On Replay and Hazards in Graphics Processing Units

by

Andrew E. Turner

B.A.Sc., B.A, University of British Columbia, 2007

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Masters of Applied Science

in

THE FACULTY OF GRADUATE STUDIES
(Electrical and Computer Engineering)

The University Of British Columbia
(Vancouver)

August 2012

Abstract

Graphics Processing Units (GPUs) have potential for more efficient execution of programs, both time wise and energy wise. They can achieve this by devoting more of their hardware for functional units. GPGPU-Sim is modified to simulate an aggressively modern compute accelerator and used to investigate the impact of hazards in massively multithreaded accelerator architectures such as those represented by modern GPUs employing a single-instruction, multiple thread (SIMT) execution model. Hazards are events that occur in the execution of a program which limit performance. We explore design tradeoffs in hazard handling. We find that in common architectures hazards that stall the pipeline cause other unrelated threads to be unable to make forward progress. This thesis explores an alternative organization, called replay, in which instructions that cannot proceed through the memory stage are squashed and later replayed so that instructions from other threads can make forward progress. This replay architecture can behave pathologically in some cases by excessively replaying without forward progress. A method which predicts when hazards may occur and thus can reduce the number of unnecessary replays and improve performance is proposed and evaluated. This method is found to improve performance up to 13.3% over the stalling baseline, and up to 3.3% over replay.

Table of Contents

List of Tables

List of Figures

Glossary

ALU	Arithmetic Logic Unit
CPU	Central Processing Unit
GPU	Graphics Processing Unit
GPGPU	General Purpose [processing on a] GPU
RAW	Read After Write
SIMD	Singe-Instruction, Multiple Data
SIMT	Single-Instruction, Multiple Thread
SFU	Special Function Unit
RISC	Reduced Instruction Set Computing
ISA	Instruction Set Architecture
IMEM	Instruction Memory
DMEM	Data Memory
DIV	Memory Divergence hazard
MSHR	Miss Status Holding Register
COMQ	Cache Communication Queue
RSV	Cache Line Reservation hazard
BANK	Bank Conflict Reservation
VLIW	Very Long Instruction Word
ICNT	Interconnect Resource Hazard
TLB	Translation Look-aside Buffer
CAM	Content Addressable Memory

PAM Private Active Mask

IP Issue Pointer

ITP Issue Tail Pointer

FP Fill Pointer

Acknowledgments

I am grateful for the support of many while completing my thesis. In particular, I wish to thank Professor Tor Aamodt for his guidance, suggestions and support.

I also would like to thank Ali Bakhoda, Wilson Fung and others from our research group for their help and camaraderie, and particularly for their advice as I prepared my final presentation.

Thank you to both of my parents for their great support, and my father for his help with many revisions. And finally, to my wife, Caitlin, whose support and love has sustained me.

Chapter 1

Introduction

Graphics Processing Units (GPUs) are a popular type of computer chip processor design. GPU type architectures are widely available separately or as co-processors in chips used in both desktop and portable devices. Gaining improvements in GPU power use and performance is a current focus of computer architecture research.

1.1 Overview

This section is an overview of computer architecture and the design and functioning of GPUs.

1.1.1 Computer Architecture

Computer architecture research is the study and design of the functional organization of computer processors. In recent years interest has been increasing in the design of highly parallel processors. Conventional monolithic processor designs generally only run a single[1] instruction stream, or program thread, at a time. Highly parallel processors have advantages in power efficiency and computing throughput. Compute accelerators, such as GPUs are the most popular parallel processor architecture currently in production, and are increasingly being used for non-graphics purposes, termed General Purpose computation on GPUs (GPGPU).

The performance of processors depends heavily on how hazards, situations where normal, fast and efficient processing cannot proceed, occur and are resolved. This problem has been quite well studied for monolithic conventional CPUs [5][14], but there is much to be understood about hazards in highly parallel architectures.

This work characterizes hazards for a GPU-like compute accelerator, and provides an analysis of possible strategies to improve performance through more effective hazard management.

[1] Or sometimes a few, such as with hyper-threading or a few on multiple monolithic cores.

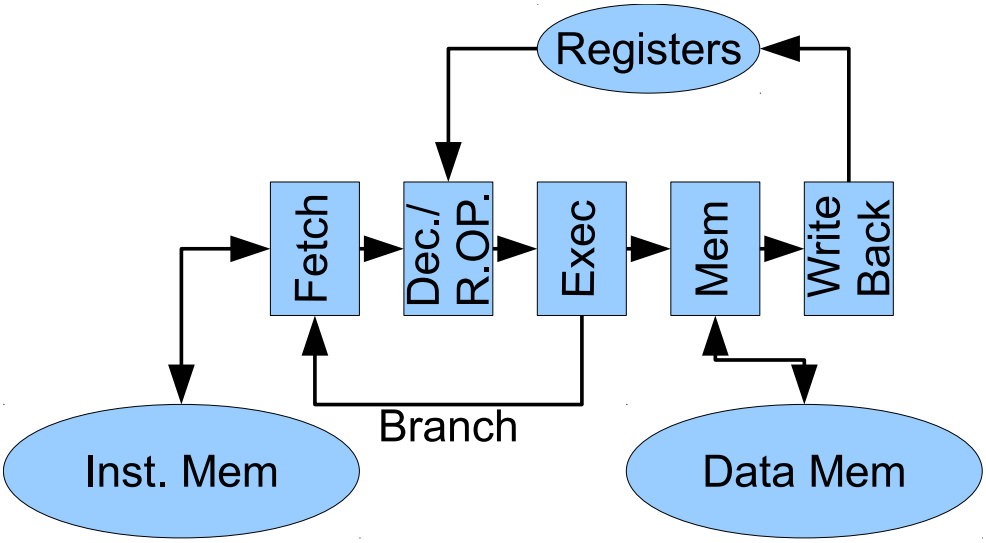

Figure 1.1: Simple core architecture.

1.1.2 Architecture Basics

A simple RISC (Reduced Instruction Set Computing) processor is depicted in Figure 1.1. A RISC processor executes a limited type of ISA (Instruction Set Architecture) where instructions either perform operations on register values, or store/load register values to/from Data Memory, but never both - this reduces the number of ISA instructions and simplifies the architecture. Elaborations and refinements of the basic RISC architecture constitute the architecture of the majority of computers today. This processor consists of a pipeline of basic steps, that Fetch, Decode/Read Operands, Execute, Access Data Memory, and Write Results, for a stream of instructions which constitute the program. The processor core generally has access to at least three types of memory: Instruction Memory (IMEM), which contains the instruction code accessed by the Fetch stage; Data Memory (DMEM), which contains data read and written by the program (Mem Stage); and Register Memory, which contains the register values for the running program (read at Read Operands, written at Write Back). The program being run is stored in the instruction memory as instruction code. In modern designs, Main memory is stored outside the core, and Instruction memory and Data memory are interfaced through caches. Modern processors use pipelining heavily. Pipelining allows multiple instructions (or more generally a stream of operations) to be processed in parallel through a number of serial steps; in the architecture of Figure 1.1, a different instruction could be processed in each of Fetch, Decode, Execute, Mem and Write Back.

The processor runs a program from the Instruction Memory, which is a sequential series of instructions (that conform to the ISA). This single sequence of instructions is a thread of execution. A single-threaded program contains only one thread whereas a parallel program is comprised of multiple threads (a highly parallel program has many threads). A thread is a program (or part of

a program) running in a single architecture thread context that handles a single linear sequence of instructions. A computer may run multiple programs concurrently by supporting multiple thread contexts.[2] The registers are a store of local memory which is small but fast, and the data memory system is a large store of distant memory which is typically large but slow. Instructions are accessed from Instruction Memory in the fetch stage based on a program counter. The program counter might be modified by a branch instruction in the execute stage. Decode/Read Operands derive control signals from the instruction and also access the Register file for the operands required by the instruction. These operands are passed along with the instruction to the execute stage where computation may create a result. Some instructions do not use the execute stage. Some data, either the computation result, or an operand value, is passed along with the instruction to the Memory stage. If the instruction is a memory operation, then the memory stage performs the appropriate operations, usually either a read or write, to or from the data memory. If the instruction is not a memory operation then the memory stage does nothing. The resultant data is further passed along to the write-back stage. If the instruction stores its result to the register file, then the result is written back to the register file. This completes the execution of the instruction.

In a RISC (Reduced Instruction Set Computing) processor (of which Figure 1.1 is an example) the instructions are either ALU operations or Memory operations. Memory operations are restricted to only transfer data between registers and main memory. ALU operations only operate on data from the registers, and if they produce a value, can only place the results back into the register memory space. For the purposes of this work we group conditional branch and jump instructions in with ALU operations since they do not access main memory and have the same data characteristics of regular ALU operations (although they may access register data differently). The RISC simplification allows instructions to be more easily encoded, reduces the size of the instruction set, and simplifies the pipeline.[3] The examined architectures are also classified as modified Harvard architectures, since both instruction and data memories are accessed independently and concurrently through dedicated caches but backed by a common unified memory.

Separating the execution of an instruction into multiple pipeline stages allows multiple instructions to be processed at the same time. For example Fetch may be loading a new instruction on the same cycle that an ALU operation is undergoing execution in the Execute stage. The instruction throughput is increased because the smaller pipeline stages have less work to do per cycle, increasing the clock frequency at which the processor can be run. Pipeline stages are generally designed so that their entire operation can be completed in a single clock cycle; the clock frequency is limited by the longest completion time of the slowest pipeline stage. Two major downsides to pipelining are that instructions are not independent from each other and pipeline stages may not always be able

[2] An architecture with only a single thread context may provide the illusion of running multiple programs or threads in parallel by switching quickly between them.

[3] The other major instruction set type is CISC (Complex Instruction Set Computing), in which a single instruction can access and do computation on data in memory.

complete their work in a single cycle. Instructions may be coupled to each other through their data dependencies. An instruction may require a result from another instruction, and if these instructions are too close to each other in sequence, the second instruction must wait in the decode stage until the first has completed the write-back stage. This is a type of hazard. A similar hazard occurs when a branch instruction relies on its calculation in execute, as the next instruction fetched depends on the outcome of that branch.

Another hazard type is when an instruction on its own cannot complete a pipeline stage in a single cycle. In this case an ALU instruction may take multiple execute stages to complete the computation, or the memory system may not be able to satisfy a memory instruction's request in a single cycle. The latter, memory hazards are typically unpredictable and are the focus of this work.

Hazards are a problem for pipelines because they cause instructions upstream (earlier in the pipeline) to stall. Stalling is when an instruction cannot proceed to the next stage at the end of a cycle. Although upstream instructions could otherwise proceed, they also stall when a downstream instruction stalls. This stalling reduces throughput and thus reduces performance. Even in more complicated architectures, unpredictable hazards causing stalls are a significant penalty on processor performance [14][29].

1.1.3 Monolithic CPU Strategies

Extensive architecture research has been done on improving single thread performance in monolithic core processors. Monolithic core processors typically expend much of their area and energy budget to optimize the performance of one or two threads. In order to combat the performance penalty of hazards, these CPUs have utilized architecture modifications to the simple pipeline. The general strategy has been to mitigate hazards by out of order effects, deep pipelining and speculation [14].

Out-of-order execution allows the processing of instructions to occur out of program order. This includes out-of-order execution, where data hazards are alleviated by executing instructions as their operands become available in multiple execution stages. This enables higher instruction throughput since many instructions may not depend on each other and may be run at the same time. Another form of out of order effect is out of order memory in conjunction with out of order execution. Because of the way data memory is constructed[4], it is possible for some requests to complete much faster than others. In an in-order pipeline, these faster requests would have to wait for slower ones to complete, whereas in an out of order pipeline these could complete as they are ready.

Out-of-order execution is usually limited to a small window of instructions from a single thread. Because this window can generally not be advanced until the oldest executing instruction is complete, the long-latency instructions become important to performance. In addition, the single threaded

[4]Memory accesses are typically accessed through a series of caches, accesses that miss in lower levels and are forced to check higher level caches or main memory and incur increasing latencies for completion.

nature of monolithic cores limits how much parallel work is available while earlier instructions are outstanding due to dependencies. Thus, memory operations, which are often the highest latency instructions in the instruction set, are optimized for latency rather than bandwidth. This, for example, leads to larger caches that support only a few outstanding accesses.

CPUs tend to also pipeline and deeply pipeline [5][14] as much possible. This allows extremely high frequencies of execution, and thus more throughput per real time.[5] The downside to this approach is that hazards carry more penalty than they would in a shorter pipeline. For example the resolution of a branch instruction may be many pipeline stages away from the fetch stage, causing a delay, in this case the branch penalty, until the next instruction can be fetched and issued.

Speculation is used by monolithic cores to reduce the effects of hazards. The primary use use of speculation targets the branch delay penalty. Typically branches are predicted based on past behavior and instructions are issued and executed based on that prediction until the branch is actually resolved. If the prediction was wrong, all speculated instructions and results are scrapped and execution begins on instructions fetched from the correct path. This can get complicated as multiple outstanding branches occur and must be handled. Another form of speculation is prefetching, in which data that may be used later is loaded into the local cache so it is available for expedient use later.

In addition to instruction throughput, processor design constraints have grown to include the area and power dissipation characteristics of processors and their components. More complicated logic and larger storage occupies more area (because it uses more transistors), whereas power dissipation is both a function of area (more transistors use and leak more power), and transistor switch rate (each transistor switch uses more power). In the monolithic CPU designs, much area and power is devoted to the execution of single threads. In speculation for example, a large area can be used, and in mispredictions a large amount of power consumed for no benefit. A class of different processor organization, GPUs, attempts to alleviate these problems by using available area and power more directly for computation with more area allocated to ALUs.

The limits of single core architectures motivates the movement to multicore architectures [28], where smaller monolithic processors are combined on a single chip. Moving further in that direction, more, smaller cores, leads towards GPU architectures.

1.1.4 Compute Accelerators

Compute Accelerators are a broad group of computer architectures designed to accomplish high throughput of highly parallel programs. GPUs are a mainstream example of this type of architecture.

The basic principle behind GPUs is that many parallel threads can run at the same time. In this scenario the accelerator is optimized for throughput: a large number of concurrent threads, a large

[5] In Computer Architecture when comparing similar designs, cycle time is assumed to be constant, so throughput is taken to be instructions / cycle. This ignores cycle timing effects such as mentioned here.

number of ALUs and high memory bandwidth. This is a different strategy from the monolithic CPU architectures where resources are expended to improve the performance of just a single or few threads.

The key to the success of GPUs is that although threads may individually experience long delays due to memory or other hazards, these delays are mitigated by the presence of other independent threads. What is important is the throughput of all threads, and this depends on the average rate of execution of the individual threads multiplied by the number running concurrently. In a single thread, performance is limited by the latency of constituent instructions: a memory access that misses in the cache may prevent that thread from doing any work for hundreds to thousands of cycles (hence the focus of Monolithic cores on latency and prefetching). In a monolithic core this delay likely leaves functional units and much of the core idle. When there are many threads executing, a large number may be unable to make progress because of cache misses, but a large number are also able to execute, utilizing the core more completely. This ability to cope with threads being blocked because of long latency occurrences, translates to a tolerance for some types of hazards which typically delay the completion of some instruction, such as a cache miss.

Because individual threads are independent and can be reordered with respect to each other, GPUs achieve some of the benefits of out-of-order execution without having to manage its downsides. Both techniques exploit parallelism in the program to keep as many functional units (ALUs, Memory) that could do useful work busy at the same time. With access to many threads ready to execute at (hopefully) different parts of their program, a selection of instruction types can fill many of the function units. Monolithic cores attempt to achieve this by running individual threads out of order, as well as running a few threads at the same time (e.g. hyper-threading [18]). Monolithic cores also must manage complexities of this out of order execution, such as data and register management and precise exceptions. GPUs do not have these specific challenges because individual threads execute in order.

Because hazards are better tolerated, GPU architectures can omit expensive modules that assist single-thread performance, such as out-of-order execution or speculation. This saving in area and power dissipation can be used for supporting more simultaneous threads, more ALUs and larger caches, further boosting throughput.

GPU compute accelerators run many threads at once primarily through three mechanisms: 1) by combining threads together in batches called warps, 2) having many warps reside actively on each core, and 3) having many cores. Figure 1.2 outlines how threads are conceptually grouped into warps, blocks and cores, and how these are run on a simple core pipeline. For a modern GPU such as the NVIDIA Fermi, concurrently executing threads can number up to 32 (threads per warp) * 48 (warps per core) * 15 (cores) = 23040 threads [25].

