EFFICIENT MEMORY-LEVEL PARALLELISM EXTRACTION WITH DECOUPLED STRANDS

BY

NEAL CLAYTON CRAGO

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Adviser:

Associate Professor Sanjay J. Patel

ABSTRACT

We present Outrider, an architecture for throughput-oriented processors that exploits intra-thread memory-level parallelism (MLP) to improve performance efficiency on highly threaded workloads. Outrider enables a single thread of execution to be presented to the architecture as multiple decoupled instruction streams, consisting of either memory accessing or memory consuming instructions. The key insight is that by decoupling the instruction streams, the processor pipeline can expose MLP in a way similar to out-of-order designs while relying on a low-complexity in-order micro-architecture. Instead of adding more threads as is done in modern GPUs, Outrider can expose the same MLP with fewer threads and reduced contention for resources shared among threads.

We demonstrate that Outrider can outperform single-threaded cores by 23-131% and a 4-way simultaneous multi-threaded core by up to 87% in data parallel applications in a 1024-core system. Outrider achieves these performance gains without incurring the overhead of additional hardware thread contexts, which results in improved efficiency compared to a multi-threaded core.

To my mother and father, who taught me that hard work is the foundation for success.

To my wife Christine, who has given me the support I needed throughout my graduate studies.

ACKNOWLEDGMENTS

I would like to thank my adviser Sanjay Patel for his encouragement and persistence throughout my graduate career. His grand challenge and expectation for greatness in the field of computer architecture have enabled higher-quality research.

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

I would like to thank my parents Steven and Louise for always making me 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 a lot 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.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AP	Access processor
CMP	Chip multiprocessor
DAE	Decoupled access/execute
EP	Execute processor
GPU	Graphics processing unit
ILP	Instruction-level parallelism
INO	In-order
L1	Level one
L2	Level two
MLP	Memory-level parallelism
OOO	Out-of-order
S#	Strand number
TLP	Thread-level parallelism

CHAPTER 1

INTRODUCTION

Execution stalls due to memory latency are the limiting factor for performance in highly parallel workloads. Current methods for mitigating the effect of memory-related execution stalls based on dynamically extracting instruction-level parallelism (ILP) or statically expressing thread-level parallelism (TLP) result in unnecessary complexity and inefficiency for both hardware and software. The inefficiencies come from the *indirect* exposure of memory-level parallelism (MLP). An architecture that can expose MLP directly at the core level, while still leveraging complexity-efficient ILP and TLP techniques, is necessary for enabling future parallel throughput-oriented chips integrating 100s to 1000s of cores. In this thesis we propose Outrider, a core micro-architecture to exploit intra-thread MLP present in data-parallel workloads with the goal of reducing complexity while increasing performance relative to existing designs that rely on ILP or TLP.

The continued increase in transistor budgets has allowed more cores to be integrated on-die with each new process generation, thus matching the execution demands of highly parallel workloads. However, there has not been a commensurate increase in memory bandwidth nor decrease in memory latency; this is a design constraint referred to as the *memory wall*. As such, current and future throughput-oriented systems are fundamentally limited by the memory subsystem. While architectural techniques to exploit TLP and ILP can be used to increase the utilization of execution resources on-die, it

is the MLP that they indirectly expose that provides aggregate speedups for designs constrained by the memory wall. Integrating larger on-die caches can mitigate the impact of the memory wall, but larger caches are not a panacea due to diminishing returns for many workloads having little temporal and spatial locality in their access streams while requiring large cache footprints to see performance improvements.

OUTRIDER is a core micro-architecture for throughput-oriented chips supporting 1000s of threads that directly leverages the MLP present in parallel workloads. The key insight is that data parallel workloads possess a significant number of memory operations that can be issued concurrently *intra-thread* if the architecture allows the memory access stream to be sufficiently decoupled from the rest of the computation stream. OUTRIDER leverages hardware and software mechanisms to decouple a single thread of execution into two or more semi-independent streams, or *strands*: those responsible for performing memory accesses and the other consuming memory values. Strands perform only a portion of the work of a sequential thread and by definition communicate data and synchronize with one another. The design achieves high performance with low complexity by increasing the level of MLP extracted from a single thread of execution without the use of complex out-of-order mechanisms or multiple threads of execution needed for multi-threading. Our results show that we can achieve 23-131% more performance than a baseline single-threaded core of comparable complexity while also providing up to 87% better performance compared to a 4-way multi-threaded core which requires more area and results in greater cache pressure.

CHAPTER 2

MOTIVATION

In this chapter we explore the impact of the memory subsystem on performance, and explore ILP, TLP, and decoupled architectures and how they help to alleviate performance loss due to exposed memory latency. We motivate the use of decoupled architectures such as OUTRIDER as an alternative to ILP and TLP processor designs.

2.1 Memory System Impact on Performance

Figure 2.1 presents the impact of the memory wall on performance for several visual computing benchmarks on a 1024-core system by comparing a baseline single-threaded two-wide in-order processor against two scenarios: the baseline augmented with an L1 data cache that never misses and has a zero-cycle access latency (Perfect L1D), and the baseline idealized with perfect branch prediction, zero-cycle functional units, and no misses and zero access latency in both the L1 instruction or data cache (Idealized INO). Additional information on the baseline used can be found in Table 7.1 (on page 39).

We find that most of the performance that is lost in our 1024-core system is attributable to the memory system, rather than fetch, branch prediction, or functional unit latencies. Removing all stalls due to memory latency more than doubles performance (2.7x), whereas idealizing the entire core increases performance by 3.6x. This demonstrates the impact of the memory system

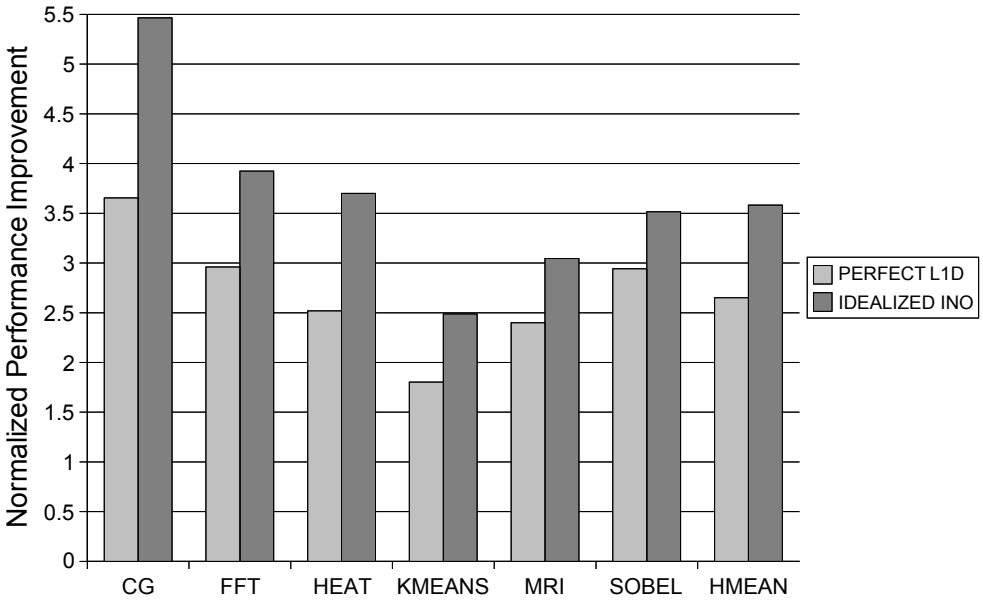

Figure 2.1: Performance limit study comparing the baseline in-order 1024-core design against perfect L1D cache and idealized case with no artificial stalls. Addressing memory stalls can bring an accelerator system significantly close to the idealized case.

on performance, and indicates the importance of efficient mechanisms for extracting MLP.

2.2 Out-of-Order Processors

Out-of-order processors enable applications to execute instructions out of order with respect to one another by using an instruction window. This enables more instructions to be available to execute at a single time, reducing the impact of memory stalls and improving performance. The additional structures that are needed to enable contemporary OOO execution include instruction schedulers, reorder buffers, physical register files, register renaming hardware, and load/store queues. Contemporary out-of-order processors such as Intel's i7 [1] and IBM's POWER7 [2] also utilize control and data

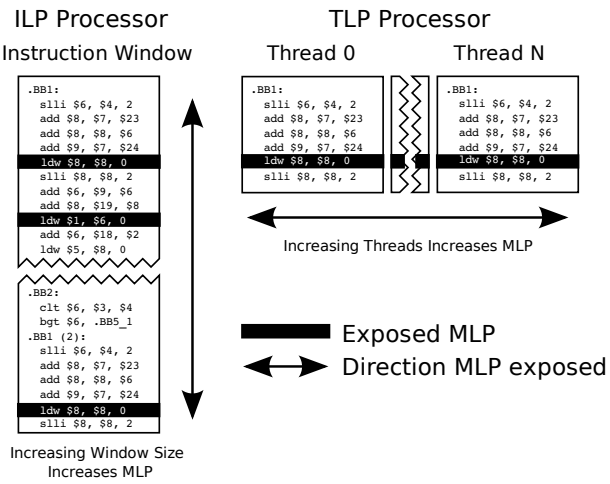

Figure 2.2: Comparison between ILP and TLP's ability to extract MLP. ILP designs expand their instruction windows to extract more MLP, while TLP designs increase the number of threads to expose more MLP. In the code excerpt, additional load operation can be executed for memory-level parallelism (MLP).

speculation to further increase the amount of instructions available for execution. Additionally, superscalar execution is used to increase performance by further exploiting ILP and issuing multiple instructions per cycle.

Figure 2.2 shows how increasing the instruction window in OOO processors enables memory latency tolerance. OOO processors can generate MLP by finding independent memory operations in the instruction window that are available to issue to the cache hierarchy. Memory latency tolerance also occurs by issuing other independent instructions. To accomplish a large instruction window, OOO processors utilize associative structures to store and schedule independent instructions for execution. Instructions that are dependent and wait on memory are kept in the associative structures until the memory access completes. During the memory access, other instructions not dependent on that memory access can be issued. The ability of the OOO processor to tolerate memory latency and execute independent instruction is

largely dependent on the amount of instructions that can be stored in the issue queue.

These associative structures do not scale well, as increasing the memory latency tolerance requires increasing the amount of instructions that must be buffered. Increasing the number of entries in these associative structures to buffer the instructions not only increases the area, but also significantly increases the energy consumed per access. As a result, a significant tradeoff of energy efficiency for performance must be made. Other structures such as register renaming logic, large physical register files, and load-store queues (LSQ) introduce additional complexity in the form of additional pipeline stages, and circuitry.

2.3 TLP Processors

Processors that exploit thread-level parallelism maintain multiple contexts per core. GPUs such as NVIDIA's Tesla [3] and CPUs such as Sun's Rainbow Falls [4] utilize high degrees of hardware multi-threading to increase pipeline utilization, especially during long-latency memory accesses. These types of TLP designs require explicit parallelism to be expressed by the software developer. Helper threads [5] and slice processors [6] implement a slightly different approach to TLP by instantiating a partial thread of execution to improve the performance of the main thread. These schemes aim to execute memory access instruction streams to generate prefetches, but require significant duplicate execution of the memory accessing instruction stream, are sensitive to timeliness and can increase cache contention. Additionally, slice processors also require significant hardware buffers and predictors, while helper threads require programmer or compiler generation.

