ENERGY-EFFICIENT LATENCY TOLERANCE FOR 1000-CORE DATA PARALLEL PROCESSORS WITH DECOUPLED STRANDS

BY

NEAL CLAYTON CRAGO

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical and Computer Engineering in the Graduate College of the University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Doctoral Committee:

Professor Sanjay J. Patel, Chair
Professor Wen-mei W. Hwu
Associate Professor Steven S. Lumetta
Associate Professor Deming Chen
This dissertation presents a novel decoupled latency tolerance technique for 1000-core data parallel processors. The approach focuses on developing instruction latency tolerance to improve performance for a single thread. The main idea behind the approach is to leverage the compiler to split the original thread into separate memory-accessing and memory-consuming instruction streams. The goal is to provide latency tolerance similar to high-performance techniques such as out-of-order execution while leveraging low hardware complexity similar to an in-order execution core.

The research in this dissertation supports the following thesis: Pipeline stalls due to long exposed instruction latency are the main performance limiter for cached 1000-core data parallel processors. Leveraging natural decoupling of memory-access and memory-consumption, a serial thread of execution can be partitioned into strands providing energy-efficient latency tolerance.

This dissertation motivates the need for latency tolerance in 1000-core data parallel processors and presents decoupled core architectures as an alternative to currently used techniques. This dissertation discusses the limitations of prior decoupled architectures, and proposes techniques to improve both latency tolerance and energy-efficiency. Finally, the success of the proposed decoupled architecture is demonstrated against other approaches by perform-
ing an exhaustive design space exploration of energy, area, and performance using high-fidelity performance and physical design models.
I dedicate this dissertation to my wife Christine.
ACKNOWLEDGMENTS

The work presented in this dissertation would not be possible without the technical and personal support of others. I have been very blessed as a doctoral student to constantly be surrounded by very able and intelligent people. That simply would not have been possible if I hadn’t studied at the University of Illinois.

I would like to thank the Rigel group for their efforts in developing the initial compiler and simulation infrastructure which I used as a basis for the architectural exploration presented in this dissertation. I would like to specifically thank John Kelm, who encouraged me early on in the formation of the early ideas presented in this work. Many thanks to Professor Steve Lumetta, Daniel Johnson, Matthew Johnson, and William Tuohy for their useful feedback on drafts and during presentations of this work.

I would like to thank my adviser Sanjay Patel for his encouragement and willingness to fund me while I worked on my dissertation. His grand challenge and expectation of greatness in the field of computer architecture encouraged me to fully develop my ideas and create high-quality research.

I also would like to thank Analog Devices for their generous support through fellowships during my first two years in graduate school. Many thanks to the Department of Electrical and Computer Engineering for providing me funding and the opportunity to teach through assistantships. I would like to thank the Semiconductor Research Corporation (SRC) which
funded this dissertation research. Finally, I would like to thank the Trusted ILLIAC Center at the Information Trust Institute for their contribution of use of the computing cluster, without which most of the contributions of this dissertation never would have happened.

Much of my inspiration to pursue my doctorate is due to my family. I have always had much admiration for my family and their passion and success in academia. My great-grandfather Raymond Starbuck was a professor of dairy science at the Ohio State University in Columbus, Ohio, and my great-grandfather Samuel Kiehl was a professor of chemistry at Columbia University in New York City, New York. Many other relatives have devoted their lives to education, including my Uncle David Crago and my grandmother Ellen Crago.

Most importantly has been the inspiration of my father and mother, who primarily inspired and encouraged me to pursue my interest in computer engineering from a young age. My father Steven first introduced me to programming when I was 8 years old, while encouraging my curiosity and initial skepticism for established thought. My mother Louise has always pushed me intellectually from a young age, taking time away from her own career as a professor of mathematics to teach my brothers and me. Most importantly, I thank Steven and Louise for always encouraging me to exceed my own expectations, and teaching me that with hard work and faith in the Lord all things can be done.

I would also like to thank my wife Christine, to whom I owe the greatest amount of credit for this work. Her constant and unconditional encouragement, friendship, and love throughout this time in graduate school have propelled me further and spurred my creativity and passion for research. Only through her personal experience while pursuing her own graduate degrees
did she truly understand all the long nights and time away from each other, and for that I am grateful.

Finally, I would like thank my Lord, my Savior, and God Jesus Christ for His grace, love, and discipline during my whole life. I pray that any success I may have in my future would honor and glorify His name.
TABLE OF CONTENTS

LIST OF ABBREVIATIONS ................................................. x

CHAPTER 1 INTRODUCTION .............................................. 1

CHAPTER 2 BACKGROUND ............................................... 7
  2.1 Rigel .......................................................... 7
  2.2 Contemporary Latency Tolerance Approaches ................. 12

CHAPTER 3 DECOUPLED ARCHITECTURES ......................... 20
  3.1 Memory Latency Tolerance through Strands ................. 21
  3.2 Control Flow Requirement .................................. 23
  3.3 Data Communication Requirement .......................... 25
  3.4 Decoupled Access/Execute Implementation ................ 26
  3.5 Decoupling as an Emerging Technique ...................... 27

CHAPTER 4 EVALUATION METHODOLOGY ......................... 29
  4.1 Performance and Energy Modeling ......................... 30
  4.2 Code Patterns for Data Parallel Applications .......... 33
  4.3 Microbenchmarks ............................................. 35
  4.4 Benchmarks .................................................. 37

CHAPTER 5 A MODERN DECOUPLED ARCHITECTURE ............ 40
  5.1 Architecture Overview ..................................... 41
  5.2 Evaluation ................................................... 44

CHAPTER 6 INCREASING STRAND-LEVEL PARALLELISM ......... 48
  6.1 Loss of Decoupling Events ................................ 48
  6.2 Memory Indirection ........................................... 50
  6.3 Floating Point Latency ....................................... 53
  6.4 Implementation .............................................. 54
  6.5 Evaluation ................................................... 56

CHAPTER 7 COMPILER-DIRECTED INSTRUCTION STEERING .... 62
  7.1 Hardware Assisted Strand Partitioning .................... 62
  7.2 Implementation .............................................. 64
  7.3 Evaluation ................................................... 69

viii
LIST OF ABBREVIATIONS

AP Access processor
CMP Chip multiprocessor
DAE Decoupled access/execute
EP Execute processor
FLOPS Floating-point operations per second
GPU Graphics processing unit
ILP Instruction-level parallelism
INO In-order
L1 Level one
L2 Level two
LOD Loss-of-decoupling
MIMD Multiple-instruction multiple-data
MLP Memory-level parallelism
OOO Out-of-order execution
S# Strand number
SIMD Single-instruction multiple-data
SMT Simultaneous multithreading
TLP Thread-level parallelism
VLIW Very long instruction word
CHAPTER 1
INTRODUCTION

Execution stalls due to memory and functional unit latency are the limiting factor for performance in highly parallel workloads. Most of the performance lost in a 1024-core data parallel processor is attributable to the memory system, rather than fetch, branch prediction, or functional unit latencies. Removing pipeline stalls due to memory latency can double performance in some cases, and tolerating functional unit latency also increases performance significantly. This indicates the importance of efficient mechanisms for tolerating memory and function unit latency. There have been many proposed solutions for tolerating instruction latency, including data prefetching, more complex cache hierarchies, multithreading, and more complex core pipelines. This dissertation focuses attention on core pipelines, and evaluates how current techniques tolerate memory and functional unit latency, and what limits their suitability for deployment in data parallel processors.

This dissertation presents a novel implementation of a decoupled architecture that reaches and can exceed the performance of other latency tolerance techniques such as multithreading and out-of-order execution on highly parallel benchmarks while maintaining energy efficiency and low hardware complexity. Decoupled architectures leverage the compiler to separate a single thread of execution into multiple instruction streams that can be executed concurrently. The thread is split into memory-accessing and memory-consuming instruction streams, which are called strands. The strands ex-
execute in separate hardware contexts while following the same control path through the program. Memory-accessing strands communicate data values from memory to the memory-consuming strands and have the ability to non-speculatively execute substantially ahead of the memory-consuming strands, thus tolerating memory and functional unit latency. By leveraging the compiler to extract some degree of parallelism, decoupled architectures can have less complex hardware than out-of-order processors, while reducing the impact of cache contention and need for increasing number of threads found in multithreading.

Despite the promise of decoupled architectures for low-complexity latency tolerance, significant performance and energy inefficiencies limit broad applicability. Specifically, code patterns commonly found in data-parallel applications can severely limit the latency tolerance capability and suitability of traditional decoupled architectures. Additionally, proper execution in traditional decoupled architectures requires a substantial number of overhead instructions, increasing energy consumption significantly. In the proposed decoupled architecture, performance and energy pitfalls are combated by extracting more parallelism, enabling control speculation, and enabling the compiler and lightweight hardware to reduce instruction overhead. When combining all these techniques together, energy consumption can be reduced and performance improved significantly over prior decoupled architectures across data-parallel benchmarks. Additionally, combining multithreading and the proposed decoupled architecture is explored as an alternative for improving energy efficiency.

High-fidelity performance and physical design models are leveraged to perform a comprehensive design space exploration and compare the energy efficiency of common latency tolerance techniques on a 1024-core data paral-
By designing a decoupled architecture specifically for energy efficiency, robust energy-efficiency across a wide range of code patterns is provided. The proposed decoupled architecture improves energy-efficiency over other techniques by 28% to 89% on data parallel benchmarks. A hybrid of multithreading and decoupling can improve energy-efficiency by another 14% on average across data parallel benchmarks.

The contributions of this dissertation are as follows:

- Techniques to avoid loss-of-decoupling events and performance pitfalls through extracting additional strand-level parallelism for superior functional unit and memory latency tolerance.

- Compiler-directed strand steering, which directly exposes strand-level parallelism to the hardware while reducing strand synchronization instruction overhead.

- Control speculation using lightweight checkpointing hardware and energy-efficient branch confidence prediction.

- Hybrid decoupled and multithreading techniques.

- Quantitative evaluation of performance against hardware prefetching, traditional decoupled, out-of-order execution, multithreading, and hardware scout prefetching on Rigel, a 1024-core data parallel processor.

A brief overview of the organization of this dissertation follows.

Chapter 2 provides background information about the current state of GPU computing, and describes the Rigel 1024-core data parallel processing platform and how it enables greater programmability. The great need for instruction latency tolerance is discussed, in particular memory latency tolerance, in data parallel processors such as Rigel. Finally, prior techniques for
latency tolerance and their pitfalls are discussed, among which five are chosen as points of comparison. This dissertation argues for decoupled architectures as a low complexity method for latency tolerance.

Chapter 3 delves more deeply into decoupled architectures. Decoupled architectures leverage the compiler to split a single thread of execution into multiple instruction streams that execute concurrently. These instruction streams, known as strands, execute as subprograms that follow the same control path through the program and communicate data values with each other. Finally, Decoupled Access / Execute is presented as an example of the canonical decoupled architecture.

Chapter 4 presents the evaluation methodology. A cycle-accurate performance model for Rigel is utilized, which is modified to run decoupled versions of parallel application kernels. Chapter 4 also explains the physical design model, which is used later in this dissertation for comparing area and energy during a design space exploration of five latency tolerance techniques.

Chapter 5 presents the vision for a modern decoupled core architecture which is used as the baseline Decoupled Access / Execute implementation. The most prominent feature in a modern decoupled architecture is the sharing of fetch and execution resources through a simultaneous multithreading approach.

Chapter 6 introduces loss-of-decoupling events, which are common code patterns that can cause performance pitfalls. In particular, memory indirection and floating point latency can cause exposed latency and reduce the ability of the memory-accessing strand to execute ahead and tolerate latency. This dissertation explores how increasing strand-level parallelism by extracting additional memory-accessing and memory-access strands can avoid loss-of-decoupling and improve performance.
Chapter 7 presents an energy-efficient mechanism to allow the compiler to express strand parallelism directly to hardware. Increasing the number of strands also significantly increases the amount of instruction overhead needed for synchronizing control flow and memory-ordering. By utilizing a single fetched instruction stream rather than multiple independently fetching and executing strands, the number of overhead instructions can be significantly reduced.

Chapter 8 investigates control speculation to avoid loss of decoupling due to data-dependent control flow. A low overhead checkpoint and rollback mechanism is utilized, along with confidence predictors to predict and reduce wasteful misspeculations.

Chapter 9 presents mechanisms to improve both area and energy efficiency for the proposed decoupled core architecture. First, the large overhead of data queues is reduced by leveraging dynamic register renaming and compiler-sized buffers. Next, the complexity of each strand is reduced, while maintaining similar levels of performance.

Chapter 10 presents the code generation for the proposed decoupled architecture. The implemented strand partitioning algorithm is presented. The code generation used for this dissertation, a binary translator, is explained.

Chapter 11 compares the proposed decoupled architecture with hardware prefetching, out-of-order execution, multithreading, and hardware scout prefetching. To facilitate this, energy-efficient mechanisms are added to each latency tolerance technique and a design space exploration is performed.

Chapter 12 explores the benefit of combining latency tolerance techniques. In particular, there is high synergy of combining decoupled architectures and multithreading. Considering that many data parallel architectures use thread-level parallelism to enhance throughput performance, the effect on
performance and energy of combining latency tolerance techniques is explored.

Finally, Chapter 13 details prior work and Chapter 14 presents conclusions of this dissertation.
CHAPTER 2

BACKGROUND

2.1 Rigel

Rigel [1] is a 1024-core data parallel processor envisioned as the evolution of current GPU computing. Similar to GPUs, Rigel targets highly parallel applications that scale up to thousands of concurrent tasks.

2.1.1 GPU Computing

In recent years, GPU computing has become very important to the computer architecture community. The end of single-processor performance scaling has increased the interest in and visibility of single-chip parallel processing. In particular, data parallel applications related to image processing, computer vision, and high-performance computing map well to multi- and many-core processors such as GPUs. In the last few years, GPUs have moved from fixed-function hardware to programmable pipelines with programming models based in high-level languages. GPUs offer substantial peak performance and power consumption advantages by focusing purely on throughput performance. The assumption with GPUs is that there are thousands of threads of available parallelism in an application, and these processors are designed to maximize FLOPS/Watt and FLOPS/mm$^2$. 
2.1.2 Rigel Architecture Overview

Rigel is a 1024-core data parallel processor envisioned as the evolution of current GPU computing. Current GPUs are continuing to improve in programmability, and do not yet have several features commonly found in general purpose processors. Rigel improves upon current GPUs to support ease of programmability and reduce programmer burden. For example, Rigel utilizes a cache hierarchy which leverages hardware to manage memory, rather than requiring the programmer to manage different address spaces. Another major improvement in Rigel is the independent execution of threads in a multiple-instruction multiple-data (MIMD) fashion. Currently, GPUs require groups of threads to execute in synchronized lockup or single-instruction multiple-data (SIMD). While SIMD execution reduces some hardware overhead, it limits performance on irregular applications. In particular, threads with different memory access latencies, or that follow different control flow paths in a program, cause stalls that limit performance. These stalls can be completely avoided through utilizing MIMD and independently executing threads. Fi-
nally, Rigel also enables a conventional programming model, which is bulk-synchronous and task-based.

A block diagram of Rigel is shown in Figure 2.1. Rigel is made up of 1024 cores arranged in a three-level cache hierarchy. The goal of Rigel is to maximize FLOP/Watt and FLOPs/mm$^2$, and thus the chip is developed to be complexity effective. Each core is an area-optimized two-wide issue in-order core with a single-precision floating point unit, private L1 instruction and data caches, and a RISC instruction set. Eight cores connect to a shared unified L2 cache via a shared bus interconnect to form a cluster. The clusters connect to a multi-banked shared last-level L3 cache through a two-level interconnect network. Groups of four L3 cache banks share an independent GDDR memory channel.

2.1.3 Cache Management and Coherence

All cores in Rigel share a memory address space, which is significantly different than GPUs which have several address spaces on a single chip. Communication between threads rarely occurs in data parallel applications, simplifying the method for keeping caches coherent with one another. While full hardware cache coherence would consume a significant amount of chip area, the memory space can instead be kept coherent by utilizing a software cache coherence mechanism [2]. L1 and L2 caches in a cluster are kept coherent utilizing a hardware snoopy protocol. However, clusters are not kept coherent with respect to one another using hardware, and the software coherence protocol must be used. Using the L3 cache as the global point of coherence, the cores operating in the cluster utilize a software coherence API and special hardware instructions to invalidate and flush data from the L1 and L2 caches.
out to the L3. This typically occurs at application synchronization points such as global barriers. Additionally, global memory operations that bypass the L1 and L2 and directly read from and write to the L3 cache can be used to ensure data is coherent among all threads on the chip.

2.1.4 Task Model

Rigel utilizes a bulk-synchronous parallel model to execute concurrent tasks. Figure 2.2 presents the high level execution of the bulk synchronous model. There is a spawn point at the start of a bulk synchronous phase, in which tasks begin execution. When all tasks have finished, a barrier synchronization is reached. To finish the bulk synchronous phase, modified data is written back with the software coherence mechanism and all data is again visible in the system.
Figure 2.3: Baseline Rigel architecture hierarchical task queue.

The programming model itself is a task queue based model, implemented in software and utilized by the programmer through API calls. A unique task descriptor is generated for each task, and the programmer can enqueue and dequeue tasks from the task queue. Figure 2.3 presents the hierarchical task queue, fully implemented in software. When the task queue is empty, cores in Rigel attempting to dequeue a task block, waiting for tasks to be enqueued. When all cores in Rigel are blocking on dequeue, an implicit barrier operation is triggered. Otherwise, when the task queue is empty, a single core will enqueue tasks to be executed. When the enqueue operation completes and tasks are available in the task queue, the waiting cores unblock, dequeue a task, and begin execution.
Figure 2.4: Categorization of latency tolerance techniques. Highlighted techniques are considered in this work.