As shown in Figure 1.2, warps are special because they are passed down the pipeline as a unit and operated on concurrently at each stage. Threads are grouped into warps such that each individual

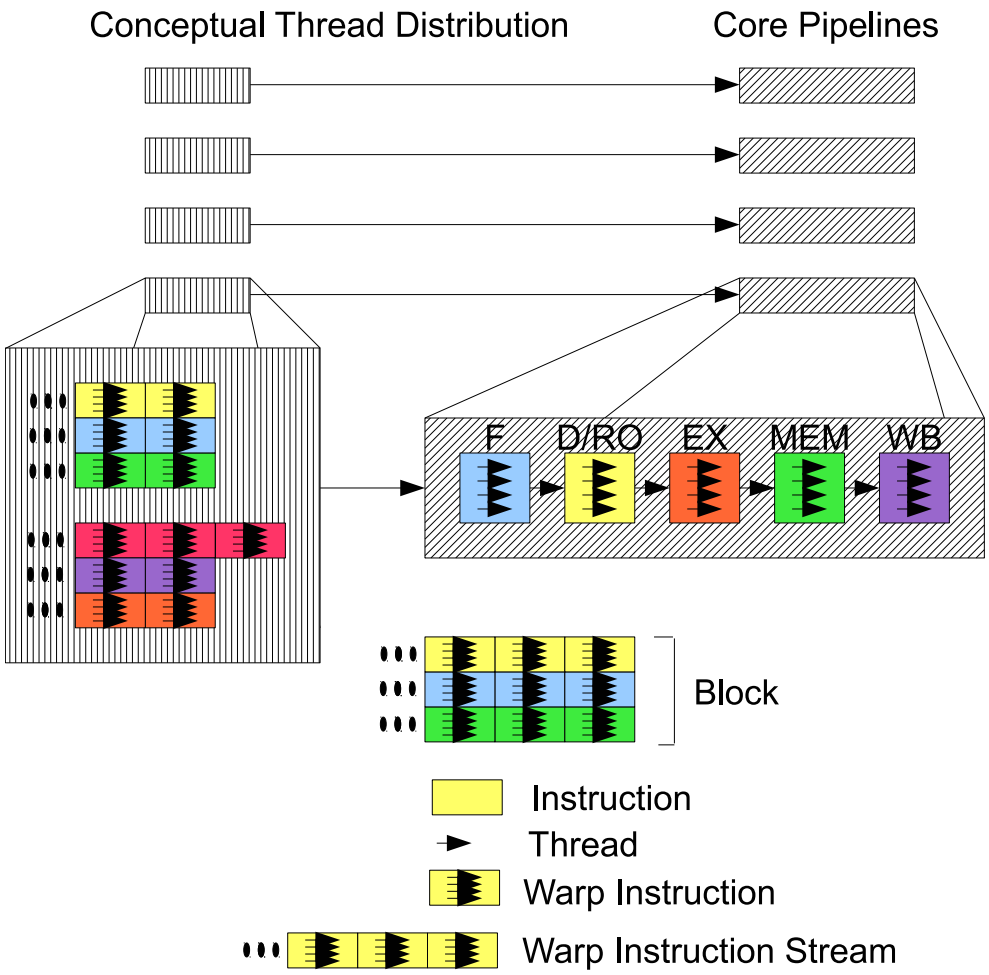

Figure 1.2: Conceptual thread grouping and execution on a GPU.

thread is at the same program instruction, in other words threads in each warp all share the same program counter value. Each thread in the packet requires the same pipeline stages and operations, so control lines are conserved. Only the data carried for each thread (which may be different for each thread in the warp) need be stored and passed between pipeline stages. Each thread accesses its own registers and reads and writes to data memory as if it were being run independently. This independence is extended to control flow as well.

This model is known as SIMT (Single Instruction, Multiple Thread) as opposed to SIMD (Single Instruction, Multiple Data) in that in addition to handling multiple data, this system handles the multiple control flow paths of individual threads in the warp packet. This allows threads to be conceptualized as separate and independent, even though they execute together in warps. This simplifies programming and prototyping, as single threaded programming methodologies can be used initially and adapted to provide increased performance. In SIMD, a major limitation is that

all threads that are packaged together (like a warp) must branch together, a condition that must be explicitly ensured by the programmer, and memory requests for each thread typically have to be contiguous. More information on programming is provided in section 1.1.5.

Control flow independence within a warp is obtained using two mechanisms. Predication of instructions[6] suffices for simple branches (a simple branch chooses between relatively few instructions; the threshold at which a branch is implemented by predication is determined by the compiler). More complicated branching is handled by a per warp stack that keeps track of the branching history of warps [9]. Each branch is systematically executed for each thread that takes that branch by masking off all other threads, and running the warp for the instructions in that branch. Branching within a warp can have significant performance penalties because each instruction in each branch requires a separate pass through the pipeline. Thus a warp that branches once executes all the warps that take the branch then all the warps that don't take the branch - the throughput is reduced by half. No penalty occurs if all threads in a warp take the same branch direction.

Another aspect of SIMT is its interaction with the memory stage. The memory stage is limited to processing a small number (usually just one) of large memory transactions per cycle. If all the memory requests for each thread are not within a single span of contiguous aligned memory then multiple cycles must be used to issue requests for all such spans required by the warp. This is another form of reduced throughput. This occurrence is called memory divergence. This is an important hazard because it occurs unpredictably at the memory stage, since is is not possible to determine if it will occur until the access addresses have been calculated for each thread.

1.1.5 Programming Languages

Programming languages and schemes for highly parallel architectures is a current research problem, and is challenging and complex. SIMT hardware attempts to simplify this problem by abstracting threads from their warp groups and allowing them to be treated as full independent threads. NVIDIA CUDA [27] is the programming language created to run on NVIDIA SIMT GPUs.

The CUDA language is C/C++ with additional adornment for specifying kernels (the code that runs on the GPU), kernel launches and special memory storage. CUDA is developed specifically for GPGPU programming. It is distinct from computer graphics programming interfaces such as OpenGL and DirectX, although they utilize the same hardware.

Although any SIMT programming language could utilize the hardware improvements proposed in this work, such as OpenCL [16], CUDA was used where not otherwise indicated.

[6]Predication involves conditioning an instruction on some per-thread data. The condition is tested separately for the data of each thread in the warp, and if and only if the condition is true then that instruction is executed on that thread. Predication is usually done in two stages, a test instruction is performed which creates a bit mask, and then the predicated instruction executes using that bit mask. This can be efficient for very small branches, but is untenable for complicated control flow.

1.1.6 Instruction / Data Flow in GPUs

In the GPU it is important to understand the progression of functional units a warp passes through in order to be executed. Each pipeline stage represents a significant step in routing and processing the instructions in a warp.

Figure 1.3 shows a simplified schematic of the baseline architecture that is used. It is similar to the NVIDIA Fermi architecture [25]. Subfigure 1.3(a) is the overall chip architecture comprised of core and L2/Memory Controller (L2+MC) units connected by a shared interconnect. The L2+MC units are a combined top level cache and memory controller. These handle memory requests from the cores, and if necessary dispatch requests off chip to the ram memory storage. Subfigure 1.3(b) shows the baseline core architecture. In contrast to the simple architecture shown in Figure 1.1, rather than a single linear pipeline, instruction warps can be steered into multiple different functional units which can operate in parallel. Pipeline stages may then take longer than a single cycle to process their instruction warp, and may themselves be pipelined or support multiple concurrent warps. ALUs may take multiple cycles to compute all the values for each thread, and some pipeline stages are designed to have high throughput but longer latency per warp, such as the scheduler, or the operand collector.

The processor can be conceptually separated into two portions, the front end, which deals with warp selection and warp issue, and the back end which deals primarily with thread data. In the diagram, Fetch, Decode, the Warp Buffer, Scheduler and Mask constitute the front end, while the Operand Collector, functional units and Operand Write Back are the back end.

The following is a short description of how each stage operates and communicates. This defines the baseline architecture which is used as a baseline to compare the benefits of architecture changes.

1.1.6.1 Fetch / Warp Buffer / Scheduler

The Warp Buffer is the primary staging area for warp instruction streams. The Warp Buffer consists of a table of storage for warp instruction streams, one entry for each warp allocated to the core. Each entry consists of a number of instruction storage areas and tracking information, such as instruction addresses, whether operand values are ready, and execution state.

When an entry in the Warp Buffer is low on instructions, the Fetch stage attempts to fill new instructions at the correct instruction address from a locally cached line obtained from the instruction cache. If the needed instruction address is not included in the cached line, the appropriate line is requested from the instruction cache memory.

Instructions are issued down the pipeline from the Warp Buffer by the Scheduler. The Scheduler typically chooses warps in a round robin fashion from those warps that are ready to execute. In the baseline design, two schedulers issue independently from exclusive sets of warps (split into even and odd warps). The scheduler also supports multiple issue, where if the first instruction in a warp

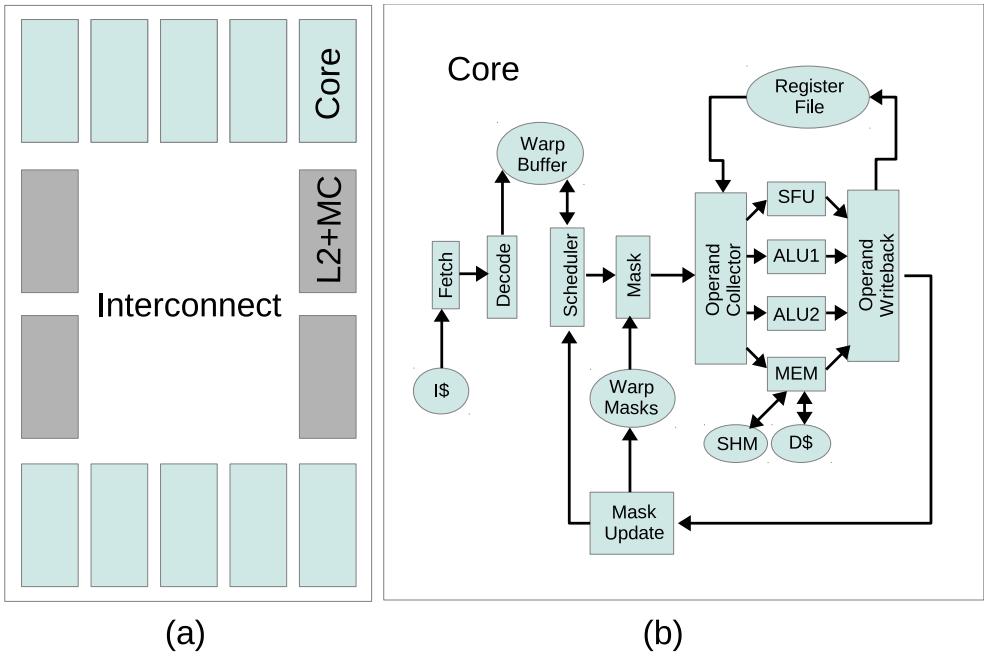

Figure 1.3: Baseline chip and core architecture. (a) Chip architecture, (b) Core architecture.

instruction stream entry can be issued, the second instruction is also issued if possible. This can improve performance when only a few warps are able to issue by helping fill function units.

Read after write dependencies occur when an instruction reads operands to which an older instruction is writing. If the older instruction has not completed, then the newer instruction cannot issue.[7] Read after write dependencies are avoided by using a separate table (the RAW table) that tracks which operands are being written to. When an instruction is issued the registers it will write to are marked off in this table. When that instruction actually writes to those registers those entries are removed from the table (the table also ensures that every active thread has written to those registers). Instructions are not issued unless the operands they read do not have entries in the table.

The baseline architecture supports stall-on-use [8] thread execution enabled by the RAW table. The processor may continue to issue newer instructions from a warp even before older instructions from that warp have completed. The warp may fetch ahead and execute in this manner until it is blocked by a read after write requirement (as determined from the RAW table) or runs out of instructions. This feature allows programs to benefit from pushing data load instructions as early as possible before the data is used. This also allows the possibility for multiple outstanding loads from a single warp as well as the possibility for computation to continue while data is being loaded. This modification can increase pressure on the memory system by increasing the number of possible outstanding loads.

[7] Write after write and write after read dependencies are dealt with in the same manner as RAW dependencies.

The baseline architecture uses a version of two level warp scheduling [23] policy to ensure a constant supply of both memory and ALU instructions. It is modified to use two schedulers. Each scheduler has separate sets of blocks to issue from.

1.1.6.2 Mask Stage

After issuing, instructions in a warp are processed and various masks are applied. These masks mark out threads in the warp that are not to be executed with this instruction. This primarily involves the branch masks, which make sure only the correct threads are executed on each side of branches. Depending on architecture this may not need to be a separate stage thus saving a cycle latency. This is done in the baseline architecture.

1.1.6.3 Operand Collector

The Operand Collector is where instructions receive data for their operands from the register file. Warps may wait several cycles here while waiting for the reads to occur. Although the register files are banked to allow multiple reads, it may not be possible to service even all of the same instructions operands at the same time. However, the operand collector can serve multiple warps at the same time, thus maintaining a higher throughput in spite of the higher latency. When an instruction has data for all its operands, it is issued to the next stage. Some scheduling may occur at this stage, because the latency for ALUs is typically fixed. This allows a scheduler to issue instructions in such a way that they complete at the correct time and don't interfere with other instructions in the same pipeline.

1.1.6.4 Functional Units

After the Operand Collector, warps diverge into different functional units for execution. Instructions either go to the Memory Unit, to one of many ALUs or to other functional units which may be present, such as Special Function Units (SFU) and a branch unit. SFUs can process more complex mathematical instructions such as square roots, and branch units handle updating the state associated with branches.

The ALUs are relatively simple from a operations perspective. The instructions have their data, and the ALUs implement the logic necessary to carry out the arithmetic operation required (chosen based on the control lines). This is very predictable, and the output is calculated in a fixed time depending on the operation.

The Memory Unit manages access to different types of memory. Three types of memory are used: constant, shared, and global. Constant memory is read only memory that is stored globally, and cached locally. Shared memory is both read and write memory and is stored locally in the core and is shared across all the warps running on the core. Threads within a core can communicate

using shared memory. Global memory is both read and write memory and is stored off chip. All warps across all cores may access the data in this memory. All threads in the GPU can communicate through global memory, although coherency and ordering is not necessarily enforced. In the baseline architecture, global memory is cached locally in an level one (L1) cache and at the chip level in an level two (L2) cache.

Texture memory is a special type of constant memory optimized for graphics texture lookups. Texture memory is implemented in special way to optimize throughput. Although this is simulated in the simulator, and some GPGPU benchmarks make use of this memory type, this memory type is not explained here. More information is available in [13].

1.1.6.5 Write Back

For instructions that return a value, that value must be written back into the register file. Instructions that are writing back to the register file coordinate with the operand collector to write their results into the appropriate banks. For ALU operations this may be already scheduled (due to predictable processing time), and there may be additional ports on the register file to maintain enough throughput for unpredictable write backs (such as may come from memory).

For example, an ALU add instruction that takes four cycles, may when it is issued, schedule the result write to occur at that time. The operand collector knows then to keep a predetermined register port clear on that cycle to allow the write back (alternatively the instruction can be delayed issue if no ports will be available). A memory instruction that writes back (a load of some kind) cannot guarantee when it can write back due to hazards it may yet encounter. In these cases a separate dedicated port may be used, or some priority mechanism to ensure memory write backs occur in timely fashion[8].

1.1.7 Hazards in the Pipeline

As noted previously, hazards that occur in the pipeline are one significant cause of reduced performance. They can cause stalls and empty pipeline stages which reduce the utilization of functional units, and thus slow down program execution. Hazards occur whenever an instruction cannot proceed through a pipeline stage at the expected rate. Henceforth pipeline hazards will be referred to as simply hazards.

Pipeline hazards may be divided into two groups, structural and dynamic. Structural hazards are *a priori* necessary for program execution on the hardware and usually known before an instruction is issued. These are usually delays that are required for proper execution based on how the program

[8] An exception to this rule is that under replay operation, it is possible to predict the latency of shared memory loads. Shared memory loads can only encounter BANK hazards which do not delay execution of the load, only signaling a replay must be done to grab all the data. A BANK hazard in stalling pipeline would cause the instruction to stall and wait until all the data is loaded on multiple cycles, meaning the latency of the operation is indeterminate.

is run. For example, an instruction's operands must be ready before the instruction can be issued for execution. If the supplying instruction is not executed far enough ahead in the program, it is known that the dependent instruction must wait (although the exact amount of time may be unpredictable). In the baseline architecture, instructions will not be issued from the scheduler until their operands are ready. Branch decisions are also structural hazards. In CPUs the effects of these structural types of hazards are reduced with out of order execution and branch prediction. In GPUs, these effects are reduced by the high amount of thread based parallelism. Threads encountering static hazards are not issued, instead other threads that are "ready" are issued. Although it is not guaranteed that there will always be ready threads, having many warps per core makes it much more likely.

Dynamic hazards are unpredictable and occur due to interactions with other warps in the core or uncertainties inherent with a particular instruction. These occur after an instruction has been issued.

Stall hazards are the simplest type of dynamic pipeline hazard. Stalls occur when an instruction cannot proceed to the next pipeline stage because that pipeline stage is not empty. Stalls are a collateral effect due to another hazard further on in the pipeline (if an instruction can't proceed for a non pipeline related issue, then it is classified as a different type of hazard). All stall hazards can be traced down the pipeline to an originating initial hazard. The initial hazard, if persistent, can cause long chains of stall hazards. Stall hazards can be the primary hazard, for example if a functional unit (usually an ALU) has waiting instructions as it processes a long latency instruction. Otherwise unlike the simple in order pipeline shown in Figure 1.1 modern architectures like the baseline architecture, have forks and joins in the pipeline, as in Figure 1.3. A stall might back up instructions past a fork and further decrease performance by choking off instructions that would otherwise bypass the initial hazard.

Figure 1.4 gives an example of how a stall might propagate through a processor with pipeline forks. Blue instructions are MEM instructions and green instructions are ALU instructions. In the initial view, no hazards are occurring and both ALU and MEM instructions obtain their operands in the operand collector and proceed to their respective pipelines. In the second view a hazard occurs for a MEM instruction, stalling that part of the pipeline. In the next view the stall has persisted and subsequent MEM instructions are unable to proceed, stalling themselves, eventually filling up into the operand collector itself. In the final view the operand collector is completely full of MEM instructions that cannot proceed, choking off instruction flow to the ALU which might otherwise have been able to do useful work.