Figure 2.2 shows how increasing the number of threads in TLP processors enables memory latency tolerance. TLP processors can tolerate latency by executing instructions from non-stalled threads while other threads are stalled. In a typical mutlithreading scheme, when a cache miss occurs the current thread is deactivated for scheduling and an active thread replaces it in the scheduler. The number of threads supported on the processor in addition to the application memory access characteristics determine the amount of memory latency that can be tolerated.

In order to guarantee that the processor will not stall, a large number of threads may be required with each requiring significant resources to store its state. The state required for multithreading includes the hardware scratch space such as the register file, and cache and memory space which holds the working dataset of the thread. As such, the scalability of multithreading is limited when area is a concern. An additional register file is required for each thread, which takes up a significant part of the processor's area. Even if the area for the scratch space per thread is justified, the overhead of increasing cache resources may be necessary in order to accomplish performance gains. If cache resources are not provisioned correctly, contention for cache resources between the thread can significantly degrade performance for many highly parallel applications.

2.4 Direct MLP with Outrider

Neither building ILP processors with large associative structures and aggressive speculation nor building TLP processors requiring a larger number of threads is an attractive solution for improving memory latency tolerance in future processors. Both ILP and TLP techniques are useful, and the latter is

necessary for utilizing on-die resources in future chips, but neither addresses memory latency tolerance extraction directly and efficiently.

OUTRIDER adopts the strategy of directly tolerating memory latency by decoupling the instruction stream. OUTRIDER enables 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, OUTRIDER avoids issuing memory operations speculatively and is not sensitive to timeliness like prefetching techniques. OUTRIDER enables a limited form of dynamic issue similar to out-of-order processors, but without area-inefficient structures such as issue queues. Finally, OUTRIDER has the ability of multi-threading through multiple instruction streams, but without increasing the aggregate working set required on chip and thus cache contention.

CHAPTER 3

DECOUPLED ARCHITECTURES

In this chapter we present the necessary elements for a software decoupled design and introduce a traditional implementation. Figure 3.1 presents the high-level design of a decoupled system. The serial instruction stream is partitioned at compile time into separate software entities which we call *strands*. Communication occurs between strands and facilities for control flow and data communication must be provided. Decoupled architectures trade off more complex software for less complex hardware implementation. 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 Strands

Decoupled architectures separate the *memory-access* and *memory-consuming* instructions into separate instruction streams, called *strands*, that are executed on logically separate processors. These strands execute parts of the original and follow the same control flow path through the 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 de-

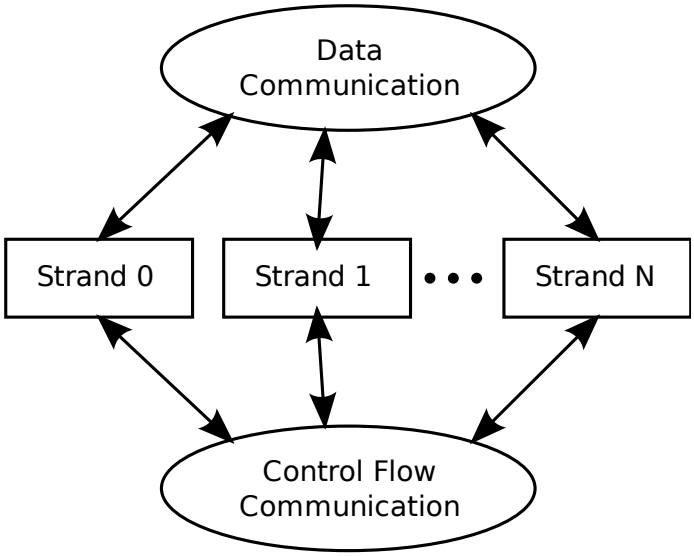

Figure 3.1: High level architecture of decoupled designs. The serial instruction stream is partitioned into strands, which communicate control flow decisions and data with one another.

pendence 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.

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. Additionally, the out-of-order execution is non-data speculative.

Strands execute in parallel with one another and persist throughout the execution of the thread. Strands have their own context of program counter,

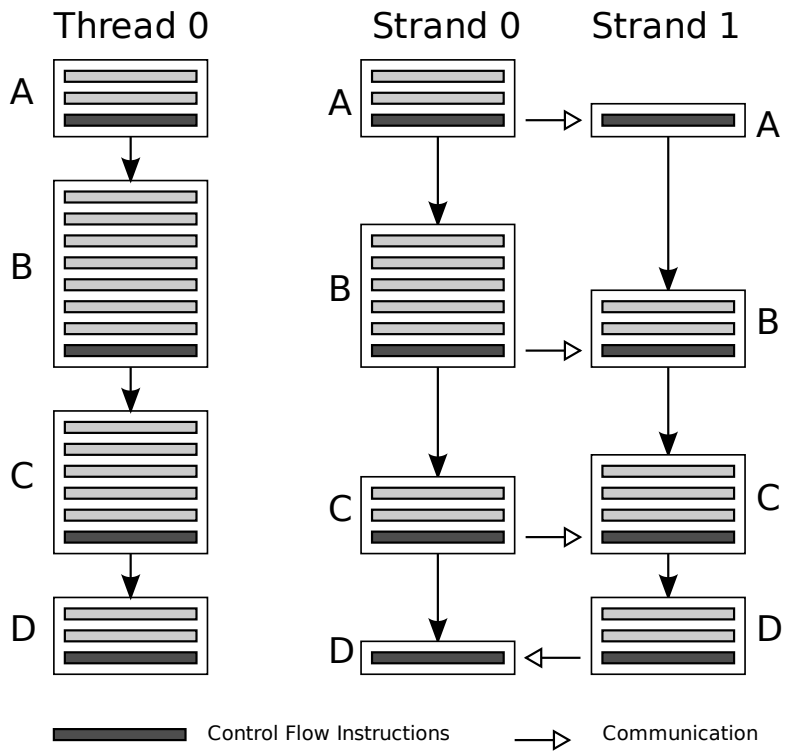

Figure 3.2: High level architecture of decoupled designs. The serial instruction stream is partitioned into strands, and communicate control flow decisions and data with one another.

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 requirements is on the order of the original sequential thread.

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 de-

sign. Though the strands must execute down the same control path in order to ensure correct execution, the execution does not need to be synchronized. That is, as long as each strand directs its execution in the correct direction, there is no need to wait on another strand. When a strand reaches a control flow instruction there is no barrier or other synchronization required before a strand can continue.

While there is no formal synchronization required by a strand when the control flow path is decided, in practice additional information may be required from another strand. In the example of compute-generated control flow, a strand must perform some number of operations before a control flow decision is made. This control flow decision cannot be generated by any other strand, and as a result, the strand generating the decision must communicate the result with all other strands.

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. We recall that in order to promote decoupling, the ideal case is to have the thread executing furthest ahead generate control flow decisions.

Outrider 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. 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 DAE, this is represented by 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 Outrider, as it eases restrictions during the partitioning and code generation process. Promoting reuse of address generation instructions is an example of when general data communication is desired. Without general data communication, the program must execute part of the address generation stream twice, as under DAE restriction. As a result, the number of states and instructions that are required can be substantially reduced.

Outrider and other decoupled architectures require the ability to communicate data between strands. As a result, 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 con-

sumed 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 that out-of-order processors have. 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 important. Potential options for facilitating data communication include FIFO data queues, rotating 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.

3.4 Decoupled Access/Execute Implementation

Figure 3.3 depicts a classic implementation of the decoupled architecture, decoupled access/execute (DAE) [7]. The access processor (AP) and execute processor (EP) are physically separate entities that are only connected

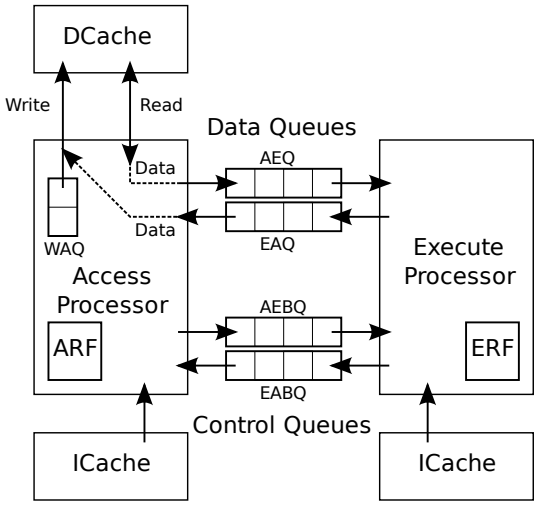

Figure 3.3: Example of a traditional decoupled-style architecture, decoupled access/execute (DAE). The access processor runs the memory program while the execute processor runs the compute program. Control flow decisions and data for computation and storage to memory are exchanged using FIFO queues. For the specific example of DAE, if the access processor does not depend on the execute processor, substantial performance improvement can be achieved.

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.

The structural requirements for decoupled designs include hardware to communicate between processors and additional hardware resources to support the additional processors. This includes register files and fetch hardware. This can result in significantly less complexity than ILP and TLP design requirements for enabling both out-of-order instruction issue and memory latency tolerance. On the other hand, the compiler must be designed to ex-

tract the separate instruction strands. This limits backwards-compatibility of code, and requires that code be re-compiled.

CHAPTER 4

TRADITIONAL LIMITATIONS

Although decoupled architectures enable memory latency tolerance, potential performance improvement is limited when the memory-accessing instruction stream cannot achieve nonblocking property with respect to the rest of the program. These situations are known as as *loss-of-decoupling* (LOD) events. Figure 4.1 presents the loss of decoupling events on traditional DAE processors, which represent a dependence between the processors that must be resolved before the AP is allowed to continue execution. Event **A** depicts the optimal case where there is no LOD event and the memory-accessing stream is not blocked. In event **B**, AP to AP 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 AP must stall. When the AP depends on data provided by the EP, LOD can also occur. This can be due to the AP needing an address generated by the EP (event **C**) and or the AP waiting on a control flow decision to be determined by the EP (event **D**). The AP must wait on the EP to proceed, which removes the ability of the AP to move ahead and uncover MLP. The LOD events significantly reduce the usefulness of DAE on common programs that exhibit memory indirection and compute-dependent behavior.

Additionally, the under-utilization of resources in traditional decoupled designs that have separate processors is also an issue. Traditional designs require separate fetch, decode, and execute resources, but these resources do

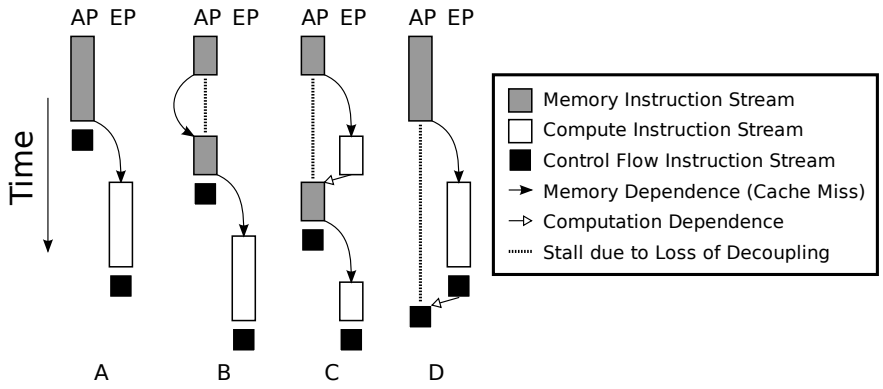

Figure 4.1: Loss of decoupling (LOD) events for access and execute Processors (AP, EP) in DAE architectures. Perfect decoupling (A), memory indirection LOD (B), compute-generated addresses (C), and compute-generated control flow (D).

not experience continuous utilization. These resources could potentially be shared, reducing area overheads. 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.