2.2 Contemporary Latency Tolerance Approaches

In this section, existing latency tolerance techniques that can be considered for future 1000-core data-parallel processors and how they can tolerate memory and floating point latency are discussed. In-order execution and hardware prefetching are used as the baseline for comparison. Figure 2.4 provides a classification of techniques, which generally fall under either instruction-level parallelism (ILP) or thread-level parallelism (TLP) techniques. ILP techniques generally focus on improving the runtime of a single thread of execution, through leveraging some combination of the static compiler and dynamic hardware techniques. TLP techniques generally focus on improving performance by increasing the number of threads in the system either by adding more threads running the original program, or by adding “helper” thread contexts to improve performance of the original program. These subcategories are explored in more depth in this section.
2.2.1 Baseline: Hardware Prefetching

Prefetching data into the caches can help avoid the effects of memory latency which can dominate memory-intensive data parallel applications. Given that the per-core additional complexity needs to be kept to a minimum, nextline and stride-based prefetching [3] are considered. Prior research has shown that stride-base prefetching performs well on GPUs and manycores [4]. When the prefetch tables are sized appropriately for data-parallel workloads, both of these approaches require a relatively low amount of additional hardware.

Prefetching is most successful when the memory access stream can be predicted accurately, and when the data is prefetched into the cache in a timely manner. If the prefetching behavior is not ideal, cache pollution can occur, which can reduce performance and increase the number of cache accesses and energy consumption. Next-line prefetching suffers the most from potential cache pollution, as it cannot predict more complicated access patterns. To provide better prefetching behavior, a table-base stride prefetching can be used to better predict the access stream and improve timeliness. Additionally, software can statically switch off prefetching if the compiler or programmer knows it will not be useful.

2.2.2 TLP: Multithreading

There has been significant research on leveraging multiple threads of execution to tolerate instruction latency. Instead of a core pipeline stalling, instructions from a separate thread are executed. These architectures assume application parallelism, and are the most common technique found in data parallel processors with hundreds and thousands of threads. Multithreading
implementations are generally categorized as coarse-grained or fine-grained, depending on the frequency at which the execution of threads interleaves.

Coarse-grained or “switch on event” multithreading suspends an active thread of execution in a processor during a long latency event, and begins execution of a previously inactive thread. While the initiating event can be a variety of different events, typically a long latency operation such as a cache miss causes a change in active thread [5, 6, 7]. Coarse-grained multithreading has been implemented in several commercial processors including Intel’s
Itanium processor, where fine-grained multithreading is not as useful due to the VLIW scheduling [8].

In contemporary data parallel processors such as GPUs, fine multithreading is used to interleave multiple in-order threads of execution to provide instruction latency tolerance [9, 10]. Figure 2.5 depicts the additional hardware required for multithreading. In addition to register files for scratch space, memory space to hold the instruction and working data sets for each the thread contribute significant overhead. Figure 2.5(b) visually depicts cache contention, which can occur if the data set for each thread does not fit in the cache.

Figure 2.6: (a) Hardware scout core diagram. Added or modified components relative to an in-order core are in blue. (b) Hardware scout is initiated on a cache miss, and generates prefetches while a core is stalled.
2.2.3 TLP: Prefetch Threading

Prefetch threading is another widely-studied technique that leverages TLP. Prefetch threads execute alongside the main thread, moving data into the caches to help avoid exposed memory latency. The prefetch thread is either generated statically at compile time or dynamically at runtime. Similarly to multithreading, there must exist hardware support in the form of additional thread contexts.

Helper threads [11] instantiate a partial thread of execution to improve the performance of the main thread. This thread is either programmer or compiler generated, and can either run completely independently or be controlled by the main thread. The main goal of the helper thread is to generate useful prefetches and warm up the data cache for the main thread. Similar to other prefetching techniques, helper threads are sensitive to timeliness and can cause cache contention and thrashing with the main thread if not properly controlled.

Hardware scout prefetching, also known as runahead execution, focuses on dynamically generating prefetches to tolerance memory latency [12, 13, 14]. The hardware scout thread operates as a separate thread of execution which speculatively preexecutes the main thread to generate prefetches while the core is otherwise stalled during cache misses. Figure 2.6 depicts the added context checkpointing hardware required to support the hardware scout prefetching thread.

2.2.4 ILP: Dynamic Hardware

There have been several core architectures researched that purely leverage hardware to tolerate instruction latency and improve performance. Perhaps
the most distinct of these architectures are out-of-order execution and in-order continual flow pipelines. There have been various versions of these techniques researched to further optimize for tolerating memory latency. Generally, these techniques rearrange instruction ordering from the original compiled program by buffering and identifying dependences between instructions.

Out-of-order (OOO) execution is the most common dynamic latency tolerance technique implemented in processors today, and thus it is considered as the representative dynamic hardware technique for 1000-core data parallel processors. OOO cores generally work by buffering a large number of instructions yet to execute, and issuing instructions whose source operands are available. To increase the amount of instructions available and reduce data hazards, large register files and register renaming are often used. Examples of processors using out-of-order execution include early commercial successes, the MIPS R10000 and Digital Alpha 21264 [15, 16]. Currently, OOO is used in multi-core processors from major vendors such as AMD, ARM,
IBM, and Intel. Figure 2.7 depicts a high level view of a typical OOO core. The additional hardware structures required over an in-order core include reorder buffers, physical register files, load-store queues, and register renaming. When sized appropriately, functional unit latency and even memory latency can be tolerated by dynamically executing non-blocked instructions in the instruction window.

Continual flow pipelines focus on tolerating data cache miss latency, and there have been several recent proposals for implementing this technique to extend in-order processors [17, 18]. The general idea is that only the dynamic instruction stream dependent on the data cache miss is buffered and executed later in a separate context, and register file checkpoints are used to correctly handle exceptions and recover from misspeculation. By buffering only waiting instructions, the amount of memory latency that can be tolerated is very high. The approach is similar to OOO techniques for buffering instructions waiting on memory [19, 20], but does not require most of the additional hardware.

2.2.5 ILP: Compiler Extracted

There has been significant research around leveraging the compiler to tolerate instruction latency. The goal of leveraging the compiler is to reduce hardware complexity.

VLIW and EPIC processors leverage the compiler to schedule instructions to avoid both functional and memory latency using loop unrolling and speculative code motion [21]. While these designs can remove much of the hardware required for out-of-order execution, these designs still require significant hardware such as large register files and memory disambiguation hardware.
Figure 2.8: (a) Decoupled core diagram. Added or modified components relative to an in-order core are in blue. (b) Decoupling provides latency tolerance by allowing the memory instruction stream to be executed ahead of the compute instruction stream.

Even with speculative code motion, VLIW and EPIC designs can still be sensitive to memory latency and can stall if not enough software pipelining is done.

Decoupled architectures focus on leveraging the compiler to statically extract instruction-level parallelism by partitioning a single thread into separate *memory-access* and *memory-consuming* instruction streams called **strands** [22, 23]. Figure 2.8 depicts a decoupled architecture, where strands execute independent from each other and exchange values through FIFO data queues. When properly decoupled, the memory-access strands continue executing instructions, thus tolerating functional unit and memory latency.
CHAPTER 3

DECOUPLED ARCHITECTURES

This dissertation adopts the strategy of directly tolerating memory latency by decoupling the instruction stream. Decoupled architectures enable the memory-accessing instruction stream to be executed well in advance of consuming instructions similar to hardware scout and helper threads, but without duplicate execution of memory accessing instructions. Additionally, decoupled architectures are not sensitive to timeliness like prefetching techniques. Decoupled architectures enable a limited form of dynamic issue similar to out-of-order processors, but without area-inefficient structures such as large physical register files, reorder buffers, and register renaming. Finally, decoupled architectures have the ability of multi-threading through multiple instruction streams, but without increasing the aggregate working set required on chip and thus cache contention.

This chapter presents the necessary elements for a software decoupled design and introduces a traditional implementation. The serial instruction stream is partitioned at compile time into separate software entities which are called strands. Communication occurs between strands and facilities for control flow synchronization and data communication must be provided. Decoupled architectures trade off more complex software for lower hardware complexity. However, the complexity increase in software is on the order of other compiler transformations and utilizes much of the knowledge the compiler has already.
3.1 Memory Latency Tolerance through Strands

Decoupled architectures separate the memory-access and memory-consuming instructions of a program into separate instruction streams, called strands, that are executed on logically separate processors. These strands follow the same control flow path through the original program, but perform a specific function within each basic block. Strands must execute together in order to perform the same function as the original sequential thread, and by definition communicate data values and control flow decisions with one another. Strands are responsible for either accessing memory or consuming memory values, with partitioning occurring along memory dependence lines. In the base case there are two strands, one accessing memory and one consuming memory values. Address generation instructions and memory operations are found in memory accessing strands, while floating point and integer arithmetic are found in memory consuming strands.

Strands execute in parallel with one another and persist throughout the execution of the thread. Strands have their own context of program counter, scratch register space, and mechanisms to communicate with other strands.
However, because an individual strand executes only a portion of an original sequential thread, the context requirements such as register working set are significantly smaller. Considering all the strands together, the aggregate register working set requirement is on the order of the original sequential thread.

The main advantage of decoupling a sequential thread into strands is the ability to tolerate memory latency. Traditional in-order processors stall when a primary data cache miss occurs and a dependent operation is waiting to be issued. Decoupling into separate strands enables the memory-accessing stream to continue to issue instruction and execute in a nonblocking manner under ideal circumstances. Essentially, decoupled architectures execute instructions out-of-order, but this parallelism is extracted by the compiler from the original program, rather than dynamically in hardware. On the other hand, the compiler must be designed to extract the separate instruction strands. This limits backwards-compatibility of code, and requires that code be re-compiled.

Figure 3.1 shows the dependency graph of the inner loop of code that adds two vectors together. In this example, only the floating-point addition (\texttt{fadd}) and store (\texttt{stw}) instructions are dependent on memory. Consider the situation when the load (\texttt{ldw}) instructions require a long latency to fill the request from memory. In-order processors stall when a primary data cache miss occurs and a dependent operation is waiting to be issued. Decoupling into separate strands enables the memory-accessing stream to continue to issue instructions while the memory-consuming strand waits on the data to return from memory.
3.2 Control Flow Requirement

A default requirement of decoupled architectures is that each strand must execute down the same control path together. Figure 3.2 depicts the execution of several basic blocks of the thread as compared with that of a decoupled design. Though the strands must execute down the same control path through the program in order to ensure correct execution, the execution does not need to be synchronized. That is, once a strand determines the direction in the control flow graph to take, there is no need to wait on another strand.

While there is no formal synchronization required by a strand when the control flow decision is generated, in practice data may be required from another strand. In the case of compute-generated control flow, the control
flow decision cannot be known ahead of time or generated independently by memory-accessing strands. As a result, the memory-accessing strands must wait for the control flow decision to be communicated by the memory-consuming strand.

Similarly, though some conditional control flow could potentially have its decisions calculated locally, this results in additional instruction overhead. An example of this is a counted loop. Though the counting instructions and control flow decision could be made locally, this duplicates effort. This instruction overhead can be avoided by calculating the control flow decision on a single strand and then communicating that decision with all other strands. In order to promote decoupling, the ideal case is to have the thread executing furthest ahead generate control flow decisions.

The proposed decoupled architecture in this dissertation and other decoupled architectures require the ability to communicate control flow decisions between strands. As a result, both software and hardware overheads must be incurred. The additional instruction overhead of branch instructions for each strand can substantially increase the number of instructions executed by decoupled systems. For applications that have small basic block sizes, this overhead can represent a large portion of the computation. Hardware for communicating the control flow decisions also must be provided. In order to ensure correct control flow, hardware must have the ability to order communicated decisions and consume them in order. In other words, the strands must consume the oldest decisions that are waiting to be consumed.
3.3 Data Communication Requirement

By definition, strands communicate data with one another. More generally, memory-accessing strands communicate data with memory-consuming strands. This reflects the compiler partitioning scheme, which creates strands based along those lines. In traditional decoupled architectures, communication is performed using fixed-function FIFO queues, which only allow the result of load operations to be communicated to the memory-consuming strand. However, enabling more general communication is appropriate for general decoupled designs such as the one proposed in this dissertation, as it eases restrictions during the partitioning and code generation process.

To support general data communication, varying degrees of hardware and software overhead may be incurred. The requirement for hardware is that a given strand may produce a value which is then consumed by another specific strand. The hardware must be able to identify each data transaction and distinguish it from others in order to ensure that the correct data is consumed by the correct instruction. Each strand must be able to tell whether there is data ready to be consumed, which strand it is from, and what instructions in its instruction stream required that data. In many ways, this requirement is similar to the requirement of out-of-order processors. Each data value produced must be identifiable so that each instruction can correctly consume it if need be. Out-of-order processors also keep track of how old values are, so that the correct value may be paired with the correct instruction.

With these requirements in mind, there are a number of possible hardware options. As decoupled architectures are meant to offer improved efficiency, a design with the smallest area and energy overhead is preferred. Potential options for facilitating data communication include FIFO data queues, ro-
tating register files, register windows, and large physical register files with register renaming hardware. Each of these possibilities require both hardware and software modification to support the requirements of the decoupled design. For example, the register renaming approach requires a mechanism to synchronize the process across strands. While some approaches such as register renaming can require an extra pipeline stage due to added hardware complexity, approaches such as FIFO data queues can add software complexity due to copy instructions. These copy instructions map the data found in the FIFO queue into the local strands working set, and are required when a single data value is used more than once.

3.4 Decoupled Access/Execute Implementation

Figure 3.3 depicts a classic implementation of the decoupled architecture, Decoupled Access/Execute (DAE) [22]. The access processor (AP) and
execute processor (EP) are physically separate entities that are only connected through FIFO data queues for communicating data values loaded from memory, data values to be stored into memory, and control flow decisions. DAE achieves memory latency tolerance by executing the memory instruction stream on the AP and the computation program on the EP. The nonblocking property of the AP requires that the AP calculate control flow decisions, which it then forwards well in advance to the EP’s control queue, which is later used by the EP’s instruction fetch hardware.

3.5 Decoupling as an Emerging Technique

There has been significant past research on DAE and derivative decoupled architectures, including cache behavior [24], dynamic execution [25], and uses in out-of-order execution [26]. However, most of this research was undertaken during the time when single-core performance was most important, and the power wall was not an issue. At that time, the difficulty of matching single-core performance to out-of-order execution and the obstacle of recompilation severely limited the broad adoption of decoupled architectures.

As the focus has shifted from single-core to many-core, the obstacles previously limiting decoupling are less of a concern. Throughput performance is more important, and the low hardware overhead required for decoupled architecture enables many active cores to be on a single chip. Perhaps more importantly, introducing additional compiler transformations to support decoupled architectures is much less of an issue for manycore processors. GPUs and manycores today compile OpenCL and CUDA applications to an intermediate representation, which is later translated to specific chip parameters [9]. Lightweight compiler transformations required for decoupled archi-
tectures can be implemented at the translation layer, enabling decoupling to be implemented today.

As a result, there has recently been renewed interest in decoupled architectures for improving performance in GPUs and manycore, driven by the desire for energy-efficiency and complexity-effective performance [27]. This recent work shows both the performance and energy benefit of leveraging decoupled architectures, including the reduction in number of active threads on the processor. Overall, recent research has shown that by focusing on improving single-thread performance, concerns with multithreading can be avoided, such as cache contention and extracting enough parallelism to saturate a growing number hardware threads on chip.
CHAPTER 4
EVALUATION METHODOLOGY

In this dissertation, five common code patterns are identified that significantly impact the performance and energy consumption of latency tolerance techniques in 1000-core data-parallel processors: compute-intensive, data-dependent control flow, data-sharing intensive, pointer-chasing, and memory-streaming intensive. Using microbenchmarks to isolate these code patterns and highly parallel benchmark kernels from visual computing, a comprehensive design space evaluation is performed with in-order with hardware prefetching, out-of-order, multithreaded, hardware scout, and decoupled techniques.

Detailed performance models of each latency tolerance technique are built and detailed physical design models are generated using synthesis and analytical SRAM models for a 45nm CMOS manufacturing process to determine benchmark runtime, dynamic energy consumption, and leakage energy consumption. Each latency tolerance technique is individually implemented with complexity-effective and energy-efficient hardware to avoid naive and unfair comparisons. The runtime and energy values for a single technique are then analyzed to determine the Pareto-optimal configurations with respect to energy-efficiency.
4.1 Performance and Energy Modeling

Figure 2.1 depicts the 1024-core data-parallel processor with a three-level cache hierarchy, based on the Rigel architecture [1]. Table 4.1 lists the chip and core design parameters. Figure 4.1 presents the flow for evaluating the performance, area, and energy consumption of the techniques in this dissertation. In this flow, each architectural configuration’s performance is measured using a cycle-accurate performance model, while the associated area and energy costs of the design are based on models derived through a combination of RTL synthesis (for logic) and CACTI 6.0 [28] (for memories).

4.1.1 Performance Modeling

The performance modeling infrastructure is an execution-driven simulator that models the cores, caches, memory controllers, and DRAM. The core
models the structure of the pipeline, including pipeline stages and storage components such as branch prediction tables, register files, reorder buffers, load-store units and instruction windows. Each structure is modeled with a specific number of read and write ports in the simulator. Each core also has private L1 instruction and data caches, a single-precision floating point unit, and a MIPS-like instruction set. Eight cores share a unified L2 cache, with lower access latency than the L3 cache. The L3 cache is connected via memory controllers to off-chip DRAM. This performance modeling infrastructure was used to obtain performance numbers in terms of execution cycles for each of the microbenchmarks and benchmarks in the study.

4.1.2 Physical Design Modeling

For modeling the energy and area of each design, a CMOS 45 nm manufacturing process was used. CACTI was used to model the register files, the caches (L1, L2 and L3), and other storage components (BTB, branch prediction tables, instruction queues, etc.); synthesized Verilog was used for all

Table 4.1: Parameters for the baseline 1024-core architecture.

<table>
<thead>
<tr>
<th>Core Base</th>
<th>8 stage, 2-wide in-order 32 entry RF 64-entry Gshare branch pred. 8 entry BTB</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 ICache</td>
<td>2kB 2-way, 1 cycle 2 MSHRs, Next-line Pref.</td>
</tr>
<tr>
<td>L1 DCache</td>
<td>1kB 4-way, 1 cycle 8 MSHRs, 4 Out. Loads, 4 Out. Stores</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>64kB Shared. 4 cycle, 8-way</td>
</tr>
<tr>
<td>Interconnect</td>
<td>Two-level tree and crossbar 16 cycle minimum latency</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>4MB Shared, 32-Bank, 4 cycle, 8-way each bank</td>
</tr>
<tr>
<td>DRAM</td>
<td>8 Channels &amp; GDDR5</td>
</tr>
</tbody>
</table>
other major core structures such as the integer execution unit, the floating point unit and bypass logic.

