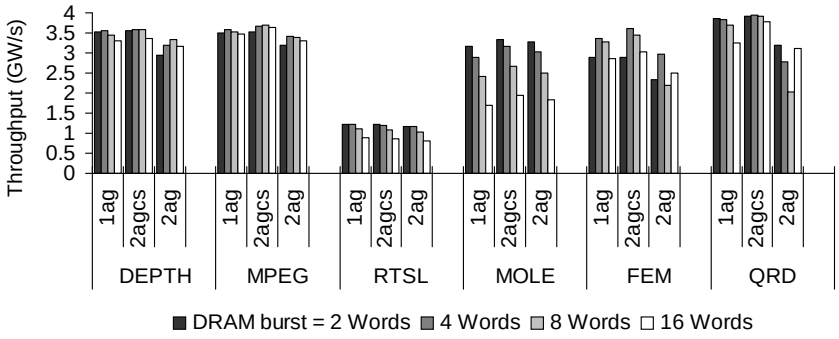
that MSHRs are being deallocated while there are still matching accesses in the channel buffers, leading to excess consumption of DRAM bandwidth and a total drop of 15% in throughput with the 1ag and 2ag configurations. Random access patterns work best with a burst of 4 or 8 words (r1rw and r4rd).

Trends are less clear in the case of the multimedia and scientific applications, which have more complicated access patterns than the micro-benchmarks (Figure 4.13b). Different access patterns work better with different values of lB, yet we observe a sweet spot with a DRAM burst of 4 words. Both the 1ag and the 2agcs configuration display near optimal performance with $lB = 4$, with the exception of MOLE where a burst of 2 words achieves 4.7% higher throughput with the 2agcs configuration.

To better understand the effect of changing the DRAM burst length, we develop an analytical model for the expected throughput based on DRAM technology parameters. Please refer to Table 4.2 in Section 4.2 for a description of the symbols used below.



(a)



(b)

Figure 4.13: Throughput vs. DRAM burst length of micro-benchmarks (a) and applications (b)

In order to keep the formula for effective random indexed bandwidth concise we make the following assumptions:

- All accesses are indexed, with the indices uniformly distributed over the entire DRAM address space.
- All accesses are to a fixed record length (lR).
- A record can be accessed by issuing at most one column command to each memory channel ($lR \leq lB \cdot nMC$).

If the index space is sufficiently large, these assumptions result in every record access requiring a new row command to an inactive bank. This leads to the following formula for the effective DRAM bandwidth[5]:

[5] $\lceil \frac{lR}{lB} \rceil$ is the number of bursts required to access a record

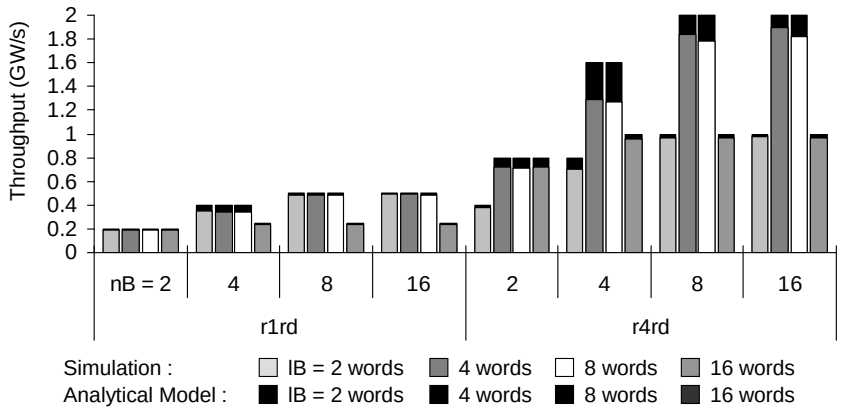


Figure 4.14: Analytical model and measured throughput of random indexed access patterns

$$BW_{rand} \simeq BW_{peak} \frac{lR}{\lceil \frac{lR}{lB} \rceil lB_{effective}} \tag{4.1}$$

The effective burst length ($lB_{effective}$) is given by:

$$lB_{effective} = W \cdot \max\left(\frac{lB}{W}, mC, tRR, \frac{tRC}{nB}\right) \tag{4.2}$$

The four arguments of the $\max$ function in Equation (4.2) correspond to four factors that may limit the rate of record accesses. The first term is the actual burst length and remaining arguments all relate to command bandwidth. It takes mC commands to access each record, so at least ($W \cdot mC$) words must be transferred on each access, or command bandwidth is the bottleneck. The third term is due to the maximum bandwidth of accesses to different banks. Finally, when tRC is large compared to nB at most one access can be made each $\frac{tRC}{nB}$ cycles.

We can now write the full equation for random throughput:

$$BW_{rand} \simeq BW_{peak} \frac{lR}{\lceil \frac{lR}{lB} \rceil \cdot W \cdot \max\left(\frac{lB}{W}, mC, tRR, \frac{tRC}{nB}\right)} \tag{4.3}$$

Figure 4.14 compares our analytical model to simulation results for the r1rd and r4rd

micro-benchmarks. We vary nB and lB, holding all other parameters to the baseline described in Section 4.3. In all cases the model matches well with simulated performance and is within 8.2% of the measured effective random bandwidth on average. The relatively large 17% error in the modeled throughput for r4rd with 4 internal DRAM banks is a result of not accurately modeling contention for DRAM command issue. With complex DRAM timing constraints and memory access scheduling, the restriction of issuing only one DRAM command per cycle to each MC can become a temporary bottleneck.

We can see the detrimental effect of limited command issue bandwidth when the number of internal banks is small. With $tRC = 20$ and only 2 or 4 banks, a new access can be made at most once every 10 or 5 cycles, which limits performance. Additionally, when $lB = 2$, two commands are required for each access in the case of r4rd, and again, performance is sacrificed. The problem of long bursts is evident when $lB = 16$ and much of the DRAM bandwidth is squandered transferring unused data.

4.7 Related Work

Traditional vector processors such as the CRAY-1 [Rus78] and the CRAY X-MP [ABHS89] have memory systems with parallel, interleaved memories allowing concurrent memory accesses between multiple vector streams [CS86]. Multiple accesses, especially from multiple streams, often result in memory channel/bank conflicts. Prior work [Soh93, OL85, Rau91] analyzed these bank conflicts and suggested techniques to alleviate performance degradation in the context of strided accesses to SRAM. One of the random interleaving schemes of [Rau91] is implemented in the memory systems evaluated in this paper. Kozyrakis [Koz02] studied memory system design space in embedded vector processors concentrating on memory channel and internal memory bank configurations for embedded DRAM technology. Our work, in contrast, focuses on multi-word strided and indexed accesses with both inter- and intra-stream access reordering to modern DRAMs.

The Stream Memory Controller [HMS+99], the Command Vector Memory System [CEV98], and more advanced Memory Access Scheduling [RDK+00a] studied hardware/software techniques for streaming accesses by exploiting the three-dimensional nature of DRAMs. However, these works did not cover the interaction of concurrent streaming access threads

and the corresponding design choices in depth. Specifically, [RDK+00a] was limited in scope to evaluating a specific choice of DRAM parameters (first generation SDRAM specification) and to a single technique of reordering memory operations to increase throughput — memory access scheduling. This technique is just one of eight different design space parameters explored in this chapter. The more thorough exploration of DRAM property trends, based on the analysis of multiple DRAM families and future projection, lead us to significantly different conclusions than prior work. For example, contrary to [RDK+00a], we conclude that the exact column/row reordering policy has only a second order effect on memory throughput. Additionally, the recent increase in read-write turnaround latencies affects performance as strongly as column/row locality, leading us to recommend a different design point than the one suggested in prior work.

Chapter 5

Scatter-Add in Stream Architectures

Stream processors exploit the underlying locality and parallelism, especially data-level parallelism, of multimedia and scientific computing applications by casting them into stream programs using the stream programming model. While much of the computation of these applications is indeed data parallel, some sections of the code require serialization which significantly limits overall performance.

One commonly used algorithm that exemplifies this is a *histogram* or binning operation. Given a dataset, a histogram is simply the count of how many elements of the dataset map to each bin as shown in the pseudo-code below [1].

```
for i=1..n
  histogram[data[i]] += 1;
```

Histograms are commonly used in signal and image processing applications to perform *equalization* and *active thresholding* for example. The inherent problem with parallelizing the histogram computation is *memory collisions* where multiple computations performed on the dataset must update the same element of the result in memory (Figure 5.1). One conventional way of dealing with this problem is to introduce expensive synchronization. For a canonical parallel architecture composed of a set of processing elements (PEs) sharing

*This chapter is based on [AED05] –

[1] In the example we refer to the case where the dataset is integer and corresponds directly to the histogram bins. In the general case, an arbitrary mapping function can be defined from the values in the dataset onto the integer bins.

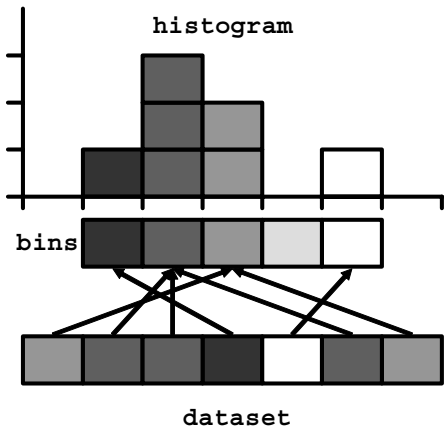


Figure 5.1: Parallel histogram computation leads to memory collision, when multiple elements of the dataset update the same histogram bin.

global memory space, before a PE updates a location in memory which holds a histogram bin it acquires a lock on the location. This ensures that no other PE will interfere with the update and that the result is correct. Once the lock is acquired, the PE updates the value by first reading it and then writing back the result. Finally the lock must be released so that future updates can occur:

```
for i=1..n
{
  j = data[i];
  lock(histogram[j]);
  histogram[j] += 1;
  unlock(histogram[j]);
}
```

This straightforward approach is complicated by the SIMD nature of the stream processors that require very fine-grained synchronization, as no useful work is performed until all PEs (clusters) controlled under an instruction sequencer have acquired and released their lock. To overcome this limitation, parallel software constructs, such as segmented scan[CBZ90], have been developed. However, the hardware *scatter-add* mechanism presented in this chapter provides up to an order of magnitude higher performance in the case of a histogram operation and a 76% speedup on a molecular-dynamics application over a

software-only approach. Moreover, these performance gains are achieved with a minimal increase in the chip area – only 2% of a $10mm \times 10mm$ chip in $90nm$ technology based on a standard-cell design. Scatter-add is a special type of memory operation that performs a data-parallel atomic read-modify-write entirely within the memory system, and is defined in the pseudo-code below. The first version of the operation atomically adds each value of an input data array or stream to the memory location it accesses, while the second version adds a constant to each located referenced:

```
scatterAdd(<T> a[m], int b[n], <T> c[n])
{
  forall i = 1..n
    ATOMIC{a[b[i]] = (a[b[i]] + c[i])};
}

scatterAdd(<T> a[m], int b[n], <T> c)
{
  forall i = 1..n
    ATOMIC{a[b[i]] = (a[b[i]] + c)};
}
```

where `a` is a *stream* (contiguous memory locations) of length `m` and an integral or floating-point type; `b` is an integral stream of length `n` whose elements are in the range $[1..\texttt{m}]$ and defines the mapping of each element of `c` onto `a` (an index stream); `c` is a stream of length `n` and the same type as `a` or a scalar of the same type. Many, and potentially all, the updates can be issued concurrently and the hardware guarantees the atomicity of each update. The scatter-add is essentially a hardware implementation of the *array combining scatter* operation defined in High Performance Fortran (HPF) [HPF93].

Applying the scatter-add to computing a histogram in a data-parallel way is straightforward, where `a` is the histogram value, `b` is the mapping of each data element – simply the data itself, and `c` is simply 1:

```
scatterAdd(histogram, data, 1);
```

Relying on the memory system to atomically perform the addition, the histogram is computed without requiring multiple round-trips to memory for each bin update and without the need for explicit and costly synchronization of the conventional implementation.

Also, the processor's main execution unit can continue running the program, while the sums are being updated in memory using the dedicated scatter-add functional units. While these observations are true for a conventional scalar fetch-and-add, our data parallel scatter-add extends these benefits to stream processors as well as vector and other SIMD processors. This new approach is enabled by the ever-increasing chip gate count, which allows us to add floating point computation capabilities to the on-chip memory system at little cost.

Histogram is a simple illustrative example, but the scatter-add operation can also be used to efficiently express other operators as well. One such important operator is the superposition operator which arises naturally in many physical scientific applications. As explained in [PTVF96], due to the inherent linearity in the physical objects simulated and due to the linearization and simplification of nonlinear problems, superposition is a prevalent operation in scientific codes. Examples include particle-in-cell methods to solve for plasma behavior within the self-consistent electromagnetic field [Wil93], molecular dynamics to simulate the movement of interacting molecules [DWP02], finite element methods [CB02] and linear algebra problems [Saa03].

The remainder of this thesis is organized as follows: we discuss related work in Section 5.1, detail the scatter-add architecture in Section 5.2, and present our evaluation results in Section 5.3.

5.1 Related Work

Previous work related to scatter-add falls into two main categories: software only techniques, and hardware mechanisms related mostly to control code and not for computation.

5.1.1 Software Methods

Implementing scatter-add without hardware support on a stream processor requires the software to handle all of the address collisions. Three common ways in which this can be done are *sorting*, *privatization*, and *coloring*. The first option is to sort the data by its target address and then compute the sum for each address before writing it to memory. Many algorithms exist for data-parallel sort suitable for stream processors such as bitonic-sort

and merge-sort, and the per-address sums can be computed in parallel using a *segmented scan* [CBZ90]. Both the segmented scan and the sort can be performed in $O(n)$ time complexity. While sorting is an $O(n \log n)$ operation in general, we only use the sort to avoid memory collisions, and process the scatter-add data in constant-sized batches. It is also important to note that the complexity constant for the sort is relatively large on stream architectures. The second software option for scatter-add implementation is *privatization* [GHL+02]. In this scheme, the data is iterated over multiple times where each iteration computes the sum for a particular target address. Since the addresses are treated individually and the sums stored in registers, or other named state, memory collisions are avoided. This technique is useful when the range of target addresses is small, and its complexity is $O(mn)$, where m is the size of the target address range. The final software technique relies on *coloring* of the dataset, such that in each color only contains non-colliding elements [CMCA98]. Then each iteration updates the sums in memory for a single color and the total run-time complexity is $O(n)$. The problem is in finding a partition of the dataset that satisfies the coloring constraint, which often has to be done off-line, and the fact that in the worst case a large number of necessary colors will yield a serial schedule of memory updates.

Software algorithms have also been developed for scatter-add implementation in the context of coarse-grained multi-processor systems and HPF's array combining scatter primitive [BR98]. One such obvious technique is to equally partition the data across multiple processors, and perform a global reduction once the local computations are complete.

5.1.2 Hardware Methods

The NYU Ultracomputer [GGK+82] suggested a hardware fetch-and-add mechanism for atomically updating a memory location based on multiple concurrent requests from different processors. An integer-only adder was placed in each network switch which also served as a gateway to the distributed shared memory. While fetch-and-add could be used to perform general integer operations [BCG+98], its main purpose was to provide an efficient mechanism for implementing various synchronization primitives. For this reason it has been implemented in a variety of ways and is a standard hardware primitive in large

scale multi-processor systems [KS93, Sco96, LL97, HT93, WRRF]. Our hardware scatter-add is a stream or vector version of the simple fetch-and-add, which supports floating-point operations and is specifically designed for performing data computation and handling the fine grained synchronization required by streaming memory systems.

Several designs for aggregate and combining networks have also been suggested. The suggested implementation for the NYU Ultracomputer fetch-and-add mechanism included a combining operation at each network switch, and not just at the target network interface. The CM-5 control network could perform reductions and scans on integral data from the different processors in the system, and not based on memory location [LAD+96]. Hoarse and Dietz suggest a more general version the above designs [HD98].

The alpha-blending mechanism [Ope04] of application specific graphic processors can be thought of as a limited form of scatter-add. However, this is not a standard interpretation and even as graphic processors are implementing programmable features [nVI06, ATI06], alpha-blending is not exposed to the programmer.