The other dynamic hazard types originate independently of the pipeline. They prevent an instruction from proceeding to the next stage because of some condition that unexpectedly causes the instruction to be unable to complete its current stage. In the baseline architecture, these types of hazards all originate in the memory stage. Memory instructions can encounter a number of different conditions which prevent them from completing.

Figure 1.5 shows a close up of the hardware components of the memory stage. Explanation of

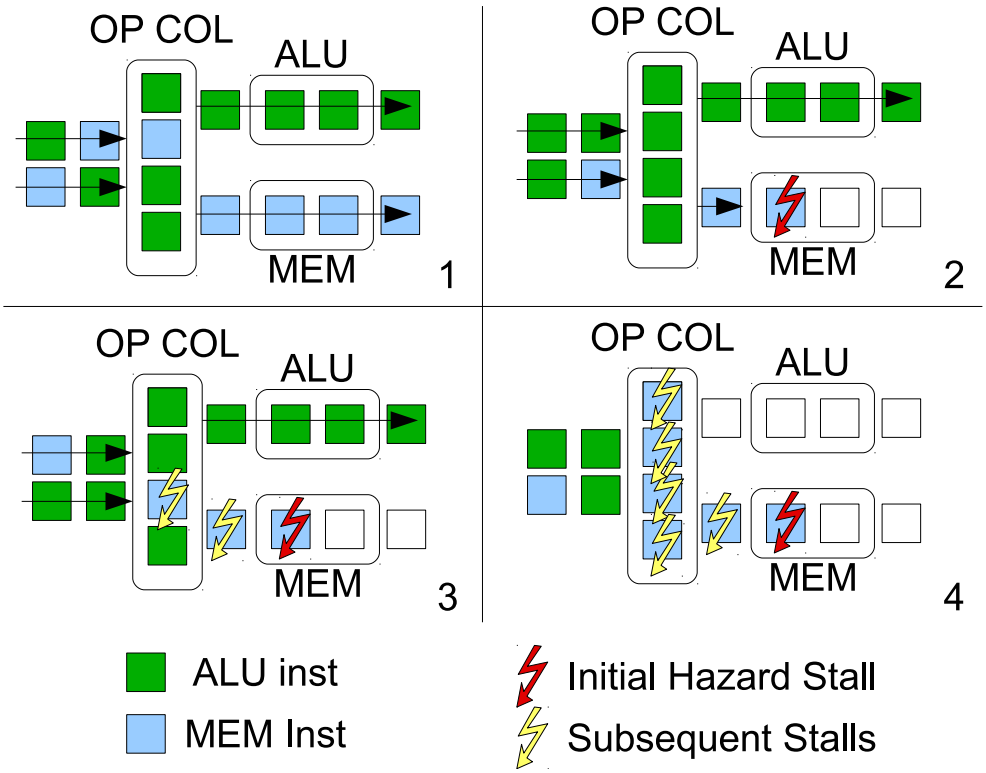

Figure 1.4: Example of stall propagation.

these components and their operation follows in relation to Figure 1.6.

Figure 1.6 is a flow chart of memory stage operation for a global memory read that demonstrates opportunities for hazards to occur. Threads in the memory instruction must first calculate their associated access addresses. These are then compared and compiled into a single large memory access. Thread accesses that do not fit into this access must wait and utilize the memory stage on another cycle. This constitutes a DIV (for divergence) hazard. The cache is then checked to see if it contains the data for the coalesced access. On a cache hit, the data is recovered and separated and copied into the corresponding individual thread's data registers. These threads then proceed on to a write back stage.

If the access misses in the cache, a memory request must be sent out of the core, over the interconnect to the memory system. In order to complete this task, an MSHR (Miss Status Holding Register) [19] in the MSHR table must be obtained. An MSHR holds information about the load such as which threads requested it, which parts of the load these threads needed, and where to write the results. A lack of MSHRs constitutes an MSHR hazard as the memory request cannot progress. In addition to an MSHR a memory request requires room to put the request before it is accepted by the interconnect. If the interconnect is full, and finite storage of additional buffers are also full, then the request can also not be made (COMQ hazard, for Communication Queue).

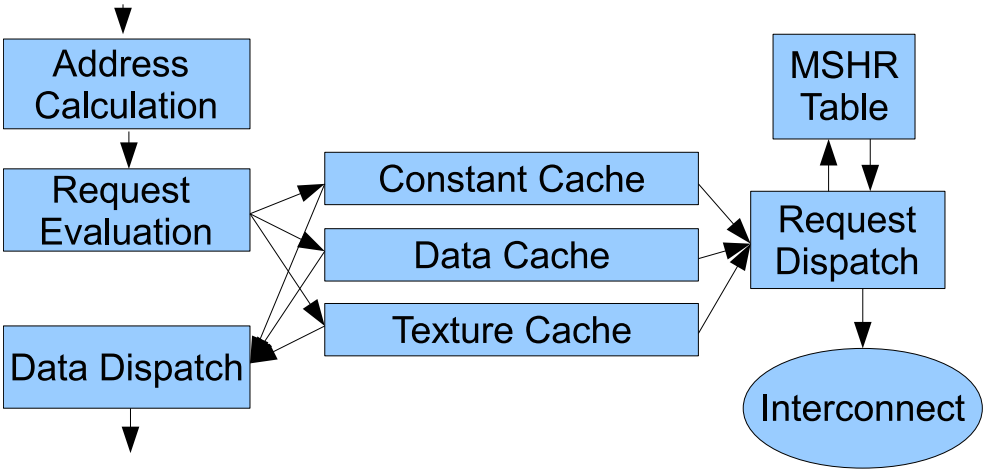

Figure 1.5: Memory Stage components.

Finally, in the read allocate, write back memory system, read accesses must reserve space in the cache when they are issued [9]. This requirement will fail if no cache line can be ejected, and this is termed a RSV (reservation) hazard. RSV hazards occur when all the valid lines that a request could be written in a cache are already reserved by other requests that have not yet returned.

If all these hazards are avoided the request is dispatched to the interconnect.

Shared memory operations do not require MSHRs, interconnect, or cache reservations and thus do not encounter these hazards. Similarly shared memory operations are not limited to accessing a single linear cache line each cycle, so do not encounter DIV hazards. However shared memory is implemented as a banked memory, and all threads in a shared memory instruction may not be able to access their data in a single cycle. For example two different threads may request different values from the same shared memory bank, in this case multiple cycles must be used to access these values. Some implementation may also be unable to broadcast a single value from a single bank to multiple threads, although in the baseline architecture used in this work this is allowed (choosing more aggressive shared memory design). This type of hazard is labeled a BANK hazard. Aside from the different cause conditions, it behaves in all other respects like a DIV hazard. The BANK hazard is the only hazard a shared memory operation may encounter.

The types of memory hazards are summarized as follows:

- **DIV** and **BANK**: The addresses accessed for each thread in a warp may not meet the conditions for completion in a single memory access. In this case the instruction must utilize the memory stage for additional cycles to perform the instruction for the remaining threads in the warp. **DIV** is used to signify hazards for constant, texture and global memory that can not be

[9] This requirement eliminates the possibility that deadlocks can occur due to the possibility of poor ordering of requests in interconnect buffers.

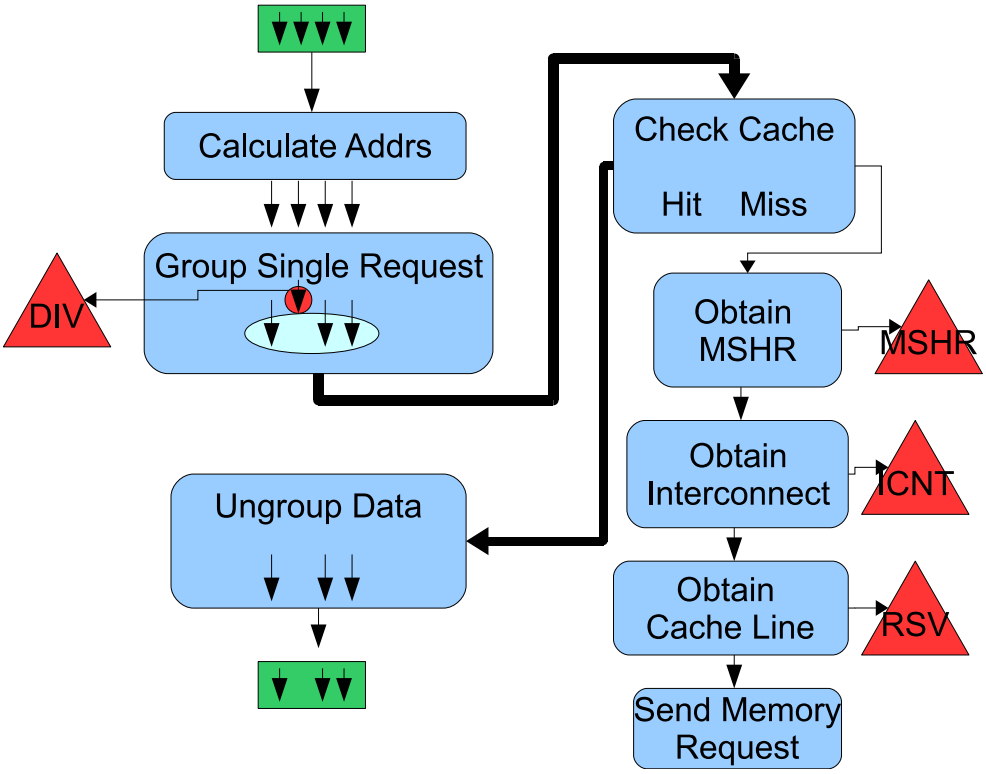

Figure 1.6: Memory Stage flow chart for a Global Memory load.

coalesced into a single memory request. **BANK** is the label for shared memory instructions that encounter bank conflicts. From the standpoint of thread behavior with this hazard, both operate in the same manner.

- **MSHR**: If a constant, texture, or global access misses in the cache then a Miss Status Holding Register (MSHR)is required as part of the request process. An MSHR keeps track of where the requested data should be stored when it returns, and also remembers which threads requested the data. If an MSHR is unavailable the request cannot be made this cycle.

- **COMQ**: If a constant, texture, or global access misses in the cache and an MSHR is available, then storage must be available for the request to enter the interconnect. In the architecture used the cache maintains an internal queue to account for clock differences and absorb fluctuations in the interconnect acceptance rate. If the interconnect is backed up long enough to fill up this queue then new requests cannot proceed, resulting in a **COMQ** hazard.

- **RSV**: If a constant, texture or global access misses in the cache and MSHR and COMQ are available, a line in the cache set must be reserved for the request (read allocate cache). If no line is available (ie. if all lines in the cache are pending reads) then the request cannot

be issued and a **RSV** hazard occurs. (RSV) hazards do not occur for the read only caches, texture and constant memory.

1.2 Contributions

This work provides the following contributions. The GPGPU simulator GPGPU-Sim was used in all contributions. Modifications were incorporated into the simulator, including implementing features similar to NVIDIA Fermi GPUs with replay, including dual schedulers and configurable operand collector and pipelines. Replay specifically was also analyzed and characterized to investigate what performance benefits it may have. Hazard prediction was proposed and implemented to correct pathologies that can occur in replay.

1.2.1 GPGPU-Sim Modifications

Changes and modifications were made to GPGPU-Sim [3]. The most significant were memory stage, pipeline and replay modifications. These are described in more detail in Section 4.3

1.2.2 Replay Analysis and Characterization

Replay represents an unorthodox approach to dealing with hazards in a pipeline. The NVIDIA Fermi [25] architecture uses replay. The performance implications of replay are not well understood. Replay could theoretically provide performance improvements in additions to implementation improvements. Replay performance was investigated to determine its value for GPGPU workloads. We characterize where it provides benefit, and identify weaknesses which cause poor performance.

1.2.3 Hazard Prediction

Replay although providing some benefit can sometimes reduce performance. Hazard prediction is proposed to alleviate deficiencies arising from replay. Hazard prediction involves two factors. The first is that instruction may or may not use resources that can cause hazards. The second is that those instructions that do require resources that cause hazards may or may not receive them. Hazard prediction is separated into two components, solving each these problems. A predictor anticipates whether a certain instruction will hit in the cache, and a tracker tracks resource usage. Combined, this hazard prediction improves performance.

1.3 Thesis Organization

Chapter 2 details previously published work related to replay and hazard prediction.

Chapter 3 provides a framework to understand how hazards affect performance and explores different solutions. Chapter 5 explains replay in detail and shows how it can be implemented. Similarly Chapter 6 explores hazard prediction implementation.

Chapter 4 details experimental setup and the benchmarks used. It also provides a summary of changes made to GPGPU-Sim.

Chapter 7 presents the experimental results and discusses their impact.

Conclusions are given in Chapter 8.

Chapter 2

Related Work

Instruction replay and stall free mechanisms have been proposed previously, and these are detailed in this chapter. Replay in GPUs is not represented in the literature, although it is very likely that the NVIDIA Fermi architecture [25] implements a form of instruction replay for least memory instructions [24]. The actual implementation of this system remains unknown, so it is not possible to compare to their implementation. The work presented in this thesis synthesizes a replay mechanism that may be similar to, but is designed independently from, the one implemented in Fermi, and analyzes its effects on performance.

Replay has been most closely examined in terms of monolithic CPU architectures. Cyclone [11] uses a replay mechanism for highly pipelined out-of-order CPUs. It uses a latency predictor and a switchback queue to replace a content addressable scheduler. If instructions are not ready to execute when they reach the front of the switchback queue, they are cycled to the back of the switchback queue. In the Cyclone architecture, instruction latency and dependence is determined speculatively. When an instruction is incorrectly scheduled (its predicted latency or dependence was wrong) then it is replayed at a later time as necessary. Cyclone uses its prediction based scheduler to remove the need for a large, cycle time sensitive, content addressable memory based scheduler. Instruction replay is used to ameliorate the problem of mispredictions made by the scheduler, by detecting when operands are not ready. Latency prediction allows the scheduler to reduce the time between when operands become ready and when they are used, even when the scheduler is many pipeline stages (and thus cycles) away from the execution units. Cache misses are a common reason for instruction latency mispredictions, causing replays of dependent instructions. In the GPU replay mechanism, replays require both a cache miss and a resource availability failure, and the memory instruction itself is replayed rather than its dependents.

The Cyclone architecture replay mechanism is useful for monolithic architectures where cycle time and instruction latency are important to performance and performance gains are gained from parallel execution of instructions from a single thread. In contrast, replay in a GPU architecture is

used to prevent pipeline stalls and thus increase instruction throughput of independent instructions. The implementation of replay differs between Cyclone and that for GPUs presented in this work. In Cyclone, replay occurs after a failed operand load, whereas in the GPU architecture, instruction dependence is enforced by the scheduler. In a GPU, instructions are not issued until their operands are ready, so the operand collector always has the correct values available. The Cyclone replay mechanism does not apply to a GPU because performance does not rely on low latency instruction execution: other independent warps may execute while one warp is blocked on a long latency instruction, or is waiting for its operands to be signaled as ready. Replay in the GPU differs also because it requires not only squashing and rescheduling but also management of warp threads. In the GPU replay implementation presented in this work, instructions are retained in the warp buffer and signaled to be reissued, rather than cycled around in a loop until executed.

Zephyr [20] is a modification to the cyclone scheduler. When Cyclone mispredicts a dependence or latency, that misprediction can caused additional mis-scheduled instructions that also must replay. These contingent replays decrease performance. Zephyr augments Cyclone's pre-scheduler with a load latency predictor and fixed length queues. This allows the pre-scheduler to introduce instructions out of order into the cyclone scheduler, and prevent it from becoming too full and congested. This reduces contention and replays in the cyclone scheduler by delaying entry of instructions that are not expected to complete in a short amount of time. The Zephyr latency predictor attempts to predict load latency, including whether a load will hit in the cache. The contention issues Zephyr addresses are similar to those that occur with GPU replay and motivate Hazard Prediction: too many replays clog the switchback queue / pipeline. While cache miss prediction is used in Zephyr, it is found to contribute to the problem of congestion. Moreover the Zephyr latency predictor depends on the observation that in the single threaded benchmarks they examined, loads are usually misses unless a store has recently written to the load's address. Hazard Prediction, in contrast, uses miss prediction in conjunction with a resource tracker to predict hazards and reduce replays, and does not rely on the same observation for cache hits. Although it is possible that the same observation could be useful for GPU programs, this behavior probably does not hold in general. GPUs have large caches where cache lines may remain valid longer, and different cache use patterns, where cache lines may be used by many threads over long time periods. The key contribution to limiting replays in Zephyr are the fixed latency queues which delay entry of long latency instructions into the scheduler. This is not directly applicable to reducing hazards in GPU replay, because excessive replays in the GPU architecture are not caused by early scheduling.

Kim and Lipasti [17] examine scalability of implementations of CPU replay. They find that replay schemes that correct speculative latency and dependence predictions do not scale well. They show that a prediction and token based system can decrease implementation complexity. The token mechanism limits the need for precise replay support by limiting which instructions are allowed to replay based on their predicted need to replay. The tokens are allocated to instructions which are

predicted to have a mispredicted latency or dependence. Dependence information is only retained for these instructions, allowing them to be replayed precisely and selectively. Instructions without a token lack dependence information and force an unselective issue queue flush. Since they focus on replay for scheduling purposes, their solution is not directly applicable to replay in GPUs. In Hazard Prediction, credits are allocated with the help of a predictor to maintain accurate estimates of resource availability. Both systems use prediction and tokens to ration core resources, although the prediction, resources, and mechanisms differ.