As the area and energy of each module depends on its implementation and how aggressively it is designed, each of the chip components was targeted for a range of delays, with different implementations yielding energy and area as a function of delay. Specifically, CACTI was modified to output the energy-area-delay points of all configurations considered during its internal design space exploration, and the synthesis toolflow was used to generate different circuit implementations at different target clock frequencies. All these data points were then used to generate Pareto-optimal curves of area, dynamic energy and leakage as function of delay. These curves allowed the selection of the most efficient implementation of a circuit based on the target frequency and were an important consideration, especially as the voltage and frequency were varied to find the most optimal design.

To model the effect of the operating voltage, voltage scaling equations extrapolated from SPICE simulations of 45 nm circuits were used. These equations model the impacts of scaling voltage on circuit delay, energy consumption, and leakage power. The voltage scaling equations were then tied together to the Pareto-optimal energy, area and delay values for each of the modules. Operating voltages from 0.7 to 1.4 V were considered. Linking these values into the core pipeline model defined by the simulator enabled the approximation of the operating frequency of each core configuration under different voltage operating points.

The final step in creating a full energy model requires linking the energy costs at the module level with the architectural activity factors to generate the total dynamic energy consumption. In addition to outputting the runtime of each benchmark, the simulation infrastructure also outputs the activity for
each component in the core and memory system. The activity factor of each component during execution is measured by counting the number of accesses performed to the structure. For storage components, reads and writes are counted, while function units count the number of active cycles. Associated lookup storage structures also count the number of lookups performed. These activity factors are then used with the module configurations and the selected operating voltage to determine area, dynamic energy and leakage.

4.2 Code Patterns for Data Parallel Applications

In this section, five key code patterns found in data-parallel programs are identified that impact the energy efficiency of the latency tolerance techniques studied in this dissertation. The code patterns are compute-intensive, data-dependent control flow, data-sharing intensive, pointer-chasing, and memory-streaming intensive. These patterns stress different aspects of the latency tolerance techniques, such as the ability to tolerate functional unit and memory latency and the ability to exploit locality in the cache hierarchy. However, this collection is not a comprehensive taxonomy; rather these are the patterns that this dissertation finds most significantly affect the techniques evaluated. Additionally, these patterns are not mutually exclusive; multiple patterns may be found in an application.

**Compute-intensive:** Compute-intensive code consists predominately of arithmetic instructions, usually floating-point computation, as opposed to memory access instructions. As a result, these pieces of code often have long chains of dependent instructions and little control flow. In an in-order core, these chains of floating-point instruction lead to significant pipeline stalls.
due to functional unit latency. Successfully tolerating these stalls requires finding independent instructions to issue during long latency instructions.

**Data-dependent Control Flow:** The *data-dependent* control flow pattern occurs when the next basic block to execute cannot be known ahead of time. Data-dependent control flow poses a potential performance problem for several of the latency tolerance techniques evaluated. In order to find independent work and continue execution under long instruction stalls, either another instruction stream must execute or speculative execution must be performed within the stalled instruction stream. If the control flow decision is difficult to predict, misspeculation can occur and result in wasted energy.

**Data-sharing intensive:** Memory-intensive code with *high reuse* exhibits a large number of memory accesses that are favorable to a cache hierarchy. Peak performance is achieved by fitting the dataset of a thread into the cache. *Sharing intensity*, that is, the fraction of data shared between threads, impacts the design of the hierarchy. When the dataset used by each thread is at least partially separate, the amount of space required on the chip increases commensurate with the number of threads. To improve performance in the presence of cache-favorable applications with low amounts of data sharing, latency tolerance techniques generally must limit contention for cache resources. Limiting contention generally requires sizing the cache hierarchy appropriately, or limiting the number of threads that simultaneously execute. When enough sharing exists such that the data fits entirely in the cache, cache contention can be avoided.

**Pointer-Chasing:** *Pointer-chasing* behavior occurs when an application predominantly traverses through linked-lists or graphs. Each node in the
linked-list or graph is a dynamic element whose address is not known ahead of time, requiring the inspection of pointers in the parent node’s data structure. The memory access patterns can be highly unpredictable, and the runtime can be dominated by the memory accesses. Depending on the data set size, each memory access can have a significant amount of latency. Additionally, instruction level parallelism can be relatively low.

**Memory-streaming intensive:** Memory-intensive code with *low reuse*, also known as the memory-streaming pattern, exhibits a large number of memory operations that are consumed by the application approximately once. Due to the low amount of reuse, the memory accesses often miss in the cache hierarchy, causing significant stalls in an in-order core. Codes with this pattern often allow exploitation of memory level parallelism. With enough memory bandwidth, a latency tolerance technique that exposes additional requests to the memory subsystem can improve performance over an in-order core.

4.3 Microbenchmarks

4.3.1 Microbenchmarks

Microbenchmarks are used to isolate the identified code patterns and determine their impact on the energy efficiency of the latency tolerance techniques.

**Compute-Intensive:** The compute-intensive microbenchmark consists of a loop that performs iterations of floating point computation on values already existing in the register file. Each iteration of the loop executes a collection of floating point instructions connected in a tree-like structure where
each stage fans into the next, much like a reduction operation. Therefore, while there is ILP that can be exploited at each stage in the tree, there is latency that must be tolerated to keep the core from stalling.

**Data-dependent Control Flow:** The data-dependent control flow microbenchmark consists of a loop where each iteration calculates a value, which is then compared to a threshold value. If the calculated value is less than the threshold, a running sum is updated. It is unknown ahead of time whether the running sum will be updated or not, corresponding to different control paths. The threshold conditional branch is statically biased to 75% taken.

**Data-Sharing Intensive:** The data-sharing microbenchmark consists of a loop dominated by memory accesses. Each thread of execution has a vector dataset where each element is iterated on in an inner loop and the entire thread’s dataset is iterated over in an outer loop. The dataset for each thread is set such that the eight baseline threads in a cluster can fit in the L2 cache without aliasing.

**Pointer-Chasing:** The pointer-chasing microbenchmark is a linked-list traversal, where a small number of floating point instructions are used to update a value in each node. Each task operates on a separate linked-list, and each node aligns to a cache line and is irregularly skewed across the address space, simulating the effects of dynamic memory allocation and linked list manipulation.

**Memory-Streaming Intensive:** A simple vector addition code to model the memory-streaming code pattern. Using this code base, the situation where the dataset is resident in the L3 cache is modeled. When the data is
Table 4.2: Benchmarks and their primary and secondary code patterns. Primary code patterns dominate the benchmark, with the secondary pattern causing significant impact.

<table>
<thead>
<tr>
<th>Code Pattern(s)</th>
<th>Benchmark</th>
<th>Primary</th>
<th>Secondary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Secondary</td>
<td></td>
<td></td>
</tr>
<tr>
<td>blackscholes</td>
<td>compute-intensive</td>
<td>data-dependent control flow</td>
<td></td>
</tr>
<tr>
<td>cutcp</td>
<td>data-dependent control flow</td>
<td>compute-intensive</td>
<td></td>
</tr>
<tr>
<td>cg</td>
<td>memory-streaming</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dmm</td>
<td>data-sharing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fft</td>
<td>compute-intensive</td>
<td>memory-streaming</td>
<td></td>
</tr>
<tr>
<td>heat</td>
<td>memory-streaming</td>
<td>compute-intensive</td>
<td></td>
</tr>
<tr>
<td>kmeans</td>
<td>memory-streaming</td>
<td>data-dependent control flow</td>
<td></td>
</tr>
<tr>
<td>mri</td>
<td>compute-intensive</td>
<td>memory-streaming</td>
<td></td>
</tr>
<tr>
<td>sobel</td>
<td>memory-streaming</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

already in the L3 cache, the access latency is in the tens of cycles and a high amount of bandwidth is available.

4.4 Benchmarks

A set of nine optimized parallel kernels from scientific and visual computing applications implemented for Rigel are used for evaluation. The benchmarks exhibit a high degree of parallelism and are written using a task-based, barrier-synchronized work queue model similar to Carbon [29]. For code generation, the LLVM compiler and backend are used [30]. LLVM compiler optimizations are turned on and code is tuned at the source level using loop unrolling and software pipelining techniques. As a result, a significant amount of latency can be tolerated over naive code.

The benchmarks include black scholes (blackscholes), conjugate gradient linear solver (cg), coulumbic potential with cutoff (cutcp), dense matrix multiply (dmm), 2D fast Fourier transform (fft), 2D stencil computation (heat), k-means clustering (kmeans), medical image reconstruction (mri),
and image edge detection (sobel). Table 4.2 lists the dominant code patterns found in these benchmarks. Each benchmark is executed for at least one billion instructions.

**Black Scholes**  The blackscholes benchmark performs pricing on a set of options using the Black-Scholes partial differential equation. The equation is highly floating-point and compute-intensive, and thus is strongly dominated by that code pattern.

**Conjugate Gradient Linear Solver**  The cg benchmark is a linear solver for sparse linear systems. The execution is dominated by sparse matrix-vector multiply operations on the dataset. The sparse matrix and vector memory accesses in the program exhibit relatively low locality, causing the benchmark to exhibit a strong memory-streaming code pattern.

**Coulombic potential with Cutoff**  The cutcp benchmark is used to calculate the potential energy between atoms. The benchmark has significant data-dependent control flow, with control flow depending on the result of the coulumbic potential calculation.

**Dense Matrix Multiply**  The dmm benchmark performs a blocked single-precision dense matrix multiply on a pair of 1024x1024 matrices. The blocking produces the potential for input data reuse among threads, making the application exhibit a strong cache-favorable code pattern with high data sharing.

**Fast-Fourier Transform**  The fft benchmark is a 2D FFT using 1D FFT operations combined with matrix transpose operations. The FFT operations
are rich in floating-point chains, leading to a compute-intensive code pattern, while the matrix transpose operations exhibit low-reuse and fall into the memory-streaming pattern.

**Heat** The heat benchmark performs a stencil computation to determine the heat flow in a 2D space. There is a small amount of data sharing exhibited between thread, due to sharing of boundary data. However, most of the data is not shared, and due to low reuse the application is memory-streaming. Additionally, the large number of floating point operations in the benchmark leads to the compute-intensive code pattern being exhibited.

**K-means Clustering** The kmeans benchmark implements a variant of the K-means clustering algorithm. Much of the execution is spent doing histogramming operations, during which memory-streaming is observed. As a secondary code pattern, the cluster computation exhibits data-dependent control flow in an outer loop.

**Medical Image Reconstruction** The mri benchmark is compute-intensive, with long chains of floating-point operations. Additionally, data accessed exhibits low reuse such that the memory streaming pattern is observed.

**Sobel** The sobel benchmark performs edge detection on a 2D image. The execution is dominated by matrix convolution operations. The kernels used for convolution exhibit a high amount of data reuse. However, the image matrix exhibits low locality, and thus the memory-streaming code pattern dominates. Additionally, there is little data shared between threads.
CHAPTER 5

A MODERN DECOUPLED ARCHITECTURE

This chapter presents the baseline core architecture used for experiments in this dissertation. Throughput-oriented data parallel processors focus on maximizing utilization of functional units while minimizing chip area to provide a large number of functional units on a single chip. A decoupled core architecture designed for a 1000-core data parallel processor should attempt to optimize its design based upon these goals.

Traditional decoupled designs such as DAE require separate fetch, decode, execute, and data queue resources for each strand processor, but these resources may not experience continuous utilization. This under-utilization results in opportunities to reduce the chip area overhead for the technique. These duplicated hardware resources could potentially be shared, improving utilization and reducing area overhead. For example, memory-consuming strands may execute integer or floating-point arithmetic, causing replication of hardware such as multipliers and shift units between decoupled processors. Traditional fetch resources such as instruction caches also represent duplicated hardware that can be shared.

The method commonly used to share resources in core architectures today is multi-threading, which executes several thread contexts on the same core. In this dissertation, a similar technique is used to execute multiple strands on the same core pipeline. The benefit of using a multi-threading technique to share hardware resources between decoupled strands is decreased area and
static energy consumption, but there is a possibility of increased contention for these shared resources, which can reduce performance. This dissertation finds that, in practice, the performance loss is generally quite low, resulting in a significant improvement in area-efficiency.

5.1 Architecture Overview

Figure 5.1 presents the core architecture used for the baseline in this dissertation. There are two strands, one memory-accessing and one memory-consuming. Each strand has its own instruction queue, register file, and FIFO data queues for communication. The memory-accessing strand can issue multiple outstanding memory requests. Both strands compete for issue slots, and execution between the strands interleaves in a simultaneous multithreading (SMT) style.
5.1.1 Shared Resources and SMT

Figure 5.2 presents the core pipeline for the modern decoupled architecture. The pipeline is seven stage with three total pipes: one for memory, one for integer/branch, and one for floating point computation. The entire pipeline is shared among the memory-accessing and memory-consuming strand. Instructions retrieved from the instruction cache at the fetch stage also go through the decode stage, at which point the instruction is stored into separate instruction queues. When instructions are ready to be issued, they are issued from the schedule and register file read stage in an SMT-style fashion, where all backend pipes are pipes are shared among strands. Two instructions can be issued per cycle total, with instructions coming from either strand.

5.1.2 Data Queues

FIFO data queues are used for communication between strands. In the modern decoupled architecture with two strands, one FIFO queue is needed for each strand. Unlike the special-purpose queues found in DAE where queues are only used for either control flow or data communication, the data queues in the modern decoupled core architecture can be used for general commu-
communication or broadcast of any value. Strands waiting for data from the queue are blocked, while other strands continue execution. The queues can achieve good performance with a small number of entries because when data is available on the queue it is likely it will be quickly consumed by a waiting strand. Additionally, since the strands are mostly independent, the frequency with which communication occurs is relatively low.

The data queues are mapped to architectural register 1 in the register file enabling each of the strands to directly communicate with one another. When register reads and writes are performed using this register, the FIFO data queue is accessed. Writing to the data queue implies producing a value for the other strand, while reading implies consuming a communicated value.

5.1.3 Memory Access Unit and Memory Ordering

Figure 5.1 shows the memory access unit (MAU) which enables multiple memory operations to be in-flight simultaneously. Each strand has its own load buffer, while the store buffer is a shared associative buffer that is used to enforce memory ordering. The low number of store entries that must be looked up associatively is effective in keeping the design of the MAU compact. The MAU is shared across all strands to enable correct memory ordering.

The store buffer is used for memory disambiguation. For each store instruction found in the original code, each strand will have either a \texttt{st\_addr} or \texttt{st\_data} instruction. The \texttt{st\_data} instruction is found in the strand providing the data for the store, while all other strands have the \texttt{st\_addr} instructions which provide the address of the original store. For each store instruction, a single entry in the store buffer is used. A single store may be completed once each strand issues its corresponding instruction. By requir-
Table 5.1: The relative distribution of dynamic instructions between the memory-accessing and memory-consuming strands. Numbers are for a two-strand decoupled core on data-parallel benchmarks.

<table>
<thead>
<tr>
<th></th>
<th>Memory-Accessing</th>
<th>Memory-Consuming</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td>68.8%</td>
<td>31.2%</td>
</tr>
<tr>
<td>cg</td>
<td>82.1%</td>
<td>17.9%</td>
</tr>
<tr>
<td>cutcp</td>
<td>75.6%</td>
<td>24.4%</td>
</tr>
<tr>
<td>dmm</td>
<td>74.0%</td>
<td>26.0%</td>
</tr>
<tr>
<td>fft</td>
<td>51.3%</td>
<td>48.7%</td>
</tr>
<tr>
<td>heat</td>
<td>62.8%</td>
<td>37.2%</td>
</tr>
<tr>
<td>kmeans</td>
<td>71.0%</td>
<td>29.0%</td>
</tr>
<tr>
<td>mri</td>
<td>48.6%</td>
<td>51.4%</td>
</tr>
<tr>
<td>sobel</td>
<td>43.3%</td>
<td>56.7%</td>
</tr>
</tbody>
</table>

ing each strand to have either the address or the data, loads can perform associative lookups into the store buffer to ensure proper memory ordering.

5.2 Evaluation

To understand the tradeoffs between reducing area and performance by utilizing SMT, a comparison between the original DAE proposal and the modern decoupled core architecture design is performed. The Rigel 1024-core data parallel processor and accompanying parallel benchmarks are considered. Both the AP and EP cores in DAE can either issue one or two instructions per cycle, while the modern decoupled core enables each strand to issue a maximum of two per cycle.

To understand hardware requirements, the instruction balance between the memory-accessing and the memory-consuming strands is investigated. Table 5.1 shows the percentage of static instructions executed in both strands. Overall, the instructions are not distributed evenly between the strands, and in some application kernels the memory-consuming strand can dominate (sobel, mri), while in others the memory-access strand clearly dominates.
Figure 5.3: Chip area consumption of DAE with one and two-wide issue compared with modern decoupled architecture normalized to the two-wide issue in-order core.

The fact that the instruction balance varies so much directly impacts the design of the instruction caches. In DAE, the L1 instruction caches should ideally be designed to capture the frequently-accessed portions of code in order to provide robust performance across applications. For the experiments, this involves completely replicating the 2kB instruction cache for both the AP and the EP.

This dissertation investigates extracting more strand parallelism to improve latency tolerance and performance. The scalability of replicating both instruction caches and functional units is limited, and thus sharing these resources provides great opportunity for area savings. The replication of instruction caches to enable robust performance is the most significant area overhead for DAE. Figure 5.3 presents the chip area for the 1024-core data parallel processor Rigel, which is largely dominated by memory structures. The functional units make a significantly smaller portion of the total chip area. Overall, the modern decoupled core interleaves strand execution, reducing area by 12% over the one-wide issue DAE and 18% over the two-wide
Figure 5.4: Performance improvement of DAE with one and two-wide issue compared with modern decoupled architecture normalized to the two-wide issue in-order core.

issue DAE. Besides the increase in throughput per chip area, having reduced area can also have a positive impact on timing and reduce latency for communication between strands. Though not investigated in detail in this dissertation, the wiring for the data queues can be substantially reduced by including all communication within a single core pipeline.