4.1 Addressing Memory Indirection

Figure 4.2 shows our approach to addressing memory indirection LOD. Memory indirection can be alleviated by adding additional memory-accessing strands. The original memory accessing stream can be split into strands, with the goal of having at least one instruction stream internally non-blocking. By adding strands, the amount of decoupling is increased and more parallelism is exposed. In order to handle compute-generated instructions, we blur the line between AP and EP by enabling the same functional units in each processor. We then can employ additional memory-accessing strands by moving

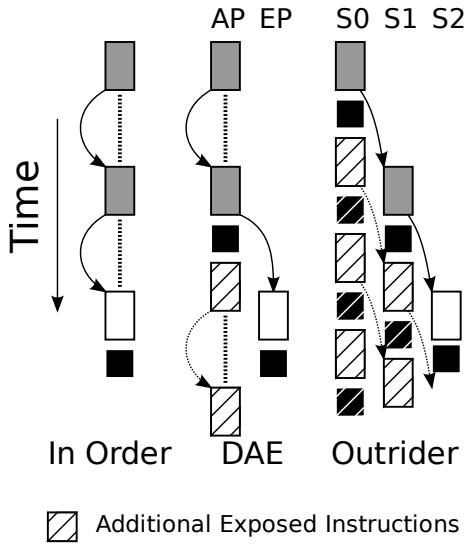

Figure 4.2: Memory indirection can cause LOD in DAE architectures. Outrider can eliminate or substantially reduce LOD in these cases by extracting more parallelism.

the critical loop from the EP and AP into a separate instruction stream that accesses memory.

Increasing the number of strands increases the amount of hardware resources required for the decoupled processor. We find that many programs we evaluate only have one or two levels of memory indirection. This leads us to choose four total strands in our design. In addition to additional register file and fetch resources, the number of data queues for communication increases with the number of strands in the system. Each instruction stream may wish to communicate with any of the other strands. The number of data queues required on such a system is an $N^2 - N$ relationship, where N is the number of strands. While scalability of increasing the number of strands is weak, we find in practice that many programs do not communicate with one or more of the other strands. Communication information is available when strands are extracted, so we can utilize a dynamically partitioned buffer and allocate a portion of the space for individual strands' data queues. The

strand's FIFO queue is in essence virtualized onto a part of the larger space. This enables area efficiency in the case that space will not be allocated to facilitate communication between two strands when such communication does not exist.

4.2 Addressing Resource Utilization

Scaling DAE to more than two strands results in many physical processor entities that have hardware resources such as instruction fetch that could be potentially shared. Resource utilization can be improved by the use of multithreading on a single processor. We propose having a general processor with all the functional units and functionality to execute memory accessing and memory consuming code, but with four contexts which each execute a single strand. Because some software may exhibit memory indirection and enable multiple strands and some may not, we also propose dynamically partitioned data queues. When strands are extracted, a portion of the data queues is allocated to each strand. These techniques are critical for enabling area efficiency in Outrider and ensure that hardware will not be left unutilized.

CHAPTER 5

OUTRIDER ARCHITECTURE

Figure 5.1 shows a block diagram of the OUTRIDER architecture which supports the decomposition of a thread into a maximum of four strands working concurrently. The dynamically partitioned shared communication queue, common and partitioned thread register files, and the out-of-order memory access unit are the main additions to the baseline in-order core. All supported structures necessary for achieving high performance with OUTRIDER consist of a low number of entries. In this section we also discuss memory consistency and binding. We propose a simple technique for detecting deadlock induced by software faults and a way to define precision of exceptions on a strand-based architecture.

5.1 Communication Queues

Figure 5.2 presents our implementation for the data queues. In OUTRIDER, strands use hardware data queues for general communication or broadcast of any value, unlike the special purpose queues found in DAE. The queues provide in-order communication that can extend across multiple iterations of a loop. 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

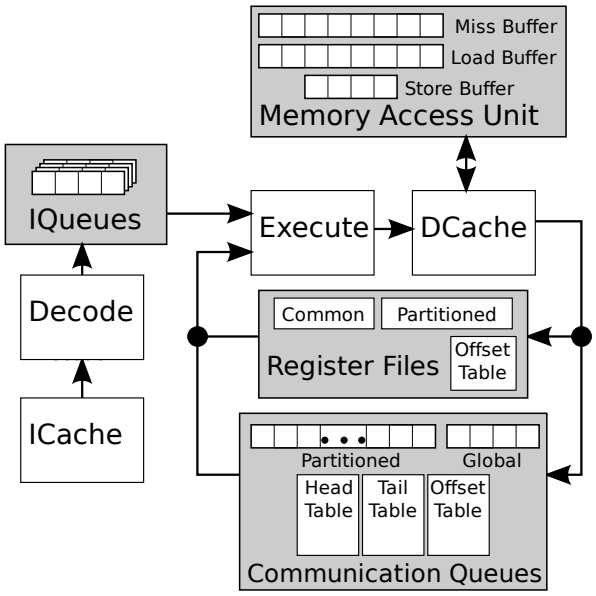

Figure 5.1: OUTRIDER expands a traditional in-order pipeline with instruction queues, communication queues, a memory access unit, and dynamically partitioned register files.

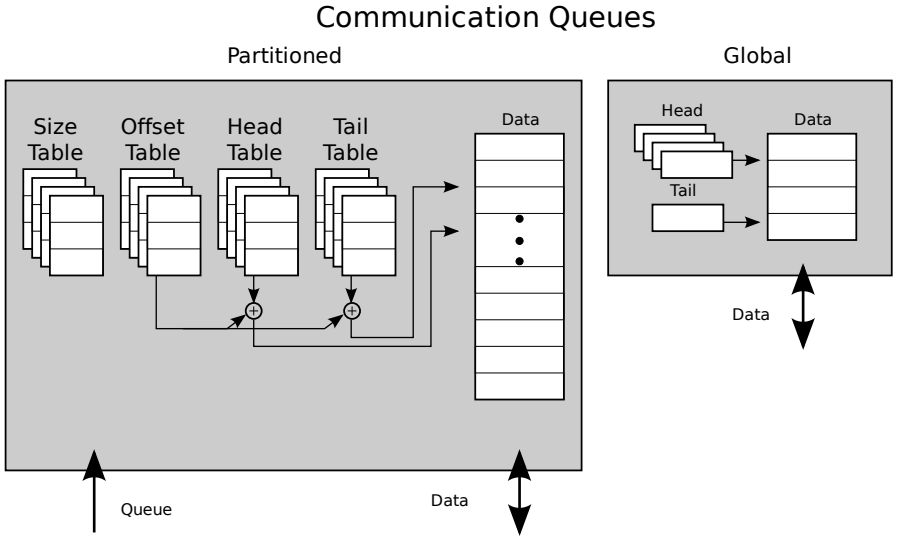

Figure 5.2: OUTRIDER uses a partitioned buffer and per-strand pointer tables to facilitate communication between each strand.

the strands are mostly independent, the frequency with which communication occurs is relatively low. The communication queue is mapped to architectural registers 1, 2, and 3 in the register file enabling each of the strands to communicate with one another. When reads and writes are performed using these registers, the hardware tables are indexed and physical locations found in the queue.

Figure 5.2 shows the communication queues and the hardware tables each strand uses to access their partition, which is configured by the compiler. For the initial study of Outrider we count the number of communication occurrences between each strand and size proportionally to the total number of communication occurrences. For strands that consume but rarely produce data, larger receive queues and smaller send queues are allocated. When the strand is initiated, the hardware tables are written using special purpose registers.

One powerful aspect of Outrider that is different from DAE architectures is the ability to complete instructions out-of-order into the communication queues. This is particularly useful with sharing general data between strands, or allowing loads to complete out of order when otherwise not blocked by a outstanding store to the same address. When a strand issues an instruction with another strand's queue as the destination, it is given a queue offset, corresponding to the tail of the queue where it will store. Using the queue offset, the instruction can execute in any order and be written correctly into the queue with in-order retirement enforced by the semantics of the queue.

A fixed size global data queue used for broadcasting common values among strands is found in Figure 5.2. The global data queue is primarily used for communicating control flow decisions between the strands. Each strand keeps a head pointer to the global data queue that advances upon a reference to

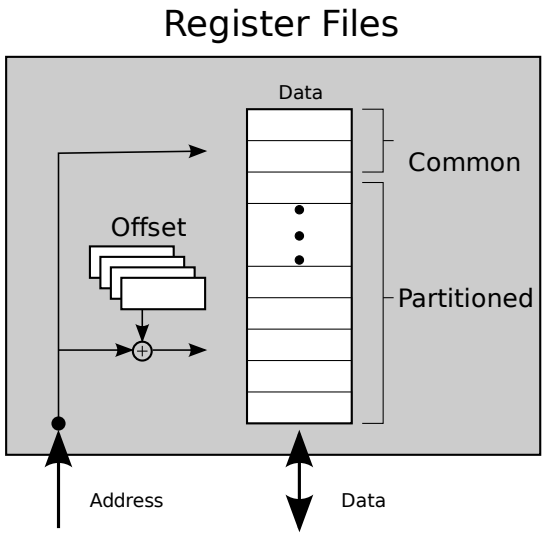

Figure 5.3: OUTRIDER partitions a register file between strands using a table of offset pointers.

the queue.

5.2 Register Files

Figure 5.3 shows the register file system used in OUTRIDER. Each strand is allocated a portion of the register file by the compiler sized relative to the working register set size of the strand. When the strand spawns, the starting offset is set by writing a special purpose register. The strand then uses architected register names 8-31 to access its portion of the register file. Dynamic allocation enables high utilization of the register file, while allowing flexibility for varying the number of registers between the strands. This provides a benefit compared with separate and statically sized register files for each strand.