Finally, processor-in-memory architectures provide functional units within the DRAM chips and can potentially be used for scatter-add. However, the designs suggested so far do not provide the right granularity of functional units and control for an efficient scatter-add implementation, nor are they capable of floating-point operations. Active Pages [OCS98] and FlexRAM [KHY+99] are coarse-grained and only provide functional units per DRAM page or more, and C-RAM[ESS92] places a functional unit at each sense amplifier. Our hardware scatter-add mechanism presented in Section 5.2 places the functional units at the on-chip cache or DRAM interfaces, providing a high-performance mechanism at a small hardware cost while using commodity DRAM.

5.2 Scatter-Add Architecture

In this section we describe the integration of the scatter-add unit with a streaming memory system and its micro-architecture, and explain the benefits and implications of the scatter-add operation.

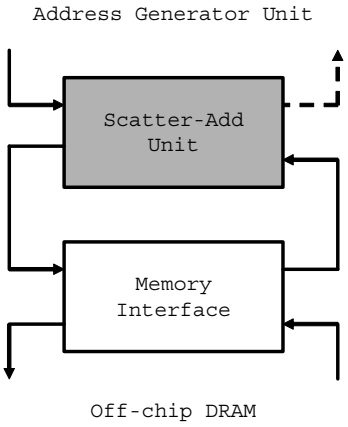


Figure 5.2: Scatter-add unit in the memory channels

5.2.1 Scatter-add Micro-architecture

As described earlier, the scatter-add operation is a parallel atomic read-modify-write operation. Another way of thinking of this operation is as an extension to the memory scatter operation where each value being written is summed with the value already in memory, instead of replacing it.

The key technological factor allowing us to provide the scatter-add functionality in hardware is the rapid rate of VLSI device scaling. While a 64-bit floating-point functional unit consumed a large fraction of a chip in the past [KM89], corresponding area for such a unit in today's $90nm$ technology requires only $0.3\mathrm{mm}^2$. As a result we can dedicate several floating-point/integer adders to the memory system and configure them to perform an atomic read-modify-write, enabling the hardware scatter-add. A natural location for the scatter-add unit is at the memory channels of the stream processor chip since all memory requests pass through this point (Figure 5.2). This configuration allows an easy implementation of atomic operations since the access to each part of global memory is limited to on-chip memory controller of that processor (Figure 2.7). A further advantage of placing the scatter-add unit in front of the memory controller is that it can combine scatter-add requests and reduce memory traffic as will be explained shortly. Another possible configuration is to associate a scatter-add unit with each cache-bank of the stream cache (if it exists) as depicted in Figure 5.3a. The reasoning is similar to the one presented above as

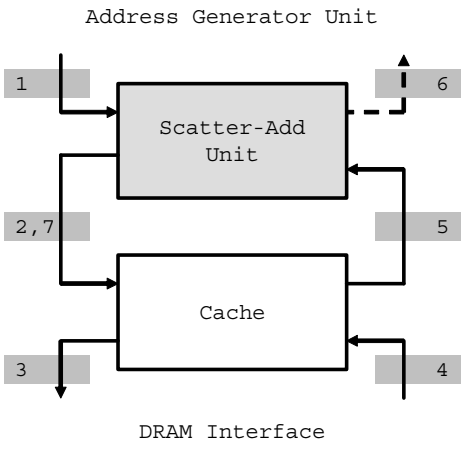
the on-chip cache also processes every memory request.

The Scatter-add unit itself consists of a simple controller with multiplexing wires, the functional unit that performs the integer and floating-point additions, and a *combining store* that is used to ensure atomicity as explained below (Figure 5.3b). The combining store is analogous to the *miss status holding register* (MSHR) and *write combining buffer* of memory data caches, and serves two purposes. First, it acts as an MSHR and buffers scatter-add requests until the original value is fetched from memory. Second, it buffers scatter-add requests while an addition, which takes multiple cycles, is performed. The physical implementation of the scatter-add unit is simple and our estimates for the area required are $0.2mm^2$, and 8 scatter-add units require only 2% of a $10mm \times 10mm$ chip in $90nm$ technology based on a standard-cell design. The overheads of the additional wire tracks necessary for delivering scatter-add requests within the memory system are negligible. Our area analysis is based on the ALU implementation of Imagine [KDR+03] designed in a standard-cell methodology and targeting a latency of 4 1ns cycles. We also include the area required for the combining store and control logic.

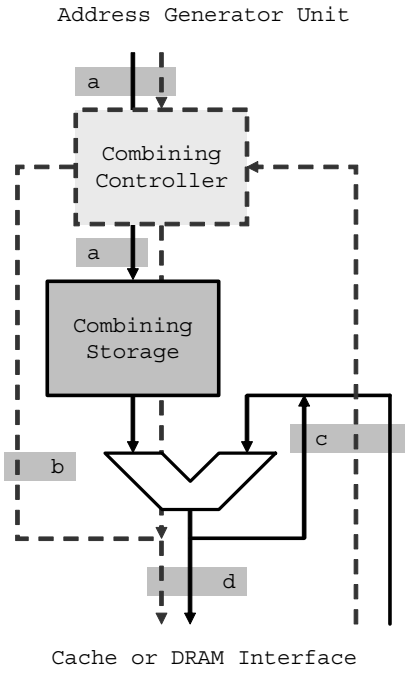
We will explain how atomicity and high throughput are achieved by walking through the histogram example in detail. Programming the histogram application on a scatter-add enabled stream architecture involves a gather on the data to be processed, followed by computation of the mapping and a scatter-add into the bins:

```
gather(data_part, data);

forall i = 1..data_part_len
  bin_part[i] = map(data_part[i]);

scatterAdd(bins, bin_part, 1);
```

We will concentrate on the histogram computation performed by the scatter-add unit as depicted in Figure 5.3, where the number annotations will be referred to in the order of operations performed, solid lines represent data movement, and dotted lines represent address paths. The process is also shown using a flow-diagram in Figure 5.4. The memory-system address generator of the streaming memory system produces a stream of memory addresses corresponding to the histogram bins, along with a stream of values to be summed (simply



(a)



(b)

Figure 5.3: Scatter-add unit in a stream cache bank (a) along with the internal micro architecture (b). Solid lines represent data movement, and dotted lines address paths. The numbers correspond to the explanation in the text.

a stream of 1s in the case of histogram). If an individual memory request that arrives at the input of the scatter-add unit (1) is a regular memory-write, it bypasses the scatter-add and proceeds directly to the cache and DRAM interface (2,3). Scatter-add requests must be performed atomically, and to guarantee this the scatter-add unit uses the combining store structure. Any scatter-add request is placed in a free entry of the combining store, if no such entry exists, the scatter-add operation stalls until an entry is freed. At the same time, the request address is compared to the addresses already buffered in the unit using a *content addressed memory* (CAM) (a). If the address does not match any active addition operations a request for the current memory value of this bin is sent to the memory system (b, 2, 3). If an entry matching the address is found in the combining store no new memory accesses are issued. When a current value returns from memory (4,5) it is summed with the combining store entry of a matching address (again using a CAM search) (c). Once the sum is computed by the integer/floating point functional unit an acknowledgment signal is sent to the address generator unit (6), and the combining store is checked once more for the address belonging to the computed sum (d)[2]. If a match is found, the newly computed sum acts as a returned memory value and repeats step c. If there are no more pending additions to this value, it is written out to memory (7). We are assuming that the acknowledgment signal is sent once the request is handled within the scatter-add unit. Since atomicity is achieved by the combining register, no further synchronization need take place. Using the combining register we are able to pipeline the addition operations achieving high computational throughput.

Multi-node Scatter-add

The procedure described above illustrates the use and details of the scatter-add operation within a single node of a DPA. When multiple nodes perform a scatter-add concurrently, the atomicity of each individual addition is guaranteed by the fact that a node can only directly access its own part of the global memory. The network interface directs memory requests to pass through the remote scatter-add unit where they are merged with local requests[3]. For

[2]Only a single CAM operation is necessary if a simple ordering mechanism is implemented in the combining store

[3]The combining store will send an acknowledgment to the requesting node once an addition is complete

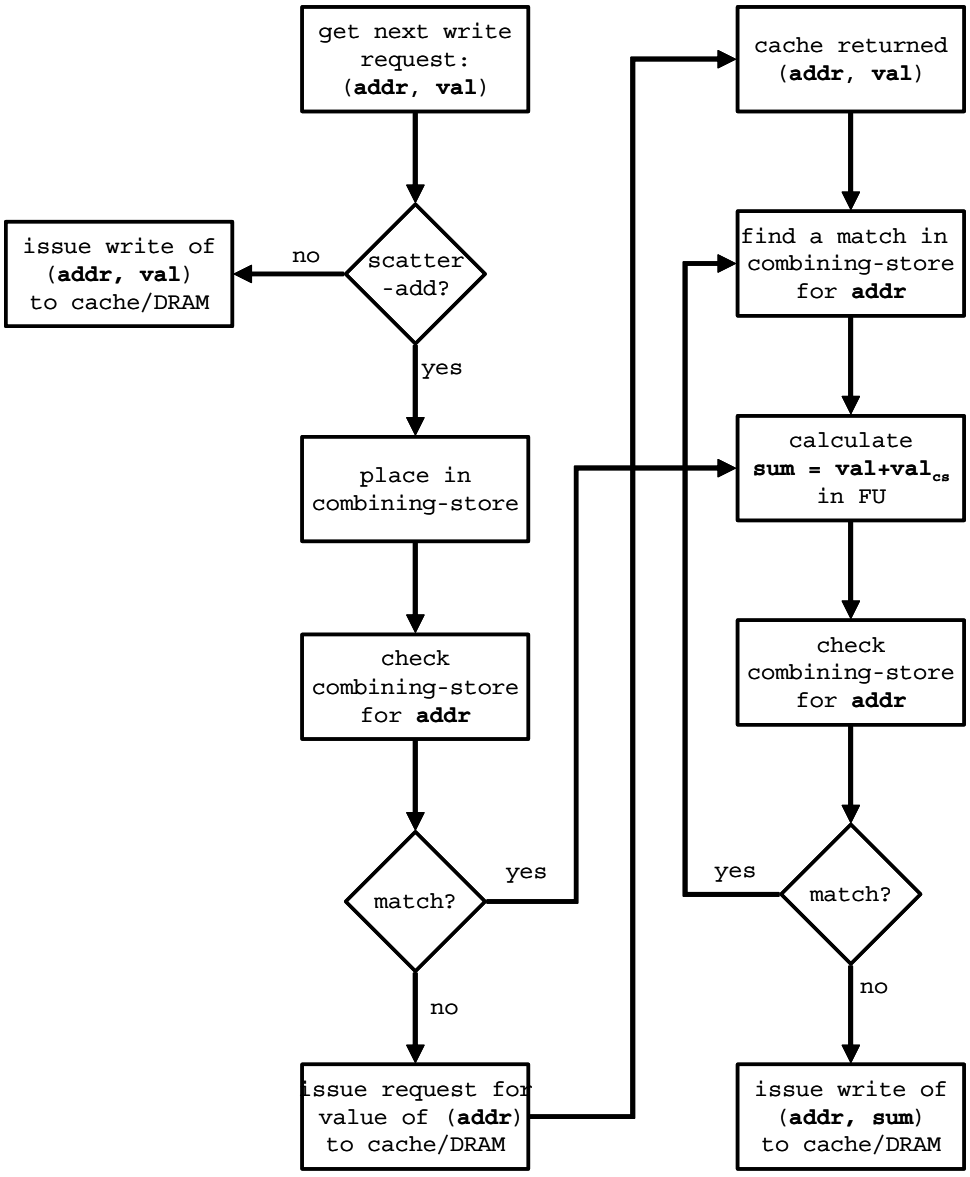


Figure 5.4: Flow diagram of a scatter-add request

multi-node configurations with local data-caches an optimization of this mechanism is to perform the scatter-add in two logical phases:

- A *local* phase in which a node performs a scatter-add on local and remote data within it's cache. If a remote memory value has to be brought into the cache, it is simply allocated with a value of 0 instead of being read from the remote node.
- In the *global* phase the global scatter-add is computed by performing a *sum-back* of the cached values. A sum-back is similar to a cache write-back except that the remote write-request appears as a scatter-add on the node owning the memory address.

The global sum is continuously updated as lines are evicted from the different caches (via sum-back), and to ensure the correct final result a flush-with-sum-back is performed as a synchronization step once all nodes complete.

5.2.2 Scatter-add Usage and Implications

The main use of scatter-add is to allow the programmer the freedom to choose algorithms that were previously prohibitively expensive due to sorting, privatization complexity, and the additional synchronization steps required. Shifting computation to the scatter-add unit from the main DPA execution core allows the core to proceed with running the application, while the scatter-add hardware performs the summing operations. Also, the combining ability of the combining store may reduce the memory traffic required to perform the computation. The performance implications are analyzed in detail in Section 5.3.

A subtle implication of using a hardware scatter-add has to do with the ordering of operations. While the user coded the application using a specific order of data elements, the hardware reorders the actual sum computation due to the pipelining of the addition operations and the unpredictable memory latencies when fetching the original value. It is important to note that while the ordering of computation does not reflect program order, it is consistent in the hardware and repeatable for each run of the program[4]. The user must be aware of the potential ramifications of this reordering when dealing with floating-point rounding errors and memory exceptions.

[4]This is not the case when dealing with multi-node scatter-add, but the lack of repeatability in multi-node memory accesses is a common feature of high performance computer systems.

The scatter-add architecture is versatile and with simple extensions can be used to perform more complex operations. We will not describe these in detail, but will mention them below. A simple extension is to expand the set of operations handled by the scatter-add functional unit to include other commutative and associative operations such as min/max and multiplication. A more interesting modification is to allow a return path for the original data before the addition is performed and implement a parallel fetch-add operation similar to the scalar Fetch&Op primitive[GGK+82]. This data-parallel version can be used to perform parallel queue allocation on vector and stream systems.

5.3 Evaluation

To evaluate the hardware scatter-add mechanism we measure the performance of several applications on a simulated stream processor system, with and without hardware scatter-add. We also modify the machine configuration parameters of the baseline system to test the sensitivity of scatter-add performance to the parameters of the scatter-add unit. Finally, we show that scatter-add is valuable in a multi-node environment as well.

5.3.1 Test Applications

The three applications we use were chosen because they require a scatter-add operation. Codes that do not have a scatter-add will run unaffected on an architecture with a hardware scatter-add capability. The inputs used for the applications below are representative of real datasets, and the dataset sizes were chosen to keep simulation time reasonable on our cycle accurate simulator.

Histogram

The histogram application was presented in the introduction of this chapter. The input is a set of random integers chosen uniformly from a certain range which we vary in the experiments. The output is an array of bins, where each bin holds the count of the number of elements from the dataset that mapped into it. The number of bins in our experiments

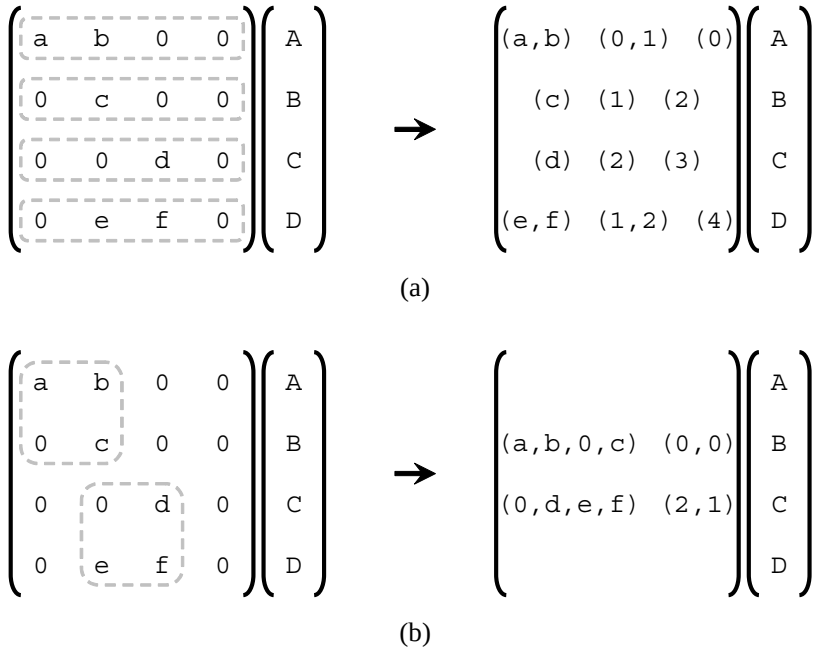


Figure 5.5: Two algorithms are implemented for the sparse matrix-vector multiply – CSR and EBE. (a) In CSR all matrix elements are stored in a dense array, and additional information is kept on the position of each element in a row and where each row begins. (b) In EBE a large sparse-matrix is divided into many small dense-matrices and dense matrix multiplies are performed in series.

matches the input range. Scatter-add is used to produce the histogram as explained in the introduction.