An architecture named "flea-flicker" [4] describes a two pass VLIW (Very Long Instruction Word) pipeline that executes out-of-order to recover from cache misses. Instructions that encounter hazards in the first pipeline complete in the second pipeline. This could be thought of as a replay-like mechanism in which the replay occurs in a separate pipeline, and only at most once per instruction.

The Stall Free Counterflow [22] architecture avoids stalls by utilizing a novel architecture where instructions and data values flow counter to each other down parallel pipelines. Instructions utilize the operands and proceed to execute once they obtain data values that match the operands they require. Stall free execution is achieved by wrapping the end of the instruction pipeline back to the start, effectively replaying instructions that failed to obtain operands or an execution unit. This replay system is part of a combined operand distribution and functional unit allocation system. Like the Cyclone [11] scheduler, replay is used to correct dependence and latency issues that might otherwise cause a stall.

Instruction replay can reduce or remove the need to stall. Other mechanisms exist for reducing problems with stalling. Hardware implementation issues with clock gating and pipeline stalling in hardware are described in [12]. They show that 70% of the power in high frequency microprocessors goes to clock circuity and latch load alone. The authors propose using an "elastic" pipelining technique for the extra storage needed for a delayed stall signal. This takes advantage of the storage available in a stalled master/slave flipflop. Detailed circuit implementations of this work are provided in [15]. Replay mechanisms that eliminate stalling would also likely avoid some of these circuit implementation problems, and may allow better and more efficient circuits designs. The "elastic" pipelining technique would not be necessary because there would be no delayed stall signal.

A wave propagation pipeline [6] forgoes pipeline registers by carefully designing the timing of circuits such that an instruction (or other signal that progresses as a unit) stays together as a "wave" throughout the pipeline. Such a system requires that signals never stall because there is no storage within the pipeline. As such it is usually proposed for ALUs and other small, non-stalling pipelines. Replay functionality could potentially open up more of the whole processor pipeline to this circuit technique.

Chapter 3

The Problem: Throughput Reduction Due to Hazards

For the purposes of this and following chapters, hazard is taken to refer to dynamic hazards (Section 1.1.7). These hazards prevent instructions from progressing down the pipeline and can lead to chains of stall hazards.

Hazards can and do restrict compute processor performance. They slow performance and reduce instruction throughput by reducing functional unit usage. Penalties can occur when functional units which could be in use are prevented from getting required resources by hazards elsewhere in the processor.

Hazards prevent a processor from maintaining the maximum throughput through each pipeline unit, given the limits imposed by the program that is being run. For the baseline architecture, two classes of instructions can interact negatively: arithmetic operations (ALU) and memory operations (MEM). These types of instructions have separate pipelines but only after the shared pipelines leading into the common operand collector. Hazards in the ALU or MEM pipelines can spill over through the operand collector and affect instructions of the other type (as shown in the example of Figure 1.4). Performance depends on both ALU and MEM instructions being executed concurrently to the full capacity of the processor, given availability. As noted before, the MEM instructions are the most significant source of hazards, and thus also can interfere with themselves.

To maximize performance (instructions per cycle) for a given architecture it is necessary to have the proper instructions available to fulfill those functional units' throughput, while at the same time, preventing hazards from also preventing functional unit use. This amounts to increasing the MEM/ALU parallelism in the processor core. Possible solutions to fixing this problem are analyzed in the following.

3.1 Taxonomy of the Solution Space

Increasing Memory / ALU parallelism is a complicated goal and in compute accelerators suffers from limitations and is amenable to certain solutions. The many-warp nature of compute accelerators makes these requirements much different from CPUs which primarily exploit memory / ALU parallelism with out of order scheduling, execution, speculation, and execute ahead options like prefetching. These monolithic CPU strategies provide the CPU core with a mix of memory and ALU instructions which can be executed concurrently. In these cases the amount of parallelism is limited by the single program thread (or a small number of threads) being run, in which many instructions in the out of order window will be dependent on each other, limiting maximum parallelism. In contrast, GPUs attempt to provide independent MEM / ALU instruction mixes by having many warps active on the same core. Because there are few[1] inherent restrictions on instructions from different warps, parallel performance depends only on access to instructions and how instructions progress through the pipelines and execution units.

3.2 ALU / MEM Instruction Mix

The first consideration towards high ALU/MEM parallelism, which maximally uses core resources, is access to instructions. Not only must enough total instructions be ready to execute every cycle, but both MEM and ALU ops must be ready to execute concurrently. For maximum performance gains, an instruction mix that matches functional unit throughput should occur for as much of the program execution as possible. Given a certain core configuration there will be an optimal ALU/MEM ratio that maximizes functional unit use assuming hazard free execution.

The instruction mix limitation specifically has to do with the mix of instructions at every moment in the program's execution. A necessary (but not sufficient) condition for a good mix at all times during execution is that the total dynamic instruction count of both MEM and ALU, the dynamic instruction mix, be as close to the optimal ratio (full functional unit throughput of the core) as possible. This ratio of these instruction counts (scheduling problems are discussed below) should match the throughput characteristics of the hardware for maximum performance improvement. For example, if a processor can complete two ALU instruction and one MEM instruction per cycle, then the optimal instruction mix of the benchmark will be two ALU operations per MEM operation. Available parallelism from a program (i.e., full functional unit occupation) will fall off increasingly for benchmarks that increasingly differ from this dynamic instruction mix ratio. Note that this limitation is between specific programs and specific hardware configurations. Different programs may be optimal on different hardware. Figure 3.1 plots the time fraction that a program would fully occupy functional units (i.e., maximum parallelism) as a function of total dynamic instruction mix

[1] A few instructions allow execution to coordinate between different active threads in CUDA. These include atomic operations and memory barriers (which serialize access) and __syncthreads() (which synchronizes the program counter across a block).

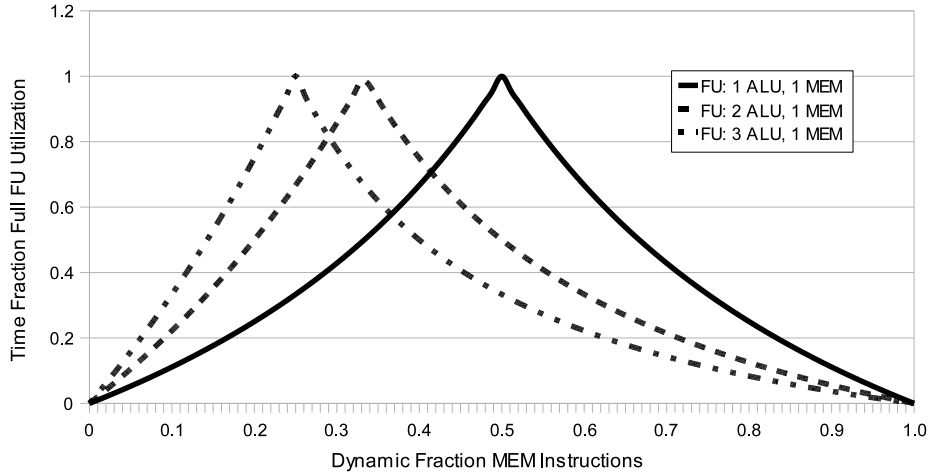

Figure 3.1: Full functional unit occupation for different dynamic instruction mix ratios. Each ALU or MEM functional unit has a single instruction per cycle throughput.

ratios (MEM instructions / MEM + ALU instructions). It is clear from the figure that available maximum parallelism drops off steeply from the optimal.

The total dynamic instruction count mix is not sufficient to ensure a good mix during the execution of the program. For example, even though a program may have an optimal dynamic instruction count mix, all warps might execute in lock step, causing at any moment instructions to be either all MEM or all ALU. In addition programs can change instruction mix between different phases of execution. There may be phases with lots of MEM operations, then phases with only ALU operations. A common programming model for CUDA programs, called the bulk synchronous model, specifically requires phases with different behaviors. In this type of program, data is loaded from global sources at the beginning of the kernel, operated on locally in the middle, and written back to global locations at the end, creating at least three different phases of execution. A program run strictly in this way will have at least one phase that does not have an optimal instruction mix, even if the total program does in aggregate. Often strict bulk synchronous operation is not necessary (or need not be provided program wide, for example only being required at the level of thread blocks). In this case a scheduling of groups from threads in different phases can provide the average instruction ratio for much of the program's execution.

Scheduling is an important component of creating a good warp mix. An optimal scheduling is an NP-hard problem for arbitrary programs [1]. Currently published solutions have been heuristic in nature [23].

One simple strategy to improve instruction mix is to increase the number of warps per core, thus increasing the chance there will be warps in the correct phase of execution to satisfy optimal loading of the core. The downside of this change however is an increase in front end resources required, for

example in the scheduler. The increased scheduler resources could possibly be handled in multiple ways, for example, the baseline architecture can use dual schedulers as in the Fermi architecture. This strategy can be enough to break up warps into different phases. Because the memory stage has a limited number of MSHRs, it can only concurrently support memory access for a fraction of the warps on the core. Thus a small set of warps are able to complete a memory instruction (in simple programs a memory instruction could constitute the entirety of a memory phase) and progress, while other warps are unable to progress. This effectively staggers execution of the small groups of warps, giving a selection of threads in different phases at any particular time. The downside of this scheme is that the process is generally unpredictable and unstable.[2] Eventually all warps could become unsynchronized. Although this is good for instruction mix, advantages related to cohesive warp execution (where similarly numbered warps execute close together) are lost. Since the data for close warps often shares cache lines (data locality), this can increase cache misses as accesses no longer happen together. Another downside is that kernels with explicit synchronization will tend to reset threads into synchronous execution of phases, as well as enforce ramp up and ramp down time (where thread groups wait to begin their staggered start and then must wait idly while other threads finish their staggered end phases).

Another scheduling heuristic is two level scheduling [23]. In this scheme warps are grouped together to execute synchronously, while the warp groups are encouraged to execute in different phases. One advantage here is that the warp groups are determined at the core architecture level and are balanced for the core.

3.3 MEM / ALU Parallelism in the Front End

The front end is comprised of pipeline stages and associated resources that mostly determine instruction control and selection, largely independent of instruction data. Because the front end treats most instructions the same regardless of MEM or ALU type, processing instructions of both types is inherently parallel. There are some scheduling decisions arbitrating between instruction selection, which could include criteria based on instruction type. This is most apparent in the scheduling policy as described above. Other scheduling type decisions occur when within the pipeline more instructions are available to enter into a pipeline stage than are able to occupy it. These decisions can be taken with a view towards maintaining a good instruction mix.

3.4 MEM / ALU Parallelism in the Back End

Parallelism in the back end can be approached broadly in two ways: decreasing MEM instruction / ALU instruction interaction via reducing hazard penalty, or reducing the number of hazards, or

[2] It is possible to create a program to explicitly enforce this staggered mode of execution, however it would amount to hardware specific tuning.

both.

3.4.1 Decreasing MEM / ALU Interaction

Opportunities to decrease MEM / ALU interaction are limited. Separate pipelines can partially remove interaction. In the baseline architecture the operand collector represents a point of MEM / ALU contention.

3.4.1.1 Multiple Pipelines

Modern processor designs separate instructions into different, non-interacting pipelines. Separate memory and arithmetic pipelines are important because of their different behavior characteristics. This is in contrast with early computers which dealt with all instruction types in the same pipeline.

Ideally different types of instructions would be separated for the entire length of the pipeline, however, scheduling and operand collection (and write back) are not easily amenable to separation based on instruction type since both types of instructions utilize and access these resources. The scheduler is warp based, and the operand collector needs to manage access to the same operand banks. Both these tasks are independent of the instruction type. The scheduler may make decisions based on instruction type, but no further separation seems useful here.

Since the front end is not amenable to separation, then in addition to separating the functional unit pipelines, it may be necessary to widen the front end pipeline to maintain equality of throughput (if one of the ends has lower maximum throughput, resources are likely being wasted in the other).

Increasing the pipeline width may also help in addition to higher throughput by allowing more ways for ALU and MEM instructions to sidestep each other as the progress down the pipeline. Increasing the width of pipeline register paths also provides some buffering capacity storage. Stall hazards must fill up twice the capacity to stall as far back up the pipeline. This improvement may be mitigated by the increased throughput, which fills registers faster.

3.4.1.2 Operand Collector

The operand collector may become a bottleneck for instructions of all types if some instructions encounter stalls further down the pipeline and pile up (as in Figure 1.4). Other than scheduling, the operand collector is the only other mandatory place of MEM/ALU instruction interaction. Stalls backing up through the operand collector are the only cause of MEM/ALU interaction in the baseline architecture.

Operand collector spaces could be logically separated based on which sort of instructions they can hold, bypassing blocked pipelines. This reserves space for instructions even if there are none available.[3] This configuration is called “separated” in contrast to the default, “unified”, which

[3]Just giving preference to certain instruction types in each slot will not solve the problem because if a non preferred

does not reserve space for specific instruction types. The separated operand collector solution does present a problem though in terms of increased operand collector requirements (such as cycle time which is sensitive to increased size) or reduced performance when the instruction mix does not match the designed optimal mix. For example a kernel could have two phases, one phase with throughput of 1 MEM + 2 ALU per cycle, and another phase with 3 ALU per cycle. To support full throughput (assuming each slot provides a throughput of 1 instruction per cycle) minimal unified operand collector would require 3 slots for both phases, but a separated operand collector would require 4 (1 MEM + 3 ALU slots). Increased latencies per slot, as is common for operand collectors (instruction may have to wait for operands), multiply slot requirements to maintain throughput.

A unified operand collector could be monitored to measure stalling that fills up the collector and which exiting pipeline (MEM or ALU) is causing the stalls. Adaptively this could then signal the scheduler to not schedule these type of instructions until the conditions clear. Although more dynamic and subject to tuning and policy this solution is similar to a separated operand collector in requiring greater operand collector size. In this case the size must be larger to tolerate the signaling delay to the scheduler, and it must also, in organization, be similar to a separated operand collector because the system must halt scheduling of hazarding instructions before the operand collector is completely full. In spite of these drawbacks this method enables complex control policies that could be tailored to specifically affect performance only when they improve performance (for example by throttling MEM instructions only in certain cases). Examination of this form of operand collector is left for future work.

3.4.2 Decrease Hazards or Their Effect

Hazards that occur in the pipeline are caused by resource problems that prevent the processing of instructions at normal throughput. These are most common in the memory system, which requires a diverse assortment of resources to function. The ALU pipelines may also encounter hazards if the injected instruction load exceeds the throughput available for that load or exceeds the write back resource requirements.

Hazards slow down the execution of instructions in a pipeline. In a typical "stalling" architecture, instructions remain at the hazard point until the hazard is resolved and they are able to continue. These instructions are stalled, and if any instructions are behind them in the pipeline these become stalled as well.

In addition to reducing the throughput of a pipeline, if the hazard persists for long enough, stalled instructions will back up into a unified operand collector that services all pipelines. As the hazard persists further the operand collector will become full and be unable to service new instructions at appropriate throughput, and eventually none at all. Because of the usually long latency of memory hazards, this type of stalling eventually chokes off instructions from the ALU

instruction happens to be in that slot when a stall hazard occurs, then the preference is of no use.

pipelines. This effectively prevents concurrent execution of memory and ALU during memory hazards (although the new memory instructions cannot proceed, memory operation is being done in parallel as evidenced by the resources being unavailable servicing other requests).

Solutions to reducing hazards or their effects presented here focus on reducing the effect of memory hazards. ALU related hazards are not as significant if throughput of the front end matches the supported throughput of the back end ALUs. In this case stalls persist only for a small time and do not back up very far.

Two methods of reducing hazards or their effect are examined. Replay removes the stalling penalty from long latency memory hazards and Hazard Prediction attempts to predict which instructions will cause memory hazards at the scheduler and avoid issuing them entirely.

3.4.2.1 Replay

Replay as described in more detail in Chapter 5 provides many benefits. Replay in particular reduces the throughput penalty of the hazards directly. Instructions that encounter hazards do not remain in the core stalling while waiting for the hazard to clear, they give a chance for other instructions to continue through. Thus throughput may be increased, although many subsequent instructions may also require the same lacking resource and also encounter a hazard. In this case the throughput is still reduced (by using cycles and resources to process hazarding instructions) but useful work may still proceed. In some cases throughput is improved because a resource required by one hazarding memory instruction is not required by other following instructions. In particular memory instructions that will hit in the cache will never encounter a hazard. In addition, RSV failures only affect accesses to one set in the cache, memory instructions that miss in the cache that map to a different set may be able to progress. In both these cases replay allows these instructions to proceed through the memory stage in spite of other memory instructions that encounter hazards and would otherwise stall the pipeline.

The lack of stalling also completely prevents stalled instructions backing up into the operand collector. Therefore direct starvation of ALUs due to memory hazards is avoided.

Replay has associated costs. Each hazard necessitates a replay, which requires front end resources to reissue the instruction, and back end resources to reprocess it. Excessive amounts of replay could use up bandwidth otherwise available to productive ALU instructions. This would only happen for programs with high amounts of hazards and ALU MEM parallelism. In addition, replaying an instruction causes increased latency for that instruction before it is executed. If a program cannot tolerate this additional latency, performance may be affected.

3.4.2.2 Hazard Prediction

Another solution is to predict memory hazards that an instruction might encounter at issue, and avoid scheduling those instructions likely to encounter hazards. This could avoid throughput issues

of hazards altogether. However the predictive nature of hazard prediction can make this a risky method. If an instruction is mispredicted in an optimistic fashion, it would encounter a hazard and possibly cause stalling and ALU/MEM interference. If on the other hand, an instruction was mispredicted pessimistically, it may be delayed from execution unnecessarily. Both situations would retard performance.