Additionally there is a small portion of the register file not privately owned by a single strand which allows constants to be shared among the strands,

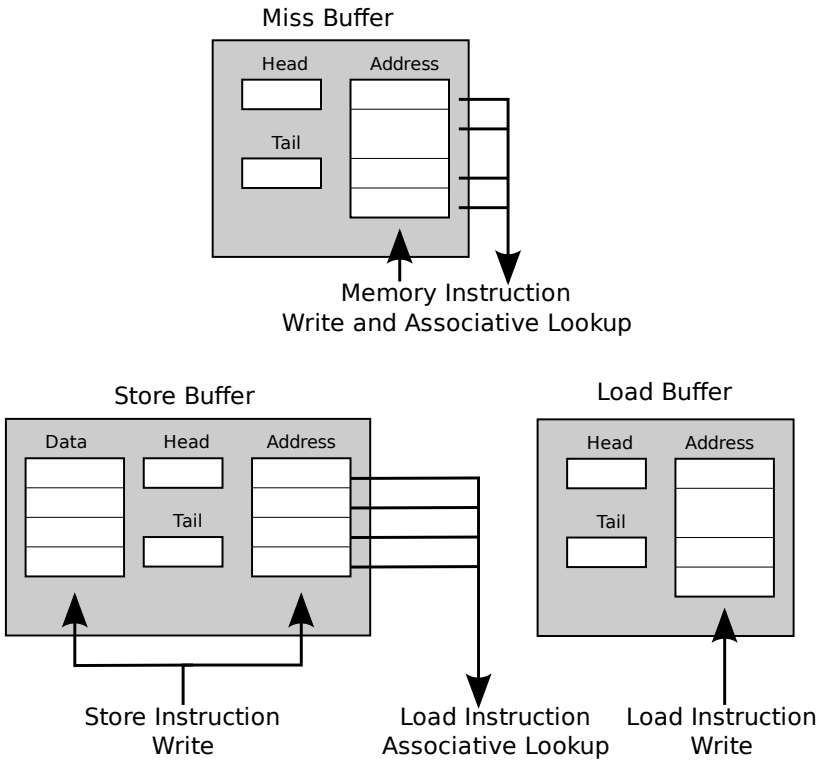

Figure 5.4: OUTRIDER utilizes a memory access unit which handles stores and misses using an associative lookup structure, and loads using an indexed structure. These buffers are utilized by the strands using hardware pointer tables.

such as the stack pointer. These shared registers are only safe to be set at the start of the function call or during a barrier synchronization between the strands and remain unchanged throughout OUTRIDER execution phases. Strands use architected registers 4-7 to directly access the shared portion of the register file.

5.3 Memory Access Unit

Figure 5.4 shows the memory access unit (MAU) which enables multiple memory operations to be in flight simultaneously. The MAU supports eight outstanding misses, eight load requests, and four store requests. The store

buffer is an 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.

Stores are handled specially in OUTRIDER as compared with DAE. Instead of function-specific queues found in DAE, OUTRIDER uses the associative store buffer. `st_addr` instructions which provide addresses are issued by the memory accessing strands and combine with `st_data` instructions which provide the data to be stored. To enforce store orderings, we require that each strand have either a `st_data` or `st_data` instruction for each store in the original program.

To take advantage of the out-of-order completion abilities enabled by OUTRIDER's communication queues, the MAU allows independent loads to complete out-of-order by checking the four-entry store buffer before servicing the request from the cache. Only storing the miss-stream instructions reduces the complexity of the MAU. Compared to previous work on DAE, the out-of-order MAU mechanism in OUTRIDER is an improvement since it does not enforce in-order completion into FIFO queues, thus increasing the concurrency in the memory system.

5.4 Memory Consistency and Binding

Memory operations from strands are issued non-speculatively and are executed in-order, thus respecting original program order. Memory dependences are enforced inter-strand and all stores are issued to the memory system in original program order, which maintains the perception of program order issue with respect to other threads. For applications that need to enforce strict

ordering between strands, OUTRIDER utilizes a pair of synchronizing instructions with full memory fence semantics. The two instructions `mem_proceed` and `mem_wait` are used to signal a particular strand in a single direction through the communication queues. A strand uses `mem_wait` before a memory access to wait for the memory fence, while another strand executes a `mem_proceed` instruction to signal that the fence has been reached and that it is safe to execute. Utilizing the memory fence can build a stronger consistency model, which is necessary to retain the memory semantics of the original thread, on top of the relaxed consistency model that OUTRIDER naturally supports.

Memory load values are bound when the operation can be completed at the L1 cache, at which point it leaves the MAU. OUTRIDER is a non-speculative architecture enforcing that every memory operation issued will be committed. Traditional load store queues found in out-of-order processors hold speculative memory operations, which means that memory values do not bind until retirement.

5.5 Deadlock and Exceptions

OUTRIDER is a software threading technique, and deadlock in OUTRIDER is similar to a software deadlock. Given correct program semantics and communication between strands, deadlock will not occur in OUTRIDER. However, it can occur in improper code if all strands are waiting on the queue for data while the queues are empty, or if all strands are waiting to insert data into the communication queues, but all the queues are full. Detection of deadlock for software debugging purposes is straightforward, and requires checking to see if all strands are blocked in the aforementioned case. When deadlock

is detected, the pipeline, instruction, and data queues are flushed and an exception is raised to allow the runtime system to recover.

Software that targets OUTRIDER is composed of multiple strands extracted from a single thread of execution. The concurrent execution of memory-accessing strands and the memory-consuming strand requires that one program counter (PC) be kept for each strand. To enable precise semantics for faulting memory instructions, we define the point of the exception in the memory-accessing strand to occur immediately before the memory access triggering the fault. The fault is initially stalled and the strand issuing the faulting instruction is blocked. The memory-consuming strands are allowed to continue executing until they reach the instruction dependent on the faulting instruction. At this point, the fault is delivered precisely across strands comprising the original thread: at the PC of the faulting instruction in the memory-accessing strand and at the PC of the first dependent instruction in the memory-consuming strand. Recovery from an exception would require addressing the fault in the memory-accessing strand and restarting it. Doing so will cause the memory-consuming strand to unblock and execution to proceed as normal.

CHAPTER 6

STRAND EXTRACTION

OUTRIDER depends on compiler extraction of strands from the original thread by examining the control and data flow graphs (CDFG) and partitioning the program into memory accessing and memory consuming instruction streams. We assume that leaf node functions make up the majority of execution time of data parallel applications for the purposes of this paper. This simplifies handling of parameter and return values, and enables different transformations on a function-by-function basis which allows additional flexibility. We also assume that the function we transform will output data directly to the memory system rather than return a value for simplicity.

Past research has demonstrated code partitioning and optimization such as [8] and [9], and the approach OUTRIDER adopts is similar. In short, memory dependence chains are identified and strands are created along memory access - memory consumption lines. The process to extract strands consists of five phases of strand assignment: load and stores, address generation, control flow, unassigned instructions, and final partitioning.

For the purposes of this thesis, manual construction of strands is performed using the partitioning process presented. The partitioning process has successfully generated the benchmarks used in the evaluation section, which includes programs with memory-indirection and compute-generated addresses. Using prior research, we have made substantial progress on automated code generation and find that performance is within 6% of the `sobel` benchmark.

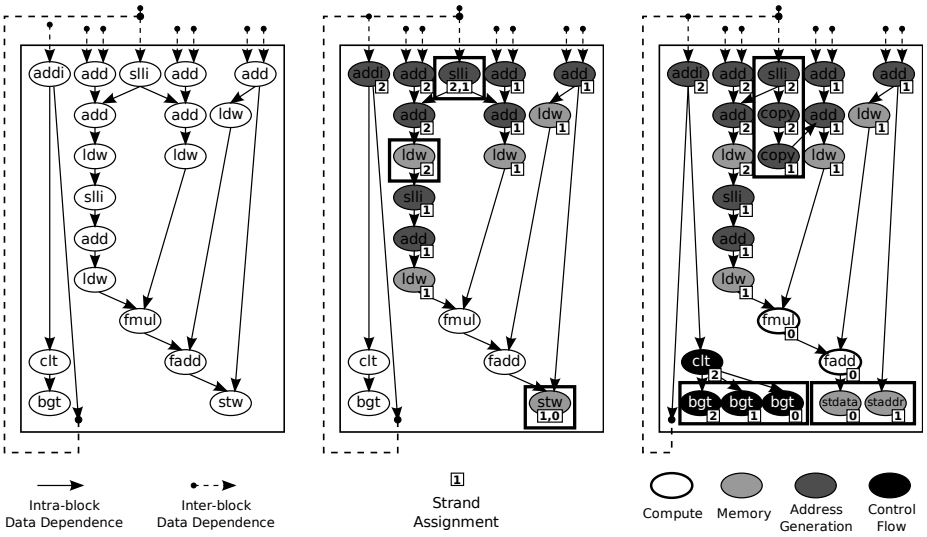

Figure 6.1: Strand extraction process. A program's control data flow graph (CFDG) (a) is used during the partitioning. The CDFG is traversed to assign memory operations and address generation instructions to strands (b). Finally, control flow and compute instructions are partitioned (c). Shared instructions must be split and communicated using copy instructions, and control flow decisions are broadcast to all strands

Our automated code generation results are preliminary, but serve to provide evidence that automatic generation is close to manual construction, and within reach. Complete details about automated code generation for OUTRIDER are outside the scope of this thesis.

We provide an example of the strand extraction process using the inner loop of the sparse-matrix vector multiply of the `cg` benchmark. Figure 6.1(a) presents the CDFG 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. Figure 6.1(b) depicts the strand assignment of the inner loop after Phases 1 and 2 are completed and the loads, stores, and address generation have been processed. Figure 6.1(c) shows the final instructions and their association after Phases 3, 4, and 5 are completed and the control flow, compute, and shared instructions have been

partitioned.

6.1 Phase 1: Partition Loads and Stores

Phase 1 identifies load operations and uses them as a partitioning point for a strand. Instructions backwards in the CDFG from a load operation represent address generation, while instructions found forward in the CDFG represent memory-dependent instructions. Stores represent an endpoint of the CDFG and are placed in the same strand to ensure proper store ordering. Stores have both address generation and data value generating instructions found backward in the CDFG.

We define a *hop* as a load in the dependency chain that must be traversed in the CDFG in order for values from a given load to reach a store operation. We are interested in the maximum number of hops and hence the strand that the load is assigned to. Using the CDFG, we identify each load and traverse forward in the CDFG, recording the maximum number of hops and assigning the load to the corresponding strand. Stores are split into multiple instructions: the data will be supplied by the lowest strand, while the address part of the instruction will be executed by all other strands. This is required to prevent ordering issues caused by aliasing of loads in high level strands with stores. Replication can add extra instruction overhead and introduce an LOD event, but can be avoided if the compiler can perform source level analysis or the programmer can provide information about variables and parameter pointers. In our manual processing, we have access to the source code and guarantee that aliasing will not occur. Figure 6.1(b) presents the CDFG with loads and stores assigned to strands. The boxed load assigned to strand 2 is a distance of two hops from the store, as its data is consumed

by another load.

6.2 Phase 2: Partition Address Generation Instructions

Phase 2 uses the identified loads and stores and their assigned strands to identify and partition the address generation instructions. Address generation instructions are defined as the back slice of instructions from a particular load or store in the CDFG that contribute to the address. Only the instructions in the back slice found before reaching a load operation are considered for inclusion in the same strand as the initial load.

To determine the address generation instructions and partition them to a strand, we first consider memory operations assigned the highest level strand. In the case that there are address generation instructions that are shared between strands, the highest level strand will be the owner of the instruction. The value will be communicated to the other strands in order to reduce circular dependences and promote decoupling.

Starting at the highest level, for each memory at the current strand level:

1. *Look backwards in the CDFG, 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*

When address generation partitioning has been performed for all memory operations at current strand level, the strand level to process is decreased, and the process continues. When all strands have been assigned their address generation instructions, the process is completed.