Sparse Matrix-Vector Multiply

The sparse matrix-vector multiply is a key computation in many scientific applications. We implement both a *compressed sparse row* (CSR) and an *element by element* [Saa03] (EBE) algorithm. Figure 5.5 illustrates both algorithms on an example matrix and vector. The two algorithms provide different trade-offs between amount of computation and memory accesses required, where EBE performs more operations at reduced memory demand. The dataset was extracted from cubic element discretization with 20 degrees of freedom using C^0 continuous Lagrange finite elements of a 1916 tetrahedra finite-element model. The matrix size is $9,978 \times 9,978$ and it contains an average of 44.26 non-zeros per row. In

CSR all matrix elements are stored in a dense array, and additional information is kept on the position of each element in a row and where each row begins (in Figure 5.5a, value, relative position per row, and starting point of each row are grouped in the right side). The CSR algorithm is gather based and does not use the scatter-add functionality. EBE utilizes the inherent structure in the sparse matrix due to the finite-element model, and is able to reduce the number of memory accesses required by increasing the amount of computation and performing a scatter-add to compute the final multiplication result. Essentially in the EBE algorithm instead of performing the multiplication on one large sparse-matrix, the calculation is performed by computing many small dense matrix multiplications where each dense matrix corresponds to an element (in Figure 5.5b, a sparse 4×4 matrix is divided into two dense 2×2 matrices, and elements and starting row and column information of each dense matrix are stored as shown in the right side).

Molecular Dynamics

We use the molecular dynamics application explained in Chapter 3. Here the simulation was performed on a sample of 903 water molecules. Dummy elements are padded to each center molecule to make the number of neighbor molecules multiple of 8, and nodes and neighbors are loaded to SRF without duplicate removal.

We implemented two versions of each application, one that uses the hardware scatter-add, and a second that performs a sort followed by a segmented scan. It is important to note that it is not necessary to sort the entire stream that is to be scatter-added, and that the scatter-add can be performed in batches. This reduces the run-time significantly, and on our simulated architecture a batch size of 256 elements achieved the highest performance. Longer batches suffer from the $O(n \log n)$ scaling of sort, while smaller batches do not amortize the latency of starting a stream operations and priming the memory pipeline. The sort was implemented using a combination of a bitonic and merge sorting phases. The histogram computation was also implemented using the privatization method.

Parameter	
Number of scatter-add units per bank	1
Latency of scatter-add functional unit	4 cycles
Number of combining store entries	8
Number of address generators	2
Peak DRAM bandwidth	4.8 GW/s
Stream cache bandwidth	8 GW/s
SRF bandwidth	64 GW/s
SRF size	128 KWords
Stream cache size	128 KWords

Table 5.1: Baseline machine parameters

5.3.2 Experimental Setup

Single node simulations were performed under the environment explained in Section 3.2, and the baseline configuration of the simulated stream processor and its memory systems is detailed in Table 5.1 for all experiments unless mentioned otherwise. The parameters for scientific applications in Table 3.2 are used unless they are defined in Table 5.1. Our multi-node simulator was derived from this single-node simulator as explained in Section 5.3.5.

5.3.3 Scatter-add Comparison

The experiments below compare the performance of the hardware scatter-add to that of the software alternatives. We will show that hardware scatter-add not only outperforms our best software implementations, but that it also allows the programmer to choose a more efficient algorithm for the computation that would have been prohibitively expensive without hardware support.

Figure 5.6 shows the performance of the histogram computation using the hardware and software (sort followed by segmented scan) implementations of scatter-add. Both mechanisms show the expected $O(n)$ scaling, and the hardware scatter-add consistently outperforms software by ratios of 3-to-1 and up to 11-to-1.

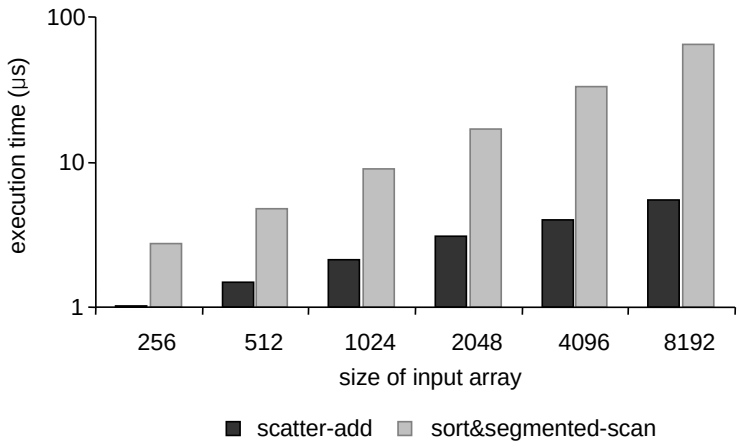


Figure 5.6: Performance of histogram computation for inputs of varying lengths and an input range of $2,048$.

While the software method using sort does not depend much on the distribution of the data, the hardware scatter-add depends on the range of indices. Figure 5.7 shows the execution time of hardware scatter-add and the sort&scan operations for a fixed dataset size of $32,768$ elements and increasing index range sizes. The indices are uniformly distributed over the range. When the range of indices is small hardware performance is reduced due to load imbalance across memory channels. The small index range causes successive scatter-add requests to map to the same memory channel, leaving some of the scatter-add units idle. As the range of indices increases, execution time starts to decreases as more of the units are kept busy. But when the index range becomes too large to fit in the cache, performance decreases significantly and reaches a steady-state. The sort&scan method performance is also lower for the large index ranges as a result of the larger number of memory references required to write out the final result.

Unlike sort, the runtime complexity of the privatization algorithm depends on the number of histogram bins. Figure 5.8 compares the performance of histogram using the hardware scatter-add to the privatization computation method, where the input data length is held constant at $1,024$ or $32,768$, and the range varies from 128 to $8,192$. As the range increases, so does the performance advantage of the hardware scatter-add enabled algorithm, where speedups greater than an order of magnitude are observed for the large input ranges.

Figure 5.9 shows the performance of the sparse matrix-vector multiplication. It is

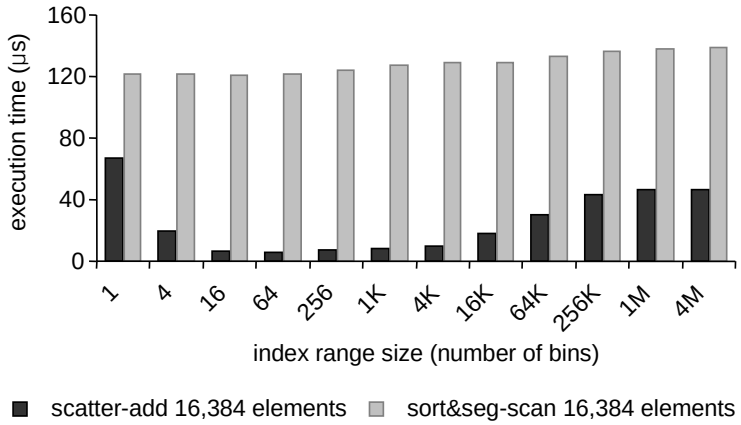


Figure 5.7: Performance of histogram computation for inputs of length $32,768$ and varying index ranges.

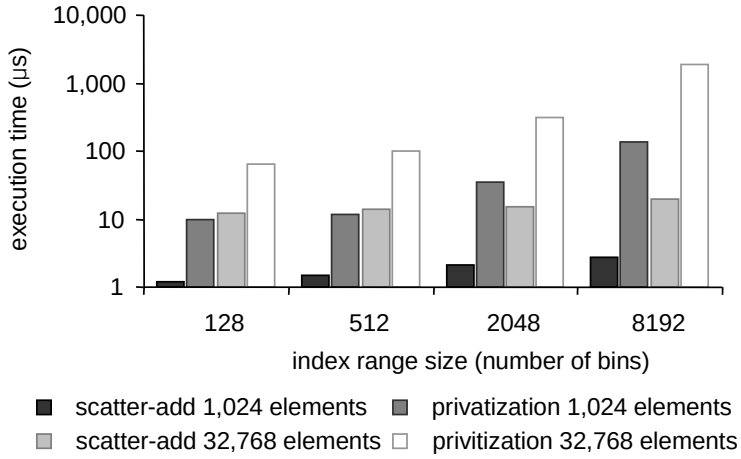
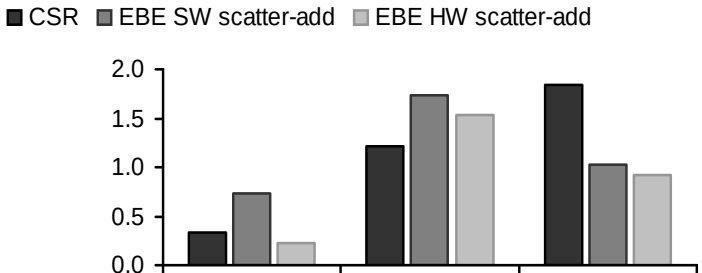


Figure 5.8: Performance of histogram computation with privatization for inputs of constant lengths and varying index ranges.



	Execution Cycles (x1M)	FP Operations (x1M)	Mem References
CSR	0.334	1.217	1.836
EBE SW scatter-add	0.739	1.735	1.031
EBE HW scatter-add	0.23	1.536	0.922

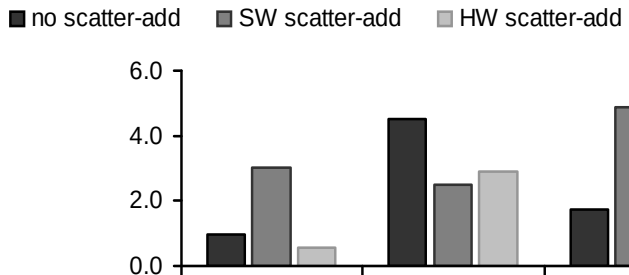
Figure 5.9: Performance of sparse matrix-vector multiplication.

clearly seen that the hardware scatter-add enables the use of the EBE method, as without it CSR outperforms EBE by a factor of 2.2. With hardware scatter-add, EBE can provide a 45% speedup over CSR.

In Figure 5.10 we see how the hardware scatter-add influenced the algorithm chosen for MOLE. The software only implementation with a sort&scan performed very poorly. As a result, the programmer chose to modify the algorithm and avoid the scatter-add. This was done by doubling the amount of computation, and not taking advantage of the fact that the force exerted by one atom on a second atom is equal in magnitude and opposite in direction to the force exerted by the second atom on the first. Using this technique a performance improvement of a factor of 3.1 was achieved. With the availability of a high-performance hardware scatter-add, the algorithm without computation duplication showed better performance, and a speedup of 76% over our best software-only code.

5.3.4 Sensitivity Analysis

The next set of experiments evaluates the performance sensitivity to the amount of buffering in the combining store (CS). In order to isolate and emphasize the sensitivity, we modify the baseline machine model and provide a simpler memory system. We run the experiments without a cache, and implement memory as a uniform bandwidth and latency structure.



	Execution Cycles (x1M)	FP Operations (x10M)	Mem References
no scatter-add	0.975	4.524	1.722
SW scatter-add	3.022	2.49	4.865
HW scatter-add	0.553	2.916	1.87

Figure 5.10: Performance of MOLE without scatter-add, with software scatter-add, and with hardware scatter-add

Throughput is modeled by a fixed cycle interval between successive memory word accesses, and latency by a fixed value which corresponds to the average expected memory delay. On an actual system, DRAM throughput and delays vary depending on the specific access patterns, but with memory access scheduling [RDK+00a] this variance is kept small.

Figure 5.11 shows the dependence of the execution time of the histogram application using hardware scatter-add on the number of entries in the combining store, and on the latencies of the functional unit memory. Each group in the figure corresponds to a specific number of combining store entries, ranging from 2 to 64, and contains seven bars. The four leftmost bars in each group are for increasing memory latencies of 8–256 cycles and a functional unit latency of 4 cycles, and the three rightmost bars show the performance for different functional unit latencies with a fixed memory latency of 16 cycles. Memory throughput is held constant at 1 word every 4 cycles. The execution time is for an input set of 512 elements ranging over $65,536$ possible bins. It can be clearly seen that even with only 16 entries in the combining store performance does not depend on ALU latency and is almost independent of memory latencies. With 64 entries, even the maximal memory latency can be tolerated without affecting performance.

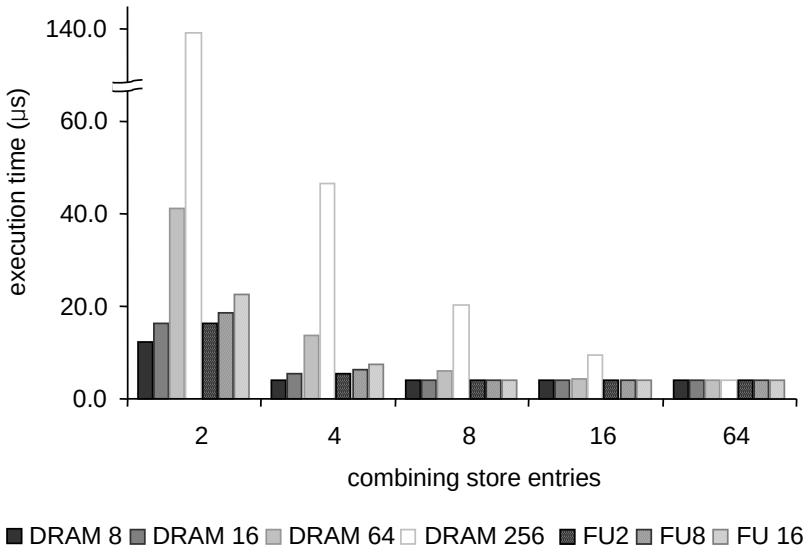


Figure 5.11: Histogram runtime sensitivity to combining store size and varying latencies.

The last sensitivity experiment gauges the scatter-add sensitivity to memory throughput. In Figure 5.12 we plot the histogram runtime vs. the combining store size and decreasing memory throughput. Memory throughput decreases as the number of cycles between successive memory accesses increases. Each group of bars is for a different number of combining store entries. Dark grey bars represent the performance of a histogram where the index range is 16, and light grey bars the case where the index range is $65,536$. Within each group, the dark-light pairs are for a certain memory throughput number. Even when a combining store of 64 entries cannot overcome the performance limitations of very low memory bandwidth. However, the effects of combining become apparent when the number of bins being scattered to is small. In this case, many of the scatter-add requests are captured within the combining store and do not require reads and writes to memory.