The modern decoupled core architecture also compares quite favorably to DAE in terms of application kernel runtime. Figure 5.4 presents the performance of the DAE configurations and the modern decoupled core normalized to a two-wide issue in-order baseline on Rigel. Overall, the modern decoupled core architecture performs competitively with the DAE configurations. The modern core improves performance over the in-order case by 31% across the application kernels, while the one-wide DAE core improves by 21% and the two-wide DAE core 39%. Considering that the DAE-1 has an aggregate two instructions issued per cycle, the substantial improvement of the modern decoupled core is due to the increased utilization of the functional units due to strand interleaving. Specifically, there is a 19% reduction of issue stalls.
due to the increase in issue slots of the modern decoupled core over DAE-1. Despite having half the aggregate issue slots, the modern decoupled core performs within 94% of the DAE-2 configuration.

Overall, the modern decoupled core reduces some significant overheads while enabling strand scalability which is evaluated in the next chapter.
CHAPTER 6
INCREASING STRAND-LEVEL PARALLELISM

Traditional decoupled core architectures can suffer from performance pitfalls due to code patterns known as *loss-of-decoupling* events commonly found in applications today. This chapter investigates how extracting more strand-level parallelism can avoid memory indirection and instruction latency loss of decoupling and improve performance. Depending on the application, more memory-accessing or memory-consuming strands can be extracted from the original program. To enable the increased strand-level parallelism, both hardware and software support are needed. In particular, changes to the core architecture must be performed, including increasing the number of data queues and register files commensurate with the number of strands. The algorithm for extracting strands is modified to clearly define which instructions are allocated to which strands. Maintaining proper memory ordering across multiple memory-accessing strands is an additional concern that is investigated.

6.1 Loss of Decoupling Events

Although decoupled architectures enable memory latency tolerance, potential performance improvement is limited when the memory-accessing instruction stream cannot achieve the nonblocking property with respect to the rest of the program. These situations are known as *loss-of-decoupling* (LOD)
Figure 6.1: Loss-of-decoupling (LOD) events for two-strand decoupled architectures which can severely limit performance. The memory-accessing strand is in strand zero (S0), while the memory consuming strand is in strand one (S1).

events [31]. Figure 6.1 presents the LOD events on traditional two-strand decoupled core architectures, which represent a dependence between the processors that must be resolved before the memory-accessing strand is allowed to continue execution.

In the optimal case there is no LOD event and the memory-accessing strand is not blocked. Memory indirection LOD events are caused by cache misses during indirect memory accesses, such as sparse matrices and multi-dimensional arrays, where the latency to access memory is exposed and the memory strand must stall. Additionally, memory indirection due to complex address generation in the memory-consuming strand also can cause LOD, such as during hash functions. If the memory-consuming strand is responsible for determining the next basic block, the memory-accessing strand is not able to execute ahead and must wait. This control flow LOD
event removes the memory-accessing strand’s ability to execute ahead and tolerate memory latency. Finally, this work introduces functional-unit latency LOD events which are caused by long functional unit latencies, such as floating point arithmetic, which can cause significant stalls and hamper performance on compute-intensive workloads. Specifically, these stalls reduce the rate at which the memory-consuming strand can consume data produced by memory-accessing strands, and in turn the memory-accessing strand must eventually stall due to the data queues becoming full.

In some cases these LOD events can be avoided, provided proper additional hardware and software. In this chapter, the memory indirection and functional-unit latency LOD events are avoided by extracting additional parallelism. For this technique, the method of extracting strands from an original program is modified, and additional strand contexts including register files, data queues, and load buffers are added to the baseline decoupled core architecture. This dissertation also researches mechanisms to avoid control flow LOD, which are discussed in Chapter 8.

6.2 Memory Indirection

Figure 6.2 shows the approach to addressing the memory indirection LOD events found in Figure 6.1. Performance loss due to these LOD events can be alleviated by extracting additional memory-accessing strands. In the case of memory indirection, the original memory-accessing stream can be split into multiple strands, with the goal of having at least one instruction stream without an LOD. Each load in the memory-indirection chain is ideally placed in separate strands. By extracting additional strands, the amount of decoupling is increased and more parallelism is exposed. In order to handle compute-
generated memory accesses, floating-point instructions are permitted to exist within a memory-accessing strand, which removes a substantial restriction from past decoupled architectures such as DAE. Using these techniques, the stalls due to memory indirection LOD can be removed in many cases and reduce exposed functional unit latency for improved runtime performance.

6.2.1 Strand Extraction Algorithm

While detailed information on the final code generation algorithm and toolflow can be found in Chapter 10, a brief overview is presented here. To accomplish the extraction of additional memory-accessing strands, the code generation toolflow must be modified. Specifically, the compiler must be augmented to keep track of how many loads are found in a backslice of a dataflow graph (DFG) of the original program. The instructions in the original program are therefore assigned to a strand based upon how many prior loads are seen in the backslice of the DFG. For example, if a floating-point instruction has a chain of two loads in its backslice, the strand assignment would be for the
third strand. Similarly, if a load has a single load in its backslice, it would be assigned to the second strand. All load operations without a prior load are assigned to the first strand which is the strand that executes the furthest ahead.

6.2.2 Memory-Ordering

Despite the potential performance improvement from avoiding LOD, increasing the number of memory-accessing strands also increases complexity. In particular, memory ordering must be maintained across the multiple memory-accessing strands. Simply separating a single memory-accessing strand into multiple strands provides no notion of strict ordering, since the execution of each strand is independent aside from communication through FIFO queues. Stores must not be allowed to complete ahead of loads to the same address, and thus at the minimum, each memory-accessing strand must have accurate information about all stores in the system. In this dissertation, this issue is addressed using a memory access unit (MAU). The MAU enables multiple loads and stores to be in-flight to tolerate memory latency, and is responsible for properly ordering loads and stores.

In decoupled architectures, store instructions are traditionally broken up into address and data instructions, which exist in separate strands. When the store buffer collects both of these instructions, the store can be committed to the memory subsystem. For every store in the proposed decoupled architecture, each memory-accessing strand is required to have a \texttt{st.addr} instruction for each store in the original program, which are collected into the store buffer. The memory-consuming strand is responsible for contributing the data for the store in the form of a \texttt{st.data} instruction. When a later
load instruction is executed and reaches the MAU, an associative lookup is done and the outstanding store addresses are checked for any conflicts. In essence, the \texttt{st_addr} acts as a point of lightweight synchronization between the memory-accessing strands facilitated through the MAU. Finally, a store is able to be committed when all \texttt{st_addr} and the \texttt{st_data} instructions have been collected.

### 6.3 Floating Point Latency

Figure 6.3 shows the approach to addressing functional unit latency LOD event found in Figure 6.1. Performance loss due to these LOD events can be alleviated by extracting additional memory-consuming strands. Dependent instructions are placed into another strand, moving the functional unit latency off the critical path and enabling one of the memory-consuming strands to execute ahead. This enables a higher rate of data consumption from the
data queues, and can increase the ability of the memory-accessing strands to execute ahead. Additionally, increasing the number of memory-consuming strands enables the strand instruction scheduler to have more instructions available to keep the functional units busy.

6.3.1 Strand Extraction Algorithm

While detailed information on the final code generation algorithm and toolflow can be found in Chapter 10, a brief overview is presented here. The additional memory-consuming strands are extracted using the compiler. After the memory-accessing strands have been extracted to avoid the memory indirection LOD event, and there are not enough extracted strands to fill all hardware strand contexts, a secondary pass is run to extract based upon floating point dependences. The memory-consuming strand is inspected, and all floating point instructions that otherwise would result in exposed latency and do not consume their operands directly from loads are placed in a second memory-consuming strand. Upon inspection of the application kernels evaluated as well as the general data parallel paradigm, a significant number of instructions primarily consume data from memory, particularly in memory-intensive workloads. Additionally, based upon the observation that many computations start with several inputs and reduce those values into a single result, decoupling along those lines results in a significant amount of instructions in both memory-consuming strands.

6.4 Implementation

The new decoupled architecture with support for multiple strands is found in Figure 6.4. The main additions to the decouple architecture found in Fig-
Figure 6.4: Multi-strand decoupled architecture.

Figure 5.1 are additional strand contexts, as well as the memory ordering support of the memory access unit (MAU). For the application kernels, supporting four total strands enables enough additional parallelism to avoid most of the LOD events. To support four strands, enough data queues are required such that all strands can communicate with one another. Each strand can communicate directly with three other strands, so the number of register mapped data queues is increased from one in the two strand case to three in the four strand case. The total number of data queues needed for the revised decoupled architecture is twelve. While this may seem like a significant chip area overhead, techniques to minimize the overhead are investigated in Chapter 9 of this dissertation. The MAU is modified to enable up to three memory-accessing strands to issue and complete load instructions to the data cache.
6.5 Evaluation

In this section the ability of additional strand level parallelism to avoid LOD events is evaluated. The Rigel 1024-core processor in Table 4.1 and the data parallel application kernels in Chapter 4 are used.

6.5.1 Memory Indirection

The impact of the memory indirection LOD avoidance technique is investigated first. Of the nine benchmarks, six have some level of memory indirection during task execution. Figure 6.5 presents these six benchmarks and the performance of the two and three strand versions versus the baseline in-order on the Rigel 1024-core processor. Increasing the number of memory-access strands provides a substantial improvement due to increased memory latency tolerance. Overall, benchmarks with memory indirection at the inner
Figure 6.6: Performance improvement of increasing the number of memory-accessing strands relative to the in-order baseline.

loop level such as \texttt{cg, dmm, heat} are most impacted by cache misses and see the most performance benefit by nearly 61% on average. The other three benchmarks are not dominated by memory indirection in the inner-loop, and the performance improvements are mixed. In the case of \texttt{blackscholes} and \texttt{cutcp} there is a slight benefit, while the added increase in dynamically executed instructions such as branches slightly lowers the performance of \texttt{kmeans}. In fact, with \texttt{kmeans} the number of instructions issued per cycle increases by 2%, although the runtime improvement does not.

6.5.2 Floating-point LOD

Building upon the memory-indirection LOD, the number of strands is increased to avoid the floating point latency LOD. This technique can impact all applications with floating point arithmetic, and so all nine benchmarks are considered. Figure 6.6 presents the nine benchmarks and the performance improvement of increasing the number of memory-consuming strands on the Rigel 1024-core processor. Overall, the results are mixed due to the poten-
tial increase in dynamically executed instructions. In fft, heat, and kmeans, performance is actually lost due to the increase in branch instructions which cause contention for the branch unit at issue time. Other benchmarks see an average of 6% performance improvement, with dmm and mri improving by more than 10%.

6.5.3 Effect of Parallelism on Dynamic Instruction Count

One significant drawback to both of these approaches is the impact on dynamic instructions executed. Increasing the amount of parallel executing instruction streams through extracting additional strands increases the amount of control flow, memory disambiguation instructions, and data movement instructions. In general, to reduce control flow instruction overhead, compiler optimizations such as loop unrolling are performed aggressively. This overhead has profound effects on energy efficiency in Rigel. Increasing the amount of dynamically executed instructions requires more energy to be spent accessing the instruction caches, register files and data queues, and functional units. As mentioned in the previous sections, the increase in overhead instructions also significantly contributes to performance loss, as extra contention for those execution resource is observed.

Figure 6.7 presents the amount of retired instructions for each of the benchmarks for two-, three-, and four-strand versions normalized to the in-order baseline. Depending on the application, the number of overhead instructions can be quite substantial. On average, the amount of overhead for the benchmarks is 12%, 21%, and 27% for two, three, and four strands respectively. In general, applications that have small basic block sizes and inner loops experience the most added overhead for both control flow and memory dis-
ambiguation instructions. Examples of such benchmarks are cg, cutcp, fft, and kmeans. Benchmarks with basic block sizes consisting of large instruction counts such as blackscholes, dmm, heat, mri, and sobel see significantly less sensitivity to instruction overhead as the number of strands extracted is increased.

Table 6.1 presents a breakdown of overhead instructions by type for the three-strand configuration for each benchmark. Overall, most of the overhead is due to control flow instructions, comprising an extra 12% on average. The extra instruction overhead required for memory disambiguation and data movement is 5% and 3% on average. The relative overhead of extra memory disambiguation and data movement instructions depends on application characteristics, such as the number of stores and the number of data values read more than once. The extra instruction overhead contributes directly to additional energy consumption and performance loss due to contention for execution resources, and methods to reduce this overhead are investigated in Chapter 7.
Table 6.1: The percent added overhead for extra control flow, memory disambiguation, and data movement instructions for three-strand versions of data-parallel benchmarks.

<table>
<thead>
<tr>
<th></th>
<th>control flow</th>
<th>memory disambiguation</th>
<th>data movement</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td>8.14%</td>
<td>1.62%</td>
<td>4.79%</td>
</tr>
<tr>
<td>cg</td>
<td>12.42%</td>
<td>11.44%</td>
<td>0.06%</td>
</tr>
<tr>
<td>cutcp</td>
<td>20.51%</td>
<td>7.53%</td>
<td>0.53%</td>
</tr>
<tr>
<td>dmm</td>
<td>4.58%</td>
<td>3.44%</td>
<td>1.62%</td>
</tr>
<tr>
<td>fft</td>
<td>18.50%</td>
<td>12.60%</td>
<td>3.78%</td>
</tr>
<tr>
<td>heat</td>
<td>7.98%</td>
<td>5.26%</td>
<td>0.15%</td>
</tr>
<tr>
<td>kmeans</td>
<td>23.97%</td>
<td>2.57%</td>
<td>5.83%</td>
</tr>
<tr>
<td>mri</td>
<td>12.05%</td>
<td>0.01%</td>
<td>0.01%</td>
</tr>
<tr>
<td>sobel</td>
<td>3.18%</td>
<td>1.40%</td>
<td>9.53%</td>
</tr>
<tr>
<td>average</td>
<td>12.37%</td>
<td>5.10%</td>
<td>2.92%</td>
</tr>
</tbody>
</table>

6.5.4 Exposure of Instruction-Level Parallelism

To understand the impact of increasing strand-level parallelism on ILP, a limit study is performed on the Rigel data parallel processor. In this study, the issue width of each core in the Rigel system is varied from one to eight, while providing enough functional units to remove all artificial stalls. For example, at the 8-wide issue point, eight floating-point instructions can be issued in a single cycle in this ideal processor. Figure 6.8 presents a limit study in the improvement in exposed ILP in the system as a result of increased decoupling. Performance is normalized to the one-wide in-order case. However, this limit study is still significantly affected by overhead instructions, which still increases the critical path execution of each benchmark.

As expected, increasing issue width past two-wide issue does not significantly improve the performance of an in-order core. Increasing issue width from one-wide to eight-wide only improves performance by 36% in an in-order core across the benchmarks. Moving to a two-strand decoupled core architectures similar to DAE improves performance significantly more. Moreover,
Figure 6.8: Extracting more strand-level parallelism improves performance as core issue width increases, despite instruction overheads.

the two-strand decouple cores can leverage the increase in issue width up to four-wide. Increasing from one-wide to two-wide improves performance 43%, while moving from two-wide to four-wide improves performance by 23% on average. However, tolerating loss-of-decoupling through additional strand-level parallelism provides superior performance. The extra instruction-level parallelism enables the average performance to be nearly 70% higher than the two-strand decoupled core. The most significant increase is moving from a two-wide issue to four-wide, which increases performance by nearly 50%. 
Despite the promise of decoupled architectures for low-complexity memory latency tolerance, there is significant energy overhead spent fetching and executing overhead instructions required to maintain separate strands. In this chapter, energy inefficiency is combated by enabling the compiler to direct hardware partitioning to maintain a single instruction stream. Leveraging the compiler and lightweight hardware, dynamic instructions for control flow and memory ordering required for correct execution in the software-only decoupled cores previously evaluated can be avoided. The result is a significant reduction in instruction overhead, reducing energy consumption and improving performance.

7.1 Hardware Assisted Strand Partitioning

Strand partitioning consists of separating a serial instruction stream into multiple fetching and executing strands, and is typically done at compile-time. To facilitate correct execution, overhead instructions such as additional control flow, data movement, and memory disambiguation instructions are inserted to correctly synchronize the strands with one another. Figure 7.1 presents the inner loop of a sparse matrix vector multiply (SMVM) operation. There are three extracted strands, and the colored instructions represent the extra instruction overhead required for data movement, memory ordering,
Figure 7.1: Sample code detailing overhead instructions in the conjugate gradient inner loop.

and control flow. The main concern with these overhead instructions is that they require extra energy to be spent fetching and executing in the core. In Chapter 6, it is shown that overhead instructions can easily increase the dynamic instruction stream by 20% to 40% or more. In addition to energy concerns, instruction overhead also potentially reduces performance improvement by contending for issue slots and execution resources.

However, it is not necessary to have separate fetching strands to achieve the performance benefit from leveraging the compiler to decouple the instruction stream. Instead of partitioning the single instruction stream into strands at compile-time, the partitioning can be done dynamically at run-time. This can be accomplished by a single instruction stream, with each instruction assigned to a specific strand by the compiler. In doing so, parallelism is extracted by the compiler, maintaining low hardware complexity for the core. The only changes are modestly increasing the size of the instruction queues and modifying the ISA to reserve two bits per instruction for strand identification on a four-strand architecture. The compiler then can identify the
parallelism at compile time and generate code marking each instruction with its strand identification. Combinational logic inspects these bits and steers the instruction into the correct instruction queue.

The main benefit to compiler-directed instruction steering is the reduction in instruction overhead. By fetching a single instruction stream, there is no longer a need to explicitly synchronize strands, either with respect to control flow or with respect to memory ordering. As a result, the branch and memory ordering overhead instructions can be avoided. Considering that these overhead instructions make up the majority of the instruction overhead, 17.4% of the overhead instructions can be removed, leaving only data movement overhead instructions at 2.9% across the application kernels. Table 6.1 shows the potential savings for each benchmark by eliminating the branch and memory ordering overhead.

7.2 Implementation

This section discusses implementation details for the compiler-directed instruction steering scheme. Figure 7.2 presents the updated decoupled core. The multiple program counters are replaced with a single program counter, and the instruction steering logic is added. The main changes to the architecture are the ways that control flow and memory ordering are handled. Additionally, strand scheduling arbitration is also updated, to take into account the instruction queue size hardware constraints.