Figure 6.1(b) presents the address generation instructions assigned to strands. The highlighted `slli` shift instruction is used by both a load in strand 2 and strand 1. Figure 6.1(c) presents the same shift instruction accompanied by copy instructions to communicate the value from strand 2 to strand 1.

6.3 Phase 3: Partitioning Control Flow

All stands progress through the program together using the same control flow decision. We enable one strand to communicate decisions to the rest of the strands. As a result, we partition the control flow decision to be handled by the highest level strand that can produce the result to enable decoupling.

We identify branch instructions and look backward through the CDFG marking unassigned instructions as control flow instructions. When an assigned memory or address generation instruction is reached, the terminating instruction is marked to indicate it will forward data to the control flow instruction stream. We record each terminating instruction's allocated strand, until the backward search has completed for a given branch instruction. At that time, the control flow instructions discovered are allocated to the highest level strand providing data to the control flow.

Figure 6.1(c) depicts the instructions marked as control flow. Though the `addi` instruction is used for control flow, it was detected as address generation belonging to strand 2. Since the `clt` instruction which generates the control flow decision has no other operands, it is also assigned to strand 2, and will broadcast to all instructions. Figure 6.1(b) and Figure 6.1(c) highlight the branch instruction which is replicated across all strands.

6.4 Phase 4: Partition Unmarked Instructions

The last stage is to mark the unmarked instructions as compute, which provide data to store operations and are placed in the lowest strand. We traverse backwards through the CDFG starting at the store instruction, and mark the instructions as compute. We terminate our search when an instruction has already been marked, at which point the terminating instruction is marked to forward data to the lowest level strand. At the end of the compute phase, all instructions will have been marked in the function. Figure 6.1(c) depicts the `fmul` and `fadd` floating point instructions being assigned to the lowest level strand, strand 0.

6.5 Phase 5: Final Partitioning

With all instructions marked, strands are created. Figure 6.1(c) 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 `staddr` assigned to strand 1 and the data providing instruction `stdata` assigned to strand 0, enabling the compute to pass the data directly to the store unit. Control flow instructions are assigned to their respective strand, with the result being broadcast to all the other strands. Branch instructions which source the globally-communicated decision are copied to all strands. Instructions that are shared among strands are placed in the highest level strand, and communicated to other strands using copy instructions.

6.6 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 OUTRIDER supports. In the case that too many strands are generated for the hardware to handle, we reduce the number of strands to the maximum size permitted by hardware by combining some of the strands.

In general, strands adjacent to one another in terms of the number of hops are considered for merging together. Specifically, we perform several passes using different priorities. During each pass, we reduce the number of strands by one. First we identify loss of decoupling events and merge those strands, as those strands will have the least amount of performance improvement. Next, strands that are allocated very few instructions incur extra overhead for communication and control flow, which can hurt performance efficiency. Finally, we choose the high level strands as a last resort to reducing 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 setup code. The number of strands extracted and resource allocation information is written to a special purpose register and a command is issued which spawns the threads, at which point they begin fetching instructions from the initiating thread's PC. A jump table is used to direct the strands to the relevant code section they are to execute. Strands execute until the function is finished, at which point decoupling is turned off.

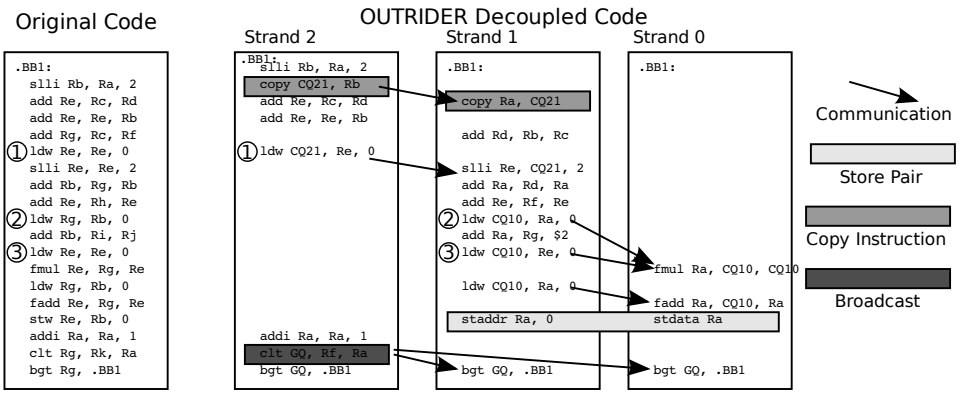

Figure 6.2: Sparse matrix vector multiply code example from the `cg` benchmark illustrating strand extraction. The basic block structure is maintained across strands, with memory-dependent instructions being placed into different strands. Control flow decisions can be broadcast from one strand to all others using the global queue (GQ). Memory dependences are communicated through the data queues (CQ). Shared data between strands is accomplished through copy instructions which enable general communication to occur across the data queues.

6.7 Code Example

Figure 6.2 presents the sparse-matrix vector multiply inner loop of the `cg` benchmark and its partitioning into strands. 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. Instruction (1) is determined to have a maximum height of 2, and thus will be allocated to Strand 2. Phase 2 detects the address generation, and Instruction (3) is shared between Instructions (1) and (2). As a result, Strand 2 is responsible for communicating the queue to Strand 0 using the data queues. Copy instructions are used to communicate the value between the two strands. Phase 3 identifies the control flow instruction, of which no instructions rely on a memory instruction. As such, they are assigned to the highest level strand, Strand 2, which broadcasts the control flow decision to the other strands. The memory store straddles Strands 1 and 0 by using a

pair of instructions that provide the address and data to the store unit.

CHAPTER 7

EVALUATION METHODOLOGY

We evaluate Outrider by comparing against traditional fine-grained simultaneous multi-threading [10] (SMT). We use the 1024-core throughput architecture shown in Figure 7.1. The cache hierarchy comprises three levels. Each core is a two-wide issue in-order with private L1 instruction and data caches and a RISC-like instruction set. Eight cores, an interconnect, and network interface form a *cluster* and share a unified L2 cache. The clusters connect to a multi-banked shared last-level L3 cache through a two-level interconnect network. Every four banks of L3 have an independent GDDR memory channel. Table 7.1 lists the chip and core design parameters.

The simulator infrastructure is execution-driven and models cores, caches, interconnects, memory controllers, and DRAM. Each benchmark is executed for at least one billion instructions. For evaluation, we use a set of six optimized parallel kernels from scientific and visual computing applications. The benchmarks exhibit a high degree of parallelism and are written using a task-based, barrier-synchronized work queue model similar to Carbon [11], but implemented fully in software. The benchmarks include conjugate gradient linear solver (`cg`), 2D fast Fourier transform (`fft`), 2D stencil (`heat`), k-means clustering (`kmeans`), medical image reconstruction (`mri`), and edge detection (`sobel`). Benchmarks are decomposed into strands manually as shown in Chapter 6.

Table 7.1: Simulation parameters for our 1024-core architecture.

Core Base	8 stage, 2-wide in-order, 32 entry RF BTFN branch prediction, 8 entry BTB
OUTRIDER	max 4 strands per thread, 32 entry shared RF 32 entry shared communication queues
Multi threading	8 entry instr. queue, 32 entry RF per thread
L1 ICache	2kB 2-way, 1 cycle, 2 Misses, Next-line Pref.
L1 DCache	1kB 4-way, 1 cycle, 8 Misses, 8 Loads, 4 Stores
L2 Cache	64kB Shared. 4 cycle, 8-way
Interconnect	Two-level tree and crossbar, 16+ cycle latency
L3 Cache	4MB Shared, 32-Bank, 4 cycle, 8-way, Next-line Pref.
DRAM	8 Channels & GDDR5

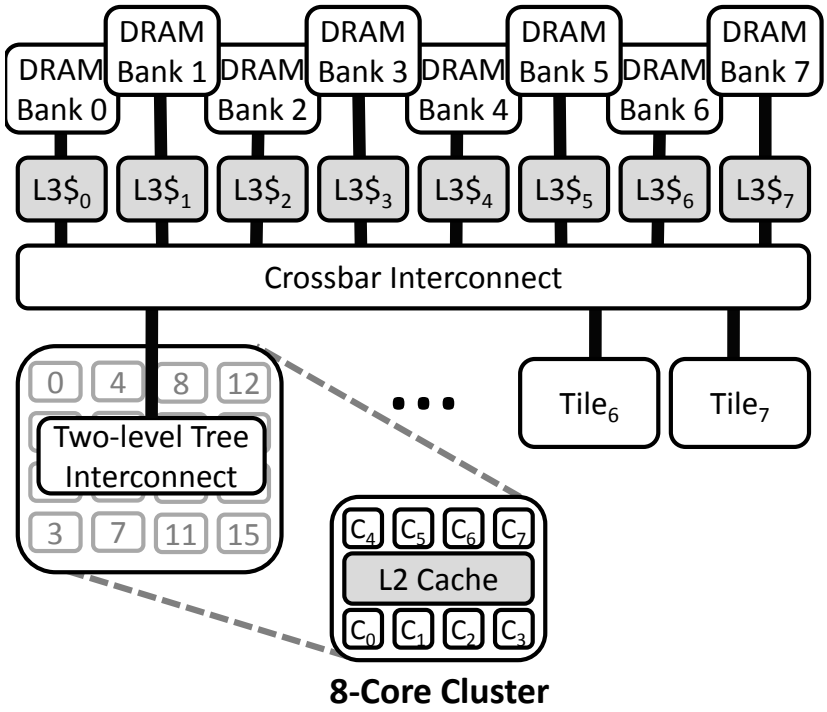

Figure 7.1: Evaluation architecture

CHAPTER 8

PERFORMANCE EVALUATION

8.1 Overall Performance

Figure 8.1 compares the performance of Outrider to the baseline single-threaded core architecture, two-way and four-way SMT. Figure 8.2 shows the harmonic mean of the increase in misses observed at the L1, L2, and L3 caches. Two-way SMT improves performance by 25% over the baseline, while four-way SMT has mixed results due to a significant amount of contention at the L3 cache which counteracts potential performance gains. Four-way SMT performs best when contention is kept to a minimum, as in `mri`. The direct extraction of MLP enables Outrider to outperform two-way and four-way SMT significantly despite being a single thread. SMT can expose a small fixed amount of MLP, while increasing contention for shared resources. Outrider does not significantly increase contention for shared resources over the baseline and SMT cores despite executing the same memory stream on the same amount of cache resources.