5.3.5 Multi-Node Results

Finally, we take a look at the multi-node performance of scatter-add with and without the optimizations described in Section 5.2.1. The multi-node system is comprised of 2–8 nodes, where each node contains a stream processor with the same parameters as used for

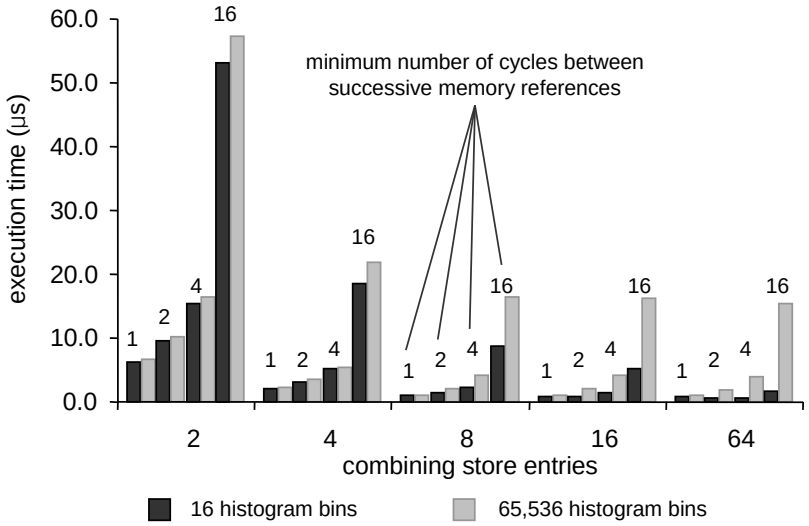


Figure 5.12: Histogram runtime sensitivity to combining store size and varying throughput.

the previous experiments, along with its cache and network interface as described in Section 2.3. The network we model is an input-queued crossbar with back-pressure. We ran experiments limiting the maximum per-node bandwidth to 1 word/cycle (*low* configuration in the results below), and also 8 words/cycle (*high* configuration) that are enough to satisfy scatter-add requests at full bandwidth. In addition to varying the network bandwidth we also turn the combining operation (Section 5.2.1) on and off. We concentrate on the performance of scatter-add alone and report the scalability numbers for only the scatter-add portion of the three test applications. For Histogram we ran two separate data-sets, each with a total of 64K scatter-add references: *narrow* which has an index range of 256, and *wide* with a range of 1M. MOLE uses the first 590K references which span $8,192$ unique indices, and SPAS uses the full set of 38K references over $10,240$ indices of the EBE method (Section 5.3.1).

Figure 5.13 presents the scatter-add throughput (additions per cycle) achieved for scaling of 1–8 nodes, where each of the four sets of lines corresponds to a different application. The *wide* version of Histogram is limited by the memory bandwidth in both the single node case and for any number of nodes when using the high bandwidth network achieving perfect scaling. When the network bandwidth is low, performance is limited by the network and the application does not scale as well. The optimization of combining in the caches

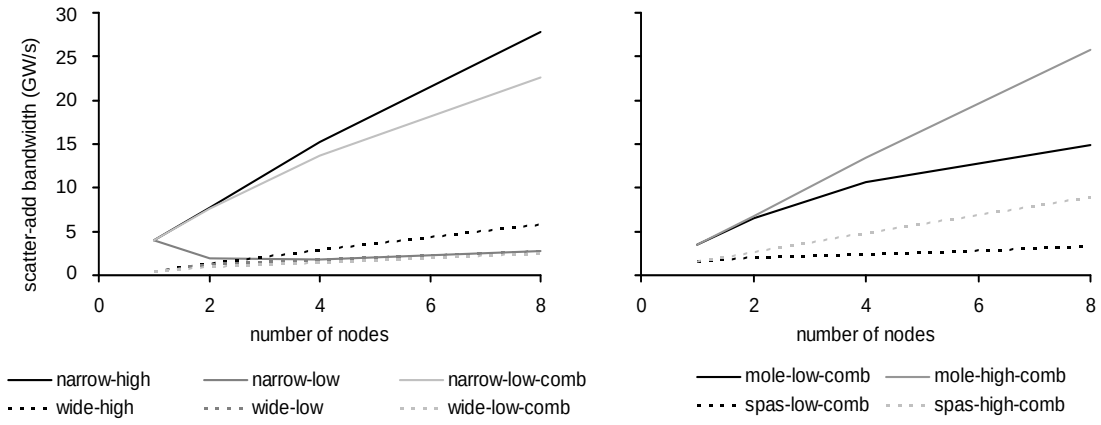


Figure 5.13: Multi-node scalability of scatter-add

does not increase performance due to the large range of addresses accessed, which lead to an extremely low cache hit rate. On the contrary, the added overhead of cache warm-up, cache combining, and the final synchronization step actually reduce performance and limit scalability.

In the *narrow* case of Histogram, the results are quite different as the high locality makes both the combining within the scatter-add unit itself and in the cache very effective. As a result, no scaling is achieved in the case of the low-bandwidth network, although with high bandwidth we see excellent speedups of up to 7.1 for 8 nodes. Employing the multi-node optimization of local combining in the caches followed by global combining as cache lines are evicted provided a significant speedup as well. The reduction in network traffic allowed even the low bandwidth configuration to scale performance by 5.7 for 8 nodes. Again, the overhead of warming up the cache, flushing the cache, and synchronization prevents optimal scaling.

The molecular dynamics application (MOLE) exhibits similar trends to *narrow*, as the locality in the neighbor lists is high. Scaling is not quite as good due to the fact that the overall range of addresses is larger, and the address distribution is not uniform. SPAS experiences similar problems but its scaling trend is closer to that of *wide*. The large index range of both these applications reduces the effectiveness of cache combining due to increased overheads incurred on cache misses and line evictions. Increasing the network

bandwidth limits the effect of this overhead and scaling is improved (*MOLE-high-comb* and *SPAS-high-comb* in Figure 5.13).

Chapter 6

Control Organizations of Stream Processors

Stream processors are optimized for the *stream execution model* [KRD+03] and not for programs that feature only instruction-level parallelism or that rely on fine-grained interacting threads. Stream processors achieve high efficiency and performance by providing a large number of ALUs partitioned into processing elements (*PEs*), minimizing the amount of hardware dedicated to data-dependent control, and exposing a deep storage hierarchy that is tuned for throughput. Software explicitly expresses multiple levels of parallelism and locality and is responsible for both concurrent execution and latency hiding [EAG+04].

In this chapter we extend the stream architecture of Merrimac to support thread-level parallelism on top of the mechanisms for data-level and instruction-level parallelism exploring control organizations of stream processors. Previous work on media applications has shown that stream processors can scale to a large number of ALUs without employing TLP. However, we show that exploiting the TLP dimension can reduce hardware costs when very large numbers of ALUs are provided and can lead to performance improvements when more complex and irregular algorithms are employed.

In our stream architecture, ILP is used to drive multiple functional units within each PE and to tolerate pipeline latencies. We use *VLIW* instructions that enable a highly area- and energy-efficient register file organization [RDK+00b], but a scalar instruction set is

also possible as in [PAB+05]. DLP is used to achieve high utilization of the throughput-oriented memory system and to feed a large number of PEs operated in a *SIMD* fashion. To enable concurrent threads, multiple instruction sequencers are introduced, each controlling a group of PEs in a *MIMD* fashion. Thus, we support a *coarse-grained independent threads of control execution model* where each ALU is under the control of a specific instruction sequencer. Each instruction sequencer supplies SIMD-VLIW instructions to its group of PEs, yielding a fully flexible MIMD-SIMD-VLIW chip architecture. More details are given in Section 6.2.

Instead of focusing on a specific design point we explore the scaling of the architecture along the three axes of parallelism as the number of ALUs is increased and use a detailed model to measure hardware cost and application performance. We then discuss and simulate the tradeoffs between the added flexibility of multiple control threads, the overhead of synchronization and load balancing between multiple threads, and features, such as fine grained communication between PEs, that are only feasible with lockstep execution. Note that our stream architecture does not support multiple execution models as described in [WTS+97, MPJ+00, SNL+03, KBH+04]. The study presented here is a fair exploration of the cost-performance tradeoff and scaling for the three parallelism dimensions using a single execution model and algorithm for all benchmarks.

The main hardware cost of TLP is in additional instruction sequencers and storage. Contrary to our intuition, the cost analysis indicates that the area overhead of supporting TLP is in some cases quite low. Even when all PEs are operated in MIMD, and each has a dedicated instruction sequencer, the area overhead is only 15% compared to our single threaded baseline. When the architectural properties of the baseline are varied, the cost of TLP can be as high as 86%, and we advocate a mix of ILP, DLP, and TLP with hardware overheads below 5% compared to an optimal organization. The performance evaluation also yielded interesting results in that the benefit of the added flexibility of exploiting TLP is hindered by increased synchronization costs. We also observe that as the number of ALUs is scaled up beyond roughly 64 ALUs per chip, the cost of supporting communication between the ALUs grows sharply, suggesting a design point with VLIW in the range of 2–4 ALUs per PE, SIMD across PEs in groups of 8–16 PEs, and MIMD for scaling up to the desired number of ALUs.

In Section 6.1 we present prior work related to this chapter. We describe the ILP/DLP/TLP hardware tradeoff of our architecture in Section 6.2. Section 6.3 discusses application properties and corresponding architectural choices. We develop the hardware cost model and analyze scaling in Section 6.4, and our experiments and performance evaluation appear in Section 6.5.

6.1 Related Work

In this section we describe prior work related to this chapter. We focus on research that specifically explored scalability.

A large body of prior work addressed the scalability of general purpose architectures targeted at the sequential, single-threaded execution model. This research includes work on ILP architectures, such as [PJS97, HBJ+02, PPE+97], and speculative or fine-grained threading as in [SBV95, RJSS97, HHS+00]. Similar studies were conducted for cache-coherent chip multi-processors targeting a more fine-grained cooperative threads execution model (e.g., [BGM+00]). In contrast, we examine a different architectural space in which parallelism is explicit in the programming model offering a widely different set of tradeoff options.

Recently, architectures that target several execution models have been developed. The RAW architecture [WTS+97] scales along the TLP dimension and provides hardware for software controlled low overhead communication between PEs. Smart Memories [MPJ+00] and the Vector-Thread architecture [KBH+04] support both SIMD and MIMD control, but the scaling of ALUs follows the TLP axis. TRIPS [SNL+03] can repartition the ALU control based on the parallelism axis utilized best by an application, however, the hardware is fixed and scalability is essentially on the ILP axis within a PE and via coarse-grained TLP across PEs. The Sony Cell Broadband Engine™ processor (Cell) [PAB+05] is a flexible stream processor that can only be scaled utilizing TLP. A feature common to the work mentioned above is that there is little evaluation of the hardware costs and tradeoffs of scaling using a combination of ILP, DLP, and TLP, which is the focus of this chapter.

The hardware and performance costs of scaling the DLP dimension were investigated in the case of the VIRAM architecture for media applications [KP03]. Media applications

with regular control were also the focus of [KDR+03], which evaluated the scalability of stream processors along the DLP and ILP dimensions. We extend this work by incorporating TLP support into the architecture and evaluating the tradeoff options of ALU control organizations for both regular- and irregular-control applications.

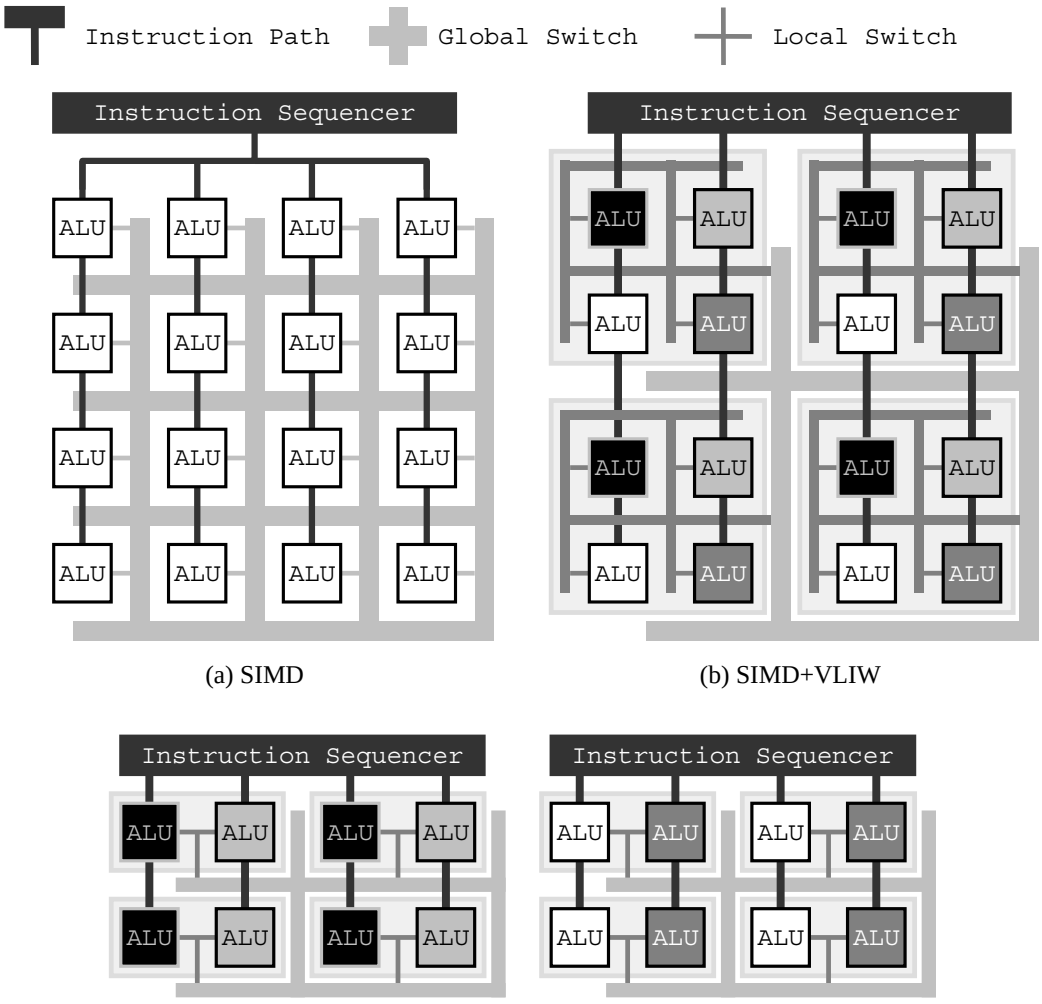
6.2 ALU Organization of Stream Architecture

In this section we focus on the mechanisms and tradeoffs of controlling many ALUs utilizing a combination of ILP, DLP, and TLP. We describe how control of the ALUs can be structured along the different dimensions of parallelism and discuss the implications on synchronization and communication mechanisms. A more thorough description of stream architectures and their merits appears in Chapter 2. A conceptual discussion of the benefits of TLP, DLP, and ILP relating to applications can be found in Section 6.3, and the analysis of the hardware costs of the various mechanisms is developed in Section 6.4 followed by experimental measurements in Section 6.5.

6.2.1 DLP

A DLP organization of ALUs takes the form of a single instruction sequencer issuing SIMD instructions to a collection of ALUs. The ALUs within the group execute the same instructions in lockstep on different data. Unlike with vectors or wide-word arithmetic, each ALU can potentially access different SRF addresses. In this organization an optional switch can be introduced to connect the ALUs. Figure 6.1(a) shows how all ALUs receive the same instruction (the “white” instruction) from a single instruction issue path and communicate on a global switch shared between the group of ALUs. Because the ALUs within the group operate in lockstep, simple control structures can utilize the switch for direct exchange of words between the ALUs, to implement the conditional streams mechanism for efficient data-dependent control operations [Kap04], and to dynamically access SRF locations across several SRF banks [JEAD04].

Another possible implementation of DLP in a stream processor is to use short-vector



(a) SIMD

(b) SIMD+VLIW

(c) SIMD+VLIW+MIMD

Figure 6.1: ALU organization along the DLP, ILP, and TLP axes with SIMD, VLIW, and MIMD mechanisms respectively.

ALUs that operate on wide words as with SSE [TH99], 3DNOW! [OFW99], and Altivec [DDHS00]. This approach was taken in Imagine for 8-bit and 16-bit arithmetic within 32-bit words, on top of the flexible-addressing SIMD, to provide two levels of DLP. We chose not to evaluate this option in this study because it significantly complicates programming and compilation and we explore the DLP dimension using clustering.

6.2.2 ILP

To introduce ILP into the DLP configuration the ALUs are partitioned into PEs or clusters. Within a cluster each ALU receives a different operation from a VLIW instruction. DLP is used across clusters by using a single sequencer to issue SIMD instructions as before, but with each of these instructions being VLIW. Figure 6.1(b) shows an organization with four clusters of four ALUs each. In each cluster a VLIW provides a “black”, a “dark gray”, a “light gray”, and a “white” instruction separately to each of the four ALUs. The same set of four instructions feeds the group of ALUs in all clusters (illustrated with shading in Figure 6.1(b)).

In this SIMD/VLIW clustered organization the global switch described above becomes hierarchical. An *intra-cluster* switch connects the ALUs and LRFs within a cluster, and is statically scheduled by the VLIW compiler. The clusters are connected with an *inter-cluster* switch that is controlled in the same manner as the global DLP switch. This hierarchical organization provides an area-efficient and high bandwidth interconnect.