7.2.1 Control Flow

Due to the single instruction stream, control flow is handled as a special case. When a branch instruction reaches the instruction steering logic, the
steering mechanism will stall until the branch is resolved. In other words, the next basic block is decided before instructions after the branch are inserted into the instruction queues. Consider the ideal case where the memory-accessing strand executing furthest ahead is responsible for handling the branch. When the branch is seen by the instruction steering logic, the stall occurs until all the instructions from the memory-accessing strand executing furthest ahead have been scheduled. In the case where all strands are active and being scheduled, the arbitration scheme may need to be aware that fetch is otherwise stalled due to waiting on the memory-access strand executing furthest ahead.
7.2.2 Memory Ordering Details

The original ordering of memory instructions is maintained by utilizing a single instruction stream. As a result, synchronization between memory-accessing strands is no longer required for correct load-store ordering, and the overhead instructions for memory ordering are eliminated. There are two ways to implement memory ordering using the MAU for disambiguation.

The simpler method is to replicate the conservative memory ordering method in the separate software strand decoupled architecture by stalling the frontend when a load operation reaches the steering logic and a \texttt{st_addr} instruction, responsible for providing the address portion of the store operation, is waiting to be issued. This is accomplished with a simple counter that tracks the number of outstanding \texttt{st_addr} instructions. This action is similar to the control flow mechanism, and \texttt{st_addr} instructions are ideally performed by the strand executing furthest ahead to reduce the chance of the instruction steering mechanism stalling.

However, considering that many data parallel applications have separate inputs and outputs in memory, with little risk of aliasing, the aforementioned approach is quite conservative. Instead, it is understood that once an instruction is steered, it will be executed due to the control flow implementation. Therefore, as loads and stores reach the instruction steering logic, they are registered in the MAU. After the loads and stores are scheduled and reach the MAU, memory disambiguation lookups can occur and orderings be enforced. In other words, known load-store queue and related techniques can be leveraged to provide correct memory disambiguation [15]. The main benefit is that regardless of load store ordering, the miss queue can be updated and the missing cache line filled into the cache, improving performance. Leverag-
ing the MAU to provide memory disambiguation is the approach leveraged in this dissertation.

7.2.3 Exception Handling

Due to the nature of Rigel being an accelerator that is controlled by a host, page faults and floating-point exceptions are envisioned to be the common events that are handled. However, a method for general hardware interrupts is presented. Is it expected for exceptions to be relatively rare and therefore service times and exception handlers are assumed to generally not be on the critical path.

If the exception is resumable, such as a page fault or floating point exception, exception handling is easily enabled through separately executing strands with private state. When an executing strand encounters an exception, only that strand needs to handle the exception. To enable precise semantics for faulting memory instructions, the point of the exception in the strand is defined to occur immediately before the instruction triggering the fault. The fault is initially stalled and the strand issuing the faulting instruction is blocked.

The instruction steering logic can be stalled once a branch has been encountered. The PC of the last branch instruction is then pushed to the stack, and the frontend of the core is flushed. The exception handler then executes on the register file of the strand issuing the faulting instruction, scheduling instructions directly out of the pipeline latches rather than the instruction queues. In this way, the instruction queue state of each strand can be maintained. The register state of the strand’s instruction queue can be written out to the stack and the fault is handled normally. When the handler fin-
ishes, the register state is restored, and the core begins fetching instructions starting at the PC of the next instruction stored in the stack. The instruction steering mechanism is unstalled, completing the recovery process, and execution continues as normal.

For more general interrupts and support for operating systems, the saving of the entire context of the decoupled execution can be performed. Preciseness is defined as the current basic block being executed by the strand executing furthest ahead. On an interrupt, the strand steering logic is stalled at the next branch, and all instruction queues and data queues are allowed to empty. The PC of the stalled branch can then be pushed onto the state, and an interrupt handler can store the register file state to memory before processing the interrupt.

7.2.4 Scheduling Arbitration Scheme

One concern is arbitration of issue slots among strands. At the high level, a compiler-directed scheme works best when there are instructions available to issue from each strand. Building on that property, a proper strand arbitration scheme to enable peak latency tolerance means managing resources on the chip. In this dissertation, several arbitration schemes are considered in addition to a baseline round-robin arbitration.

**Leading Strand First and Following Strand First** Strands are prioritized based upon strand number, which indicates the relative position in the software pipeline. For example, the strand executing furthest ahead is given the number zero, while a strand executing behind is given a larger number. The *Leading Strand First* arbitration scheme prioritizes strands executing ahead. The insight behind this approach is that memory requests
are generated as early as possible, in order to tolerance the most amount of memory latency. The *Following Strand First* arbitration scheme prioritizes strands executing behind. This scheme enables eager consumption of data values from the FIFO queues, and assumes that the critical path of program completion is the strand executing furthest behind.

**Full Resource First** When resources such as data queues, instruction queues, and the MAU start to fill up, significant stalls occur across the chip. In the *Full Resource First* scheme, if the instruction queue or data queue of a particular strand is full, that strand is given priority to schedule.

**Branch Strand First** When a branch is observed by the instruction steering logic, instruction fetch stalls. While the branch is waiting to be resolved, there is a significant probability that another strand will run out of instructions, reducing the instantaneous instruction issue rate. *Brand Strand First* prioritizes the strand with the branch instruction, in order to determine the next basic block as soon as possible. This helps ensure that each strand will have instructions to issue at any point in time, while not being biased to a particular strand. In other words, the branch can exist in any strand, and that strand will execute quickly to determine the next section of code.

### 7.3 Evaluation

#### 7.3.1 Instruction Queue Sizing

One of the concerns with using a hardware partitioning scheme is the potential hardware overhead for instruction queues. As a single instruction stream is used, there must be enough entries in the instruction queues to
buffer the instructions in strands executing behind and waiting on data from memory. When the instruction queues fill up, the frontend of the core must stall, therefore removing the chance of further instruction execution from strands executing ahead. Figure 7.3 presents the performance improvement of the hardware steering scheme compared with an in-order core baseline while varying the number of entries in the instruction queue. With 16 entries in each instruction queue, a performance improvement of nearly 58% is achieved on average. Moving to 32 and 64 entries for each instruction queue provides an extra 8% and 2% performance improvement. Overall, an instruction queue with size 64 and above does not provide much extra benefit, and only the cg and heat benchmarks see a 7% performance boost.

7.3.2 Scheduling Arbitration

Figure 7.4 presents the evaluation of the strand scheduling arbitration schemes. Overall, the baseline round robin scheme never performs the best on any of the data parallel benchmarks. In the case of the cg benchmark, it performs
nearly 20% behind other arbitration schemes. Prioritizing either the leading or following strand can lead to significant variation in performance. Depending on the benchmark, both schemes either have higher or lower performance, and usually are opposite of each other. Moreover, there is not a high correlation of application patterns on performance with the leading or following strand prioritization schemes. The resource full arbitration scheme works generally well and improves over the round-robin baseline by 5% on average. However, in the blackscholes and cutcp benchmarks, this scheme performs poorly due to data-dependent control flow. Finally, prioritizing strands with a branch, then falling back to round-robin arbitration, is the best arbitration scheme. This arbitration scheme does not suffer any performance pitfalls on any of the benchmarks. As mentioned previously, eagerly executing branches avoids the case when instruction fetch stalls, and helps ensure that there are a significant number of instructions available from each strand in the instruction queues.
### Figure 7.5: Performance of software and hardware instruction partitioning normalized to the in-order core baseline.

#### 7.3.3 Performance and Energy Evaluation

With the instruction queue sizing fixed and the strand arbitration fixed, the impact of the techniques on the performance of the decoupled core is evaluated. Figure 7.5 presents the performance of the software and hardware instruction partitioning schemes. Overall, with the decrease in overhead instructions, performance is improved by over 13% when utilizing hardware partitioning and a single instruction stream. In some cases, the reduction in branch and memory disambiguation instructions can improve performance by 29% up to 47% in the `cg` and `sobel` benchmarks.

The most significant effect is on energy consumption. Using the physical design modeling for Rigel as found in Chapter 4, Figure 7.6 presents the relative energy consumption for both approaches. Overall, the energy consumption is reduced significantly across the benchmarks, by 11% on average. The reduction in static instructions by moving to a single instruction stream significantly reduces the number of dynamically executed instructions. Energy is significantly reduced because instruction fetch is one of the largest contributors.
Figure 7.6: Energy consumption of hardware instruction partitioning normalized to software partitioning.

contributors to dynamic energy consumption on Rigel. Specifically, the reduction in instruction cache energy dominates the relatively small increase in energy consumption for the increased instruction queue size.
CHAPTER 8

AVOIDING CONTROL FLOW LOD

Data-dependent control flow is a loss-of-decoupling event that can severely impact performance. Figure 6.1 depicts that the memory-accessing strand must stall and wait until the next basic block is resolved. This control flow LOD event removes the memory-accessing strand’s ability to execute ahead and tolerate memory latency. Avoiding this LOD event is important as it is a common code pattern found in parallel applications today. In this chapter, the control flow LOD event is avoided through control speculation. For this technique, mechanisms for checkpointing and restoring strand context state are added, as well as methods for reducing energy consumption on misspeculations.

8.1 Control Speculation

Figure 8.1 shows the approach to the data-dependent control flow LOD event found in Figure 6.1. In the two strand case, control flow stalls occur in the memory-accessing strand when it is waiting for the memory-consuming strand’s result to decide the next basic block to execute. Recall that all strands in a decoupled architecture travel down the same control flow path together. Requiring the memory-accessing strand to wait on the memory-consuming strand restricts its performance to that of in-order execution.
Control speculation, traditionally used in wide-issue OOO processors, can enable the memory-accessing strand to continue execution and tolerate memory latency. This dissertation investigates a checkpoint and rollback speculation mechanism, where the memory-accessing strand checkpoints its state and continues execution. When the speculated branch in the memory-consuming strand is executed, a correct prediction results in the retirement of the checkpoint and signifies correct execution. A misprediction of the branch represents misspeculation, which results in restoration of the state of the memory-accessing strand and execution down the correct control path.

One of the main concerns with utilizing control speculation on throughput-oriented data parallel processors is energy efficiency. When branch mispredicts occur causing restoration of strand state, the energy spent during the speculation becomes wasted energy. The success and energy efficiency are therefore tied to successful branch prediction. While many branches are biased to a certain path and can be easily predicted, a significant num-
ber data-dependent branches may not be biased and be difficult to predict. These unpredictable branches are the source of energy-efficiency, and in this dissertation branch confidence prediction is leveraged to help combat mis-speculation and energy waste.

8.2 Confidence Prediction

One concern with speculation is the additional energy overhead spent during mis-speculations. When a branch prediction is incorrect and the checkpointed state needs to be recovered, energy is wasted on the instructions whose results will never be used. The execution of those instructions requires energy from all parts of the core pipeline including the instruction caches, scheduling logic, register files, and data caches. From an energy-efficiency perspective, the core pipeline should only speculate when the prediction is perfect and there is little chance for a mis-speculation.

The method proposed for speculation in decoupled architecture only speculates across branches not found in the memory-access strand executing furthest ahead. As a result, the number of dynamic branches that can be speculated across is greatly reduced, along with any potential energy overhead. However, for the remaining branches that can initiate speculation, it is desired to know which of those instructions have a high chance of correct prediction.

One way to do this is by augmenting the branch predictor with confidence prediction [32, 33]. In general, confidence prediction works by observing the number of correct predictions versus incorrect predictions, and comparing the two in order to decide the likely outcome of a branch prediction. Confidence prediction has been proposed for reducing power consumption on wide-issue
Figure 8.2: Control speculation enabled decoupled architecture.

single-core processors [34], and this dissertation investigates its relevance to 1000-core throughput-oriented data parallel processors.

For each entry in the branch prediction table, a confidence entry is added that consists of a saturating counter that has the ability to be reset [32]. When a correct prediction occurs, the entry in the table is updated by incrementing the counter; on a misprediction the counter is reset to zero. In other words, the counter keeps track of how many times in a row the branch was correctly predicted. A branch is considered confident when the counter reaches a threshold, the particular value of which is evaluated in this chapter.

8.3 Implementation

To enable control speculation, the most significant change to the core architecture is the addition of checkpoints. Figure 8.2 presents the updated core,
with checkpointing enabled for the register files, and tail pointers for the load buffers, store buffer, and data queues. These checkpoints are required for correct restoration of strand state after a misspeculation. To reduce area overhead, the register file checkpoints are shadow-bitcells [35] which simplify the SRAM design. The data queue checkpoints require storing the state for the speculation strand’s tail pointers. Additionally, a bit is stored to mark whether the head is valid and comparator hardware is also required to keep track of the case when all non-speculative data is removed and the restored size should be zero. Similar checkpoint hardware is used in the load and store buffer tails found in the MAU. For each branch that needs to be speculated, a register file checkpoint is required. The hardware requirement for control speculation is relatively modest as compared to control speculation on wide-issue OOO processors. Speculation only occurs on data-dependent branch instructions, a subset of all branches.

### 8.3.1 Initiating Speculation

The speculation process begins when a branch belonging to a strand not executing the furthest ahead is seen in the strand steering logic. In the non-speculation case, the instruction fetch would stall until the branch was resolved. Speculation enables the instruction fetch to avoid stalling. Provided there are checkpoints available, the branch instruction is inserted into the correct instruction queue and special “register checkpoint” instructions are inserted into all strands executing ahead. Checkpoints of the load and store buffers are taken immediately, while the data queues and register files checkpoints are performed later. The participating strands are registered, so that proper recovery actions can be taken later. Instruction fetch contin-
ues as usual, and only stalls if a data-dependent control flow instruction has reached the strand steering logic and there are no checkpoints free.

When a “register checkpoint” instruction reaches the head of the instruction queue for a strand, the scoreboard is inspected to see whether there are outstanding instructions that have yet to write to their destination registers. When there are no such outstanding instructions, the strand’s register files and data queues are checkpointed. Execution continues normally until the initiating branch instruction is executed.

8.3.2 Retirement and Rollback Process

When all the strands have checkpointed, the initiating instruction is eligible to be executed. If the branch target of the initiating instruction was correctly predicted, the checkpoint across all strands is retired. Retirement has low overhead, as it simply involves updating the head pointer to what is the active checkpoint. More importantly, retirement is not on the critical path, so it can be done without affecting instruction schedule and issue.

If the branch target was mispredicted, recovery must occur and the state is restored using the checkpoints. Fetch and instruction issue is stalled, enabling the last instructions to be written successfully into the instruction queues. The pipeline must be flushed, including flushing the instruction queues for the participating strands. The checkpoints are restored for the data queues, register files, and MAU. The strand register scoreboards are cleared, the branch predictor updated, and the program counter is set to the correct branch target. The pipeline flush takes three cycles, which is the critical path for checkpoint recovery. The other restoration can happen in the background,
Figure 8.3: Performance improvement of control speculation.

resulting in the mispredict penalty to be three cycles, one cycle larger than a non-speculative branch mispredict.

8.3.3 Confidence Prediction

Confidence prediction is implemented by using a saturating counter table in conjunction with the branch predictor table. When a branch instruction’s target is determined, the counter entry is updated. If the prediction was correct, the entry is incremented or remains at the saturated value. If the prediction was incorrect, the counter is reset. To avoid warmup times, the counter originally is initially set to being saturated. When a data-dependent branch reaches the strand steering logic, the associated confidence is compared against the threshold. If the branch is not confident, fetch stalls and the core must wait on that branch to resolve until it may continue. If the branch has high confidence, normal execution and speculation can occur.
Figure 8.4: Performance improvement as the number of checkpoints for control speculation is varied.

8.4 Evaluation

In this section the effectiveness of the proposed control speculation mechanisms are evaluated. Specifically, the ability of speculation to avoid control flow LOD events and improve performance is explored. Additionally, the number of checkpoints needed to achieve high performance, as well as the success of confidence prediction to eliminate control speculation waste, is investigated. These experiments are performed on the Rigel 1024-core data parallel processor with parallel application kernels that exhibit data-dependent control flow.

Figure 8.3 presents the performance of two- and three-strand versions of the blackscholes and cutcp application kernels both with and without speculation. In this experiment the number of checkpoints is assumed to be unlimited. Both of these application kernels also suffer from the memory indirection LOD, and thus cache misses can be exposed in the two-strand case. Despite the memory indirection LOD, speculation provides a substantial performance boost of 14.7% and 12.4% even in the two-strand case.
Figure 8.5: The correct prediction rate of the speculation mechanism as the branch confidence is increased.

However, once the memory indirection LOD is removed and another strand is extracted, the full potential of control speculation is revealed. In the three-strand version of the application kernels, performance is improved by 38.7% and 33.7%. The furthest executing strand can continue to issue instructions to create a software pipeline of memory accesses. The achievable performance improvement over the in-order processor is 71.2% on average, significantly better than the 17.0% in the two-strand non-speculative case similar to DAE.

Figure 8.4 presents how the number of checkpoints affects the performance of speculation on applications with control flow LOD. Specifically, the effect on the three-strand version of the application kernels is shown. Overall, increasing the number of checkpoints to one, two, three and four improves performance incrementally by 15.8%, 10.1%, 9.0%, and 1.0% on average. For more than three checkpoints, the performance benefit is negligible, mainly due to the fact that data-dependent branches represent a small subset of the total number of branches in these and many application kernels that need to be speculated on. Because the number of checkpoints is tied to the number of instructions executed ahead to tolerate memory latency, more checkpoints
Figure 8.5 presents the impact of using the confidence prediction scheme on the ability of control speculation to improve the prediction success rate. The threshold for confidence is set at one, two, four, and eight correct predictions. With blackscholes, many of the data-dependent branches are highly biased, resulting in a low mispredict rate of 3%. However, on the cutcp benchmark, the misprediction rate is initially 17%. By using branch confidence prediction requiring 1, 2, 4, and 8 correct predictions, the misprediction rate is reduced to 10%, 7%, 4% and 1% respectively. Overall, branch confidence prediction successfully eliminates biased branches from being speculated.

In terms of performance, utilizing a confidence predictor reduces the ability of the decoupled core to speculate. Figure 8.6 presents the result of using branch confidence prediction on performance. Overall, the effect on performance correlates with the threshold used: the larger the threshold, the lower the performance. The performance reduction is due to the fact that...
when mispredictions occur, the saturating counter is reset, requiring several
correct predictions before speculation is again available.