When combined with replay this approach may become beneficial. Instructions that unexpectedly encounter a hazard will replay, avoiding pipeline stalls, and replays that are known to require unavailable resources can be avoided for scheduling, avoiding excessive amounts of replay. In this case it is still very important to avoid pessimistic mispredictions (predicting an instruction will encounter a hazard when in actuality it would not), as these might retard execution by not issuing hazard free instructions.

3.5 Estimating Performance Loss Due To Hazards

Performance loss due to hazards is a function of both the architecture configuration and the behavior of the running program. A method for estimating performance loss due to ALU/MEM interference hazards is presented here. Predicted maximum speedup attempts to estimate the performance improvement of mechanisms which remove the stalling penalty from hazards, such as replay or hazard prediction or both.

A first order estimation of posible performance improvement could consider the dynamic MEM / ALU fraction of the benchmark. This fraction shows how much total ALU/MEM parallelism is available in the program as per Figure 3.1. The baseline architecture has two ALUs, one SFU, and one MEM unit. This places the 100% parallel peak at a dynamic memory instruction fraction between 0.3 and 0.25. This however is imprecise because it assumes perfect single cycle throughput for all functional units. In particular the memory unit will likely have much less than single cycle throughput. In that respect, full parallelism could be obtained at dynamic memory instruction fractions much greater than 0.3, depending on the benchmark. The dynamic memory instruction fraction is shown in Figure 3.2. The benchmarks are introduced in Section 4.1.

Possibilities for improved performance are also dependent on the number of hazards encountered by a program. If the program has a large fraction of hazards occurring then it is likely to benefit from performance improvement schemes. Figure 3.3 shows the fraction of execution cycles in which hazards of different types occur for each benchmark in the baseline architecture. Although not a direct predictor of performance, both ALU/MEM instruction mix and the effects of hazard rates are included in the calculation of predicted maximum speedup.

Predicted maximum speedup is calculated by using dynamic runtime ALU and MEM instruction counts as well as the nuber of cycles the operand collector was blocked due to stalling (thus taking account of the real impact of hazards). It assumes that every cycle in which the operand collector was blocked (assuming it was blocked by a MEM hazard) could have instead been used to speed up

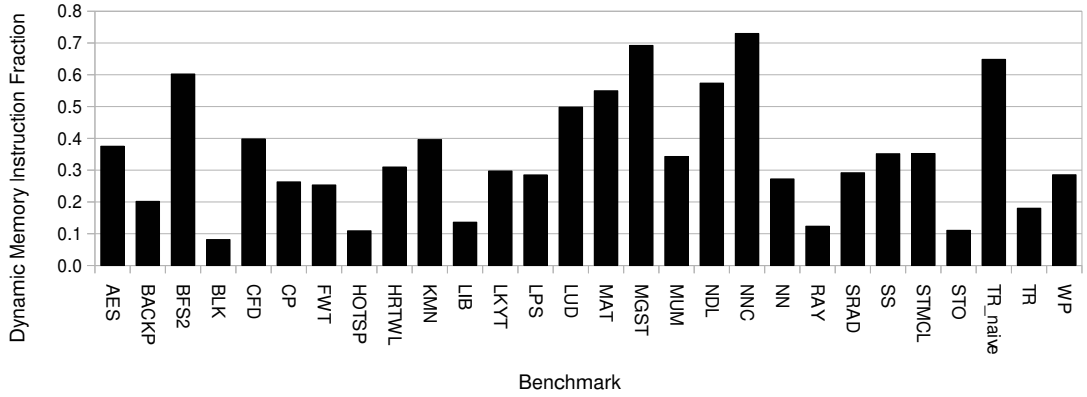

Figure 3.2: Dynamic memory instruction fraction for each benchmark.

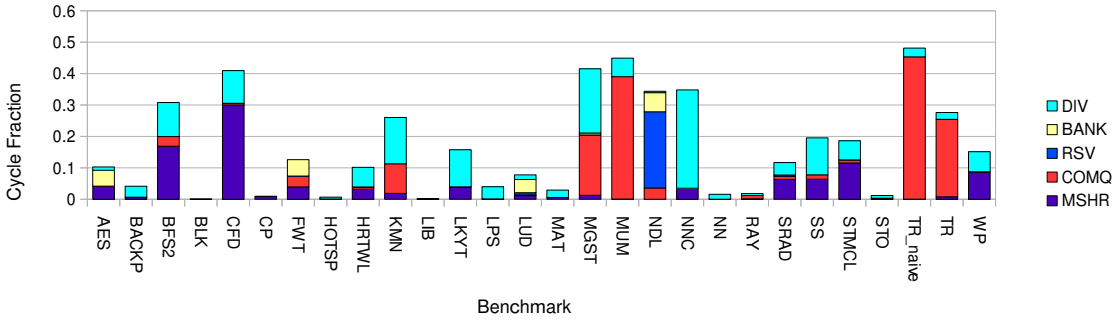

Figure 3.3: Hazard cycle fraction for each benchmark.

the program by executing ALU instructions earlier than it did. No more ALU instructions can be executed earlier than there are ALU instructions in the program (thus taking account of MEM/ALU mix). Thus execution time is shortened by the minimum of blocked cycles and the total number of ALU instructions divided by the full rate they can be executed:

$$new\_cycles = old\_cycles - min(blocked\_cycles, \frac{ALU\_inst}{ALU\_rate})$$

$$speed\_up = \frac{new\_cycles}{old\_cycles} - 1$$

Figure 3.4 shows the predicted maximum speedup for the benchmarks in an architecture with 2 ALUs, 1 SFU, and 1 MEM unit.

The predicted maximum speedup is a very optimistic strong maximum of speedup by preventing interference (it is not absolute because other mechanisms may allow higher speedups in practice, such as higher cache hit rates). For most programs, ALU instructions may not be available to

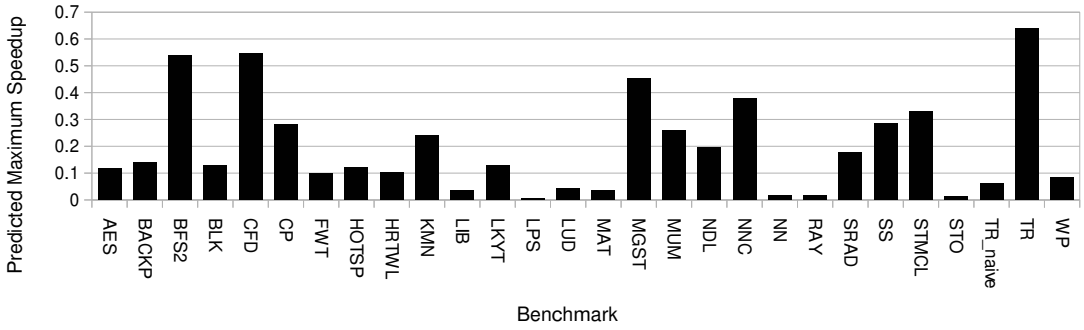

Figure 3.4: Predicted maximum speedup for each benchmark.

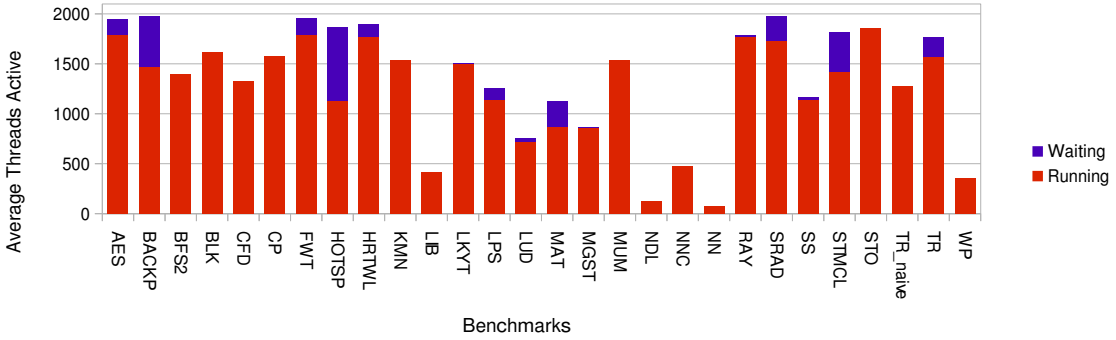

Figure 3.5: Average thread availability.

fill avoided stall cycles. This can be due to instruction dependence, imperfect scheduling between threads, and explicit dependence such as barriers, short kernels and atomic operations. These effects are not accounted for in the predicted maximum speedup estimation.

Availability of instructions can be estimated by measuring the issuable warp availability during execution. This is calculated by measuring the average number of active warps per cycle over the entire execution of a benchmark. Higher numbers represent higher availability for scheduling and more opportunities for hazard avoidance mechanisms. This value is presented for each benchmark in Figure 3.5. Some benchmarks have fewer instructions available to execute due to synchronization requirements in the program. Counts of these are also shown as "Waiting". Note that for the baseline architecture, 2048 threads = 64 warps is the maximum hardware limit.

The hazard prediction presented here specifically targets MSHR hazards (although it could be easily adapted to the other types of hazards). In this respect performance improvement would only be expected for benchmarks with a large fraction of MSHR hazards.

Chapter 4

Methodology

To evaluate architecture design choices GPGPU-Sim[3] was used and modified to model compute accelerator execution. GPGPU-Sim is a cycle accurate timing simulator which runs compiled CUDA binaries by simulating architectures similar to NVIDIA GPUs. GPGPU-Sim is highly configurable and, being written in C++, is highly modifiable.

GPGPU-Sim was significantly modified to support multiple pipelines, replay functionality, and hazard prediction, as well as numerous gathered statistics.

4.1 Benchmarks

Numerous benchmarks were used in evaluating the performance of architectural changes. The GPGPU benchmarks are from numerous sources and cover a broad range of programs and program types. These are summarized in Table 4.1.

The benchmarks are listed in Table 4.1. They are classified according to behavior. Benchmarks with high maximum predicted speedup, high average thread activity, and high number of hazards would be expected to perform the best with replay. Benchmarks with a large number of MSHR hazards would further be expected to benefit from hazard prediction targeting MSHR hazards.

4.2 Baseline Architecture Configuration

The baseline architecture configuration was chosen to simulate an aggressive compute accelerator as might be built in the next few years. Configuration parameters are given in Table 4.2.

4.3 Modifications to GPGPU-Sim

This section describes the major modifications to GPGPU-Sim.

Abr.	Max.Pred.Spedup	Occ.	Hazards	MSHR hazard	Desc.
AES			low	low	AES cypher[21]
BACKP					Back Propagation[7]
BFS2	high		high	high	BFS Graph Traversal[7]
BLK					BlackScholes[26]
CFD	high		high	high	Redundant Flux Computation[7]
CP	medium				Coulomb Potential[3]
FWT			low	low	Fast Walsh Transform[26]
HOTSP		medium			Hotspot[7]
HRTWL			low	low	Heart Wall Detection[7]
KMN	medium		high		K Means Kmeans Clustering[7]
LIB		low			LIBOR[3]
LKYT			low	low	Leukocyte[7]
LPS		medium			3D Laplace Solver[3]
LUD		low			LU Decomposition[7]
MAT		medium			Matrix Multiply[26]
MGST	high	low	high		Merge Sort[7]
MUM	medium		high		MUMMER-GPU[3]
NDL	medium	low	high		Dynamic Programming[7]
NNC	high	low	high	low	NN_cuda[7]
NN		low			Neural Network[3]
RAY					Ray Tracing[2]
SRAD	medium		low	low	Image Processing[7]
SS	medium	medium	low	low	Web Document Clustering[7]
STMCL	medium		low	low	Data Mining[7]
STO					StoreGPU[3]
TR			high		Matrix Transpose[26]
TR_naive	high		high		Matrix Transpose (naive version)[26]
WP		low	low	low	Weather Prediction[3]

Table 4.1: Benchmark abbreviations, classifications, descriptions and sources. Benchmarks are classified qualitatively (missing qualifications default to the most common). Max.Pred.Speedup: Maximum Predicted Speedup (default: low). Occ.: Average Thread Activity (default: high). Hazards: Amount of hazards (default: very few). MSHR Hazard: Amount of MSHR hazards (default: very few).

Property	Value
Number of Cores	10
Number of Memory Controllers	6
Interconnect	Crossbar
Clocks (MHz)	Core 700, Interconnect+L2 1400, Memory 1800
Warps Per Core	64 (2048 Threads)
Shared Mem Size	26k
L1 Data Cache (per Core)	sets:64, line size:128, assoc.:6, 32 MSHRs
L1 Constant Cache (per Core)	sets:64, line size:64, assoc.:2, 32 MSHRs
L1 Texture Cache (per Core)	sets:8, line size:32, assoc.:20
L2 Combined Cache (per M.C.)	sets:64, line size:32, 8 way
Schedulers (per Core)	2
Per Warp Instruction Buffer Size	8
Operand Collector	8 slots (generic)
Functional Units (per Core)	2 ALU, 1 SFU, 1 MEM
Hazard Handling	Stalling

Table 4.2: Baseline architecture configuration.

4.3.1 Configurable Pipeline

Code was introduced to make the core pipeline highly configurable. The number of schedulers was changed to be variable. In addition the operand collector was adjusted to be configurable as to the type and number of instruction slots. To support the higher rate of instructions from multiple schedulers and larger operand collectors, all the pipeline registers were given configurable width.

4.3.2 Memory System

The core memory system was restructured to improve reliability and readability. This included merging similar code between memory interfaces. Design approaches of these modifications were carried into GPGPU-Sim 3.0.

4.3.3 Replay

Instruction replay was implemented in the simulator. Replay was implemented so that it could apply to any type of instruction, although in this work only memory instructions are replayable. In addition to replay functionality (as described in Chapter 5), tracking and debugging utilities were included, most useful of which was a circular buffer that is used to track the replay history of each warp. This was used to verify the implementation.

Of particular import to the implementation of replay, the method of tracking instructions as they move through the pipeline was modified. In the standard version of GPGPU-Sim, these are tracked through the pipeline with pure pointers. Since instructions are only issued once they can

be easily allocated at fetch and deallocated at completion. Advancing an instruction down the pipeline is simply copying the pointer. However in replay, the same instruction may be issued multiple times, and may be held in different places at the same time (for example, in both the warp buffer and somewhere in the pipeline). To address this, a smart pointer class was created to manage the complexities of replay. This class explicitly manages warp instructions and only allows operationally correct operations on them.

4.3.4 Hazard Prediction

Hazard prediction was also implemented in the simulator. Chapter 6 details hazard prediction operation. Although this work only examines hazard prediction in relation to MSHR hazards, the implementation includes infrastructure that can be adapted to easily track and utilize prediction of any other type of Hazard.

Chapter 5

Replay

The use of replay is motivated as a method of reducing hazard interaction with other warps in the pipeline and consequently increasing GPGPU performance. Replay also has other advantages. These are described below, followed by the details of replay implementation.

As individual cores become larger and faster it becomes more important to reduce pipeline stalling. In a SIMT pipeline most unexpected events center around the unpredictable nature of memory operations. Memory instructions can fail to acquire an MSHR, or interconnect buffer space fails to push a request (COMQ resource failure), or the memory stage may not be able to service all threads in a warp (a memory divergence (DIV) failure). Additionally, memory requests may fail because they cannot acquire a free line in the cache (a RSV failure) or there may be more TLB misses than can be handled by the hardware (a TLB failure). In contrast conventional superscalar processors do not face these issues as strongly because sufficient resources are available to tolerate the expected number of hazards and allow ALU operations to execute out of order. For memory intensive SIMT benchmarks it is not possible to provide enough resources to handle even a small fraction of the hazard load. For example on a core with 32 active warps of 32 threads, each thread may load a different address (non coalesable) from memory in a short amount of time, producing 1024 different memory requests. Typically MSHR resources available are much less than this.

5.1 Other Motivations for Replay

Stalling presents particular difficulties at the hardware circuit level. At high clock frequencies it becomes impossible to stall an entire pipeline in a single clock cycle. At the current scale of modern processors it takes more than a single cycle time to send a signal across a compute core. For instance, the signal distance on single stream multipliers on an NVIDIA Fermi core from its center to its edge is about 4mm, which in a 65nm process signal travel time is 10mm in four clock cycles, which gives at least two cycles to route a stall from the center across the whole core [10]. Pessimistically if the stall cannot be initiated in the center, traversing the whole core would take the entire four cycles. If

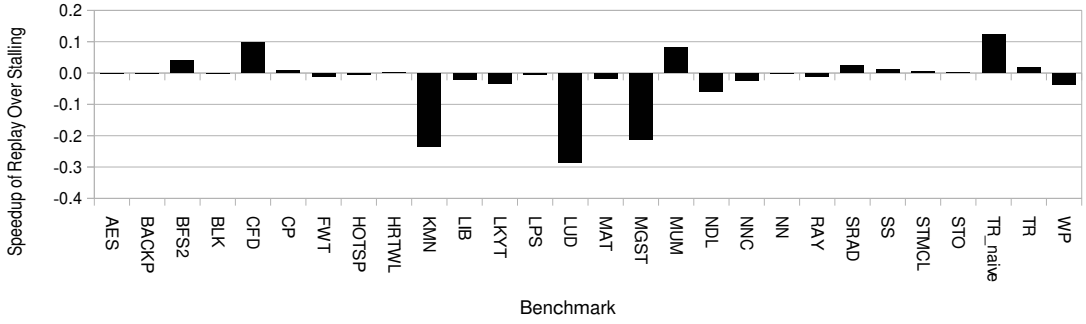

Figure 5.1: Comparison of replay strategy speedup over stalling.

one still wants to use stalling, additional buffers must be added to store the pipeline registers that pile up while the stall signal propagates and progressively stalls the pipeline. This however would incur a large area penalty. For example, additional buffering in front of the memory stage in a 32 wide SIMT architecture would need to buffer 32 x 64 bits = 2k bits of memory addresses per cycle of storage required. Other pipeline registers would need to be buffered as well.