6.2.3 TLP

To address TLP we provide hardware MIMD support by adding multiple instruction sequencers and partitioning the inter-cluster switch. As shown in Figure 6.1(c), each sequencer controls a *sequencer group* of clusters (in this example four clusters of two ALUs each). Within each cluster the ALUs share an intra-cluster switch, and within each sequencer group the clusters can have an optional inter-cluster switch. The inter-cluster switch can be extended across multiple sequencers, however, doing so requires costly synchronization mechanisms to ensure that a transfer across the switch is possible and that the data is coherent. A discussion of such mechanisms is beyond the scope of this study. It is possible to design an interconnect across multiple sequencer groups and partition it in software to allow a reconfigurable TLP option, and the hardware cost tradeoff of a full vs. a partitioned switch is discussed in Section 6.4. Another option is to communicate between sequencer groups using coarser-grained messages that amortize synchronization costs. For example, the Cell processor uses DMA commands to transfer data between its 8 sequencer groups (SPEs in Cell terminology) [PAB+05].

We do not evaluate extending a stream processor in the TLP dimension by adding virtual contexts that share the existing hardware as suggested for other architectures in [TEL95, KBH+04]. Unlike other architectures a stream processor relies heavily on explicitly expressing locality and latency hiding in software and exposes a deep bandwidth/locality hierarchy with hundreds of registers and megabytes of software controlled memory. Replicating such a large context is infeasible and partitioning the private LRFs reduces software's ability to express locality.

6.3 Impact of Applications on ALU Control

In order to understand the relationship between application properties and the mechanisms for ALU control and communication described in Section 6.2, we characterize numerically-intensive applications based on the following three criteria. First, whether the application is throughput-oriented or presents real-time constraints. Second, whether the parallelism in the application scales with the dataset or is fixed by the numerical algorithm. Third, whether the application requires *regular* or *irregular* flow within the numerical algorithm. Regular control corresponds to loops with statically determined bounds, and irregular control implies that the work performed is data dependent and can only be determined at runtime.

6.3.1 Throughput vs. Real-Time

In throughput-oriented applications, minimizing the time to solution is the most important criteria. For example, when multiplying large matrices, simulating a complex physical system as part of a scientific experiment, or in offline signal and image processing, the algorithms and software system can employ a batch processing style. In applications with real-time constraints, on the other hand, the usage model restricts the latency allowed for each sub-computation. In a gaming environment, for instance, the system must continuously respond to user input while performing video, audio, physics, and AI tasks.

The Imagine and Merrimac stream processors, as well as the similarly architected ClearSpeed CSX-600, are designed for throughput-oriented applications. The ALUs in

these processors are controlled with SIMD-VLIW instructions along the ILP and DLP dimensions only. The entire on-chip state of the processor, including the SRF (1MB/576KB in Merrimac/CSX600) and LRF (64KB/12KB), is explicitly managed by software and controlled with a single thread of execution. The software system exploits locality and parallelism in the application to utilize all processor resources. This organization works well for a large number of applications and scales to hundreds of ALUs [KDR+03, DHE+03]. However, supporting applications with real-time constraints can be challenging. Similarly to a conventional single-threaded processor, a software system may either preempt a running task due to an event that must be processed, or partition tasks into smaller subtasks that can be interleaved to ensure real-time goals are achieved.

Performing a preemptive context switch requires that the SRF be allocated to support the working set of both tasks. In addition, the register state must be saved and restored potentially requiring over 5% of SRF capacity and consuming hundreds of cycles due to the large number of registers. It is possible to partition the registers between multiple tasks to avoid this overhead at the expense of a smaller register space that can be used to exploit locality.

Subdividing tasks exposes another weakness of throughput-oriented designs. The aggressive static scheduling required to control the stream processor's ALUs often results in high task startup costs when software pipelined loops must be primed and drained. If the amount of work a task performs is small, because it was subdivided to ensure interactive constraints for example, these overheads adversely affect performance.

Introducing MIMD to support multiple concurrent threads of control can alleviate these problems, by allocating tasks spatially across the sequencer groups. In this way resources are dedicated to specific tasks leading to predictable timing. The Cell processor, which is used in the Sony PlayStation 3 gaming console, takes this approach. Using our terminology a Cell is organized as 8 sequencer groups with one cluster each, and one non-ALU functional unit and one FPU per cluster. The functional units operate on 128-bit wide words representing short vectors of four 32-bit operands each. As mentioned in Section 6.2, we do not directly evaluate vector-style SIMD in this thesis and focus on an architecture where SIMD clusters can access independent SRF addresses.

6.3.2 Scaling of Parallelism

Many numerical algorithms and applications have parallelism that scales with the dataset size. Examples include n-body system simulation where interactions are calculated for each particle, processing of pixels in an image, images within a video stream, and points in a sampled signal. In such applications the parallelism can typically be cast into all three parallelism dimensions. Multiple threads can process separate partitions of the dataset, multiple clusters can work simultaneously on different elements, and loop unrolling and software pipelining transform data parallelism into ILP. The resulting performance depends on multiple factors such as the regularity of the control, load balancing issues, and inherent ILP as will be discussed in Section 6.5.

On the other hand, some numerical algorithms place a limit on the amount of parallelism, reducing scalability. For example, the deblocking filter algorithm of H.264 is limited in parallelism to computations within a 4×4 block because of a data dependency between blocks [Ric03]. This leads to fine-grained serialization points of control, and the application may be more amenable to a space-multiplexed TLP mapping. The mapping of this type of limited-parallelism algorithm to highly parallel systems is a topic of active research and its evaluation is beyond the scope of this thesis.

6.3.3 Control Regularity

All control decisions within a *regular control* portion of an algorithm can be determined statically. A typical case of regular control is a loop nest with statically known bounds. Regular control applications are very common and examples include convolution, FFT, dense matrix multiplication, and fixed degree meshes in scientific codes. Such algorithms can be easily mapped onto the DLP dimension and executed under SIMD-type control, and can also be cast into ILP and TLP. Converting DLP into TLP incurs overheads related to synchronization of the threads due to load imbalance. Casting DLP as ILP increases pipeline scheduling overheads related to software pipelining and loop unrolling, that are required to effectively utilize many ALUs. As a result, we expect that a high SIMD degree is necessary to achieve best performance on regular control applications.

In code with *irregular control*, decisions on control flow are made based on runtime

information and cannot be determined statically. For example, when processing an element mesh the number of neighbors each element has may vary, leading to irregular control. Mapping irregular algorithms to the TLP dimension is simple as each sequencer follows its own control path. Mapping along the DLP dimension onto SIMD hardware is more challenging, and is discussed in detail in prior work [Kap04, Ere06].

We evaluate the benefits and overheads of utilizing parallelism along the three axes for both regular and irregular applications in Section 6.5.

6.4 VLSI Area Models

Our hardware cost model is based on an estimate of the area of each architectural component for the different organizations normalized to a single ALU. We focus on area rather than energy and delay for two reasons. First, as shown in [KDR+03], the energy of a stream processor is highly correlated to area. Second, a stream processor is designed to efficiently tolerate latencies by relying on software and the locality hierarchy, thereby reducing the sensitivity of performance to pipeline delays. Furthermore, as we will show below, near optimal configurations fall within a narrow range of parameters limiting the disparity in delay between desirable configurations. In this study we only analyze the area of the structures relating to the ALUs and their control. The scalar core and memory system performance must scale with the number and throughput of the ALUs and do not depend on the ALU organization. Based on the implementation of Imagine [KDR+03] and the design of Merrimac [Ere06] we estimate that the memory system and the scalar core account for roughly 40% of a stream processor's die area.

6.4.1 Cost Model

The area model follows the methodology developed in [KDR+03] adapted to the design of the Merrimac processor [DHE+03, JEAD04, Ere06] with the additional MIMD mechanism introduced in this chapter. Table 6.1 and Table 6.2 summarize the parameters and equations used in the area cost model. The physical sizes are given in technology independent *track* and *grid* units, and are based on an ASIC flow. A *track* corresponds to the minimal distance

Parameter	Value	Description (unit)
b	64/32	Data width of the architecture (bits)
A_{SRAM}	16	Area of a single ported SRAM bit used for SRF and instruction store (grids)
A_{sb}	128	Area of a dual ported stream-buffer bit (grids)
G_{SRF}	0.18	Area overhead of SRF structures relative to SRAM bit
w_{ALU}	1754	Datapath width of a 64-bit ALU (tracks)
w_{nonALU}	350	Datapath width of non-ALU supporting functional unit (FU) (tracks)
w_{LRF}	281	Datapath width of 64-bit LRF per functional unit (tracks)
h	2800	Datapath height for 64-bit functional units and LRF (tracks)
G_{COMM}	0.25	COMM units required per ALU
G_{ITER}	0.5	ITER units required per ALU
G_{sb}	1	Capacity of a half stream buffer per ALU (words)
I_0	64	Minimal width of VLIW instructions for control (bits)
I_N	64	Additional width of VLIW instructions per ALU/FU (bits)
L_C	4	Initial number of cluster SBs
L_N	1	Additional SBs required per ALU
L_{AG}	8	Bandwidth of on-chip memory system (words/cycle)
S_{SRF}	2048	SRF capacity per ALU (words)
S_{SEQ}	2048	Instruction capacity per sequencer group (VLIW words)
C	—	Number of clusters
N	—	Number of ALUs per cluster
T	—	Number of sequencers

Table 6.1: Summary of VLSI area parameters (ASIC flow). *Tracks* are the distance between minimal pitch metal tracks; *grids* are the area of a (1×1)track block.

between two metal wires at the lowest metal layer. A *grid* is the area of a (1×1) track block. Because modern VLSI designs tend to be wire limited, areas and spans measured in grids and tracks scale with VLSI fabrication technology.

We only briefly present the parameters and equations of the area model and a more detailed description can be found in [KDR+03].

The first parameter in Table 6.1 is the data width of the architecture, and we show results for both 64-bit and 32-bit arithmetic. The second set of parameters corresponds to the per-bit areas of storage elements, where the G_{SRF} parameter accounts for the multiplexers and

decoders necessary for the SRF structures. The third group relates to the datapath of the stream processor and gives the widths and heights of the functional units and LRF.

Because the stream architecture is strongly partitioned, we only need to calculate the area of a given sequencer group to draw conclusions on scalability, as the total area is simply the product of the area of a sequencer group and the number of sequencers. The main components of a sequencer group are the sequencer itself, the SRF, the clusters of functional units, LRF, and intra-cluster switch. A sequencer may also contain an inter-cluster switch to interconnect the clusters.

The sequencer includes the SRAM for storing the instructions and a datapath with a simple and narrow ALU (similar in area to a non-ALU numerical functional unit) and registers. The instruction distribution network utilizes high metal layers and does not contribute to the area. Because each VLIW instruction controls N functional units, the area of the sequencer scales roughly linearly with N. Note that some designs, such as the Cell, processor share the SRF structure for both data and instructions. In this case as well the model must account for the additional capacity required to support the instructions.

In our architecture, the SRF is banked into lanes such that each cluster contains a single lane of the SRF, as in Imagine and Merrimac. The capacity of the SRF per ALU is fixed (S_{SRF}), but the distribution of the SRF array depends on C and N. The SRF cost also includes the hardware of the stream buffers. The amount of buffering required depends on both the number of ALUs within a cluster, and the bandwidth supplied by the wide SRF port. The number of SBs is equal to the number of external ports in a cluster P_e. The number of ports has a minimal value L_C required to support transfers between the SRF and the memory system and a term that scales with N. The capacity of each SB must be large enough to effectively time multiplex the single SRF port, which must support enough bandwidth to both feed the ALUs (scales as $G_{sb}N$) and saturate the memory system (scales as $2L_{AG}/C$). The SRF bandwidth required per ALU (G_{sb}) will play a role in the performance of the MATMUL application described in Section 3.1.

Note that the total amount of buffering in the SBs is larger when N or C is smaller due to the minimal buffering required to support multiple streams in a kernel and to maintain a minimum throughput between the SRF lane within a cluster and the global memory system.

The intra-cluster switch that connects the ALUs within a cluster and the inter-cluster

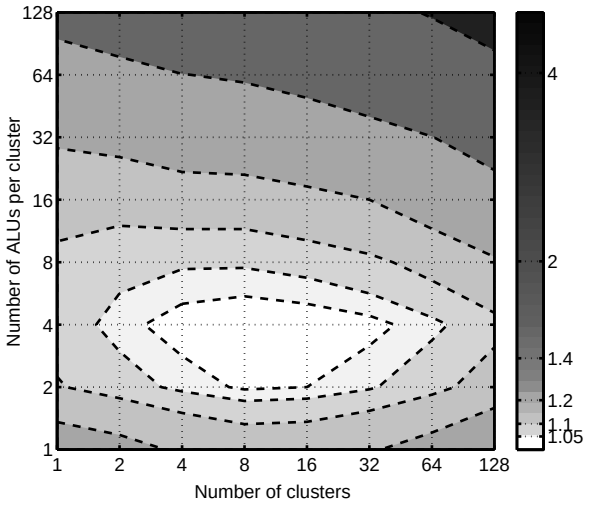
Component	Equation
COMMs per cluster	$N_{COMM} = (C = 1 \; ? \; 0 : \lceil G_{COMM} N \rceil)$
ITERs per cluster	$N_{ITER} = \lceil G_{ITER} N \rceil$
FUs per cluster	$N_{FU} = N + N_{ITER} + N_{COMM}$
External Cluster Ports	$P_e = L_C + L_N N$
COMM bit width	$b_{COMM} = (b + log_2(S_{SRF} NC)) N_{COMM}$
Sequencer area	$A_{SEQ} = S_{SEQ}(I_0 + I_N N_{FU}) A_{SRAM} + h(w_{nonALU} + w_{LRF})$
SRF area per cluster	$A_{SRF} = (1 + G_{SRF}) S_{SRF} N A_{SRAM} b$ $+2A_{sb} b P_e max(G_{sb} N, L_{AG}/C)$
Intra-cluster switch area	$A_{SW} = N_{FU}(\sqrt{N_{FU}} b)(2\sqrt{N_{FU}} b + h + 2w_{ALU} + 2w_{LRF})$ $+\sqrt{N_{FU}}(3\sqrt{N_{FU}} b + h + w_{ALU} + w_{LRF}) P_e b$
Cluster area	$A_{CL} = N_{FU} w_{LRF} h$ $+(N w_{ALU} + (N_{ITER} + N_{COMM}) w_{nonALU}) h + A_{SW}$
Inter-cluster switch area	$A_{COMM} = C b_{COMM} \sqrt{C}(b_{COMM} \sqrt{C} + 2\sqrt{A_{CL} + A_{SRF}})$
Total area	$A_{TOT} = T(C(A_{SRF} + A_{CL}) + A_{COMM} + A_{SEQ})$

Table 6.2: Summary of VLSI area cost models

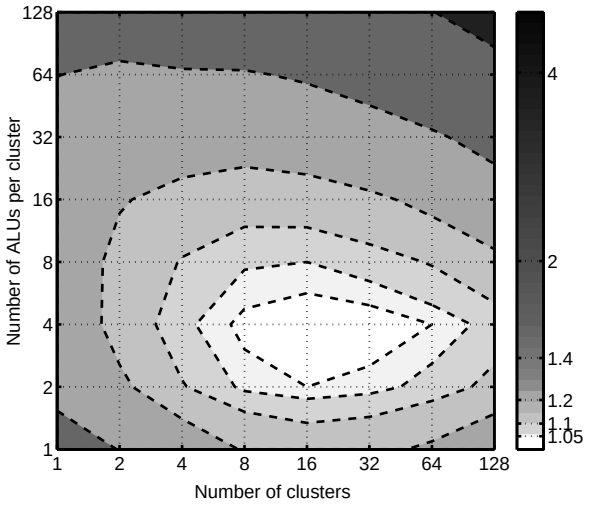
switch that connects clusters use two dimensional grid structures to minimize area, delay, and energy as illustrated in Figure 6.1. We assume fully-connected switches, but sparser interconnects may also be employed.

The intra-cluster switch area scales as N^2, but for small N the cluster area is dominated by the functional units and LRF, which scale with N but have a larger constant factor.