However, performance is still greatly improved over a decoupled core with-\nout speculation. With blackscholes and cutcp, the performance improve-\nment is 30% on average when a threshold of two is utilized. Performance
improvement drops to 25% and 17% on average when thresholds of 4 and
8 are used. The promise of utilizing branch confidence prediction is high,
and an interesting area of future work is investigating other branch confi-
dence schemes. Performance loss can be avoided with a more sophisticated
branch prediction methodology utilizing history bits and larger tables, at
some additional cost in hardware complexity.
Decoupled cores provide substantial performance improvement over in-order processors, while leveraging compiler information to keep hardware complexity lower than pure hardware methods such as out-of-order execution. However, a significant amount of extra area and energy are spent over an in-order processor to provide this performance. In this chapter, mechanisms are created and tradeoffs made to create more area- and energy-efficient decoupled cores for data-parallel processors. The most significant hardware additions are that of the per-strand register files for scratch space and the data queues for communication between strands.

9.1 Data Queues

Critical to the design of a decoupled architecture are the data queues used for inter-strand communication, which make up a large portion of the hardware complexity overhead. Traditional decoupled architectures use FIFOs for each communication lane between the strands. Considering the performance improving techniques previously discussed in the last chapter, the increased number of simultaneously executing strands also increases the number of FIFO data queues. In the proposed multiple-strand decoupled core, four strands of parallelism can be extracted resulting in twelve FIFO data queues needed to facilitate all the possible communication lanes.
9.1.1 Virtualized Data Queue Scheme

A significant amount of hardware waste exists in the traditional data queue scheme due to the lack of extracted strands. While one approach might be to power-gate the unused FIFOs to save leakage power, this does not promote area efficiency as the transistors for the unused FIFOs still take up considerable space on the chip. An approach that is also area-efficient is to
Figure 9.2: Assembly code for sparse-matrix vector multiply inner loop. Arrows and circled registers denote inter-strand communication through data queues. The table counts the communication occurrences between strands.

virtualize the FIFO data queues onto shared hardware buffers. Virtualization is powerful, as the size of each queue can be tailored to the requirements of the application based upon information known by the compiler. The compiler has accurate information about how many strands are extracted and which inter-strand communication lanes are not being used. There is also static information about communication frequency that the compiler can exploit.

In particular, the number of communication occurrences between strands as found in loops can potentially give a strong indication of what the relative sizing and allocation of each virtual data queue should be.

Figure 9.1 shows the hardware buffers and the configuration tables each strand uses to access their virtual FIFO data queues, which is configured by the compiler. For each strand, a single hardware buffer is used, and data about the index location of the head pointer, tail pointer, the allocated size, and starting offset for each virtual data queue is stored. The compiler determines the size and starting offset by identifying loops in the code and the proportion of static communication occurrences between each pair of strands. For strand communication lanes that frequently communicate data, larger portions of the hardware buffer are allocated. Figure 9.2 depicts this
process in a code example, identifying strand communication using arrows and counting the communication occurrences in the table. For example, strand 2 receives two data messages from strand 1 for every message from strand 0, and thus sizes its hardware buffer into approximately one-third and two-thirds partitions. The hardware tables are written using special instructions when the strand is initiated, and data queue resizing can only be safely performed during a barrier operation across strands.

9.1.2 Hardware-managed Register-renamed Data Queue Scheme

Leveraging compiler knowledge to virtualize data queues can provide significant improvement in hardware utilization. However, the compiler may only have static knowledge of program execution, thus the exact amount of communication between strands at runtime is not known. As a result, the partitioning provided by the compiler may not be ideal, and significant hardware waste may exist.
An alternative is utilizing dynamic register renaming to facilitate data movement between the strands. The approach trades off additional hardware complexity for high utilization of the hardware buffer used for inter-strand communication. However, the added complexity is much less than register renaming found in contemporary out-of-order execution, as renaming will only occur on registers to handle inter-strand communication. The private register file for each strand will not be renamed, as each strand still executes in order. In other words, this is a technique not strictly for eliminating register data hazards, but for improving utilization of the dedicated hardware to improve area and energy efficiency.

Figure 9.3 presents the approach, which utilizes the data queues as a separate physical register file and adds renaming tables and a freelist. The renaming tables are FIFO queues to enable multiple entries, which is necessary for instructions consuming both operands from the data queue. The dynamic register renaming scheme operates similarly to those found in out-of-order architectures, though only for strand communication.

When an instruction has a destination register that is mapped to the data queue, corresponding to the fact that inter-strand communication should occur, a free physical register is pulled from the freelist for use in renaming. The instruction for the strand sending data is updated with the destination physical register name so that the result of the operation can be written successfully into the physical register file. The corresponding location in the table of the destination strand is updated with the physical register name, so that the later instruction consuming the communicated data can read the physical register name and successfully read the operand from the physical register file used for the data queues. For the renaming mechanism to work properly, the single instruction stream from the hardware partitioning scheme
must be used. This ensures that the order in which the renaming table is read is synchronized correctly. Later consuming instructions read their rename table and update their operand register name. These rename tables are small, as for a core supporting four strands each strand only needs three rename FIFO queues with a size of two entries. Once the operand is read, the register name can be released back onto the freelist.

9.2 Strand Context and Core Complexity

Each strand has a private instruction queue, register file, and hardware buffer for storing the virtual data queues. Typically, each thread context in multithreading has the ability to issue multiple instructions in a superscalar core. However, decoupled architectures have properties which potentially enable the reduction in hardware context complexity significantly over multithreading. In practice, each strand consists of a subset of the instruction stream, either memory-accessing or memory-consuming instructions. One strand may consist purely of memory load and address generation instructions, while another strand may consist only of floating point instructions. Considering that Rigel and other data parallel processors architect a single load-store unit and floating point unit [1] per core, enabling a strand to issue more than one instruction per cycle can result in overprovisioning.

The number of instructions that each strand can issue per cycle is reduced. Considering that decoupled architectures interleave execution between strands on a single core, there is little impact on the peak throughput of each core. Reducing the issue width for each strand has profound implications on hardware complexity. To facilitate a two-wide issue strand, the register files must be designed with four read ports and two write ports.
Figure 9.4: Performance of the data queue implementations normalized to the 16-entry FIFO.

If each strand is architected to issue one instruction, only two read ports and one write port are required, leading to substantial energy consumption improvement. Similar benefits in complexity reduction are observed in the instruction and data queues.

9.3 Evaluation

In this section, the impact on performance and energy consumption for the techniques proposed in this chapter are evaluated.

9.3.1 Data Queue Implementation

Figure 9.4 presents the performance comparison between the three techniques as the aggregate number of data queue entries among the strands is increased from 16 to 128. The performance numbers are normalized to the 16-entry
FIFO case. As there are twelve FIFO queues for communication in the baseline data queue design, entries are evenly allocated and the remaining entries are given to downstream strands.

There are certain cases where additional buffering in the data queues has minimal impact on the performance of the benchmark. For example, blackscholes, cutcp, and kmeans see low benefit from increasing the size of the data queues bigger than 16. This is because of the relatively small memory latencies that need to be tolerated, and immediate consumption of values from the data queues when available. However, the other six benchmarks see substantial improvement when increasing the size of the data queues. While all techniques can achieve similar performance with enough entries in the data queues, the compiler and hardware managed techniques often require 2x to 4x fewer entries to do so. Most interesting is the competitiveness of the compiler management technique compared to the hardware renaming technique.

9.3.2 Strand Context Complexity

Reducing strand complexity has a relatively small effect on performance. Figure 9.5 presents the effect of reducing issue rate of a two-wide and four-wide core in the Rigel data parallel processor. In the two-wide case, the number of instructions a strand can issue per cycle is reduced to one, while in the four-wide case the maximum strand issue rate is reduced to three and two instructions per cycle. The result is significantly reduced complexity at a 9% cost in performance on average in the two-issue case and a 4% in the four-issue case. The mri benchmark is a particular outlier, where limiting the issue width of a strand limits the peak issue rate significantly to reduce the
performance more than 21%, though only in the two-wide issue case. With more aggregate issue width, the decoupled architecture is less susceptible to decreasing per-strand issue width in half. This is mainly due to limited ILP found in each strand, which is generally around two instructions in the in-order execution strand.

Reducing strand complexity has a much larger effect on reducing energy consumption. As mentioned before, the number read and write ports on each strand’s register files, data queues, and instruction queues are reduced, leading to greater opportunities to reduce energy consumption. Figure 9.6 presents the effect on energy consumption of reducing issue rate of a two-wide and a four-wide core in the Rigel data parallel processor. Generally, the amount of energy saved is highly correlated to the type of instructions executed in the benchmark. Memory operations which access the cache hierarchy see less effect, while two-operand instructions that also write to the register file see a larger impact. Overall, the impact on energy is relatively consistent across the benchmark, with an average energy savings of nearly 10% for the two-wide case. Despite the performance improvement over two-wide issue
of 34% on average, four-wide issue spends an additional 30% extra energy to achieve the improvement, a negligible improvement in energy-efficiency. However, reducing strand complexity improves energy-efficiency significantly, through reducing energy consumption by 11% and 19% by enabling three-wide and two-wide issue per-strand. The 30% performance improvement compared with the 10% energy overhead makes moving to four-wide issue a net energy efficiency win.
Decoupled architectures depend on compiler extraction of strands from the original thread by examining the dependency graph and partitioning the program into memory-accessing and memory-consuming instruction streams. This chapter presents the strand extraction algorithm used on the benchmarks in this dissertation, including the techniques described in Chapter 6 and 7 for handling loss-of-decoupling. For the purposes of this dissertation, the partitioning process is performed on leaf node functions which make up the majority of execution time of data parallel applications. That is, the tasks being executed in parallel are expressed as a single function, using aggressive inlining of smaller function whenever possible. Using leaf node functions simplifies handling of parameter and return values, and enables different transformations on a function-by-function basis which allows additional flexibility.

Past research has demonstrated code partitioning and optimization [36, 37] for decoupled architectures, and the approach adopted in this dissertation is similar. In short, memory dependence chains are identified and strands are created along memory-access and memory-consumption lines. However, support for avoiding the loss-of-decoupling techniques and the support for compiler-directed instruction steering to the partitioning scheme are added. The process to extract strands consists of four phases of strand assignment:
loads, stores, and control flow; unassigned instructions; final partitioning; and hardware mapping.

While the techniques can be easily included into a compiler as other decoupled techniques before it such as DSWP [38], for the purposes of this dissertation binary translation of the identified functions is performed. More detail on the implemented toolflow is presented later in this chapter. Complete details on implementation in LLVM, GCC, or other compiler toolchains are outside the scope of this dissertation.

10.1 Strand Partitioning Scheme

This section presents that strand partitioning scheme used for the proposed decoupled architecture and provides an example of the strand extraction process using the inner loop of the sparse-matrix vector multiply of the cg benchmark. Figure 10.1 presents the dependency graph of the original inner loop in which the addi instruction acts as a loop counter which provides data for itself and the slli shift instruction used for calculating memory addresses.

10.1.1 Phase 1: Partition Loads, Stores, and Control Flow

Phase 1 partitions the load, store, and control flow instructions into their proper strands in order to achieve decoupling. Loads are partitioned into strands according to how many levels of loads are required to generate its address. Floating point instructions that get all of their operations from loads are marked and allocated into one strand. Control flow instructions should ideally be serviced in strand 0, then communicated to all other strands. If
the decision cannot be determined by strand 0, loss of decoupling can occur. Stores that may alias with loads must have their addresses calculated in strand 0 in order to prevent loss of decoupling caused during the distribution of \texttt{st_addr} instructions. Similarly to partitioning the load instructions, both the store and control flow instructions are partitioned by determining how many levels of loads must be traversed.

Floating point chains are analyzed, and the compiler creates a preliminary code schedule based upon data flow and floating point instruction latency. If floating latency cannot be tolerated by overlapping the schedule in floating point computation, the floating point chains are broken into separate strands at the point at which the largest exposed latency exists. At this phase, the maximum number of strands to be extracted is determined, which can potentially be more than hardware supports. In this case, Phase 4 reduces the number of strands during hardware mapping.
Figure 10.2: Diagram of the example code’s dependency graph after Phase 1 is complete.

Figure 10.2 presents the result of the partitioning of Phase 1 on the dependency graph of the example loop from cg. As both the address of the stw and inputs of the bgt instructions do not depend on memory, they can be placed in the lowest level strand, strand 0. As these instructions exist in all strands, they are partitioned as such. There is a single level of indirection found in the loop, which results in the dependent load being assigned to strand 1.

10.1.2 Phase 2: Partition Unassigned Instructions

Phase 2 uses the identified loads, control flow, and stores and their assigned strands to identify and partition the address generation instructions. The backslice of instructions from a particular load, branch, or store in the dependency graph are considered. Only the instructions in the backslice found before reaching a load operation are considered for inclusion in the same strand as the initial instruction. As the backslice of some loads, branches,
and stores inspected may share instructions, priority is given to the lower level strand. In the case that there are address generation instructions that are shared between strands, the value is communicated to the other strands in order to reduce circular dependences and promote decoupling. As a special case, the backslice of instructions providing data to store instructions are placed in the highest level strand.

Starting at strand 0, for each load, store, or control flow at the current strand level:

1. Look backwards in the dependency graph, marking all unassigned instructions as belonging to this strand.

2. Terminate when an instruction has already been assigned.

3. Mark terminating instruction as also assigned to this strand.
Original C Code: \[
\text{for (int}\ i=0;\ i<\ \text{array\_size};\ i++)
\text{vec\_out}[i] = \text{vec\_a}[i] + \text{vec\_b}[i];
\]

<table>
<thead>
<tr>
<th>Original ASM Code</th>
<th>Compiler directed</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;\text{BB1}&gt;:)</td>
<td>(&lt;\text{BB1}&gt;:)</td>
</tr>
<tr>
<td>add $r8, $r6, $r3</td>
<td>0 add $r8, $r6, $r3</td>
</tr>
<tr>
<td>add $r9, $r5, $r3</td>
<td>0 add $r9, $r5, $r3</td>
</tr>
<tr>
<td>ldw $r9, $r9, 0</td>
<td>0 ldw $DQ, $r9, 0</td>
</tr>
<tr>
<td>ldw $r8, $r8, 0</td>
<td>0 ldw $DQ, $r8, 0</td>
</tr>
<tr>
<td>fadd $r8, $r9, $r8</td>
<td>1 fadd $r8, $DQ, $DQ</td>
</tr>
<tr>
<td>add $r3, $r7, $r3</td>
<td>0 add $r9, $r7, $r3</td>
</tr>
<tr>
<td>add $r2, $r2, 1</td>
<td>0 add $r3, $r3, 4</td>
</tr>
<tr>
<td>stw $r8, $r9, 0</td>
<td>0 add $r2, $r2, 1</td>
</tr>
<tr>
<td>clt $r8, $r4, $r2</td>
<td>0 staddr $r9, 0</td>
</tr>
<tr>
<td>bne $r8, &lt;BB1&gt;</td>
<td>1 stdata $r8, 0</td>
</tr>
</tbody>
</table>

Figure 10.4: Assembly code example illustrating strand extraction on inner loop of vector addition.

10.1.3 Phase 3: Final Partitioning

With all instructions marked, strands are created. Figure 10.3 shows the final partitioning of instructions. Loads and their address generating instructions are included in their assigned strand, using the data queues to communicate their resulting values to the dependent strand utilizing copy instructions. Stores are split, with the address providing instruction \text{st\_addr} assigned to strands 0 and the data providing instruction \text{st\_data} assigned to strand 2, enabling the compute to pass the data directly to the MAU. Instructions that are shared among strands are placed in the lowest level strand, and communicated to other strands using copy instructions.

10.1.4 Phase 4: Mapping Strands to Hardware

During partitioning, more strands can be created than hardware has resources for. For example, a function with many levels of memory indirection may generate five strands, more than the four strands that our decoupled
architecture supports. In the case that too many strands are generated for the hardware to handle, the number of strands is reduced to the maximum size permitted by hardware by combining some of the strands.

In general, strands adjacent to one another are considered for merging together. Specifically, several passes are performed using different priorities. During each pass, the number of strands is reduced by one. The first pass identifies loss of decoupling events and merges those strands, as those strands have the least amount of performance improvement. Next, strands that contain few instructions incur extra overhead for communication and control flow, which can hurt performance efficiency. Finally, lower level strands are chosen as a last resort to reduce the amount of strands to the maximum allowed by the architecture.

After the code has been partitioned into strands, the compiler is responsible for generating the machine code. At this phase, the bits in the instruction set are written to identify a particular instruction with a particular strand. Aside from that difference, the code is generated as in-order code, respecting original memory ordering found in the higher-level language.

10.1.5 Code Example

Figure 10.4 presents the inner loop of vector addition example code and its partitioning into strands. This code reads in two vectors, adds them, and stores the result. The original code is presented alongside the partitioned code, with corresponding instruction even between the two. During Phase 1, all the loads and stores are identified and heights recorded. The loads in this example are not memory dependent, and so they are assigned to strand 0. The store instruction is split across the two strands, with the address being
provided by strand 0 and the data by strand 1. The control flow is placed in strand 0 and the result of this partitioning is that strand 0 can continue to execute and generate memory requests while strand 1 is still waiting on data from memory to perform the floating-point addition. As such, memory latency tolerance is exhibited.

10.2 Compiler Support and Implementation

In this dissertation, the aforementioned strand extraction algorithm is implemented in a binary translator. The toolflow used in the experiments is presented in Figure 10.5. The original code compiled using LLVM, resulting in a ELF binary. The developed binary translator decompiles the binary into a CDFG of the program for a provided function. The decoupled strand extraction scheme is performed on the function, and the resulting function...
assembly code is generated. This assembly code is then recompiled with the original program with LLVM to create the final ELF binary.

While this toolflow is functional and bug-free for the benchmarks, there are several manual steps that could be automated. First, the identification of the function to transform is performed manually, though all functions in the code can potentially be transformed. Similarly, the generated assembly code is separate from the original program, while it could be directly rewritten in the binary. Finally, the binary translation implementation does not perform register allocation, and leverages the original allocation from the LLVM compiler.
In this dissertation, an optimized decoupled architecture which avoids loss-of-decoupling pitfalls while maintaining low complexity and energy overhead is proposed. This chapter compares the proposed decoupled architecture with other latency tolerant techniques such as those mentioned in Chapter 2. In-order with hardware prefetching, out-of-order, multithreaded, hardware scout, and decoupled architectures are compared. Specifically, absolute performance and performance per unit energy are compared. Each approach has fundamental strengths depending on the code pattern. In addition to application kernels, microbenchmarks are used to understand key differences between the latency tolerance techniques.