In modern SIMT architectures, design considerations for power, performance and area have produced complicated pipeline implementations. The NVIDIA Fermi architecture schedules 48 warps into two pipelines that share a memory port, and has two schedulers which schedule mutually exclusive sets of warps [25]. This likely reduces area and cycle time as complex scheduling operations can consider fewer warps at at time. Content Addressable Memory (CAM) can be used in warp selection (NVIDIA's actual implementation is unknown) and the size and cycle time of CAM structures is sensitive to the number of entries. Implementing stalling in this configuration would be complicated by needing to stall both pipeline paths and have storage for the data that piles up from each. In addition, stalling caused by one scheduling stream or pipeline would cause both pipelines to stall.

Stalling the pipeline for MSHR, COMQ, TLB and RSV failures prevents useful work from entering the pipeline. TLB failures, although rare, could block the memory stage for hundreds of cycles. Even if ALU operations can bypass the stalled Memory stage, a second memory instruction will cause the whole pipeline to necessarily stall. DIV failures do not have this problem because only a subset of the threads of a warp suffer this failure, and the rest successfully complete the pipeline stage. In addition to allowing ALU operations to proceed by not stalling the pipeline, replaying TLB failures allows memory instructions that do not miss in the TLB to proceed. Figure 5.1 shows the performance of replay over stalling. It can be seen that memory intensive benchmarks gain performance.

As compute architectures go forward it is predicted that compute resources will grow faster than memory bandwidth [10]. This increased demand for memory will increase contention for limited

memory resources such as MSHRs. Other architectural improvements, such as allowing multiple outstanding non dependent loads from threads via stall-on-use [8] could further increase pressure on memory resources and MSHRs.

By replaying instructions that encounter hazards, these problems can be avoided. A SIMT based architecture with replay as described in Section 5.2 has been implemented in this work.

In addition, once replay is supported it becomes very simple to deal with any sort of hazard, especially corner case conditions where performance is not an issue. Such corner case conditions may include rare resource contentions and hardware related constraints such as circuit timing issues. Even rare timing faults or data faults at the circuit level might be rectified in this way. To handle any of these issues hazard conditions need only be detected and the offending instruction squashed and set for replay.

Instruction replay has been used in highly pipelined superscalar processors to deal with instructions that have been scheduled but cannot obtain their operands [11][20][17]. Although the idea is the same, replay treats fundamentally different problems in each architecture. In a SIMT pipeline, memory hazards would otherwise cause the pipeline to stall. In a superscalar pipeline, memory hazards are handled by the out-of-order nature of the processor, and instructions that depend on these for operands are replayed. Moreover, the size of the problem is larger for a SIMT machine because many more threads compete (at least 32 for current GPUs) for resources in a pipeline, and all are stalled if even one thread's resource needs are not met.

5.2 Baseline Replay Implementation

The following describes how the baseline architecture is modified to form the baseline replay architecture.

5.2.1 Fundamental Operation of Replay

As in previous CPU designs which use replay, instructions that cannot complete their way through the pipeline are removed (squashed) to prevent stalling. The scheduler replays these squashed instructions. The scheduler may reschedule these instructions based on knowledge or estimations of when the instruction may proceed, or may be scheduled as normal along with other instructions.

5.2.2 Overview

In contrast to CPU based replay, replay on compute architectures serves a fundamentally different purpose. Compute accelerators require a way to deal with memory instructions that cannot complete in a single cycle, whereas CPUs can utilize replay to avoid processing instructions that lack ready registers [11]. For CPUs, instructions are squashed relatively early in the pipeline, before execute,

whereas for compute accelerators, instructions are squashed deeper in the pipeline, at the memory stage.

5.2.3 Scheduling and Front End

As with the baseline architecture, the scheduling logic utilizes the data stored in the instruction buffer. Replay leverages the instruction buffer structure to coordinate the information necessary for replay. The warp buffer keeps track of whether instructions in a warp are issuable, whether they should be issued (possibly for a replay) as well as per instruction thread information. Instruction, per thread, handling is described in Subsection 5.2.4.

In the instruction buffer, each warp has a segment of storage, which holds a number of instructions (the control information thereof), as well as other per warp information. The instructions in the line form a circular buffer, where new instructions can be inserted after older instructions have finished and been removed. The warp maintains instruction order with an issue pointer that points to which instruction the scheduler will consider for execution, and advances once that instruction has been issued. In addition there is a fill pointer, indicating which instruction slot is free to be filled, which is advanced when instructions are added from the instruction memory (possibly many at a time). The fill pointer cannot advance past the issue pointer, so an instruction fill is never performed if this would occur.

In the baseline architecture, instructions are issued in order from the the instruction storage in the warp buffer data line, as indicated by the issue pointer. When issued, instructions will always complete (if they encounter a hazard, they will stall until the hazard clear), so may be immediately freed from the instruction circular buffer as the issue pointer advances.

In a replay architecture, instructions that might replay must be retained in the instruction buffer as they are not guaranteed to complete when issued. These replayable instructions remain active until they are signaled to have completed, and their entry can be freed. To ensure this is maintained correctly there is an issue tail pointer, which points to the entry before the oldest replayable instruction that has not completed and thus may still replay. The fill logic may not fill past this point. This creates a region of instructions in which some instructions may be replayable and issue again. When an instruction is signaled to be replayed, the selection logic for that warp only presents the oldest replay signaled instruction to be selected by the scheduler for issue. In this way forward progress is assured.

Figure 5.2 shows the data organization in the modified replay warp buffer. An example of warp buffer operation is presented in Section 5.3.

Stall on use functionality is not affected by replay and proceeds as normal.

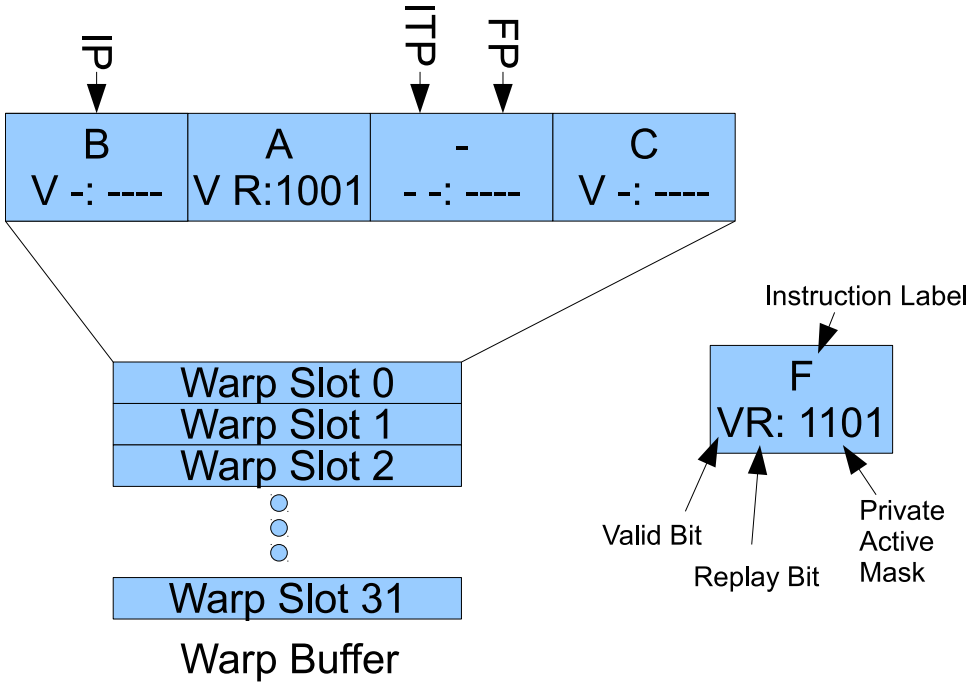

Figure 5.2: Warp Buffer data organization.

5.2.4 Replay with Threads

Orchestrating replay in compute accelerators is more complicated than squashing and replaying all the threads in a warp at the same time. Often only some threads in the warp cannot proceed. Bank conflicts and memory divergence (BANK and DIV hazards) cause only some threads to be processed at the same time. These individual sets of threads in the instructions can complete, or encounter a hazard and need to be replayed independently. Thus the issuing logic needs to keep track of which instructions have been completed, so they are not issued again for execution (multiple execution of the same instruction is inefficient and could break program consistency).

The instruction buffer retains a field of one bit per thread (32) per instruction slot. This bit field is the private active mask (PAM). When an instruction is first issued (at the issue pointer) the private active mask is initialized to the current active mask for that instruction. Recall that the active mask is a bit mask which indicates which threads in a warp should be issued; threads may be excluded from execution because of branch divergence or because the number of threads being run in a block is not an exact multiple of the warp size (thus such a warp will only be partially filled). This allows the instruction to use the correct mask at issue and thereafter since the active mask may change as newer non-dependent instructions are issued from the warp. The instruction needs to base future replays on that initial mask. This fully takes account of branch divergence, as the instruction completes the active mask appropriate to its branch. This method can also handle predication if the result of the predication test is returned and factored into this bit field.

The instruction buffer accepts a thread complete signal. When threads are known not to need to replay, a complete signal with a signature (of one bit per thread) is sent to the instruction buffer. This occurs whenever threads in an instruction are known to complete. For example, when a memory request hits in the cache or is accepted into the memory system, the threads involved in these occurrences will not need to replay. When this signal is received, the indicated threads are removed from the instruction's private active mask bit field. If there are active threads remaining (the private active mask is not all zero) then it is known that a replay is required, and the instruction is marked as ready to issue. If the instruction is the oldest ready to issue instruction then it will be presented to the scheduler for consideration on the next cycle.

Alternatively, if no active threads remain, i.e., the private active mask is all zero, the instructions will no longer be replayed, and its entry is freed. If the freed entry is the oldest replayable instruction (between the issue tail pointer and the issue pointer), the issue tail pointer is advanced to one entry behind the next oldest replayable instruction. If no such instruction exists, then the issue tail pointer is set to the entry before the issue pointer. This frees up space for newer instructions to be filled from instruction memory, and the fill pointer may advance accordingly as additional entries are filled.

This single signal system requires that there is a single place where a replayable thread may signal being known to complete. If this restriction is true it is known that all threads that are not known to complete will not be able to complete and must be replayed. This logic allows the complete signal to indicate if a replay is required when compared with the private active mask. If a single instruction type can signal completion in multiple places then a separate signal must be sent to notify which threads need to be replayed. Since all the hazards considered occur in the memory stage during a single cycle, only the single signal type is required. Since the complete signal in essence doubles as a replay signal, the complete signal must be sent even if no threads will complete (they all need to be replayed). This requirement establishes a replay checkpoint where instructions must be evaluated before continuing, and must always signal to the instruction buffer.

Dealing with hazards in the memory stage is straightforward. When threads successfully access the cache or are successfully issued to the memory system they are known to complete and the signal is sent for those threads. The checkpoint can be placed at this point in the memory stage. All other threads that have not reached this point must have failed to complete due to some hazard.

It is notable that this fall-through hazard management may further simplify hardware implementation of corner cases and diverse hazards of the types previously mentioned. Replays may be simply signaled by stopping processing of threads that encounter hazards, wherever they occur in the logic (the control signals, including the warp ID number must still reach the checkpoint). Only signals for successful threads need to be routed through the processor to the checkpoint; threads that do not reach the checkpoint will be replayed.

The replay logic does not require knowledge of the hazard that created the need for a replay. However, this information could be recorded for other purposes; for example, scheduling decisions

might be made based on when and why a hazard occurred. This information is made use of in hazard prediction (see Chapter 6).

5.2.5 Notes on the Implementation

This architecture has been represented as storing the private active mask and implementing the associated logic in the warp buffer. This is probably inefficient from a design standpoint, as the warp buffer and associated scheduling logic is on the critical path for the scheduling stage. However the private active mask logic is not on that critical path and in fact is not required for scheduling. For this reason it would likely be moved to separate storage further down the pipeline. The mask stage is an excellent place since other masks are applied here and the initialization of the private active mask from the active mask is straight forward.

In this configuration the private active masks are consolidated into a single table in the mask stage. The table is indexed by the warp slot number and the instruction entry number. The warp buffer would retain just the per instruction status indicators: whether an instruction is active (and thus replayable) and whether is is marked as needing to be replayed. The scheduler does not need per thread information, so the correct thread mask can be applied in the mask stage. This architecture is shown in Figure 5.3.

Signals from the thread completion checkpoint in the memory stage are routed to the private active mask table and logic in the mask stage. The private active masks are updated as in the unified case. Upon receiving a signal, the evaluation of needing a replay or completion is determined as before, and this determination is then sent to the instruction buffer. This new signal to the instruction buffer can either indicate that the instruction should be replayed or is complete and should be freed.

This may add cycles before the need to replay is received at the instruction buffer. There is likely already a delay here due to the distance involved in the signal transfer across the chip. In terms of program execution, this delay is not necessarily on the program's critical path, in that the scheduler could schedule a ready instruction from another warp, or even an instruction from the same warp if its operands are ready.

In addition, a circular buffer may be replaced with a static buffer for per warp instruction storage in the warp buffer. This may however, limit the number of outstanding replayable instructions.

5.3 Replay Example

Figure 5.4 is an example of how the warp buffer handles replay. The example focuses on the execution of a single warp's instruction stream and shows a number of snapshots of the warp buffer contents pertaining to that warp over time. Each box corresponds to an instruction in the kernel for that warp. Instructions are labeled A, B, ..., E to denote the order in which they will be executed. The bits in the instruction boxes correspond in order: V if valid, R if ready to be replayed, and the

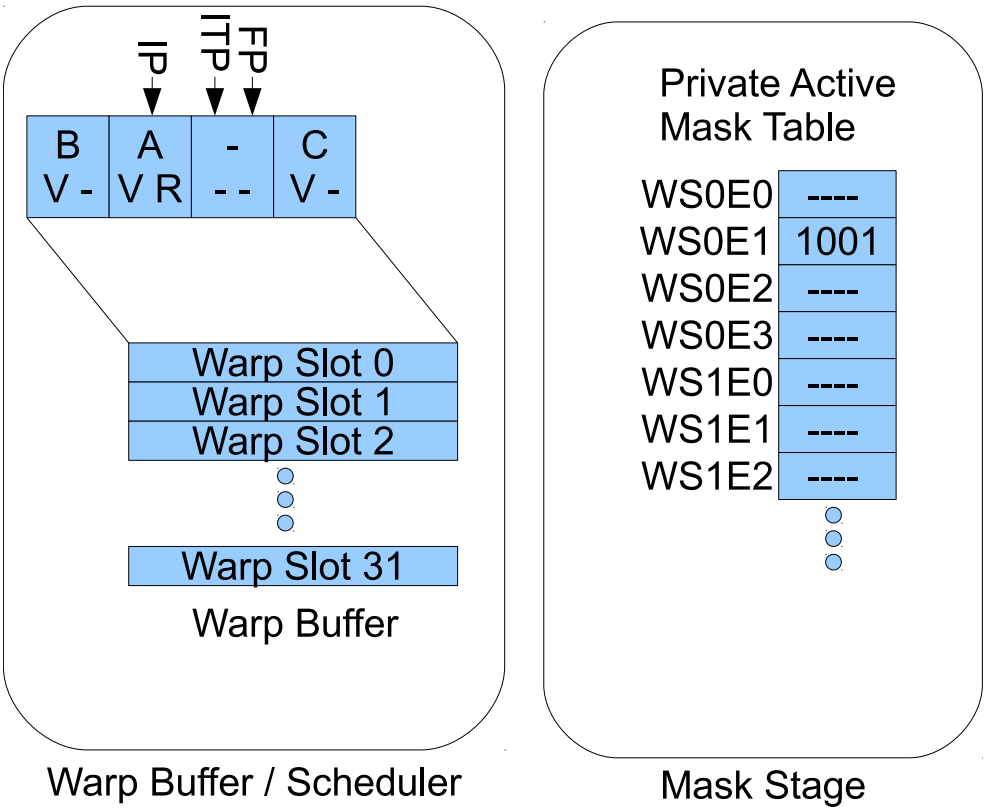

Figure 5.3: Modified replay data organization.

private active mask for 4-wide SIMT. The pointers are the Issue Pointer (IP), the Issue Tail Pointer (ITP) and the Fill Pointer (FP).

The first diagram in the series shows instruction A being issued for the first time. Instruction A is first in line to be executed since the Issue Pointer points to it and the Issue Tail Pointer is pointing to the slot behind it, indicating that there are replayable instructions pending.

In the second diagram, instruction C is being issued, and B has already been issued. Instruction B was not a replayable instruction so was immediately freed (the valid bit was cleared). Instruction A was a replayable instruction, and was retained in the warp buffer. The IP advanced to instruction C but the ITP was unable to advance due to instruction A. The Fill Pointer has not advanced because this warp buffer design fills two instructions at a time from instruction memory, and requires instruction A's space to continue. Instruction A's private warp mask has been updated with the warp mask that it encountered when it was issued: all threads were executing together.

In the third diagram, instruction C's private warp mask has been updated with the warp mask it encountered when it was issued. In this case only the first two threads were active when it was executed. Instruction B was a branch instruction, and the last two threads took that branch, in this case leaving the first two threads to continue executing consecutive instructions.