The inter-cluster switch can be used for direct communication between the clusters, and also allows the clusters within a sequencer group to share their SRF space using the cross-lane indexed SRF mechanism [JEAD04]. We use a full switch to interconnect the clusters within a sequencer group and its grid organization is dominated by the C^2 scaling term. However, the dependence on the actual area of a cluster, the SRF lane, the SBs, and the number of COMM units required (N_{COMM} scales linearly with N) plays an important role when C is smaller.



(a) 64-bit datapath



(b) 32-bit datapath

Figure 6.2: Relative area per ALU normalized to optimal ALU organization with the baseline configuration: (8-DLP,4-ILP) and (16-DLP,4-ILP) for 64-bit and 32-bit datapaths respectively.

6.4.2 Cost Analysis

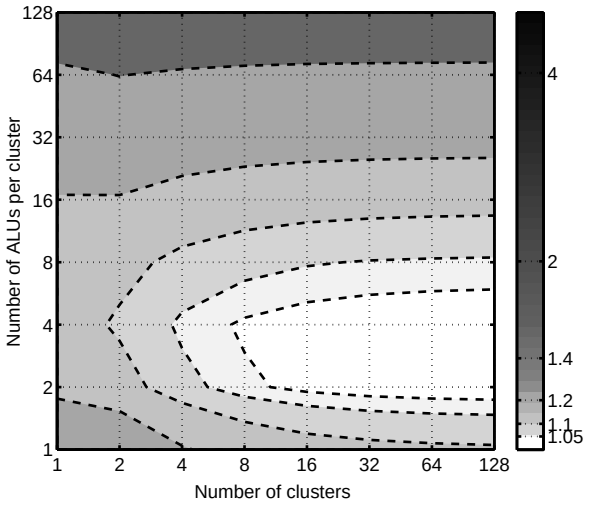
As the analysis presented above describes, the total area cost scales linearly with the degree of TLP, and is equal to the number of sequencer groups (T) multiplied by the area of a

single sequencer group. Therefore, to understand the tradeoffs of ALU organization along the different parallelism axes we choose a specific number of threads, and then evaluate the DLP and ILP dimension using a heatmap that represents the area of different (C, N) design space points relative to the area-optimal point. Figure 6.2 presents the tradeoff heatmap for the baseline configuration specified in Table 6.1 for both a 64-bit and a 32-bit datapath stream processor. The horizontal axis corresponds to the number of clusters in a sequencer group (C), the vertical axis to the number of ALUs within each cluster (N), and the shading represents the relative cost normalized to a single ALU of the optimal-area configuration. In the 64-bit case, the area optimal point is (8-DLP,4-ILP) requiring 1.48×10^7 grids per ALU accounting for all stream execution elements. This corresponds to 94.1mm^2 in a 90nm ASIC process with 64 ALUs in a (2-TLP,8-DLP,4-ILP) configuration. For a 32-bit datapath, the optimal point is at (16-DLP,4-ILP).

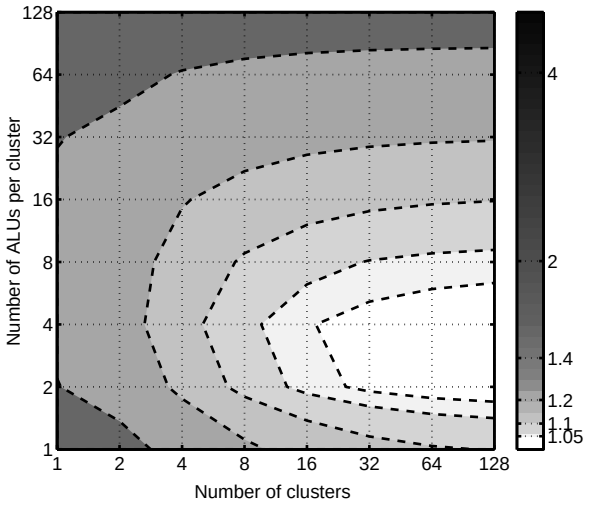
Figure 6.2 indicates that the area dependence on the ILP dimension (number of ALUs per cluster) is much stronger than on DLP scaling. Configurations with an ILP of $2-4$ are roughly equivalent in terms of hardware cost, but further scaling along the ILP axis is not competitive because of the N^2 term of the intra-cluster switch and the increased instruction store capacity required to support wider VLIW instructions. Increasing the number of ALUs utilizing DLP leads to better scaling. With a 64-bit datapath (Figure 6.2(a)), configurations in the range of $2-32$ and 4 ALUs are within about 5% of the optimal area. When scaling DLP beyond 32 clusters, the inter-cluster switch area significantly increases the area per ALU. A surprising observation is that even with no DLP, the area overhead of adding a sequencer for every cluster is only about 15% above optimal.

Both trends change when looking at a 32-bit datapath (Figure 6.2(b)). The cost of a 32-bit ALU is significantly lower, increasing the relative cost of the switches and sequencer. As a result only configurations within a $2-4$ ILP and $4-32$ DLP are competitive. Providing a sequencer to each cluster in a 32-bit architecture requires 62% more area than the optimal configuration.

Scaling along the TLP dimension limits direct cluster to cluster communication to within a sequencer group, reducing the cost of the inter-cluster switch, which scales as C^2. The inter-cluster switch, however, is used for performance optimizations and is not necessary for the architecture because communication can always be performed through



(a) 64-bit datapath



(b) 32-bit datapath

Figure 6.3: Relative area per ALU normalized to optimal ALU organization for configurations with no inter-cluster switch: (128-DLP,4-ILP).

memory. Figure 6.3 shows the area overhead heatmaps for configurations with no inter-cluster communication. The area per ALU without a switch improves with the amount of DLP utilized, but all configurations with more than 8 clusters fall within a narrow 5% area overhead range. The relative cost of adding sequencers is larger when no inter-cluster

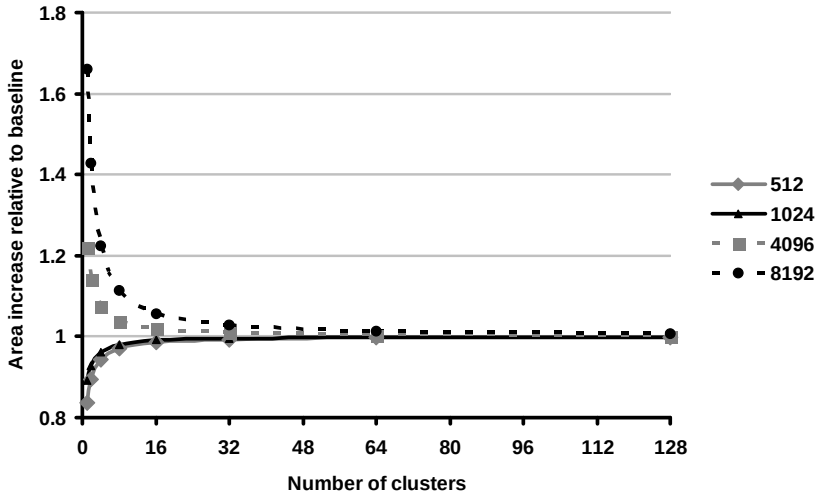
switch is provided because partitioning the control does not reduce the inter-cluster switch area. Thus, the single-cluster sequencer group configurations have an overhead of 27% and 86% for a 64-bit and a 32-bit datapath respectively. Note that the area-optimal configurations with no inter-cluster switch are 9% and 13% smaller than the area-optimal baseline configurations for 64-bit and 32-bit datapaths respectively.

We evaluate two extreme configurations of no inter-cluster communication and a fully-connected switch. Intermediate tradeoff points include lower bandwidth and sparse interconnect structures, which will result in overhead profiles that are in between the heatmaps shown in Figure 6.2 and Figure 6.3.

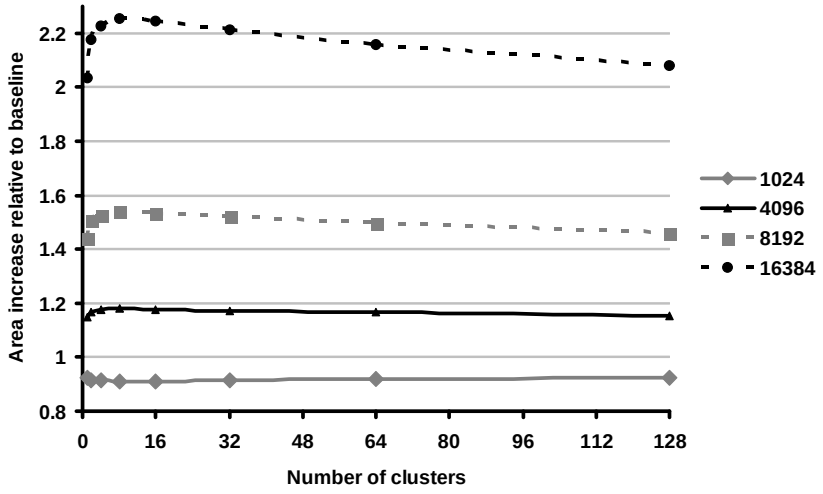
Figure 6.4 shows the sensitivity of the results presented above to the amount of on-chip storage in the sequencer instruction store (Figure 6.4(a)) and SRF (Figure 6.4(b)). The results are presented as a multiplicative factor relative to the baseline configuration shown in Figure 6.2(a). In the case of instruction storage, there is a minor correlation between the area overhead and the degree of ILP due to VLIW word length changes. This correlation is not shown in the figure and is less than 1% for the near-optimal $2-4$-ILP configurations. The area overhead of increasing the instruction store is a constant because it only changes when sequencer groups are added along the TLP dimension. Therefore, the overhead decreases sharply for a larger than baseline instruction store, and quickly approaches the baseline. Similarly, when the instruction store is smaller than the baseline, the area advantage diminishes as the number of clusters increases. The trends are different for changing the amount of data storage in the SRF as the SRF capacity scales with the number of ALUs and, hence, the number of clusters. As a result, the area overhead for increasing the SRF size is simply the ratio of the added storage area relative to the ALU area for each configuration. The area per ALU has an optimal point near $8-16$ clusters, and the SRF area overhead is maximal.

6.5 Performance Evaluation

In this section we evaluate the performance tradeoffs of utilizing ILP, DLP, and TLP mechanisms using three regular and three irregular control applications that are all throughput-oriented and have parallelism on par with the dataset size. The applications used here –



(a) Sensitivity to instruction store capacity



(b) Sensitivity to SRF capacity

Figure 6.4: Relative increase (multiplicative factor) of area per ALU as the number of clusters and capacity of storage structures is varied. In each sub-figure, the different lines indicate a change in capacity. Area is normalized to the baseline 64-bit configuration.

CONV2D, FFT3D, MATMUL, MOLE, FEM, and CDP – are explained in Section 3.1.

6.5.1 Experimental Setup

We used the simulation environment for scientific applications explained in Section 3.2, which has a 64-bit datapath. Our baseline configuration uses a 1MB SRF and 64 multiply-add floating point ALUs arranged in a (1-TLP,16-DLP,4-ILP) configuration along with 32 supporting iterative units, and allows for inter-cluster communication within a sequencer group at a rate of 1 word per cluster on each cycle.

Applications are modified from their original implementation described in Section 3.1 to support various ALU organizations with different number of clusters and sequencer groups. When there are multiple sequencer groups, dataset is distributed over sequencer groups to make the amount of computation similar among them. In all the applications, the amount of working set is determined by the size of SRF per cluster. In the case of MATMUL and irregular applications using duplicate removal, it also depends on the number of clusters per sequencer group if the inter-cluster switch is used to share dataset among clusters. These two factors – the size of SRF per cluster and the number of clusters per sequencer group – also determine the lengths and the access patterns of stream memory transfers providing difference in the amount of locality, which are further presented in the remainder of this section.

6.5.2 Performance Overview

The performance characteristics of the applications on the baseline configuration with the 1-TLP, 16-DLP, 4-ILP (1, 16, 4) configuration of Merrimac are reported in Table 3.3. In the remainder of this section we will report run time results relative to this baseline performance.

Figure 6.5 shows the run time results for our applications on 6 different ALU organizations. The run time is broken down into four categories: all sequencers are busy executing a kernel corresponding to compute bound portion of the application; at least one sequencer is busy and the memory system is busy, indicating load imbalance between the threads, but performance bound by memory throughput; all sequencers are idle and the memory system is busy during memory bound portions of the execution; at least one sequencer is busy and the memory system is idle due to load imbalance between the execution control threads.

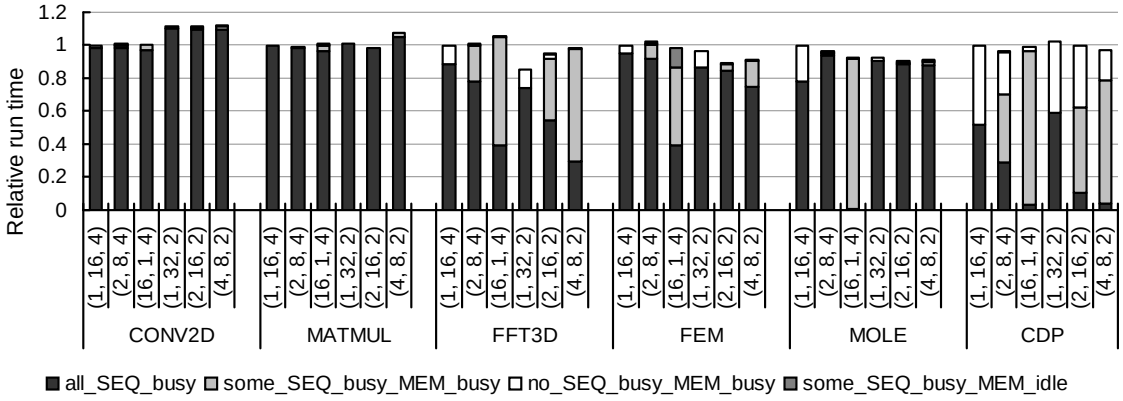


Figure 6.5: Relative run time normalized to the (1-TLP, 16-DLP, 4-ILP) configuration of the 6 applications on 6 ALU organizations. Total height of each bar indicates application run time.

The six configurations were chosen to allow us to understand and compare the effects of controlling the ALUs with different parallelism dimensions. The total number of ALUs was kept constant at 64 to allow direct performance comparison. The degree of ILP (ALUs per cluster) is chosen as either 2 or 4 as indicated by the analysis in Section 6.4, and the amount of TLP (number of sequencer groups) was varied from 1 to 16. The DLP degree was chosen to bring the total number of ALUs to 64. The total size of SRF was kept constant at 1MB as well unless mentioned otherwise.

Overall, we see that with the baseline configuration, the amount of ILP utilized is more critical to performance than the number of threads used. In FFT3D choosing a 2-ILP configuration improves performance by 15%, while the gain due to multiple threads peaks at 7.4% for FEM.

We also note that in most cases, the differences in performance between the configurations are under 5%. After careful evaluation of the simulation results and timing we conclude that much of that difference is due to the sensitivity of the memory system to the presented access patterns as shown in Chapter 4. Below we give a detailed analysis of the clear performance trends that are independent of the memory system fluctuations. We also evaluate the performance sensitivity to the SRF size and availability of inter-cluster communication.

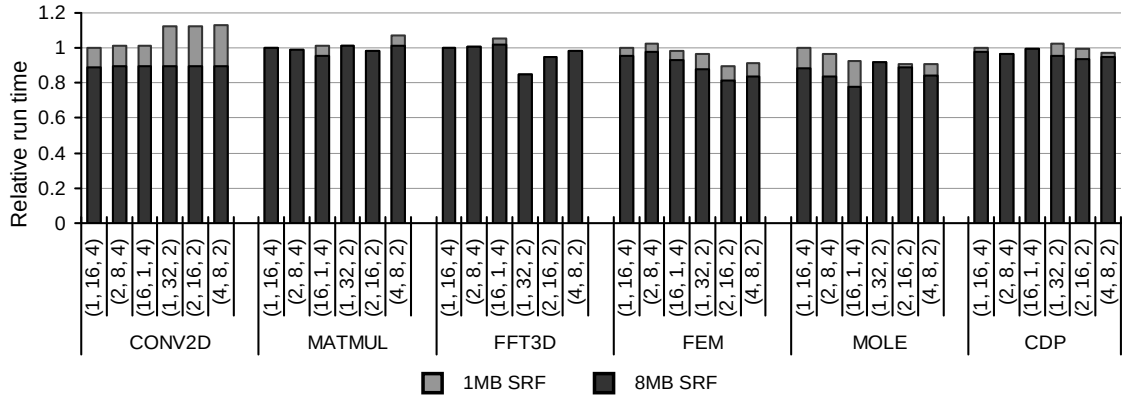


Figure 6.6: Relative run time normalized to the (1-TLP, 16-DLP, 4-ILP) configuration of the 6 applications comparing a 1MB SRF (light) and a 8MB SRF (dark).