The performance gains in Outrider come from both the SMT interleaving of strands and the MLP generated by strands. The SMT interleaving effect is especially clear in `kmeans`, where performance is substantially improved over the perfect cache case. Memory-intensive benchmarks such as `cg` and `fft` do not see as much benefit from Outrider. This is due to extreme and irregular memory accesses that result in reduced utilization of cache resources, a

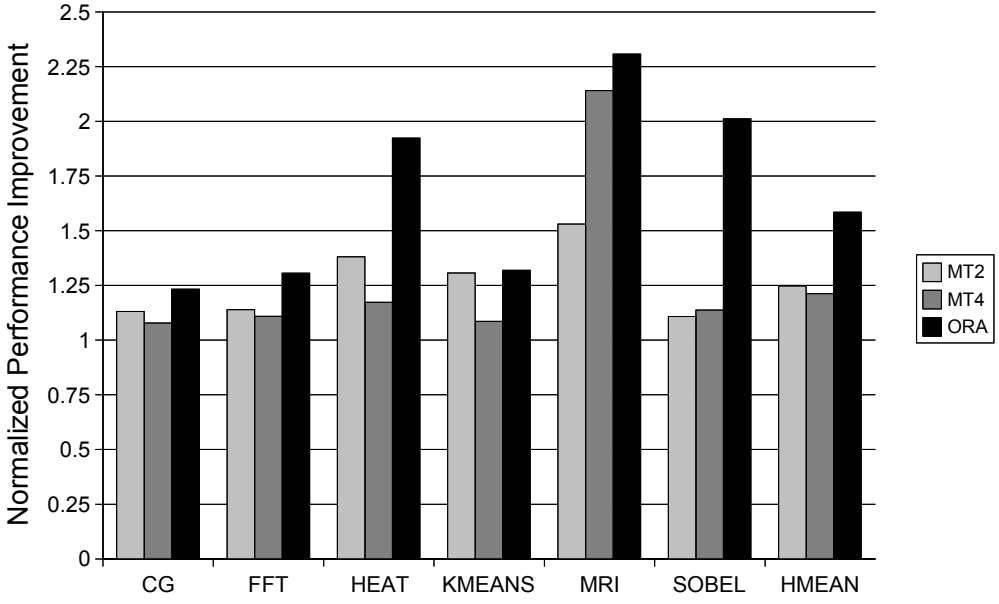

Figure 8.1: Overall performance of two-wide in-order baseline, two-way and four-way SMT, and OUTRIDER architecture relative to baseline.

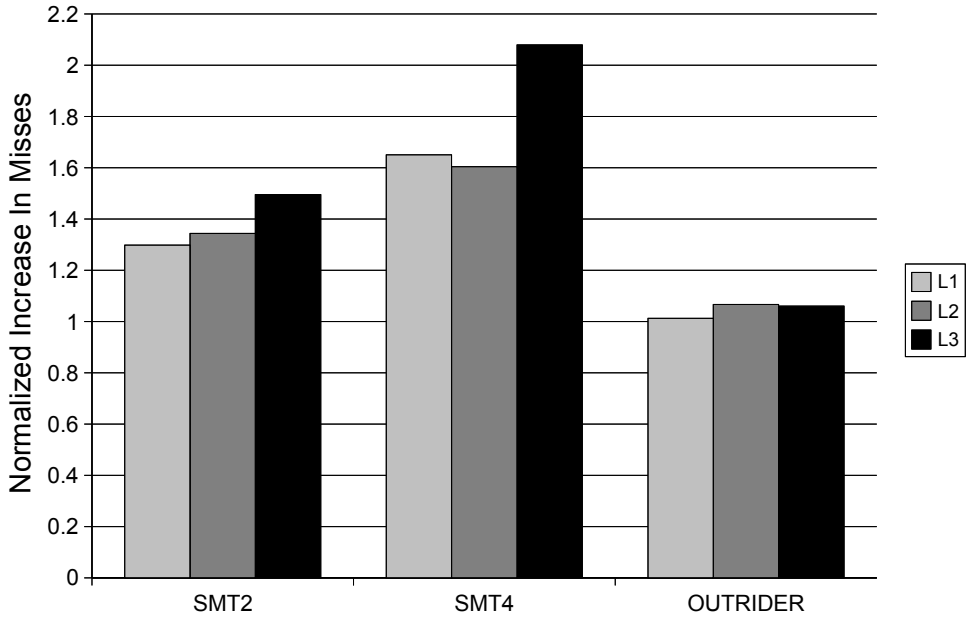

Figure 8.2: Increase in cache misses in two-way and four-way SMT, and OUTRIDER architecture relative to two-wide in-order. Harmonic mean across all benchmarks is presented.

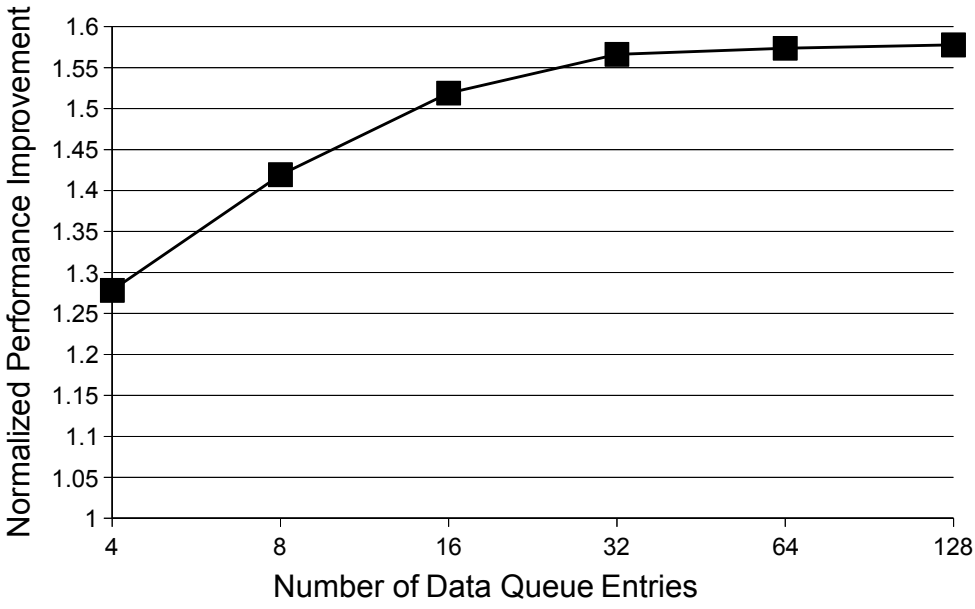

Figure 8.3: OUTRIDER sensitivity study for data queue sizing. Performance is harmonic mean of all benchmarks.

performance limiter also found in the baseline. On memory-intensive benchmarks such as `heat` and `sobel`, which exhibit locality favorable to our cache hierarchy, OUTRIDER outperforms SMT by up to 87%.

8.2 Communication Queue Sizing

Figure 8.3 shows the mean sensitivity of OUTRIDER to the size of the communication queues. The harmonic mean for all benchmarks is presented. OUTRIDER requires a modest 32 entries to achieve nearly all of the performance benefit on the data parallel benchmarks. The number of entries in the communication queues represents the number of in-flight data that have yet to be consumed by the memory-consuming strands and therefore the ability to tolerate latency. We observe that when data is written into

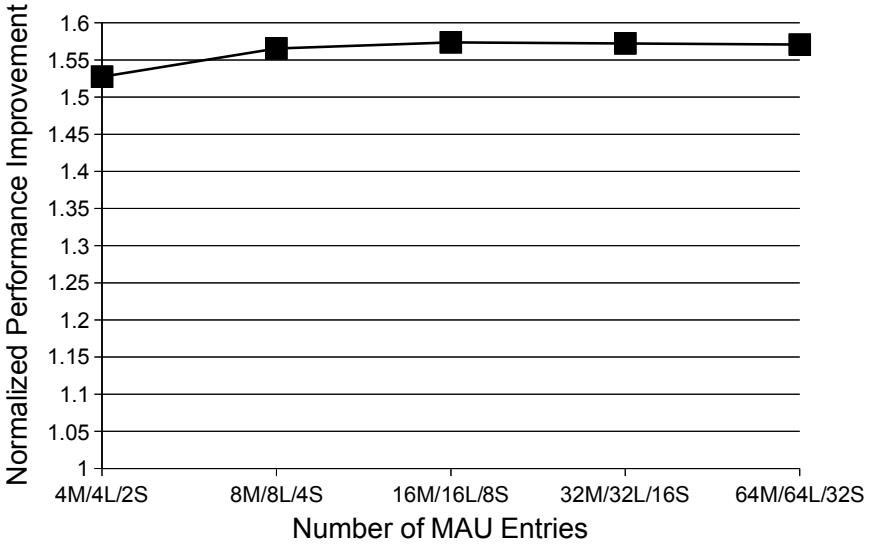

Figure 8.4: OUTRIDER sensitivity study for memory access unit sizing. Performance is harmonic mean of all benchmarks.

the data queues, it is usually quickly consumed by a waiting strand. For some applications, a communication queue of size eight entries is sufficient to reach the peak performance possible. This is not true for applications such as `mri`, where long chains of floating point operations dominate. This application shows more bursty behavior, in which it consumes more data in some parts of the computation chain than others. In this case, increasing the communication queue to 64 entries can benefit performance.

8.3 Memory Access Unit Sizing

Figure 8.4 shows the mean sensitivity of OUTRIDER to the size of the MAU. The harmonic mean for all benchmarks is presented. Our results for OUTRIDER demonstrate that our proposal can achieve good performance with 8 miss, 8 load, and 4 store entries in the MAU. OUTRIDER sees relatively low

performance gains from increasing the MAU to a larger size. This is due to high L1 and L2 data locality that is exploited in our benchmarks. As a result we see a majority of performance increases from primary data cache and L2 cache latency tolerance. A low number of entries allows for area- and power-efficient implementation. Unlike the LSQ found in out-of-order cores, the MAU must only track misses and load-store dependences. Additionally, only the store and miss buffer are associative look-up structures. OUTRIDER enables this efficiency versus OOO cores because strands are non-data-speculative and allow out-of-order load completion into the data queues through indexing.

8.4 Cache Latency Sensitivity

Figure 8.5 shows the mean sensitivity to L2 cache latency across all benchmarks. We find that OUTRIDER is not as sensitive to memory latency as the baseline and SMT cores; a two-way SMT processor with a 16-cycle L2 Cache latency performs as well as OUTRIDER with 64-cycle L2 Cache latency. SMT cores can only extract a small amount of MLP per thread, while OUTRIDER's explicit decomposition into non-blocking memory strands enables more MLP to be uncovered. In some cases four-way SMT outperforms two-way SMT as L2 cache latencies increase, as the extra MLP provides benefit greater than the degradation due to L3 cache contention. In some benchmarks, such as `mri` and `sobel`, insensitivity exists up to 64 cycles without significant performance degradation. OUTRIDER begins to exhibit sensitivity to L2 latency at 32 cycles due to reaching MAU and communication queue capacity limits. Additional MLP exists in the memory strands, and increasing the MAU and communication queue size allows additional insensitivity to latency.

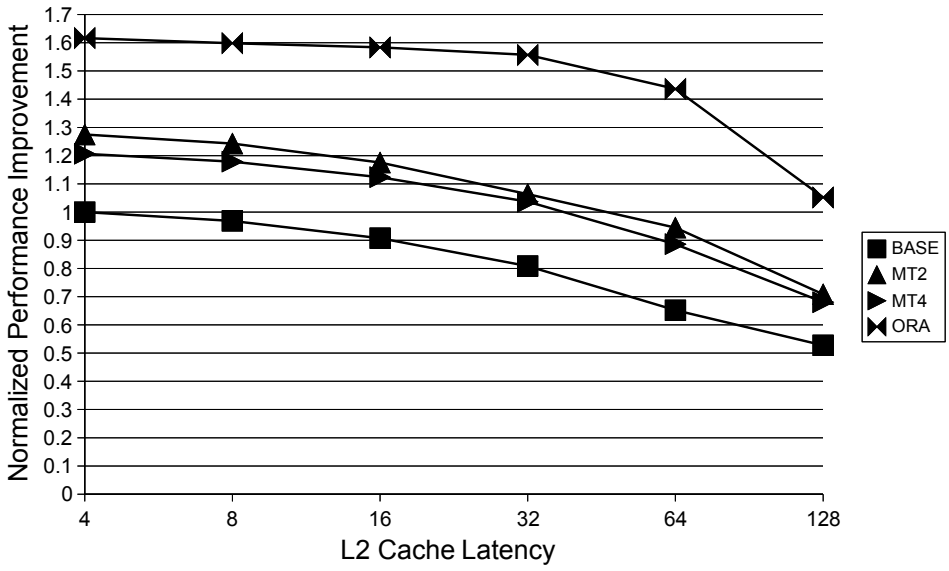

Figure 8.5: L2 cache latency sensitivity study, comparing performance of in-order baseline, two-way and four-way SMT, and OUTRIDER architecture relative to in-order baseline with L2 latency of four cycles.