In the fourth diagram both instruction A and C have received completion signals from the memory stage. None of the threads in instruction C have completed. Only two of the four threads in instruction A have completed. Instruction C encountered a hazard where no instructions could make forward progress, either a MSHR, COMQ, or RSV hazard. Some of the threads in instruction A made forward progress, meaning that that instruction encountered either a BANK or a DIV hazard. The completed threads were removed from the private active masks of the instructions, and both instructions were marked for replay. Being the oldest instruction ready to issue, instruction A is issued.

In the final, fifth diagram, the remaining threads in instruction A have completed. The completed threads matched the remaining private active mask, signaling that instruction A was complete, and instruction A was freed. Freeing instruction A allowed the Issue Tail Pointer to advance to the entry after instruction C. The advance in the ITP allowed the instruction fill mechanism to fill instructions D and E, advancing the Fill Pointer as well. The Issue Pointer advances to instruction D. Instruction C, being the oldest instruction ready, is selected for issue.

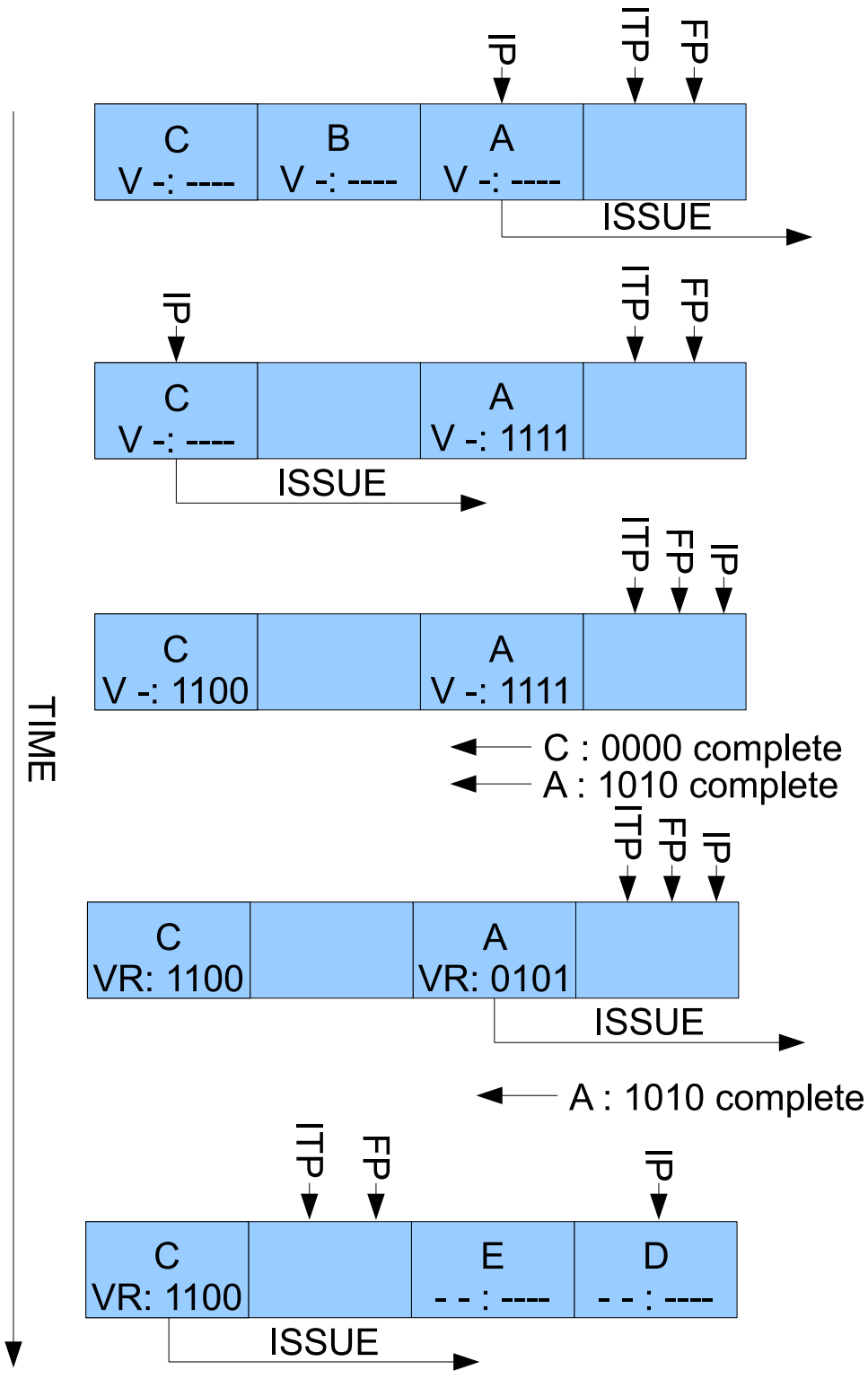

Figure 5.4: Example of Warp Buffer replay operation.

Chapter 6

Hazard Avoidance

Replay allows a stall free pipeline, and allows warps which do not use memory or use memory in a way that does not cause hazards (such as cache hits) to continue running. Even so, instructions which will be replayed use up storage in the pipeline. Each instruction that needs to replay is using up a spot in the pipeline that potentially could have been used for an instruction that makes forward progress. Figure 6.1 shows how cycles of a program are divided by hazard type that caused the replay. Also shown is hazard breakdowns for varying number of data cache MSHRs. This shows it is possible to trade-off between different types of replay architecturally. For example, decreasing the number of MSHRs reduces the number of possible outstanding memory requests which reduces pressure on the DRAM and interconnect and thus reduces replays due to COMQ failures. Conversely, increasing MSHRs may reduce MSHR hazards but increase COMQ hazards. This is evident for BFS2, CFD and STMCL. Figure 6.2 shows the issue cycle breakdown across the same set of varied MSHRs. A speedup is indicated by a lower bar. As can be seen, although hazard distribution is changed, performance is not greatly affected. Lower performance with increased COMQ hazards (for example BFS2 and CFD) likely derives from increased interconnect congestion (which is non linear in response to load). It is observed that replays for MSHR hazards are the major component of hazards for most benchmarks. Although hazard prediction could be used to handle other types of hazard using similar means, MSHR hazard replays represent the most significant contribution.

To prevent crowding out of useful instructions (or indeed the problem of pipeline hazards entirely) an optimal solution is never to schedule warps that will encounter a hazard and require a replay[1]. In the case of MSHR failures, this means never scheduling a load which needs an MSHR when none will be available (when it reaches the memory stage). Sometimes whether a load needs an MSHR is known with some certainty, for example when it has previously failed due to a MSHR, COMQ or RSV failure (all of which can only occur for a cache miss), or when it has a memory

[1] Mandatory hazards such as BANK and DIV would still be required for correct execution. In these cases the instruction is always doing useful work however, even when encountering the hazard.

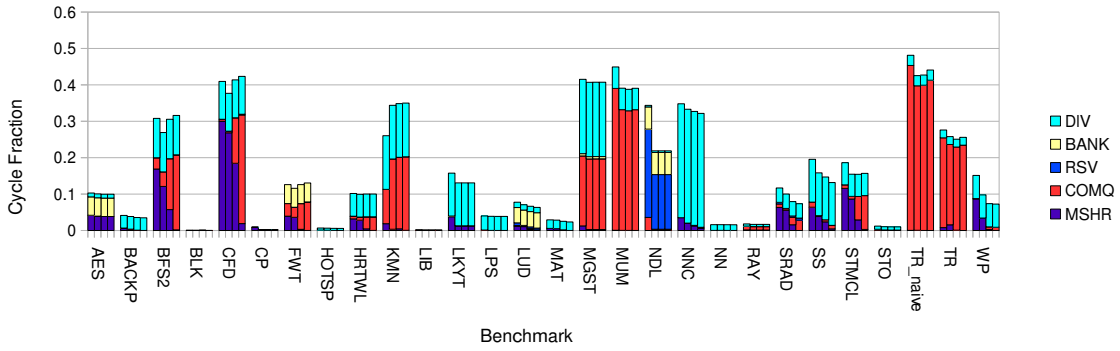

Figure 6.1: Breakdown of hazard cycles according to type for varying number of data cache MSHRs. Bars are: Baseline 32 MSHRs, Replay 32 MSHRs, Replay 64 MSHRs, Replay 128. Normalized to total cycles of Baseline 32 MSHRs.

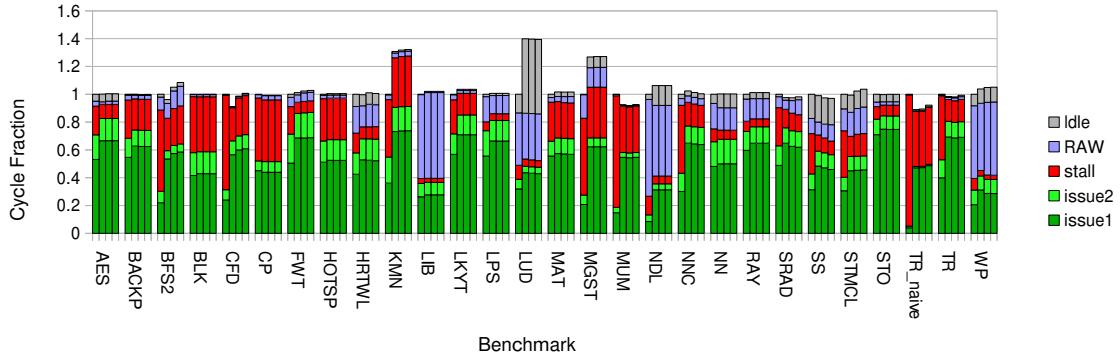

Figure 6.2: Breakdown of issue cycles according to type. Bars are: Baseline 32 MSHRs, Replay 32 MSHRs, Replay 64 MSHRs, Replay 128. Normalized to total cycles of Baseline 32 MSHRs.

pickup available (in which case it should hit in the cache). In the other cases however, whether the request will hit in the cache is unknown. If it is in the cache we would like to schedule it, if it is not we would like to ensure that it is not scheduled until we can be sure it will receive an MSHR.

To this end two mechanisms for tracking MSHRs have been implemented and predictors which predict cache hits have been examined. These two components are referred to as trackers and predictors. The tracker tracks the use of MSHRs, and the predictor speculates whether a specific instruction will miss in the cache and require an MSHR.

6.0.1 MSHR Trackers

The MSHR tracker's function is to determine whether an instruction that will require an MSHR (or is predicted to require an MSHR) should be scheduled or not. The idea is that instructions that will require an MSHR when they reach the memory stage are only scheduled if an MSHR will be available at that time. The main difficulty here is to overcome the delay between the scheduling stage, and the memory stage, and keeping track of how many MSHRs will be free at that future time.

Two MSHR trackers are implemented: naive and credit. The naive merely checks the current number of MSHRs and only schedules warps that are predicted to need an MSHR when an MSHR is available. The naive system is simulated with no delay in reading the current number of free MSHRs (a value available in the memory stage and used in the scheduling stage). This may be impractical due to the distance between these structures in real hardware implementations. In any case such a delay could only make this tracker worse. The naive tracker makes no attempt to keep track of the number of instructions capable of MSHR requests it has allowed into the pipeline. Thus this tracker may suffer from the pipeline delay between issuing instructions and when they reach the memory stage by placing more instructions requiring an MSHR in the pipeline than there are MSHRs available (or will be available). This observation leads to the design of the credit system (below) which ensures that no more requests requiring MSHRs are scheduled than can be accommodated by the available MSHRs.

The credit tracker implements a token type system where credits are allocated to instructions to indicate that an MSHR will be available to that instruction. A credit in this system represents the right to obtain an MSHR in the memory stage. The system is not strict in that instructions without a credit may still acquire an MSHR in the memory stage. This ensures that the credit system is simple and can only affect the operation of the core in the scheduling stage.

The credit tracker is implemented as follows. A core has an equal number of credits as MSHRs. The credit store (the number of free credits) is kept at the issue stage. A warp predicted to require an MSHR is not scheduled unless it can be given a credit. The credit is returned when the MSHR is freed. The system is not strict because a load instruction without a credit may still claim an MSHR in the memory stage, and an instruction with a credit which does not need an MSHR returns its credit at that point. This loose behavior is necessary to tolerate the uncertain nature of whether loads hit in the cache. When a load acquires an MSHR without a credit, a virtual credit is issued to the instruction. The virtual credit is the same as a normal credit except that when it is issued, a credit is removed from the available pool in the scheduler, which unlike regular credits, may make the credits available to the scheduler negative. In this case another warp was issued with a credit which corresponded to the MSHR that was just taken, and the virtual credit will be returned when that warp fails to acquire an MSHR or a memory request returns and releases an MSHR along with a credit. This system works properly even when the credit counter update is delayed by signaling

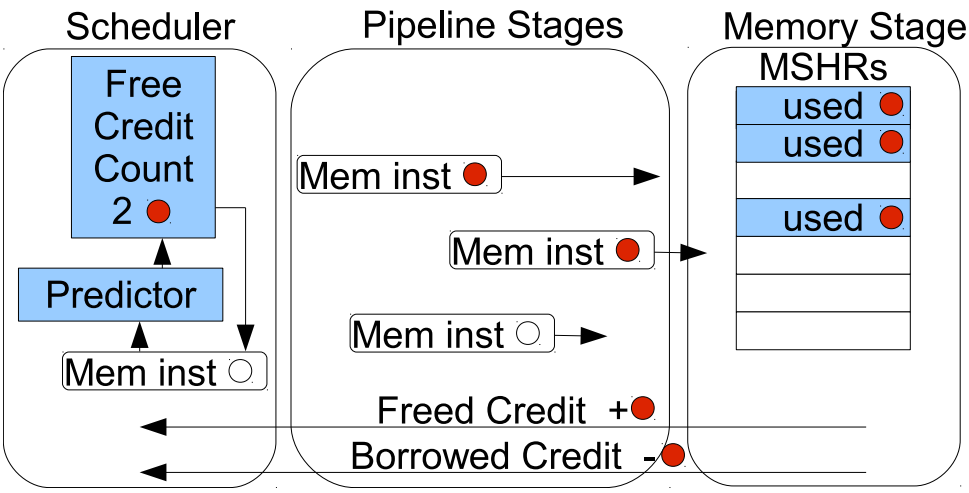

Figure 6.3: Overview of credit transactions and storage.

latency.

Figure 6.3 is a overview of credit transactions and storage. Credits operations are shown in the scheduler, memory stage, and intervening pipeline stages.

The credit system solves the problem of delay in the sense that the pipeline is filled with requests for MSHRs up to the number available. It still suffers from the delay in that instructions are not scheduled unless an MSHR is already available, and doesn't take into account that an MSHR may become available by the time the instruction reaches the memory stage.

6.1 MSHR Use Predictors

To achieve accurate prediction of hazard encounters, cache hit predictors are used. Four types are examined, 1) Predict Hit, 2) Predict Miss, 3) Saturating Counter, and 4) Oracle. These predictors (with the exception of Oracle) use information available to the scheduler to predict whether a memory instruction will be a cache miss and require an MSHR. Only constant, global, local and texture load instructions are considered as these are the only ones that can require an MSHR. A conservative predictor will over predict needing MSHRs, and thus slow down the program by not scheduling instructions that will hit in the cache, but will rarely suffer MSHR replays. Erring in the other direction, an optimistic predictor under predicts the need for MSHRs causing MSHR replays.

All predictors regardless of type first predict based on known information and if not covered by these cases makes its own prediction. In particular if an instruction that has been squashed because of an MSHR, COMQ, or RSV resource failure has missed in the cache then it is known to miss in the cache and require an MSHR, and this can be used for the prediction. For an instruction that was squashed and classified as a DIV failure, it is going to access a different address next time it is replayed and thus it is not known whether that address will be in the cache and no prediction can be

made on this information.

Two simple predictors are implemented. These either always predict miss, or conversely always predict hit. Predicting miss assumes all loads require an MSHR, and thus is overly conservative. Predicting hit assumes all loads are hits unless they have replayed and it is known they need an MSHR. Predict Hit limits MSHR replays to a maximum of once per load (barring MSHR tracking errors), since each load can miss and be squashed once before it is known to require an MSHR.

To more accurately predict whether an instruction needs an MSHR a saturating address based predictor has been implemented. Since access patterns for the same load instruction is probably similar across all warps, current hit/miss behavior for that load is likely a good predictor for other warps executing that same instruction. This predictor is implemented to explore the advantages of prediction. In particular, the saturating counter table is shared across all cores. Reads and updates are communicated instantly. In real hardware this organization would likely be impractical, given that the prediction result lookup is tightly incorporated into the scheduling logic and would likely need to be local. Because the predictor has access to information on behavior from all the cores, it should make better predictions in fewer training cycles than individual predictors in each core. Taken together these considerations place the performance of prediction as a loose upper limit on the efficacy of saturating counter type predictors in general.

Finally, to compare our predictors an optimal oracle predictor is implemented which checks the cache at issue time and predicts whether a load will hit based on whether its line is in the cache at that time (and thus is nearly optimal). This is an oracle operation because in hardware operation the issue stage does not have access to the memory stage caches and could in any case not check the cache until the address was known from loading the operands and doing the address calculation.

There may be some danger for hazard prediction in terms of self-fulfilling prophesy. If a load address is correctly predicted as a miss for many threads that would all hit to the same cache line (i.e., the first thread prefetches for the other threads), then every load of that data may be delayed. However this is unlikely to be a problem for this scheme since loads are only prevented from issuing if they are predicted to need an MSHR and there are no MSHRs available. Issuing this "self-fulfilling" load would do no good unless it was able to obtain an MSHR. Therefore there should be no chance of this danger.