6.5.3 ILP Scaling

Changing the degree of ILP (the number of ALUs per cluster) affects performance in two ways. First, the SRF capacity per cluster grows with the degree of ILP, and second, the VLIW kernel scheduler performs better when the number of ALUs it manages is smaller.

Because the total number of ALUs and SRF capacity is constant, a 2-ILP configuration has twice the number of clusters of a 4-ILP configuration. Therefore, a 2-ILP configuration has half the SRF capacity per cluster. The SRF size per cluster can influence the degree of locality that can be exploited and reduce performance. We see evidence of this effect in the run time of CONV2D. CONV2D processes the 2048×2048 input set in 2D blocks, and the size of the blocks affects performance. Each 2D block has a boundary that must be processed separately from the body of the block, and the ratio of boundary elements to body elements scales roughly as $1/N$ (where N is the row length of the block). The size of the block is determined by the SRF capacity in a cluster and the number of clusters. With a 128KB SRF, and accounting for double buffering and book-keeping data, the best performance we obtained for a 2-ILP configuration uses 16×16 blocks. With an ILP of 4 the blocks can be as large as 32×32 leading to an 11.5% performance advantage.

Looking at Figure 6.6 we see that increasing the SRF size to 8MB reduces the performance difference to 1%, since the proportion of boundary elements decreases and it is largely due to imperfect overlap between kernels and memory transfers at the beginning

Application	Kernel	Fraction of Run Time (1, 32, 2)	Fraction of Run Time (1, 16, 4)	Initiation Interval (1, 32, 2)	Initiation Interval (1, 16, 4)
FFT3D	FFT_128_stage13	20.7%	23.1%	28	16
	FFT_128_stage46	45.6%	47.2%	53	33
MOLE	mole_DR_XL_COND	98.2%	98.7%	125	68

Table 6.3: Scheduler results of software-pipeline initiation interval for critical kernels.

and the end of the simulation.

The effectiveness of the VLIW kernel scheduler also plays an important role in choosing the degree of ILP. Our register organization follows the stream organization presented in [RDK+00b], and uses a distributed VLIW register file with an LRF attached directly to each ALU port. As a result, the register file fragments when the number of ALUs is increased, reducing the effectiveness of the scheduler. Table 6.3 lists the software pipelined loop initiation interval (II) achieved for 3 critical kernels. We can clearly see the scheduling advantage of the 2-ILP configuration as the II of the 2-ILP configurations is significantly lower than twice the II achieved with 4 ALUs. This leads to the 15% and 8.3% performance improvements observed for FFT3D and MOLE respectively.

6.5.4 TLP Scaling

The addition of TLP to the stream processor has both beneficial and detrimental effects on performance. The benefits arise from performing irregular control for the applications that require it, and from the ability to mask transient memory system stalls with useful work. The second effect plays an important role in improving the performance of the irregular stream-program control of FEM. The detrimental effects are due to the partitioning of the inter-cluster switch into sequencer groups, and the synchronization overhead resulting from load imbalance.

First, we note that the addition of TLP has no impact on CONV2D beyond memory system sensitivity. We expected synchronization overhead to reduce performance slightly, but because the application is regular and compute bound, the threads are well balanced and proceed at the same rate of execution. Therefore, the synchronization cost due to waiting

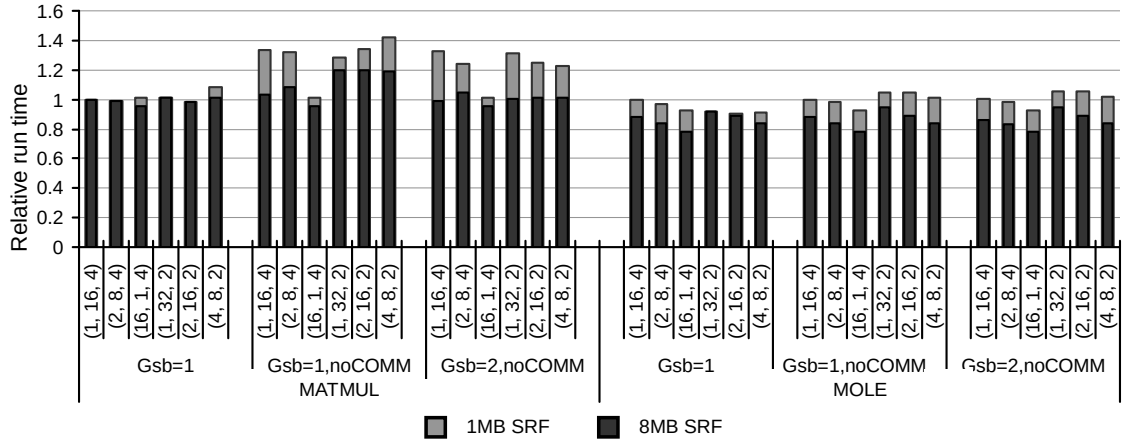


Figure 6.7: Relative run time normalized to the (1-TLP, 16-DLP, 4-ILP) configuration of MATMUL and MOLE with and without inter-cluster communication.

on threads to reach the barrier is minimal and the effect is masked by memory system performance fluctuation.

The performance of MATMUL strongly depends on the size of the sub-matrices processed. For each sub-matrix, the computation requires $O(N_{sub}^3)$ floating point operations and $O(N_{sub}^2)$ words, where each sub-matrix is $N_{sub} \times N_{sub}$ elements. Therefore, the larger the blocks the more efficient the algorithm. With the baseline configuration, which supports direct inter-cluster communication, the sub-matrices can be partitioned across the entire SRF within each sequencer group. When the TLP degree is 1 or 2, the SRF can support 64×64 blocks, but only 32×32 blocks for a larger number of threads. In all our configurations, however, the memory system throughput was sufficient and the performance differences are due to a more subtle reason. The kernel computation of MATMUL requires a large number of SRF accesses in the VLIW schedule. When the sub-matrices are partitioned across the SRF, some references are serviced by the inter-cluster switch instead of the SRF. This can most clearly be seen in the (4,8,2) configuration, in which the blocks are smaller and therefore data is reused less requiring more accesses to the SRF. The same problem does not occur in the (16,1,4) configuration because it provides higher SRF bandwidth in order to support the memory system throughput (please refer to discussion of stream buffers in Section 6.2). The left side of Figure 6.7 presents the run time results of

MATMUL when the inter-cluster switch is removed, such that direct communication between clusters is not permitted, and for both a 1MB and a 8MB SRF. Removing the switch prevents sharing of the SRF and forces 16×16 sub-matrices with the smaller SRF size, with the exception of (16,1,4) that can still use a 32×32 block. As a result, performance degrades significantly. Even when SRF capacity is increased, performance does not match that of (16,1,4) because of the SRF bandwidth issue mentioned above. The group of bars with $Gsb = 2$ increases the SRF bandwidth by a factor of two for all configurations, and coupled with the larger SRF equalizes the run time of the applications if memory system sensitivity is ignored.

We see two effects of TLP scaling on the performance of FFT3D. First, FFT3D demonstrates the detrimental synchronization and load imbalance effect of adding threading support. While both the (1,16,4) and (1,32,2) configurations, which operate all clusters in lockstep, are memory bound, at least one sequencer group in all TLP-enabled configurations is busy at all times.

The stronger trend in FFT3D is that increasing the number of threads reduces the performance of the 2-ILP configurations. This is a result of the memory access pattern induced, which includes a very large stride when processing the Z dimension of the 3D FFT (Figure 6.8). Introducing more threads changes the blocking of the application and limits the number of consecutive words that are fetched from memory for each large stride. Here the amount of spatial locality in memory system is proportional to the number of clusters because the array is stored in row-major order making data in a different plane (with different Z values) far apart in memory address space. This effect has little to do with the control aspects of the ALUs and is specific to FFT3D.

FEM demonstrates both the effectiveness of dynamically masking unexpectedly long memory latencies and the detrimental effect of load imbalance. The structure of the FEM application limits the degree of double buffering that can be performed in order to tolerate memory latencies. As a result, computation and memory transfers are not perfectly overlapped, leading to performance degradation. We can see this most clearly in the (1, 32, 2) configuration of FEM in Figure 6.5, where there is a significant fraction of time when all clusters are idle waiting on the memory system (white portion of bar). When multiple threads are available this idle time in one thread is masked by execution in another thread,

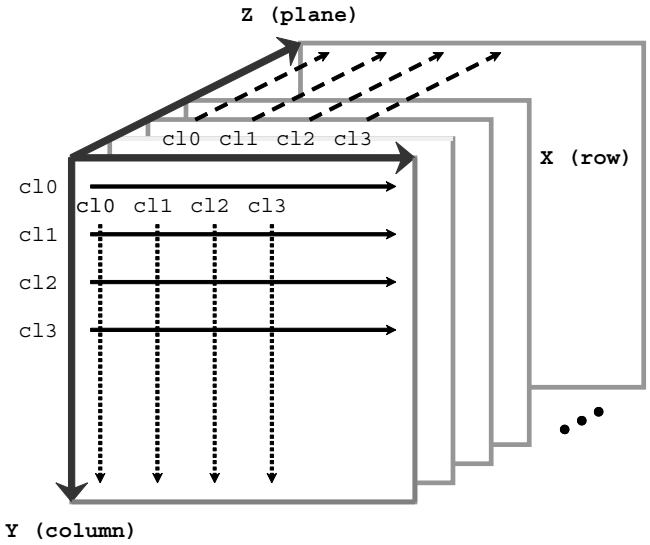


Figure 6.8: Array access patterns of FFT3D. Arrays are stored in row-major order – sorted by planes, columns, then rows. So when FFTs are computed over Z dimension there is least amount of spatial locality in memory, and it is proportional to the number of clusters.

reducing the time to solution and improving performance by 7.4% for the (2, 16, 2). Increasing the number of threads further reduces performance by introducing load imbalance between the threads. FEM contains several barrier synchronization points, and load imbalance increases the synchronization time. This is most clearly seen in the extreme (16,1,4) configuration.

Based on the results presented in [Ere06] we expect the benefit of TLP for irregular control to be significant (maximum 20% and 30% for MOLE and CDP respectively). Our results, however, show much lower performance improvements. In the baseline configuration, with 1MB SRF and inter-cluster communication, we see no advantage at all to utilizing TLP.

For MOLE the reason relates to the effective inter-cluster communication enabled by SIMD. In the baseline configuration with 2-ILP, the algorithm utilizes the entire 1MB of the SRF for exploiting locality (cross-lane duplicate removal method of [Ere06]) and is thus able to match the performance of the MIMD enabled organizations. As discussed earlier, the 4-ILP configurations perform poorly because of a less effective VLIW schedule. If

we remove the inter-cluster switch for better area scaling (Section 6.4), the performance advantage of MIMD grows to a significant 10% (Figure 6.7). Similarly, increasing the SRF size to 8MB provides enough state for exploiting locality within the SRF space of a single cluster and the performance advantage of utilizing TLP is again 10% (Figure 6.6).

In the case of CDP, the advantage of TLP is negated because the application is memory throughput bound. Even though computation efficiency is improved, the memory system throughput limits performance as is evident by the run time category corresponding to all sequencers being idle while the memory system is busy (white bars in Figure 6.5).

Chapter 7

Conclusion

Stream processing has been shown as an attractive solution to map a large body of scientific and multimedia applications to modern VLSI implementation technology by achieving high performance and efficiency without losing programmability. In a stream program, datasets are formatted into streams. Then communication and computation of each block are explicitly specified by stream transfer and kernel invocation exposing locality and parallelism. Subsequently, a stream processor exploits parallelism by utilizing many ALUs and memory channels as well as overlapping computation and global communication. This achieves high performance and exploits locality by providing storage hierarchy and combining multiple memory requests into a single DRAM burst. In this dissertation, we studied memory and control organizations of stream processors in search of memory system structures and ALU control combinations leading to better application performance while using the same amount of hardware resources.

We explored the design space of streaming memory systems in light of the DRAM technology trends of rapidly increasing DRAM bandwidth and a very slow improvement in DRAM latency. We have shown that these trends lead to growing DRAM access granularity and high sensitivity of throughput to application access patterns. Specifically, we identified the importance of maintaining locality in the reference pattern applied to the DRAM in order to achieve high performance. This reduced internal DRAM bank conflicts and read-write turnaround penalties. We presented a detailed taxonomy of the memory system design space, and examined two hardware configurations that are able to exploit locality.

The first is a single, wide address generator architecture that operates on one access thread or stream at a time. The single-AG configuration takes advantage of locality, however, the reliance on only a single thread can lead to a load imbalance between memory channels, and hence, to reduced performance. Based on these observations, we designed the novel channel-split hardware mechanism that is able to achieve locality and balance load across multiple channels at the same time.

We introduced the hardware scatter-add operation for stream architectures. Scatter-add allows global accumulation operations to be executed efficiently and in parallel, while ensuring the atomicity of each addition and the correctness of the final result. This type of operation is common in many application domains such as histogram computations in signal and image processing, and expressing superposition in scientific applications for calculating interactions and performing algebraic computations. We described the general scatter-add micro-architecture and showed that the availability of this hardware mechanism allows the programmer to choose algorithms that are prohibitively expensive to implement entirely in software.

Then we extended the stream architecture to support and scale along the three main dimensions of parallelism. VLIW instructions utilize ILP to drive multiple ALUs within a cluster, clusters are grouped under SIMD control of a single sequencer exploiting DLP, and multiple sequencer groups rely on TLP. We explored the scaling of the architecture along these dimensions as well as the tradeoffs between choosing different values of ILP, DLP, and TLP control for a given set of ALUs. Our methodology provides a fair comparison of the different parallelism techniques within the scope of applications with scalable parallelism, as the same execution model and basic implementation were used in all configurations. We developed a detailed hardware cost model based on area normalized to a single ALU, and showed that adding TLP support is beneficial as the number of ALUs on a processor scales above $32-64$. However, the cost of increasing the degree of TLP to an extreme of a single sequencer per cluster can be significant and ranges between $15-86\%$. Our performance evaluation shows that a wide range of numerical applications with scalable parallelism are fairly insensitive to the type of parallelism exploited. This is true for both regular and irregular control algorithms, and overall, the performance speedup is in the $0.9-1.15$ range. We explained in detail the many subtle sources of the performance

difference and discussed the sensitivity of the results.

7.1 Future Work

The work presented in this dissertation leads to a number of other interesting fields of future research:

Memory System Analysis

In Chapter 4, we used the memory accesses extracted from stream memory transfers of multimedia and scientific applications to study the design space of streaming memory systems. However, many stream processors share DRAM to serve requests from both stream and scalar core. In a multi-processor system, the memory system in each node also needs to process requests from other nodes. Therefore, in the future, it would be important to analyze the performance of memory systems using all these heterogeneous sources of memory accesses and to design memory systems that achieve good performance on a wide variety of access patterns.

Scatter-add

Chapter 5 analyzed the performance implications of a single-node stream processor. We also described multi-node implementation and explored the scalability of scatter-add as well using on-chip caching to optimize multi-node performance. An interesting area of future work would be to enhance hardware scatter-add to allow efficient computation of scans (parallel prefix operations) and implement system wide synchronization primitives for stream architectures. Another area of future work would be to consider an optimization to the multi-node cached algorithm that will arrange the nodes in a logical hierarchy and allow the combining across node to occur in logarithmic instead of linear complexity.

Applications

In Chapter 6, we characterized numerical applications using three criteria: whether they are throughput oriented or present real-time constraints, whether the parallelism scales with the dataset or not, and whether they have irregular or regular flow in algorithm. However, we focused on the applications which are throughput oriented and have the parallelism scaling with the dataset during experimental study. Therefore, one interesting area of future work would be to analyze the effects of real-time constrains in stream processors, and to find good combinations of ALU control structure or design new architectural supports for these real-time applications. Also mapping applications with limited-parallelism or at least limited data parallelism to stream architectures will show different performance characteristics between control organizations, possibly in favor of multiple sequencer groups per processor. These applications may perform better in more heterogeneous configurations where the amount of SIMD or VLIW controlling ALUs is different between sequencer groups.