8.5 Cache Size Sensitivity

Figure 8.6 shows the mean sensitivity of OUTRIDER and SMT to cache sizing for all benchmarks. As the size of the L1 data cache is increased, all processors see improvement. The increase in cache size decreases the amount of cache contention which improves SMT, but the additional storage space also benefits OUTRIDER and the baseline. OUTRIDER experiences a larger benefit due to increased cache sizing, as the MAU and L1 data cache can be utilized more efficiently.

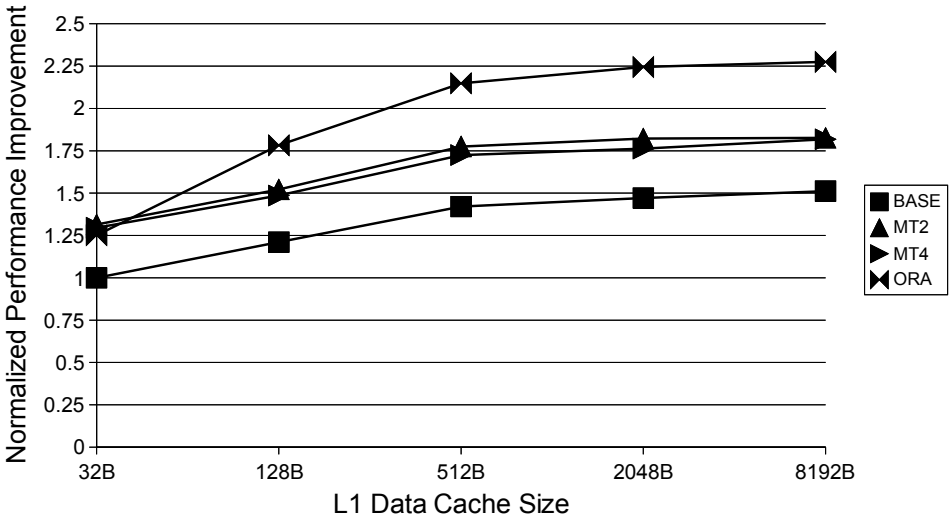

Figure 8.6: L1 data cache size sensitivity study, comparing performance of in-order baseline, two-way and four-way SMT, and OUTRIDER architecture relative to in-order baseline with L1 data cache size 32B.

CHAPTER 9

OVERHEAD EVALUATION

Table 9.1 presents the amount of added instruction overhead among the benchmarks. Instructions that copy shared data between strands, and control flow instructions needed to direct dependent strands make up the overhead; The number of memory operations and computation instructions remains the same. The harmonic mean of the instruction overhead is 14%, which compared with the 62% mean performance improvement is a performance-overhead advantage. Further reduction in the branch overhead can be achieved by traditional techniques such as loop unrolling. While the overhead can reach 38%, these instructions consume less energy as compared with memory and floating point operations, which dominate the application's instruction stream.

We use Cacti 6.5 [12] to estimate the power and area cost for implementing Outrider in a 45nm process. The added structures are the communication queues, offset tables, instruction queues, and the MAU. We also model the cost of adding threads for the SMT processor. The resulting area is found in Table 9.2. For register files and communication queues we assume SRAM, while for instruction queues, MAU, and hardware tables we assume latches with size 8 μm^2 per bit including overhead. From Table 9.2, we see that the majority of area is taken up by the communication and instruction queues. Past work such as Rigel [13] is able to fit 1024 single-threaded cores in 320 mm^2 in a 45nm technology. The total extra area for Outrider compared to

Table 9.1: Total instruction overhead for OUTRIDER compared to the baseline in-order design. OUTRIDER copy instructions and replicated branch instructions are also shown.

	%Total	%Copy	%Branch
cg	14.59%	8.39%	6.19%
fft	37.82%	14.74%	23.08%
heat	14.10%	9.87%	4.22%
kmeans	24.37%	11.30%	13.07%
mri	6.05%	4.03%	2.02%
sobel	14.29%	13.17%	1.12%
hmean	13.57%	8.56%	3.15%

Table 9.2: Area overhead of OUTRIDER, SMT2 and SMT4 in regards to additional storage required.

Item	Addition	ORA Area	SMT2 Area	SMT4 Area
MAU	(1) 8/8/4x32	5,124 μm^2	5,124 μm^2	5,124 μm^2
Reg. Files	32x32 bit	None	12,071 μm^2	36,214 μm^2
Comm.Queues	32x32 bit	12,071 μm^2	None	None
Instr. Queues	8x32 bit	8,192 μm^2	4,096 μm^2	8,192 μm^2
CQ tables	(12) 4x5 bit	1,920 μm^2	None	None
RF tables	(4) 5 bit	20 μm^2	None	None
Total		27,327 μm^2	21,291 μm^2	49,530 μm^2

such a system is 27.9 mm^2 or about 8.7%. SMT2 systems require 21.8 mm^2 (6.8%) and SMT4 systems require 50.7 mm^2 (15.8%). As such, we believe that from a performance per area perspective, OUTRIDER is a significant improvement over SMT.

CHAPTER 10

RELATED WORK

In this section, we discuss other reduced-complexity processor architectures that leverage either hardware or software approaches to provide some level of memory latency tolerance. Table 10.1 presents the high level features and differences between several designs that are most related with Outrider. Outrider relies on the compiler to extract parallelism instead of costly hardware structures and as a result does not require duplicated execution of memory access instructions or compute instructions as found in other designs. Outrider is a proactive and non-speculative mechanism that provides both memory and functional unit latency tolerance through extracting up to four semi-independent strands of execution. Additionally, Outrider leverages hardware and software techniques to minimize hardware and instruction execution overheads. While other designs require entire thread context hardware state per thread or even processor fetch, decode and functional units to be replicated, Outrider enables an area efficient design without this requirement.

10.1 Compiler-Enabled Techniques

VLIW and EPIC processors leverage the compiler to schedule instructions to avoid both functional and memory latency. The instruction scheduling is best performed by using a software pipelining approach and loop unrolling. This

Table 10.1: Differences between related work and OUTRIDER.
OUTRIDER trades off hardware complexity for software complexity, reducing duplicated instruction execution and large hardware structures.

	Overheads								Benefits					
	Hardware/Software Overheads				Speculation	Duplicate Execution Overhead				Latency Coverage		Memory Access Support		Extracted Concurrency
	Large Inst. Buffers	Increased Context Hardware	Hardware Data Queues	Compiler Support	Speculative Memory Access/ Prefetching	Memory Address/ Access Instructions	Compute Instructions	Branch Instructions	Proactive	L1 Access Latency	L2/L3 Access Latency	Memory Disambiguation	Memory Indirection	More Than Two Memory Strands
VLIW/EPIC		X		X	X				X	X		X		
DSWP		X	X	X					X				X	X
Helper Thread		X		X	X	X			X		X		X	
Slice Processors	X	X			X	X			X		X		X	
Hardware Scout		X			X	X	X	X			X			
Flea-Flicker	X	X			X	X		X			X	X	X	
ICFP	X	X			X					X	X	X	X	
DAE		X	X	X				X	X	X	X			
MISC		X	X	X				X	X	X	X			
OUTRIDER			X	X				X	X	X	X	X	X	X

enables instructions from multiple loop iterations to be executed at the same time in order to overlap functional unit latency. Besides loop control structures, additional memory latency can be tolerated using speculative code motion [14]. VLIW and EPIC machines have been implemented as products and the approach is found in Intel Corporation's Itanium and Transmeta's Crusoe processors [15, 16]. VLIW and EPIC designs have significant hardware overheads such as very complex register file, due to increased entries required for more in-flight (live) values and more ports for larger issue widths. OUTRIDER minimizes hardware overheads through hardware data queues which store only inflight values from long-latency operations as opposed to every value. Utilizing code motion techniques to move load instructions significantly before their use can require data to be speculated and load instructions to be moved in front of stores. This requires significant hardware resources to perform memory disambiguation, and requires software fixup code to be included in the program binary. OUTRIDER is non-speculative, which removes the need for techniques like code motion. Even with code motion, VLIW and EPIC designs can still be sensitive to memory latency and can stall if not enough software pipelining is done. OUTRIDER enables even L2 and L3 latency tolerance and tolerates memory indirection non-

speculatively, unlike VLIW and EPIC designs. Additionally, VLIW designs lead to increased code footprint size, which impacts instruction cache design. Outrider does not require unrolling of code to achieve performance improvement and can tolerate latency using its low-complexity communication queues.

DSWP and STLP are general approaches that extract thread-level parallelism from loops, while Outrider focuses purely on overlapping memory latency in code with or without loops. STLP partitions loop iterations into separate threads speculatively. DSWP uses a loop's dependency graph and static estimated instruction latency to allocate instructions into threads. Both schemes allow memory dependences to exist within a single thread. While complex EPIC or OOO cores these approaches target can tolerate variable load latency in these dependences, simple cores cannot. Outrider uses a stricter and fine-grained scheme that partitions precisely along memory access/consumption lines to avoid costly exposed latency in high-throughput systems with simple inorder cores. Outrider is complementary as a memory latency tolerance technique to parallel partitioning techniques (DSWP authors in TACO 2008).

Decoupled software pipelining (DSWP) is a compiler technique that creates parallel tasks to be generated from sequential programs [17] with loops. A loop is partitioned into pipeline tasks and those tasks are mapped into physical thread contexts in a CMP system, made up of either out-of-order or VLIW/EPIC processors. The motivation is to create parallelism from sequential code and is targeted at high-performance wide-issue processors that already have some degree of memory latency tolerance. This is a different motivation than Outrider, which targets highly parallel systems and applications on simple in-order processors. DSWP partitions 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. Caches misses at these instructions can lead to exposed latency on simple in-order processors. OUTRIDER assumes that variable-latency memory instructions are the most costly, and specifically partitions along memory-access memory-consumption lines to avoid exposed latency. DSWP is complementary to memory-latency tolerant techniques such as those found in OUTRIDER and can improve performance [18].