The techniques are compared using the Rigel 1024-core data parallel processor. Each technique is first optimized by utilizing energy-efficient mechanisms, with similar effort as used in the decoupled technique. The energy-efficiency mechanisms include complexity-effective hardware, speculation pruning, and memory access distribution. Considering the optimized techniques, the ability of each technique to scale performance to cores with wider issue widths is compared.

Finally, the techniques and energy efficiency are compared by performing a comprehensive design space exploration using microbenchmarks to isolate these code patterns and highly parallel benchmark kernels from visual computing. This is accomplished through detailed performance models of each
latent tolerance technique and a detailed physical design model generated using synthesis and analytical SRAM models for a 45nm CMOS manufacturing process.

## 11.1 Energy-efficient Latency Tolerance

In this dissertation, the energy efficiency of five latency tolerance techniques for future 1000-core data-parallel processors is compared while advocating for the proposed decoupled latency tolerance technique. For each latency tolerance technique, energy-efficient extensions to improve these techniques over performance-focused implementations are implemented. Figure 11.1 presents the impact on energy efficiency when deploying these techniques on a 1024-core data parallel processor with a two-wide issue core and the data parallel benchmarks found in Section 5. The extensions improve hardware prefetching by 8%, out-of-order execution by 36%, multithreading by 23%, hardware scout by 32%, and decoupled by 8% on average across the benchmarks. Additional details on evaluation methodology are found in Section 5.

**In-order with Hardware Prefetching:** In the design space exploration, both next-line and stride-based prefetching are considered [3]. Table-based stride prefetching is used to improve energy-efficiency by better predicting the access stream and improving timeliness over next-line prefetching. The software is enabled to statically switch off prefetching if the compiler or programmer knows it will not be useful.

**Out-of-Order Execution:** Complexity-effective out-of-order hardware is included in the design space exploration to improve energy efficiency. To simplify the instruction window, dependence-based instruction steering which
Figure 11.1: Average energy-efficiency improvement of each latency tolerance technique using complexity-effective tuned implementations on two-wide core and data-parallel benchmarks.

uses simple hardware FIFOs is included [39]. Aggressive partitioning of the core into floating point and integer pipes is also enabled to reduce physical register file complexity, and also partition the reorder buffer and register renaming free list to reduce the number of read and write ports required [16].

**Multithreading:** Cache set hashing is implemented at the L1 and L2 cache to more uniformly distribute memory addresses, reducing the probability of set conflicts between threads and therefore cache contention [40, 41]. Specifically, a simple bitwise XOR hashing function using the memory address as an input is implemented, which minimizes added hardware complexity. Additionally, software is enabled to statically configure the number of active threads on each core to further reduce the effects of cache contention.

**Hardware Scout Prefetching:** The extra energy spent fetching and preeexecuting the instruction stream is a concern, and can be improved by avoiding preeexecution of useless instructions [42]. For example, floating point instructions do not generally contribute to generating prefetches and can
be not executed to reduce functional unit energy consumption. Finally, to reduce cache contention the scout thread is limited to generate four prefetcheds during a preexecution session.

**Decoupled Execution:** The instruction overhead commonly found in decoupled architectures can be partially alleviated by reducing the complexity for each strand context. Each strand has relatively low ILP due to dependencies, due to the decoupled partitioning placing memory-accessing and memory-consuming instructions in separate strands. Therefore the number of instructions that each strand can issue per cycle is reduced, leading to a lower number of read and write ports in the register files, instruction and data queues.

11.2 Performance

11.2.1 Issue Width Sizing

Figure 11.2 presents the performance of the four latency tolerance techniques normalized to a one-wide issue in-order core on the Rigel 1024-core data parallel processor. The four-wide issue pipeline builds upon the two-wide issue pipeline by enabling some combination of one store, one load, one floating point addition, one floating point multiplication, and two integer operations to issue each cycle, similarly to current Intel multi-core processors. For multithreading and out-of-order, the best performing configuration is presented. If prefetching is not useful, hardware prefetching and hardware scout present the in-order execution only.

Overall, performance scalability on in-order execution is very limited, though using hardware prefetching can improve performance by 19% and 7% using
Figure 11.2: Comparing performance of in-order baseline, two- and four-way SMT, OOO with 16- and 64- entry instruction window, hardware scout, and decoupled architecture as issue width scales from one-, two-, and four-wide with realistic number of functional units.

two- and four-wide issue across the benchmarks. Hardware scout is also somewhat limited, and nearly matches hardware prefetching in performance. In some benchmarks such as blackscholes, dmm, fft, and heat hardware scout improves performance by 1%, 5%, and 5% over hardware prefetching on average as issue width is increased from one to four. As issue-width increases, there is less competition for execution resources between the scout thread and the main thread. Additionally, the additional execution slots enable prefetches to be uncovered more quickly while the cache miss is outstanding.

Simultaneous multithreading, out-of-order execution and decoupled techniques see substantial benefits, even at the four-wide issue width. Overall, multithreading increases performance by 10%, 10%, and 17% over hardware prefetching on average as issue width is increased from one to four. Multithreading does not perform as well overall, due to memory-intensive benchmarks where cache contention can occur such as cg, dmm, fft, heat, and
Overall, out-of-order improves performance by 21%, 37%, and 40% and decoupled 22%, 34%, and 28% over hardware prefetching on average as issue width is increased from one to four. Decoupled generally performs similarly to out-of-order execution, though performance can be limited in data-dependent control flow benchmarks such as blackscholes and kmeans. Out-of-order is the most robust across all the benchmarks, but realistically-sized instruction windows limit the performance on streaming benchmarks such as cg and sobel.

11.3 Design Space Exploration

In this section a comprehensive design space exploration is performed for the four latency tolerance techniques. The performance, chip area, and energy consumption are compared between the techniques.

11.3.1 Design Space Setup

The design space is listed in Table 11.1. All major components in the 1024-core Rigel chip including issue width, number of functional units, and cache sizing are varied. Additionally, specific parameters for each latency tolerance technique are varied.

Figure 11.3 details how Pareto curves are presented. The energy consumption and the performance improvement of a configuration are normalized to a one-wide issue in-order baseline. Energy efficiency of a configuration is defined as the ratio of normalized performance improvement to the normalized energy consumption. As performance increases and runtime decreases, component activity and power consumption increase, as seen in the curve.
Table 11.1: Design space exploration parameters.

<table>
<thead>
<tr>
<th>Shared Parameters</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue Width</td>
<td>1-4</td>
</tr>
<tr>
<td>BTB</td>
<td>8-64 entry</td>
</tr>
<tr>
<td>GShare Table</td>
<td>32-256 entry</td>
</tr>
<tr>
<td>LDQ and STQ</td>
<td>8-32 entry</td>
</tr>
<tr>
<td>Miss Queue</td>
<td>4-16 entry</td>
</tr>
<tr>
<td>Caches</td>
<td></td>
</tr>
<tr>
<td>L1D (4-way)</td>
<td>1-4kB</td>
</tr>
<tr>
<td>L2 (8-way)</td>
<td>32-128kB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hardware Prefetch Parameters</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Nextline Prefetching</td>
<td>1-4 cache lines</td>
</tr>
<tr>
<td>Stride Prefetching Table</td>
<td>8-32 Entries</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Out-of-Order Parameters</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Reorder Buffer</td>
<td>32-128 entry unified or partitioned</td>
</tr>
<tr>
<td>Instruction Window</td>
<td>32-entry Assoc. or (16) 16-entry FIFOs</td>
</tr>
<tr>
<td>Physical Registers</td>
<td>128-256 Unified or Split FP/INT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Multithreaded Parameters</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>2-4 threads</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decoupled Parameters</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Queues</td>
<td>16-128 entry partitioned</td>
</tr>
</tbody>
</table>

Figure 11.3: Sample figure showing energy-efficiency contours and curve for fixed power consumption.
representing a fixed power budget. By comparing points on the power curve, the relative performances of the latency tolerance techniques can be assessed.

In this section the comprehensive design space exploration for the five latency tolerance techniques is performed. Performance, chip area, and energy consumption are compared between the techniques. In improving performance significantly, total energy is actually lower than the baseline as leakage energy is avoided.

### 11.3.2 Energy Efficiency on Microbenchmarks

The energy efficiency of the latency tolerance techniques on the identified code patterns is investigated using microbenchmarks.

Figure 11.4 shows the compute-intensive microbenchmark that exhibits exposed functional unit latency. In this pattern there are very few memory accesses and so prefetching techniques such as hardware prefetching and hardware scout cannot provide any benefit. Out-of-order execution can ex-
tract ILP from a single thread, with moderate hardware complexity and a small instruction window. Multithreading is able to tolerate even more floating point latency with four threads, with low hardware complexity providing the best energy-efficiency overall. The proposed decoupled architecture is able to extract three ways of strand parallelism to tolerate floating unit latency similar to out-of-order but with lower hardware complexity. Overall, energy-efficiency is improved in out-of-order by 17%, in multithreading by 30%, and in decoupled by 28% over the in-order core with hardware prefetching at the 150 W power budget.

Figure 11.5 shows the data-dependent control flow microbenchmark when the inner-loop branch is taken only 75% of the time. Similar to the compute-intensive microbenchmark, hardware prefetching and hardware scout cannot improve performance. Both out-of-order, decoupled, and multithreading have the ability to tolerate data-dependent control flow. However, the speculation and dynamic hardware in out-of-order consumes significant energy and
Figure 11.6: Energy efficiency on cache-favorable microbenchmark with low degree of data sharing.

results in multithreading being more energy efficient. Additionally, the decoupled technique experiences loss-of-decoupling when data-dependent control flow is present, enabling efficient speculation with lower hardware complexity than out-of-order execution. Overall, energy-efficiency is improved in out-of-order by 13%, in multithreading by 77%, and in decoupled by 27% over the in-order core with hardware prefetching at the 150 W power budget.

Figure 11.6 shows the data-sharing intensive microbenchmark where there is no data sharing. With a significant amount of predictable misses, hardware prefetching with in-order execution provides substantial performance. While hardware scout provides better prefetching capability, the added extra performance is offset by the extra energy required to dynamically preexecute the instruction stream. When multithreading is chosen, high levels of cache contention increase cache activity and energy consumption. In general, multithreading prefers larger caches to reduce contention, at a higher price in energy consumption. Out-of-order with small instruction windows and low
complexity is required to tolerate the relatively small L2 access latency. The decoupled architecture provides high memory-level parallelism and utilizes low hardware complexity, yielding high energy-efficiency. Overall, energy-efficiency is improved in out-of-order by 12% and in decoupled by 28% over the in-order core with hardware prefetching at the 150 W power budget.

Figure 11.7 shows the pointer-chasing microbenchmark where each node in the linked list is updated. Neither hardware prefetching nor hardware scout can easily predict the access stream, leading to lower performance and significant energy waste. Out-of-order only needs a small instruction window to tolerate memory latency and execute the update portion of the microbenchmark while a memory access is outstanding, leading to performance improvement at low added complexity. The decoupled architecture proposed in this dissertation exhibits similar behavior, at significantly lower complexity and higher energy-efficiency. The independent nature of multithreading enables more memory accesses to be outstanding, which enables substantial
Figure 11.8: Energy efficiency on memory-streaming microbenchmark with data in the L3 Cache.

performance improvement over out-of-order or decoupled execution. Overall, energy-efficiency is improved in out-of-order by 23%, in multithreading by 39%, and in decoupled by 28% over the in-order core with hardware prefetching at the 150 W power budget.

Figure 11.8 shows the memory-streaming microbenchmark, when the data is found on-chip in the L3 Cache. Both hardware prefetching and hardware scout can tolerate a significant amount of memory latency. However, prefetches are reactive and only generated on cache misses, resulting in the core spending significant time stalled. Out-of-order can execute independent work and tolerate long memory latencies, though a large instruction window is needed to do so. Multithreading is able to uncover more memory misses, tolerating memory latency successfully. Scaling the number of threads to tolerate more latency is possible, at the added cost of register file scratch space and additional cache resources to support the additional thread contexts. Decoupling is the more efficient technique, enabling similar memory latency
tolerance to out-of-order but with a much larger effective instruction window and substantially reduced hardware complexity. Overall, energy-efficiency is improved in hardware scout by 4%, in out-of-order by 17%, in multithreading by 39%, and in decoupled by 56% over the in-order core with hardware prefetching at the 150 W power budget.

11.3.3 Average Energy and Area Efficiency on Microbenchmarks

Figure 11.9(a) presents the mean result of the design space exploration, with the performance improvement and energy consumption of each architectural configuration averaged across the five microbenchmarks. Overall, the proposed decoupled core architecture is the most energy-efficient on average, with multithreading being preferred over out-of-order. While out-of-order is not the most energy efficient on any single microbenchmark, out-of-order provides robust energy-efficiency across the benchmarks as compared with decoupled or multithreading, which can experience pitfalls from data-dependent control flow and cache contention. Prefetching can tolerate some memory latency, but cannot help tolerate floating-point unit latency, or tolerate much memory latency in randoms walks of pointer-based structures. The decoupled proposal is robust across all the microbenchmarks, and does not suffer either a performance or energy pitfall. Across the microbenchmarks, energy-efficiency is improved on average in hardware scout by 2%, in out-of-order by 20%, in multithreading by 24%, and in decoupled by 36% over the in-order core with hardware prefetching at the 150 W power budget.

Averaging energy-efficiency across the microbenchmarks weights the importance of each code pattern equally. The weighting and relative importance of the five microbenchmarks were explored and varied from 0 to 100%
Figure 11.9: Pareto optimal curves for energy consumption and chip area of each latency tolerance technique averaged across all five microbenchmarks.
in increments of 10%, and the most energy-efficient technique for all possible weighting combinations was determined. The result was the proposed decoupled latency tolerance technique being most energy-efficient across 49% of the weighting combinations, followed by out-of-order at 33% and multithreading at 18%.

Figure 11.9(b) presents the mean performance improvement compared with the absolute chip area for the design space exploration on the 1024-core data-parallel processor. The five microbenchmarks are equally weighted. Voltage scaling provides significant performance improvement over in-order core with hardware prefetching without cost in area. At lower performance improvement levels, multithreading and hardware scout are more area efficient than the proposed decoupled approach. To reach higher performance, in-order with hardware prefetching, multithreading and hardware scout must rely on larger caches to continue to improve performance, and eventually become less area-efficient than the decoupled core architecture. The memory-latency tolerance ability of decoupling is less sensitive to cache-sizing and as a result the area-efficiency scales much more favorably when equally weighting the microbenchmarks. Overall, the proposed decoupled architecture can substantially improve performance over in-order with hardware prefetching while keeping chip area under 400 mm$^2$.

11.3.4 Energy Efficiency on Visual Computing Benchmarks

Overall, the microbenchmarks illustrate the profound effect of the code patterns on both performance and energy consumption of the latency tolerance techniques as no single technique emerges as the clear choice. To further understand the implication of code patterns, a design space exploration is
Figure 11.10: Pareto-optimal energy efficiency for latency tolerance techniques for each data-parallel benchmark and averaged across benchmarks. Data is normalized to hardware prefetching.
undertaken for the latency tolerance techniques on the benchmarks from visual computing applications. Figure 11.10(a) presents the most energy-efficient configurations for each benchmark compared with the in-order baseline. Figure 11.10(b) presents the average of a single configuration across all the benchmarks. Again, energy-efficiency is determined by the ratio of normalized performance improvement over normalized energy consumption.

On average, out-of-order, multithreading and decoupled techniques provide similar levels of energy-efficiency, with decoupled being preferred due to the majority of benchmarks having memory-intensive patterns. These techniques provide 66%-143% better energy efficiency than the baseline and hardware scout on average. However, upon closer inspection, decoupled and multithreading techniques perform quite differently depending on the code pattern. Data-dependent control flow dominated benchmarks such as blackscholes, cutcp, and kmeans favor multithreading due to the ability to execute another thread instead of stalling while waiting to determine the next basic block. Memory-intensive benchmarks such as fft, heat, and sobel strongly favor decoupled, as the cache footprint is kept low by maintaining fewer threads per core. On the benchmarks that favor multithreading, the difference in choosing multithreading over decoupled as the latency tolerance technique improves energy efficiency by 11% to 61%, while in the memory-intensive benchmarks choosing decoupled over multithreading improves it by 38% to 112%. Out-of-order provides general performance improvement across all the benchmarks similar to the microbenchmarks, but again this benefit is negated by the energy overhead required for dynamic scheduling. Hardware scout is not competitive with hardware prefetching in terms of energy-efficiency, being dominated by the extra energy to preexecute the instruction stream to generate prefetches.
Overall, the proposed decoupled architecture proposal enables a much larger degree of energy efficiency. By enabling control speculation, the performances of blackscholes and cutcp are competitive with those of out-of-order and multithreading. The addition of loss-of-decoupling avoidance through extracting additional memory-accessing and memory-consuming strands provides additional performance over DAE on all of the benchmarks. On average the energy-efficiency improvement of the proposed decoupled latency tolerance technique is 28% to 89% over hardware prefetching, out-of-order, multithreading, and hardware scout prefetching alone.
Decoupled architectures are an ILP technique to improve latency tolerance and high performance for a single thread. To improve throughput performance, many chips combine ILP and TLP latency tolerance techniques. The combination of VLIW or OOO and multithreading is attractive to tolerate long memory latencies and fill otherwise unused issue slots. Effectively, combining techniques also improves performance robustness across different code patterns while improving energy efficiency.

In this chapter, the relative benefits of combining latency tolerance techniques is investigated. Particular, the ability of combining multithreading and decoupling to provide further improvement in energy efficiency is researched. While the proposed decoupled architecture can provide robust energy-efficiency on irregular codes through the use of control speculation, a hybrid approach can lead to even larger improvements, particularly when there is low cache contention.

12.1 Implementation

The varying capabilities of the latency tolerance techniques imply that no single scheme will be able to offer robust energy efficiency across a wide variety of workloads. A hybrid directly combining these techniques is possible, but the additive cost of supporting multiple techniques can be prohibitive.
To minimize additional hardware overhead and energy consumption, this dissertation proposes combining multithreading and decoupled execution. Both techniques provide both memory and functional unit latency tolerance and require similar hardware requirements. By leveraging these two techniques, the significant hardware complexity of out-of-order execution, as well as the speculation waste and lack of functional unit latency tolerance of both hardware prefetching and hardware scout, are avoided.