Scaling Inter- and Intra-cluster Switches

Throughout the dissertation it was assumed that the inter-cluster switch and the intra-cluster switch are fully connected. In the future, it would be important to study the effects of applying sparser interconnects for these switches. For example, we can divide ALUs in a cluster into multiple groups and provide different amounts of connectivity within groups and between groups, which will give more challenges to the VLIW kernel scheduler. Making connectivity among clusters sparser provides another interesting tradeoff between hardware cost, complexity, and performance since the throughput of interconnect depends on the communication patterns generated from clusters. This makes control logic more complex while it saves more space for datapath than fully-connected crossbars.

Bibliography

[ABHS89] M. C. August, G. M. Brost, C. C. Hsiung, and A. J. Schiffleger. Cray X-MP: The Birth of a Supercomputer. *IEEE Computer*, 22(1):45–54, January 1989.

[AD05] Jung Ho Ahn and William J. Dally. Data Parallel Address Architecture. *Computer Architecture Letters*, 4, 2005.

[ADK+04] Jung Ho Ahn, William J. Dally, Brucek Khailany, Ujval J. Kapasi, and Abhishek Das. Evaluating the Imagine Stream Architecture. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 14–25, Munich, Germany, 2004.

[AED05] Jung Ho Ahn, Mattan Erez, and William J. Dally. Scatter-Add in Data Parallel Architectures. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture*, San Francisco, California, February 2005.

[AED06] Jung Ho Ahn, Mattan Erez, and William J. Dally. The Design Space of Data-Parallel Memory Systems. In *SC'06*, Tampa, Florida, November 2006.

[ATI05] ATI. Radeon X1800 Memory Controller, 2005. http://www.ati.com/products/radeonx1k/whitepapers/X1800_Memory_Controller_Whitepaper.pdf.

[ATI06] ATI. RADEON X1900 Graphics Technology - GPU Specifications, 2006. http://www.ati.com/products/radeonx1900/specs.html.

[Bac78] John Backus. Can Programming be Liberated from the von Neumann Style? *Communications of the ACM*, 21(8):613–641, August 1978.

[BCG+98] J. Boisseau, L. Carter, K. Gatlin, A. Majumdar, and A. Snavely. NAS benchmarks on the Tera MTA. In *Proceedings of the Multithreaded Execution Architecture and Compilation Workshop*, February 1998.

[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.

[BGM+00] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th International Symposium on Computer Architecture*, Vancouver, Canada, June 2000.

[BR98] S. Bae and S. Ranka. Array Combining Scatter Functions on Coarse-Grained, Distributed-Memory Parallel Machines. In *International Parallel Processing Symposium and Sympoisum on Parllel and Distributed Processing*, March 1998.

[CB02] Tirupathi R. Chandrupatla and Ashok D. Belegundu. *Introduction to Finite Elements in Engineering*. Prentice Hall, 2002.

[CBZ90] Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha. Scan primitives for vector computers. In *Proceedings Supercomputing '90*, pages 666–675, November 1990.

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

[Clo53] Charles Clos. A Study of Non-Blocking Switching Networks. *Bell Systems Technical Journal*, 32:406–424, 1953.

[CMCA98] L. Catabriga, M. Martins, A. Coutinho, and J. Alves. Clustered Edge-by-Edge Preconditioners for Non-Symmetric Finite Element Equations. In *4th World Congress on Computational Mechanics*, Buenos Aires, Argentina, 1998.

[CS86] Tony Cheung and James E. Smith. A simulation study of the CRAY X-MP memory system. *IEEE Transactions on Computers*, 35(7):613–622, 1986.

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

[Dav01] Brian Thomas Davis. *Modern DRAM Architectures*. PhD thesis, The University of Michigan, Ann Arbor, 2001.

[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.

[DDM06] Abhishek Das, William J. Dally, and Peter Mattson. Compiling for Stream Processing. In *The 15th International Conference on Parallel Architectures and Compilation Techniques*, Seattle, Washington, September 2006.

[DHE+03] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonte, J. Ahn, N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck. Merrimac: Supercomputing with streams. In *SC'03*, Phoenix, Arizona, November 2003.

[DWP02] E. Darve, M.A. Wilson, and A. Pohorille. Calculating Free Energies using a Scaled-Force Molecular Dynamics Algorithm. *Molecular Simulation*, 28(1–2):113–144, 2002.

[EAG+04] Mattan Erez, Jung Ho Ahn, Ankit Garg, William J. Dally, and Eric Darve. Analysis and Performance Results of a Molecular Modeling Application on Merrimac. In *SC'04*, Pittsburgh, Pennsylvaniva, November 2004.

[EJKD05] Mattan Erez, Nuwan Jayasena, Timothy J. Knight, and William J. Dally. Fault Tolerance Techniques for the Merrimac Streaming Supercomputer. In *SC'05*, Seattle, Washington, November 2005.

[ELP05] ELPIDA Memory, Inc. 512M bits XDR^{TM} DRAM, 2005. http://www.elpida.com/pdfs/E0643E20.pdf.

[Ere06] Mattan Erez. *Merrimac - High-Performance and High-Efficient Scientific Computing with Streams*. PhD thesis, Stanford University, 2006.

[ESS92] D. Elliott, M. Snelgrove, and M. Stumm. Computational RAM: A memory-SIMD hybrid and its application to DSP. In *Custom Integrated Circuits Conference*, May 1992.

[FAD+05] B. Flachs, S. Asano, S.H. Dhong, P. Hotstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H. Oh, S.M. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, and N. Yano. A Streaming Processing Unit for a Cell Processor. In *2005 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pages 134–135, February 2005.

[GGK+82] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The NYU Ultracomputer: designing a MIMD, shared-memory parallel machine. In *Proceedings of the 9th Annual Symposium on Computer Architecture*, pages 27–42, April 1982.

[GHL+02] Brian R. Gaeke, Parry Husbands, Xiaoye S. Li, Leonid Oliker, Katherine A. Yelick, and Rupak Biswas. Memory-Intensive Benchmarks: IRAM vs. Cache-Based Machines. In *International Parallel and Distributed Processing Symposium*, April 2002.

[GR05] Jayanth Gummaraju and Mendel Rosenblum. Stream Programming on General-Purpose Processors. In *Proceedings of the 38th annual ACM/IEEE international symposium on Microarchitecture*, Barcelona, Spain, November 2005.

[HBJ+02] M. S. Hrishikesh, Doug Burger, Norman P. Jouppi, Stephen W. Keckler, Keith I. Farkas, and Premkishore Shivakumar. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 14–24, Anchorage, Alaska, May 2002.

[HD98] Hoare and Dietz. A Case for Aggregate Networks. In *Proceedings of the 11th International Parallel Processing Symposium*, pages 162–166, Geneva, Switzerland, April 1998.

[HHS+00] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael Chen, and Kunle Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, 2000.

[HMS+99] Sung I. Hong, Sally A. McKee, Maximo H. Salinas, Robert H. Klenke, James H. Aylor, and William. A. Wulf. Access Order and Effective Bandwidth for Streams on a Direct Rambus Memory. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, pages 80–89, Orlando, Florida, January 1999.

[HPF93] High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, High Performance Fortran Forum, Houston, TX, 1993.

[HT93] W. Daniel Hillis and Lewis W. Tucker. The CM-5 Connection Machine: a scalable supercomputer. *Communications of the ACM*, 36(11):31–40, 1993.

[JEAD04] Nuwan Jayasena, Mattan Erez, Jung Ho Ahn, and William J. Dally. Stream Register Files with Indexed Access. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, Madrid, Spain, February 2004.

[Kap04] Ujval J. Kapasi. *Conditional Techniques for Stream Processing Kernels*. PhD thesis, Stanford University, 2004.

[KBH+04] Ronny Krashinsky, Christopher Batten, Mark Hampton, Steve Gerding, Brian Pharris, Jared Casper, and Krste Asanovic. The Vector-Thread Architecture. In *Proceedings of the 31st International Symposium on Computer Architecture*, pages 52–63, Munich, Germany, June 2004.

[KDR+01] Brucek Khailany, William J. Dally, Scott Rixner, Ujval J. Kapasi, Peter Mattson, Jinyung Namkoong, John D. Owens, Brian Towles, and Andrew Chang. Imagine: Media Processing with Streams. *IEEE Micro*, pages 35–46, March/April 2001.

[KDR+03] Brucek Khailany, William J. Dally, Scott Rixner, Ujval J. Kapasi, John D. Owens, and Brian Towles. Exploring the VLSI Scalability of Stream Processors. In *Proceedings of the 9th Symposium on High Performance Computer Architecture*, Anaheim, California, February 2003.

[KDTG05] John Kim, William J. Dally, Brian Towles, and Amit K. Gupta. Microarchitecture of a High-Radix Router. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 420–431, Madison, Wisconsin, June 2005.

[KHY+99] Y. Kang, W. Huang, S. M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas. FlexRAM: Toward an Advanced Intelligent Memory System. In *International Conference on Computer Design*, pages 192–201, October 1999.

[KM89] Les Kohn and Neal Margulis. Introducing the Intel i860 64-bit Microprocessor. *IEEE Micro*, 9(4):15–30, August 1989.

[Koz02] Christoforos Kozyrakis. *Scalable vector media-processors for embedded systems*. PhD thesis, University of California, Berkeley, 2002.

[KP03] Christos Kozyrakis and David Patterson. Overcoming the Limitations of Conventional Vector Processors. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 399–409, San Diego, California, June 2003.

[KPP+97] Christoforos Kozyrakis, Stylianos Perissakis, David Patterson, Thomas Anderson, Krste Asanovic, Neal Cardwell, Richard Fromm, Jason Golbus, Benjamin Gribstad, Kimberly Keeton, Randi Thomas, Noah Treuhaft, and Katherine Yelick. Scalable Processors in the Billion-Transistor Era: IRAM. *Computer*, 30(9):75–78, 1997.

[KRD+03] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. Programmable Stream Processors. *IEEE Computer*, pages 54–62, August 2003.

[Kro81] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 81–87, Minneapolis, Minnesota, 1981.

[KS93] R. E. Kessler and J. L. Schwarzmeier. Cray T3D: a new dimension for Cray Research. In *COMPCON*, pages 176–182, February 1993.

[LAD+96] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St Pierre, David S. Wells, Monica C. Wong-Chan, Shaw-Wen Yang, and Robert Zak. The network architecture of the Connection Machine CM-5. *Journal of Parallel and Distributed Computing*, 33(2):145–158, 1996.

[Lit61] John D. C. Little. A Proof for the Queuing Formula: L= λW. *Operations Research*, 9(3):383–387, May–June 1961.

[LL97] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server . In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, June 1997.

[LM87] E. A. Lee and D. G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, January 1987.

[LMB+04] Francois Labonte, Peter Mattson, Ian Buck, Christos Kozyrakis, and Mark Horowitz. The Stream Virtual Machine. In *Proceedings of the 2004 International Conference on Parallel Architectures and Compilation Techniques*, Antibes Juan-les-pins, France, September 2004.

[MDR+00] Peter Mattson, William J. Dally, Scott Rixner, Ujval J. Kapasi, and John D. Owens. Communication Scheduling. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 82–92, Cambridge, Massachusetts, November 2000.

[Mic06] Micron. Reduced Latency DRAM, September 2006. http://download. micron.com/pdf/datasheets/rldram/256M_16_32_RLDRAM.pdf.

[MPJ+00] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart Memories: A Modular Recongurable Architecture. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 161–171, Vancouver, Canada, June 2000.

[nVI06] nVIDIA. *n*VIDIA GeFORCE 7 Series GPUs Specifications, 2006. http://www.nvidia.com/object/7_series_techspecs.html.

[OCS98] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Active Pages: A Computation Model for Intelligent Memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 192–203, Barcelona, Spain, June 1998.

[OFW99] Stuart Oberman, Greg Favor, and Fred Weber. AMD 3DNow! Technology: Architecture and Implementations. *IEEE Micro*, 19(2):37–48, 1999.

[OL85] Wilfried Oed and O. Lange. On the Effective Bandwidth of Interleaved Memories in Vector Processing Systems. *IEEE Transactions on Computers*, 34(10):949–957, 1985.

[Ope04] OpenGL Architecture Review Board. *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.4*. Addison Wesley Professional, 2004.

[Owe02] John D. Owens. *Computer Graphics on a Stream Architecture*. PhD thesis, Stanford University, 2002.

[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 *Proceedings of the IEEE International Solid-State Circuits Conference*, pages 184–185, San Francisco, California, February 2005.

[PJS97] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 206–218, Denver, Colorado, 1997.

[PMTH01] Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A Real-Time Procedural Shading System for Programmable Graphics Hardware. In *Proceedings of ACM SIGGRAPH*, pages 159–170, August 2001.

[PPE+97] Yale N. Patt, Sanjay J. Patel, Marius Evers, Daniel H. Friendly, and Jared Stark. One Billion Transistors, One Uniprocessor, One Chip. *Computer*, 30(9):51–57, 1997.

[PTVF96] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brain P. Flannery. *Numerical Recipes in Fortran 90*. Cambridge University Press, 1996.

[Rau91] B. Ramakrishna Rau. Pseudo-Randomly Interleaved Memory. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 74–83, Toronto, Canada, 1991.

[RDK+00a] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory Access Scheduling. In *Proceedings of the 27th International Symposium on Computer Architecture*, Vancouver, Canada, Jun 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 *Proceedings of the 6th International Symposium on High Performance Computer Architecture*, pages 375–386, Toulouse, France, January 2000.

[Ric03] Iain E.G. Richardson. *H.264 and MPEG-4 Video Compression: Video Coding for Next-generation Multimedia*. John Wiley & Sons, Ltd, 2003.

[RJSS97] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and Jim Smith. Trace Processors. In *Proceedings of the 30th ACM/IEEE International Symposium on Microarchitecture*, pages 138–148, North Carolina, 1997.

[Rus78] Richard M. Russell. The CRAY-1 Computer System. *Communications of the ACM*, 21(1):63–72, Janunary 1978.

[Saa03] Youssef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial & Applied Mathematics, second edition, 2003.

[Sam05] Samsung Electronics. 512Mbit SDRAM Specification, October 2005. http://www.samsung. com/Products/Semiconductor/DRAM/.

[Sam06a] Samsung Electronics. 512Mbit DDR SDRAM Specification, March 2006. http://www.samsung. com/Products/Semiconductor/DRAM/.

[Sam06b] Samsung Electronics. 512Mbit DDR2 SDRAM, October 2006. http://www.samsung.com/ Products/Semiconductor/DRAM/.

[Sam06c] Samsung Electronics. 512Mbit GDDR3 SDRAM, June 2006. http://www.samsung.com/ Products/Semiconductor/GraphicsMemory/.

[SBV95] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 414–425, S. Margherita Ligure, Italy, 1995.

[Sco96] Steven L. Scott. Synchronization and communication in the T3E multiprocessor. In *Proceedings of the 7th International Conference on Architectural*

Support for Programming Languages and Operating Systems, pages 26–36, Cambridge, Massachusetts, 1996.

[SNL+03] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 422–433, San Diego, California, June 2003.

[Soh93] G. S. Sohi. High-Bandwidth Interleaved Memories for Vector Processors - A Simulation Study. *IEEE Transactions on Computers*, 42(1):34–44, 1993.

[TEL95] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 392–403, S. Margherita Ligure, Italy, June 1995.

[TH99] Shreekant (Ticky) Thakkar and Tom Huff. The Internet Streaming SIMD Extensions. *Intel Technology Journal*, (Q2):8, May 1999.

[TKA02] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, Grenoble, France, April 2002.

[Wil93] Timothy J. Williams. 3D Gyrokinetic Particle-In-Cell Simulation of Fusion Plasma Microturbulence on Parallel Computers . In *High Performance Computing: Grand Challenges for Computer Simulation*, March/April 1993.

[WRRF] Michael Woodacre, Derek Robb, Dean Roe, and Karl Feind. The SGI Altix 3000 Global Shared-Memory Architecture. SGI White Paper, 2003.

[WTS+97] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua,

Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to Software: Raw Machines. *IEEE Computer*, pages 86–93, September 1997.