10.2 Preexecution Techniques

Hardware scout was proposed to enable memory latency tolerance for in-order [19, 20] and out-of-order designs [21]. On an L2 cache miss, the processor checkpoints its state and proceeds to pre-execute instructions during the duration of the miss with the goal of generating prefetches. When the cache miss returns, execution resumes at the instruction that caused the miss. This technique works well when only single levels of cache misses are observed. When memory indirection causes several memory accesses which each miss in the cache, hardware scout is not effective. OUTRIDER extracts multiple strands, which allows higher performance in the case of memory indirection. Hardware scout is speculative and performance depends on correct branch prediction during the pre-execute period. The prefetches generated by hardware scout are sensitive to correct branch prediction, timeliness, and cache contention, unlike OUTRIDER in which data is not prefetched or speculated. Pre-execution of address-generation, control, and compute instructions duplicates work that is done by the main program. As a non-speculative program,

OUTRIDER does not duplicate address generation and compute instructions.

Flea-flicker [22] 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. However, address generation instructions and control flow instructions are reexecuted, unlike OUTRIDER. Flea-flicker enables prefetching multiple levels of memory indirection through making multiple passes over the instructions in the instruction buffer. Like hardware scout, prefetches can be sensitive to correct branch prediction, timeliness, and cache contention. Unlike Flea-flicker, OUTRIDER does not require a large instruction buffer to handle memory indirection and instead explicitly exposes MLP in multiple strands to achieve performance. Additionally, a result store buffer is not needed as data dependences can be targeted using the software partitioning into strands in OUTRIDER.

In-order continual flow pipelines (iCFP) [23], and Simultaneous Speculative Threading (SST) [24] 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 preexecution 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. OUTRIDER 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 stream of execution, while Outrider can extract up to four.

10.3 Helper Thread Techniques

Slice processors [6] 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. Outrider is not 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. Outrider requires much more meager hardware overheads, only enough to buffer instructions and the data communicated between strands.

Helper threads [5] 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. OUTRIDER is not a prefetching technique and is not sensitive as helper threads are. Also, helper threads duplicate execution of the address generation stream while OUTRIDER does not.

10.4 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 [7]. 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 [25, 26, 27, 28]. While DAE enables parallelism and can allow the memory thread to create MLP, it is unable to handle memory indirection or compute dependent memory accesses which degrade performance. OUTRIDER utilizes additional strand parallelism 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. OUTRIDER enables out-of-order completion of messages and general data communication through the communication queues. Finally, OUTRIDER utliizes SMT to share fetch and execution resources and enable efficiency not found in DAE.

The Multiple Instruction Stream Computer (MISC) attempts to improve over DAE by providing four one-wide issue in-order processors [29]. Up to four strands can be extracted, with two of those strands allowed to access memory. MISC relies on the compiler to provide guarantees of no memory aliasing and proper load-store ordering. For programs that use pointer struc-

tures, there is no guarantee of correct memory ordering for multiple memory strands and the partitioning must be restricted to a single memory strand. OUTRIDER enables all four strands to access memory with correct memory ordering, which is enabled by a mixed hardware and software approach to memory aliasing detection. The compiler conservatively places `st_addr` instructions in all strands which are fed into the MAU. Later loads look in the store buffer to maintain proper ordering. MISC has a downfall of poor hardware utilization due to the strands executing on separate processors. Each processor has an instruction cache, decode, functional units, and full hardware queues to connect strands which, depending on the strand's instruction stream, may not be fully utilized. OUTRIDER uses SMT to share the instruction cache, decode, and functional units, which enables high area efficiency. OUTRIDER removes the need for separate memory response and general data queues, unifying them into a partitioned buffer. Using the partitioned buffer split into dynamically sized queues allows higher area efficiency and utilization of area, unlike MISC which requires a total of 24 separate queues statically sized to the highest amount of slip and memory latency tolerance desired.

More contemporary decoupling work involves hardware partitioning and SMT [30]. In this work, the authors propose hardware partitioning of integer and floating-point instructions into separate threads in order to extract MLP using large instruction queues to hold dependent floating-point instructions while they wait for the miss to return. These instruction queues needed can be more than an order of magnitude larger that those required for OUTRIDER, and this technique is limited to floating point applications. Additionally, this technique suffers from memory indirection and compute-dependent memory accesses, which OUTRIDER supports. The technique also only supports SMT

of different threads on either the EP or AP, unlike OUTRIDER which uses SMT across strands.

10.5 Orthogonal Techniques

Software pipelining [31] and loop unrolling allow the compiler to schedule memory and other dependent operations in advance of their use as a way to exploit ILP. Software pipelining involves unrolling loops to expose MLP followed by rescheduling memory operations in advance of consuming instructions. Software pipelining increases the code footprint of the application and more hardware registers can be required to hold in-flight values. In OUTRIDER, the values communicated are consumed almost immediately, reducing storage overhead.

Hardware and software prefetching often found in cached systems can be effective in alleviating the memory latency problem by anticipating memory accesses and warming up caches. Hardware prefetching involves SRAM-based structures that monitor the memory access stream and identify patterns, while software techniques require duplication of part of the instruction stream and user or compiler effort to insert prefetches. Unlike prefetching, OUTRIDER is non-speculative and has 100% utilization of all memory requests, completely eliminating waste found in prefetching.

CHAPTER 11

CONCLUSION

In this thesis, we present OUTRIDER, an architecture for directly exposing MLP in highly parallel workloads. The memory wall is making the efficient extraction of MLP critical to scaling performance on future CMPs. OUTRIDER has the goal of increasing the efficiency of highly parallel CMPs by decoupling the memory access streams from the rest of the computation. Doing so allows for increased concurrency in the memory system with minimal additional cost over our in-order baseline micro-architecture and without additional thread contexts found in multi-threaded architectures. We find that the key advantage OUTRIDER provides over previous decoupled access-execution architectures is the ability to continue decoupled execution when memory indirection and data-dependent control flow are present in applications. Our results comparing OUTRIDER to a conventional multi-threaded architecture show that directly expressing MLP via strands rather than indirectly through threads can provide performance advantages of 23–131%. Our limits studies demonstrate that the hardware overhead of OUTRIDER structures relative to our in-order baseline can be modest and much lower than the cost of additional register files and increased cache sizing necessary to support more threads. The result is a micro-architecture for parallel systems that can take advantage of both TLP and MLP while relying on a simple in-order pipeline and a lower number of explicit software threads.

REFERENCES

[1] S. Rusu, S. Tam, H. Muljono, J. Stinson, D. Ayers, J. Chang, R. Varada, M. Ratta, and S. Kottapalli, "A 45nm 8-core enterprise Xeon processor," in *IEEE International Solid-State Circuits Conference*, 2009, pp. 56–57.

[2] M. Ware, K. Rajamani, M. Floyd, B. Brock, J. Rubio, F. Rawson, and J. Carter, "Architecting for power management: The IBM POWER7 approach," in *IEEE 16th International Symposium on High-performance Computer Architecture*, 2010, pp. 1–11.

[3] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[4] J. Shin, K. Tam, D. Huang, B. Petrick, H. Pham, C. Hwang, H. Li, A. Smith, T. Johnson, F. Schumacher, D. Greenhill, A. Leon, and A. Strong, "A 40nm 16-core 128-thread CMT SPARC SoC processor," in *IEEE International Solid-State Circuits Conference*, 2010, pp. 98–99.

[5] C.-K. Luk, "Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001, pp. 40–51.

[6] A. Moshovos, D. N. Pnevmatikatos, and A. Baniasadi, "Slice-processors: An implementation of operation-based prediction," in *Proceedings of the 15th International Conference on Supercomputing*, 2001, pp. 321–334.

[7] J. E. Smith, "Decoupled access/execute computer architectures," in *Proceedings of the 9th Annual Symposium on Computer Architecture*, 1982, pp. 112–119.

[8] K. D. Rich and M. K. Farrens, "Code partitioning in decoupled compilers," in *Proceedings of the 6th European Conference of Parallel Processing*, 2000, pp. 1008–1017.

[9] N. Topham, A. Rawsthorne, C. McLean, M. Mewissen, and P. Bird, "Compiling and optimizing for decoupled architectures," in *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, 1995, p. 40.

[10] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995, pp. 392–403.

[11] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: Architectural support for fine-grained parallelism on chip multiprocessors," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007, pp. 162–173.

[12] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007, pp. 3–14.

[13] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel, "Rigel: An architecture and scalable programming interface for a 1000-core accelerator," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009, pp. 140–151.

[14] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W.-m. W. Hwu, "Dynamic memory disambiguation using the memory conflict buffer," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994, pp. 183–193.

[15] H. Sharangpani and H. Arora, "Itanium processor microarchitecture," *IEEE Micro*, vol. 20, no. 5, pp. 24–43, 2000.

[16] J. Dehnert, B. Grant, J. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The Transmeta Code Morphing^{TM} software: Using speculation, recovery, and adaptive retranslation to address real-life challenges," in *International Symposium on Code Generation and Optimization*, 2003, pp. 15–24.

[17] G. Ottoni, R. Rangan, A. Stoler, and D. I. August, "Automatic thread extraction with decoupled software pipelining," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, 2005, pp. 105–118.

[18] R. Rangan, N. Vachharajani, G. Ottoni, and D. I. August, "Performance scalability of decoupled software pipelining," *ACM Transactions on Architecture and Code Optimization*, vol. 5, no. 2, pp. 8:1–8:25, 2008.

[19] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay, "High-performance throughput computing," *IEEE Micro*, vol. 25, pp. 32–45, 2005.

[20] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," in *Proceedings of the 11th International Conference on Supercomputing*, 1997, pp. 68–75.

[21] O. Mutlu, H. Kim, and Y. N. Patt, "Efficient runahead execution: Power-efficient memory latency tolerance," *IEEE Micro*, vol. 26, pp. 10–20, 2006.

[22] R. D. Barnes, S. Ryoo, and W.-m. W. Hwu, ""Flea-flicker" multipass pipelining: An alternative to the high-power out-of-order offense," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, 2005, pp. 319–330.

[23] A. Hilton, S. Nagarakatte, and A. Roth, "iCFP: Tolerating all-level cache misses in in-order processors," *IEEE Micro*, vol. 30, no. 1, pp. 12–19, 2010.

[24] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, "Simultaneous speculative threading: A novel pipeline architecture implemented in Sun's rock processor," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009, pp. 484–495.

[25] J. E. Smith, G. E. Dermer, B. D. Vanderwarn, S. D. Klinger, and C. M. Rozewski, "The zs-1 central processor," in *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 1987, pp. 199–204.

[26] J. R. Goodman, J.-t. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young, "PIPE: A VLSI decoupled architecture," in *Proceedings of the 12th Annual International Symposium on Computer Architecture*, 1985, pp. 20–27.

[27] L. K. John, V. Reddy, P. T. Hulina, and L. D. Coraor, "Program balance and its impact on high performance RISC architectures," in *Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture*, 1995, pp. 370–379.

[28] L. Kurian, P. T. Hulina, and L. D. Coraor, "Memory latency effects in decoupled architectures with a single data memory module," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992, pp. 236–245.

[29] G. Tyson, M. Farrens, and A. R. Pleszkun, "MISC: A multiple instruction stream computer," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, 1992, pp. 193–196.

[30] J.-M. Parcerisa and A. Gonzalez, "Improving latency tolerance of multithreading through decoupling," *IEEE Transactions on Computers*, vol. 50, no. 10, pp. 1084–1094, 2001.

[31] M. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, 1988, pp. 318–328.