Figure 12.1 presents the proposed core architecture used to implement hybrid latency tolerance. A total of four contexts can be supported, each of which can either be a strand context from decoupled execution or a hardware thread context. Unlike prior decoupled architecture proposals [22], all strands interleave their execution on a single pipeline. To facilitate strand communication during decoupled execution, data queues are provided using the virtualization technique. Finally, each context has support for issuing multiple outstanding load instructions. The parameters of this core architecture including issue width, functional units, and cache sizing, are varied for the design space exploration.
The architecture supports the following execution modes:

1. **Decoupled Execution** (1 thread of up to 4 strands)

2. **Multithreaded Execution** (4 threads of 1 strand each)

3. **Multithreaded Decoupled Execution** (2 threads with 2 strands each)

### 12.1.1 Selection of Execution Mode

The execution mode for a data parallel task is selected based upon behavior to improve performance and to avoid energy waste. There are several ways that the mode can be selected, including dynamically with runtime and operating system support, or statically by the compiler or programmer. In this dissertation automatic static selection is considered, which performs within 15% on average of oracular static selection. While dynamic selection is left for future work, the potential benefit of implementing it seems to be relatively low.

The proposed scheme for choosing the execution mode focuses primarily on leveraging decoupled execution to provide latency tolerance, and falls back to multithreaded execution when data-dependent control flow exists and decoupling is not useful. If the inner-loop of a task of a data parallel program has data-dependent control flow determined by computation, multithreading is chosen. Otherwise, decoupled execution is used. If only two strands can be extracted from the code, two-way multithread decoupled execution mode is used. If more than two strands can be extracted, single-threaded decoupled execution is enabled.

At the end of the selection process, one version of task code is generated and run on the Rigel 1024-core data parallel processor described. The hybrid
Figure 12.2: Pareto-optimal energy efficiency for latency tolerance techniques for each data-parallel benchmark and averaged across benchmarks. Data is normalized to hardware prefetching.

mode selection scheme often correctly chooses the best mode, and is within 15% energy efficiency of the best selection on average for the benchmarks. Although the direct knowledge of the application is used to statically pick the execution mode, the scheme can also be implemented into a compiler by detecting data-dependent control flow.

12.2 Evaluation

12.2.1 Energy Efficiency on Visual Computing Benchmarks

Figure 12.2 presents the most energy-efficient configurations for each benchmark compared with the in-order baseline. The proposed decoupled and multithreading hybrid is also presented. Again, energy-efficiency is determined by the ratio of normalized performance improvement over normalized energy consumption.
Figure 12.3: Pareto optimal energy-efficiency of multithreading, decoupled, the statically-chosen hybrid, and an oracle choosing the best hybrid configuration of multithreading, multithreaded decoupled, and decoupled on data-parallel benchmarks. Data is normalized to hardware prefetching.

Overall, the hybrid proposal enables a much larger degree of energy efficiency. By detecting data-dependent control flow and enabling the more efficient multithreading on blackscholes and cutcp, energy efficiency is substantially improved. All other benchmarks operate in decoupled mode, as multiple strands can be extracted. On average the energy-efficiency improvement of the hybrid latency tolerance technique is 14% to 91% over hardware prefetching, out-of-order, multithreading, hardware scout prefetching, and decoupled techniques alone.

12.2.2 Static Hybrid Latency Tolerance Mode Evaluation

The ability of the static mode selection scheme for hybrid latency tolerance technique is evaluated. Figure 12.3 presents the comparison of multithreading, decoupled execution, and the hybrid latency tolerance technique with an oracle choosing the best configuration. Overall, the static algorithm per-
forms within 15% of the oracle. The scheme is able to statically choose the best mode on many of the benchmarks, with the exception of cutup, dmm, kmeans, and mri. In these benchmarks, choosing multithreading over decoupled (dmm), two-way multithreaded decoupled over three-strand decoupled (kmeans) and two-way multithreaded decoupled over multithreading (cutup and mri) can improve performance significantly. Though it may be possible to dynamically select the latency tolerance mode and is potentially interesting future work, the benefit is likely low.

12.2.3 Combining other Latency Tolerance Techniques

The other possible combinations of latency tolerance techniques are compared by performing a design space exploration. Figure 12.4 presents the average improvement of combinations of hardware prefetching, out-of-order execution, multithreading, and decoupled execution. Unlike the proposed
design in this dissertation, the maximum number of contexts was not limited when considering decoupling and multithreading together. Considering the similarities of decoupled execution and out-of-order, they were not combined. Overall, adding hardware prefetching improves energy-efficiency by 5% on out-of-order, 8% on multithreading, and 7% on decoupled on average. When multithreading is combined with out-of-order, energy-efficiency cannot be substantially improved due to high hardware complexity. However, combining decoupling with multithreading provides the biggest boost, providing an additional 28% improvement in energy-efficiency over decoupled alone. The low hardware complexity and robust performance across varying workloads result in combining decoupling with multithreading being the best choice by a significant margin. Finally, adding prefetching to the decoupled multithreading combination does not improve energy-efficiency significantly, as any potential performance improvement is offset by an increase in cache activity.
In this chapter, prior research related to the decoupled architecture presented in this dissertation is discussed. Decoupled architectures rely on the compiler to extract parallelism instead of costly hardware structures and as a result do not require duplicated execution of memory access instructions or compute instructions as found in other designs. The proposed decoupled architecture provides both memory and functional unit latency tolerance through extracting up to four semi-independent strands of execution. Additionally, the proposed decoupled architecture leverages hardware and software techniques to minimize hardware and instruction execution overheads. While other designs require register files per thread or even processor fetch, decode and functional units to be replicated, the proposed decoupled architecture enables an area-efficient design without this requirement.

13.1 Latency Tolerance Techniques

13.1.1 Compiler-Enabled Techniques

VLIW and EPIC processors leverage the compiler to schedule instructions to avoid both functional and memory latency using loop unrolling and speculative code motion [21]. While these designs can remove the need for associative instruction windows, they require both large register files to hold
in-flight values and memory disambiguation hardware. Additionally, loop unrolling and code motion techniques grow the code footprint considerably, which causes greater impact on accelerator architectures with small instruction caches per core. Even with speculative code motion, VLIW and EPIC designs can still be sensitive to memory latency and can stall if not enough software pipelining is done. The proposed decoupled architecture is a non-data speculative technique that tolerates all levels of memory latency using completely separate streams of execution and hardware data queues which store only in-flight values from long-latency operations as opposed to every value.

Decoupled software pipelining (DSWP) is a compiler technique that creates parallel tasks from loops in sequential programs [38]. These pipeline-parallel tasks are mapped onto physical thread contexts in a CMP system, made up of either high-performance wide-issue out-of-order or VLIW/EPIC processors that already have some degree of memory latency tolerance. This is a different motivation than the proposed decoupled architecture, which targets highly parallel systems and applications on simple in-order processors. DSWP partitions are based upon strongly-connected components in the dependency graph, while estimating the latency per instruction to combine these SCCs into the threads run on the processor. Following this partitioning scheme can result in memory dependences existing in a single thread which can lead to exposed latency on simple in-order processors. The proposed decoupled architecture assumes that variable-latency memory instructions are the most costly, and specifically partitions between memory access instructions and their consuming instructions to avoid exposed latency. DSWP is complementary to memory-latency tolerant techniques such as those found in the proposed decoupled architecture and can improve performance [43].
13.1.2 Pre-execution Techniques

Hardware scout threads were proposed to enable memory latency tolerance for in-order [13, 14] and out-of-order designs [42]. This is done by pre-executing the memory access stream. Hardware scout is not as effective for programs which have memory indirection. Flea-flicker two-pass pipelining [12] improves on hardware scout by adding a large instruction buffer to handle dependent memory operations and adds a result store buffer to enable the reuse of pre-executed instructions to combat data dependences. These schemes duplicate execution of the memory access instruction stream and only extract two ways of parallelism, and are sensitive to traditional prefetching concerns such as timeliness and accuracy of speculation. The proposed decoupled architecture does not require duplicate execution of the memory access stream and is not data speculative, while having up to four strands of concurrent execution.

In-order continual flow pipelines (iCFP) [44] and Simultaneous Speculative Threading (SST) [18] allow execution to continue normally under a cache miss by deferring dependent instructions and their operands to a hardware queue. The deferred instructions are executed once the cache miss returns. This is an improvement over previous pre-execution work as no duplicate instruction execution is required except under a misspeculated branch dependent on a cache miss. However, memory disambiguation hardware is required in order to detect violations. The proposed decoupled architecture does not rely on adding large structures, such as large deferred instruction queues, or multiple checkpoints to provide memory latency tolerance. Another difference is that iCFP and SST spend overhead cycles fetching and decoding instructions only to defer them to the deferred queue. This is a reactive mechanism that can
potentially waste issue slots that could be used for executing independent instructions. By using the compiler to partition dependent instructions into strands, independent work can potentially be uncovered more quickly. iCFP and SST also are limited to extracting only two streams of execution, while the proposed decoupled architecture extracts up to four.

13.1.3 Helper Thread Techniques

Slice processors [45] implement prefetching by dynamically extracting the memory miss instruction stream and then executing that stream in parallel with the main thread to prefetch data. When a miss occurs, the backslice of instructions is identified that caused the miss. The extracted stream can then be used to actively prefetch into the data cache. Like other prefetching techniques, accuracy and timeliness are not guaranteed and executing the prefetching instruction stream creates duplication of executed instructions. The proposed decoupled architecture is not data speculative and does not prefetch data, nor does it require duplicate execution of the memory accessing stream. Slice processors require several large additional data structures, including a slice cache, an instruction stream slicer, and the candidate selector predictor table. The proposed decoupled architecture requires much more meager hardware overheads, only enough to buffer instructions and the data communicated between strands.

Helper threads [11] instantiate a partial thread of execution to improve the performance of the main thread. This thread is either programmer or compiler generated, and can either run completely independently or be controlled by the main thread. The main goal of the helper thread is to generate useful prefetches and warm up the data cache for the main thread. Similar to
other prefetching techniques, helper threads are sensitive to timeliness and can cause cache contention and thrashing with the main thread if not properly controlled. The proposed decoupled architecture is not a prefetching technique and is not sensitive as helper threads are. Also, helper threads duplicate execution of the address generation stream while the proposed decoupled architecture does not.

13.1.4 Decoupled Techniques

Decoupled access/execute (DAE) provides memory latency tolerance by partitioning a program into two strands, one for executing memory instructions and one for executing compute instructions [22]. The programs are run on separate processors and to handle dependences between compute and memory, hardware queues are used for message passing communication. Later related work to DAE included investigating silicon implementations, code partitioning, strand balancing and memory latency tolerance limitations [46, 47, 24, 25]. While DAE enables parallelism and can allow the memory thread to provide memory latency tolerance, it is unable to handle loss-of-decoupling events which degrade performance. The proposed decoupled architecture utilizes additional strand parallelism and control speculation to remove this performance degradation. Additionally, DAE requires in-order completion of memory accesses into the FIFOs, and restricts data to only be from loads or to stores. The proposed decoupled architecture enables out-of-order completion of messages and general data communication through the communication queues. Finally, the proposed decoupled architecture utilizes SMT to share fetch and execution resources and enable efficiency not found in DAE.
An alternative design, the Multiple Instruction Stream Computer (MISC),
atttempts to improve over DAE by executing up to four concurrent strands
on separate processors [23]. However, this design does not ensure correct
load-store ordering between strands and can only have two strands that ac-
cess memory. MISC requires 24 statically-allocated hardware queues, and
the efficiency of the MISC design is degraded if these queues or the separate
processors cannot be fully utilized. The proposed decoupled architecture en-
ables all four strands to access memory with correct memory ordering, which
is enabled by a mixed hardware and software approach to memory aliasing
detection. MISC is also limited and does not support control speculation.
The proposed decoupled architecture utilizes SMT to share fetch and execu-
tion resources and enable efficiency not found in MISC.

The DS architecture is a DAE design utilizing OOO processors with asso-
ciative instruction windows and reorder buffers [48]. In addition to utilizing
in-order hardware, the proposed decoupled architecture enables control spec-
culation and up to four strands with correct load-store ordering while sharing
frontend, execution, and data queue resources.

Other contemporary decoupling work involves hardware partitioning and
SMT [49]. In this work, the authors propose hardware partitioning of inte-
ger and floating-point instructions into separate threads in order to provide
memory latency tolerance using large instruction queues to hold dependent
floating-point instructions while they wait for the miss to return. These in-
struction queues needed can be more than an order of magnitude larger than
those required for the proposed decoupled architecture, and this technique
is limited to floating-point applications. Additionally, this technique suffers
from memory indirection and compute-dependent memory accesses, which
the proposed decoupled architecture supports. The technique also only sup-
ports SMT of different threads on either the EP or AP, unlike the proposed decoupled architecture which uses SMT across strands.

13.2 Energy-efficient Mechanisms

13.2.1 Efficient Memory Latency Tolerance

There has been substantial work done on developing a single energy-efficient memory latency tolerance technique. Energy-efficient techniques were proposed to reduce the number of dynamic instructions executed during hardware scout sessions [42]. Other techniques such as BOLT [50] and Flea-flicker [12] also dynamically tolerate memory latency and reduce energy over OOO processors. These investigations are done purely in the context of out-of-order uniprocessors and single-threaded applications. This dissertation uses a similar hardware scout technique in context of 1000-core data parallel processors, highly parallel workloads, while being implemented on in-order and multithreaded cores.

13.2.2 Compiler-directed strand partitioning

The braids architecture also uses dynamic partitioning to execute braids, or independent dataflow subgraphs, in parallel on distributed execution units [51]. In the braids architecture, the compiler uses a single bit in the ISA to denote the start of a braid, which the hardware uses to allocate the instructions execution resources. However, the braids architecture focuses on wide-issue single-threaded processors with dynamic hardware of out-of-order designs, including register renaming, control speculation using checkpoints, and reorder buffers, making it significantly more complex than the proposed decoupled
architecture for 1000-core data parallel processors. Also, the proposed decoupled architecture has different parallelism extraction, focusing on tolerating memory latency through decoupling memory-access and consumption.

13.2.3 Compiler Management of Register Files

Prior work has investigated utilizing the compiler to manage register files to save energy using operand register files [52, 53]. Operand register files rely on the compiler to keep data closer to the functional units by controlling data movement between levels in a multi-level register file hierarchy. In contrast, this dissertation leverages the compiler to allocate data queue resources to strands, in order to maximize the utility of the architected hardware buffers. Using a loop-based scheme and a hardware buffer per strand enables lower complexity and more energy efficiency. Finally, this dissertation is the first to study the energy efficiency of leveraging compile-time information to virtualize data queues on a hardware buffer.

13.3 Energy and Performance Modeling

13.3.1 Energy of Cores

Recent research has developed methods for investigating energy-performance tradeoffs when considering in-order and out-of-order uniprocessors [54]. Using CACTI and synthesis flows, the work explores the design space of a single processor, varying the architectural parameters. This dissertation utilizes a similar approach, but builds upon it considerably. Multithreading, hardware scout, and decoupled architectures are investigated as potential latency tolerance techniques and evaluated on a 1000-core data-parallel processor.
Additionally, key code patterns and benchmarks are used to help evaluate these approaches and their suitability for use in future chips.

13.3.2 Design Space Exploration for Throughput-Oriented Architectures

Several recent works investigate the design space for multi- and manycore processors. [55] investigated area-efficient throughput-oriented core architectures and found that in-order multithreaded processors were preferred over out-of-order architectures. [56] presented early work on the design space of multicore CMPs, and also enumerated application characteristics that are important. [57] presented GPU benchmarks and performed sensitivity analysis to chip design parameters. This dissertation improves over past work by focusing on exploring the design space and giving special attention to modeling the physical design, including energy consumption. Additionally, this dissertation identifies five distinct code patterns that greatly affect the performance and energy consumption of latency tolerance approaches.

13.4 Hybrid Processor Design

Hybrid latency tolerance approaches have been previously investigated and implemented in the context of serial-performance focused processors. Intel’s i7 [58] and IBM’s POWER7 [59] implement out-of-order and simultaneous multithreading. IBM’s POWER6 architecture implements multithreading and a restricted form of hardware scout, called load-lookahead prefetching [60]. This dissertation investigates these techniques in the context of 1000-core data-parallel processors, and performs a thorough design space exploration of energy consumption on data-parallel workloads. This dissertation
finds that multithreading combined with decoupled latency tolerance techniques provides significant benefit over prior approaches.

Other work proposes dynamic hardware partitioning of integer and floating-point instructions into separate threads in order to provide memory latency tolerance [49]. While this technique also supports multithreading of different threads on separate processors, this approach relies on out-of-order hardware for dynamic scheduling. The DS architecture is a hybrid decoupled and out-of-order processor with associative instruction windows and reorder buffers [48]. This dissertation finds that dynamic-scheduling hardware is relatively energy-inefficient on highly parallel workloads and 1000-core data-parallel processors.
CHAPTER 14

CONCLUSIONS

Throughput performance on data parallel processors such as GPUs is fundamentally limited by chip power budgets. Therefore, memory and functional unit latency tolerance, the key enabler for throughput performance, must be designed with energy-efficiency in mind. However, prior latency tolerance techniques either have too much complexity, or suffer performance and energy pitfalls when commonly occurring code patterns are exhibited during application runtime.

This dissertation proposes a novel decoupled architecture developed specifically for high energy-efficiency. While traditional decoupled architectures have energy consumption and performance pitfalls, techniques to extract more strand parallelism, implement control speculation, and enable a single decoupled instruction stream are developed. Additionally, hybrid latency tolerance techniques leveraging both multithreading and decoupling are developed to provide robust performance and energy-efficiency. While multithreading and decoupling in isolation have performance pitfalls on different code patterns commonly found in data parallel workloads, enabling a hybrid latency tolerance can avoid these pitfalls and improve energy-efficiency significantly.

High-fidelity performance and physical design models are leveraged to perform a comprehensive design space exploration to compare the energy efficiency of common latency tolerance techniques on a 1024-core data paral-
lel processor. By designing a decoupled architecture specifically for energy efficiency, robust energy-efficiency across a wide range of code patterns is achieved. The proposed decoupled architecture improves energy-efficiency over other techniques by 28% to 89% on data parallel benchmarks. A hybrid of multithreading and decoupling can improve energy-efficiency by another 14% on average across data parallel benchmarks.
REFERENCES