Chapter 7

Discussion

7.1 Analysis of Architecture Configurations

Two configuration options important to hazards in replay operation are examined. Firstly, different configurations of operand collectors are examined in Section 7.1.2. Secondly, the effect of dual schedulers is examined with respect to interactions with replays in Section 7.1.3.

7.1.1 Warp Buffer Penalty

As described in Section 5.2.3, during replay, instructions that may replay must be retained in the warp buffer until they are known to not need any more replays. This puts a penalty on replay operation compared to the baseline architecture, which can "fire and forget" instructions. In the baseline architecture new instructions can continue to be loaded into the warp buffer and subsequently issued, even when replayable instructions have not completed. For example a load instruction may be stalled in the memory stage for many cycles, but the warp buffer associated with that warp can continue to issue independent ALU instructions without limit. In the case of replay, the number of independently issued instructions would be limited by the warp buffer size (since filling the buffer could not advance past the retained load instruction). Essentially this limit reduces the window in which a single thread can issue instructions. Figure 7.1 attempts to quantify this penalty by imposing the replayable instruction retention on the baseline stalling architecture. The warp buffer size used for all architectures is 8. The effects of this limit are relatively minor for most benchmarks. LUD and NDL suffer the most, possibly due to their reduced warp availability (a larger window provides more instructions to issue, when constrained by low warp count). Other benchmarks with low warp availability may not suffer (e.g., NNC) because dependencies between instructions prevent use of the larger window. For replay to provide benefit it must overcome this inherent limitation.

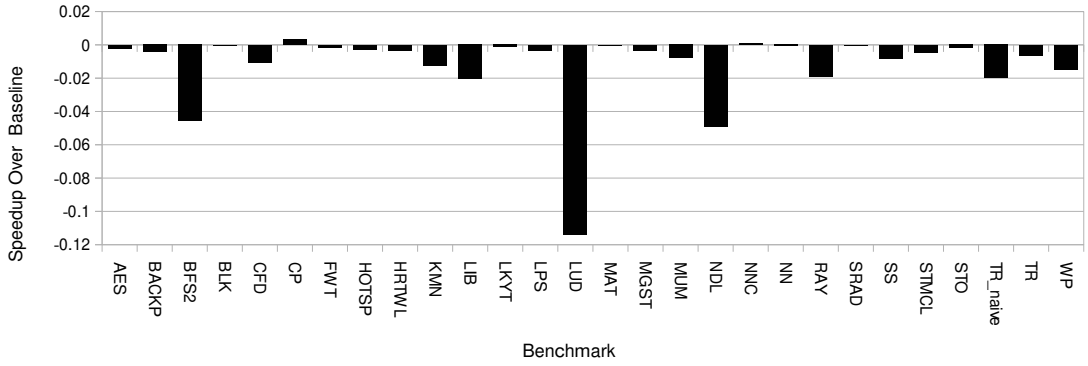

Figure 7.1: IPC speedup of retaining replayable instructions in the warp buffer (but no replay) over baseline architecture.

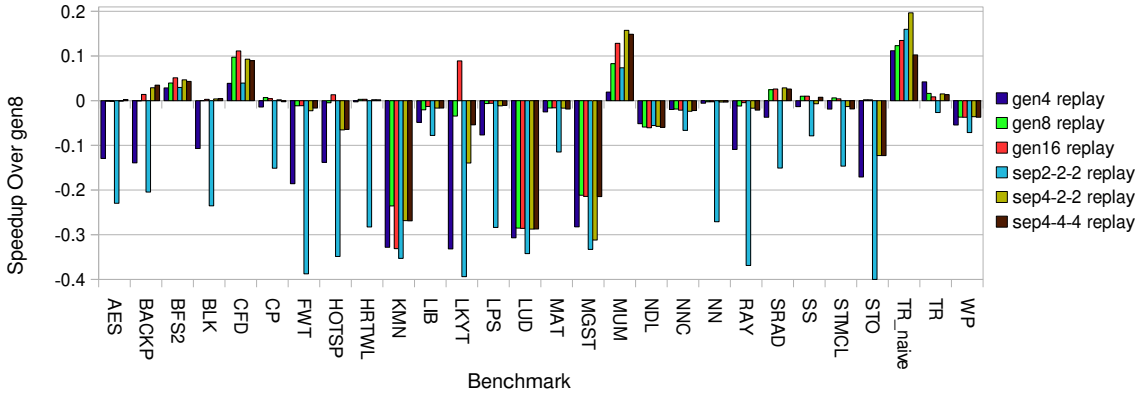

Figure 7.2: IPC Speedup over baseline (gen8 stalling) for various operand collector structures with stalling

7.1.2 Performance of Operand Collector Composition and Sizing

Figures 7.2 and 7.3 show the performance changes that occur when changing the operand collector configuration. The configurations are abbreviated as follows; “gen” represents a generic operand collector with the number representing its capacity; “sep” represents a separated operand collector with the three numbers denoting the size of the slots for instructions of type ALU, SFU, and MEM respectively. Figure 7.2 shows the varying performance of different collectors in a stalling configuration and Figure 7.3 shows this in a replay configuration. It is clear that a generic operand collector is more space efficient than separate configurations. This is likely due to the ability of generic configurations to use their space for any instruction, whereas separate configurations could be limited since size requirements could be exceeded individually. Capacity to accommodated stalls is limited in separated operand collectors.

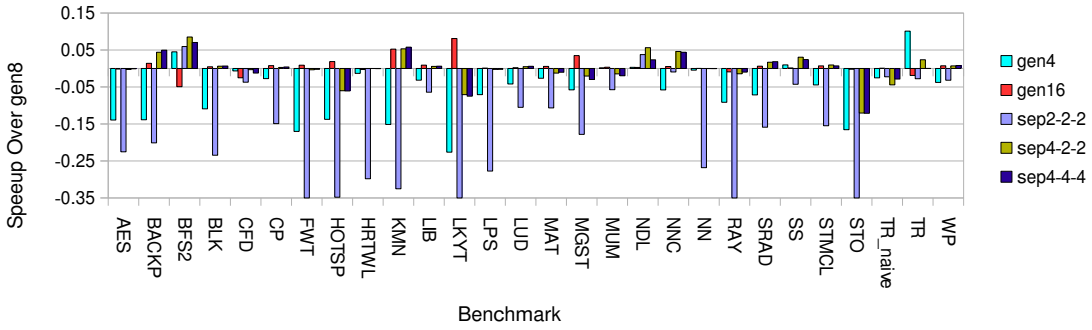

Figure 7.3: IPC Speedup over baseline (gen8 stalling) for various operand collector structures with replay

Separated operand collectors do sometimes provide additional performance benefits over generic ones. This is likely do to reduced MEM / ALU interference, although the improved performance of NNC, CFD, and MUM even with replay indicate that there are additional reasons for improvement. Since these benchmarks are memory heavy and suffer from large numbers of hazards (indeed these benchmarks improve the most using hazard avoidance mechanisms see Section 7.2), the separated operand collector will also prevent excessive memory replays from interfering with ALU instructions, possibly explaining the improved performance of separated operand collectors in these instances. As such, although possibly more expensive, sufficiently large separated operand collectors could be effective in solving MEM / ALU interference.

Because generic operand collectors generally are implemented with a CAM type structure for selection, their size might be a bottle neck on timing performance. An 8 slot generic operand collector was chosen as a reasonable baseline. This design point achieves within 5% of the maximum tested or most benchmarks.

7.1.3 Dual Scheduler Performance

Figure 7.4 compares the speedup of dual schedulers over a single scheduler in both with and without replay. It is clear that for many benchmarks there is a greater benefit of dual schedulers for replay; CFD, HRTWL, MAT, NCC, SRAD, SS, WP benefit greatly with two schedulers and replay.

Figure 7.5 provides a breakdown of the per cycle operations of the schedulers for each configuration. Scheduler cycles are classified as either “Idle” (no work in scheduler), “RAW” (no ready instructions due to Read After Write hazards), “stall” (scheduler cannot issue due to a stall in the pipeline), “issue1” (scheduler issues one instruction) and “issue2” (scheduler issues two instructions). Note that stalls do occur even in replay as the front end can provide instructions faster than the back end can consume them even if no memory hazards occur (this is due to the ability of sched-

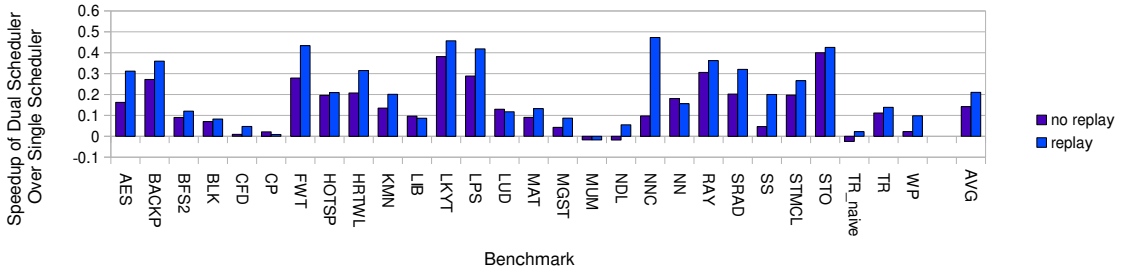

Figure 7.4: Speedup of two schedulers over one scheduler (one scheduler baseline) without and with replay.

ulers to issue two consecutive instruction in the same cycle). Since these classifications are mutually exclusive and complete, their total sums to the total execution time of the program, this total is normalized to the baseline stalling single scheduler configuration (the first bar in each set sums to 1.0). Dual scheduler configuration double the number of scheduler cycles, while keeping runtime the same, so these approach a sum of 2.0 for runtimes of equivalent duration. Bars below 1.0 for single scheduler and 2.0 for dual scheduler indicate a speedup. The bars in order are: baseline stalling single scheduler, stalling dual scheduler, replay single scheduler, replay dual scheduler.

Dual schedulers can increase throughput to the benefit of both stalling and replay configurations. The "issue1" and "issue2" totals include counts of both the first issue and replayed issues of instructions, so replay configurations experience large amounts of memory hazards have an increased number of instructions that need to be issued. Dual schedulers allow for this increased front end load to be appropriately handled. This can be seen especially in AES, BFS2, CFD, MUM, NNC, SRAD, WP where two schedulers provides a greater benefit for replay than it does for stalling. Some of these benchmarks show a marked improvement from replay and dual schedulers suggesting that the increased issue bandwidth is necessary for performance (e.g., NNC).

Figure 7.6 provides cycle breakdown counts for each of the hazard types perviously discussed. In stalling configurations these become stalls at the memory stage, and in replay configurations these become replays. Bar orders are the same as in Figure 7.5 and are normalized to the total cycles in the stalling single scheduler baseline (and so are directly comparable between graphs). For BFS2, CFD, STMC and WP MSHR, replays seem to be limited in the single scheduler replay configuration, implying that some restriction is slowing down memory operations. This observation fits the hypothesis that a single scheduler causes a bottleneck which restricts replay performance. As two schedulers are required for best operation (the number of functional units is designed for this amount of issue throughput), the baseline architecture outside of this section includes them.

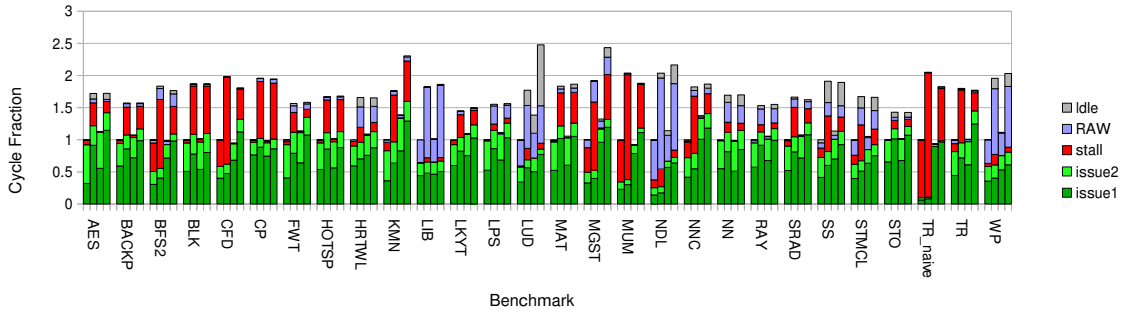

Figure 7.5: Fetch cycle breakdown of single and dual schedulers, with and without replay, normalized to single scheduler without replay. Each sub bar in order is no replay single, no replay dual, replay single, replay dual. Single scheduler results indicate a speedup when total is less than 1.0; dual scheduler results indicate a speedup when the total is less than 2.0 (since 2 fetch events happen every cycle).

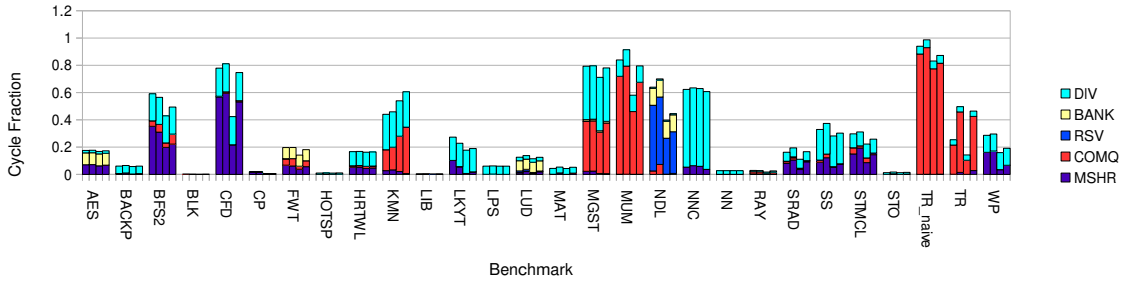

Figure 7.6: Hazard occurrence breakdown normalized to no replay single total cycles. Each sub bar in order is, no replay single, no replay dual, replay single, replay dual.

7.2 Hazard Prediction

This section demonstrates the relative performance improvements of the various hazard prediction schemes.

Figure 7.7 shows a scatter plot of predicted maximum speedup versus actual speedup with replay. Predicted maximum speedup is calculated by using data from the stalling baseline. It is calculated as described in Section 3.5 and represents an upper limit to performance improvement for that benchmark that can be gained from MEM / ALU parallelism.

Figure 7.7 shows a small noticeable correlation between predicted maximum and actual speedup with replay: the larger the predicted maximum the more likely the benchmark benefited from replay. However it seems that it is possible that there is more performance to gain from benchmarks with higher predicted speedup. These under-performing benchmarks, AES, FWT, CFD, BFS2, and SS are the same ones that show greatly expanded instruction issue numbers due to replay as shown in the last bar in Figure 7.5. This indicates that these excessive replays may retard performance.

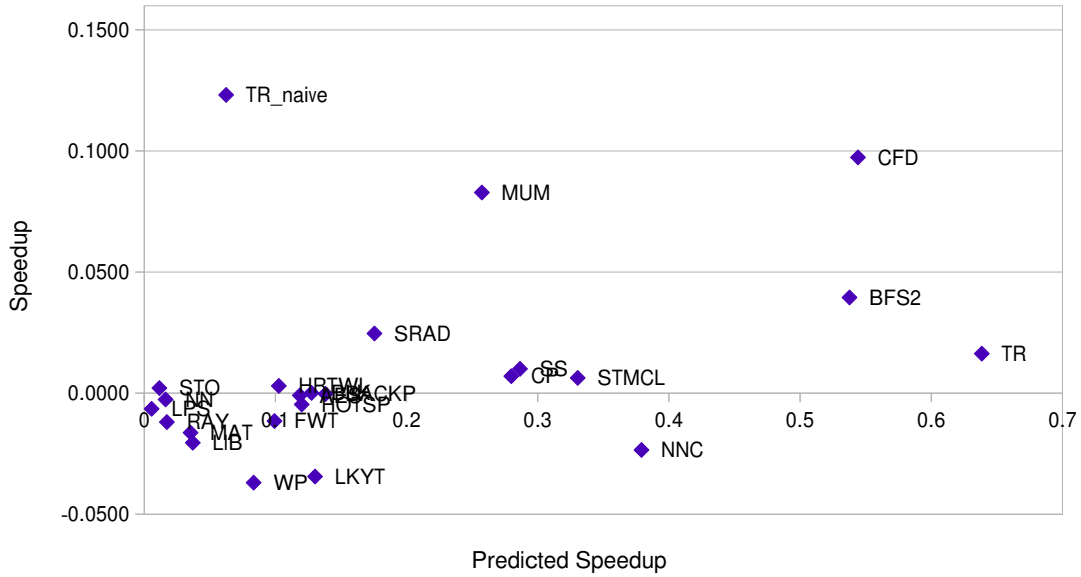

Figure 7.7: Scatter plot showing replay speedup versus predicted maximum speedup.

TR_naive is an outlier in Figure 7.7 in that it achieves almost double the predicted maximum speedup with replay. TR_naive is a very simple memory intensive benchmark that performs matrix transpose. It does not attempt to manage accesses in an optimal way (such as implemented in TR). Replay does create a slightly higher l1 cache hit rate (0.23 rather than 0.22), presumably by allowing new instructions to check for hits against the cache even while all MSHRs are in use. It is not clear if this explains the extra performance over maximum predicted speedup. TR completes the same transpose operation in approximately one fifth the total cycles, by manually caching data that will be used in shared memory. This indicates that a slightly higher cache hit rate may lead to significantly higher performance for this benchmark.

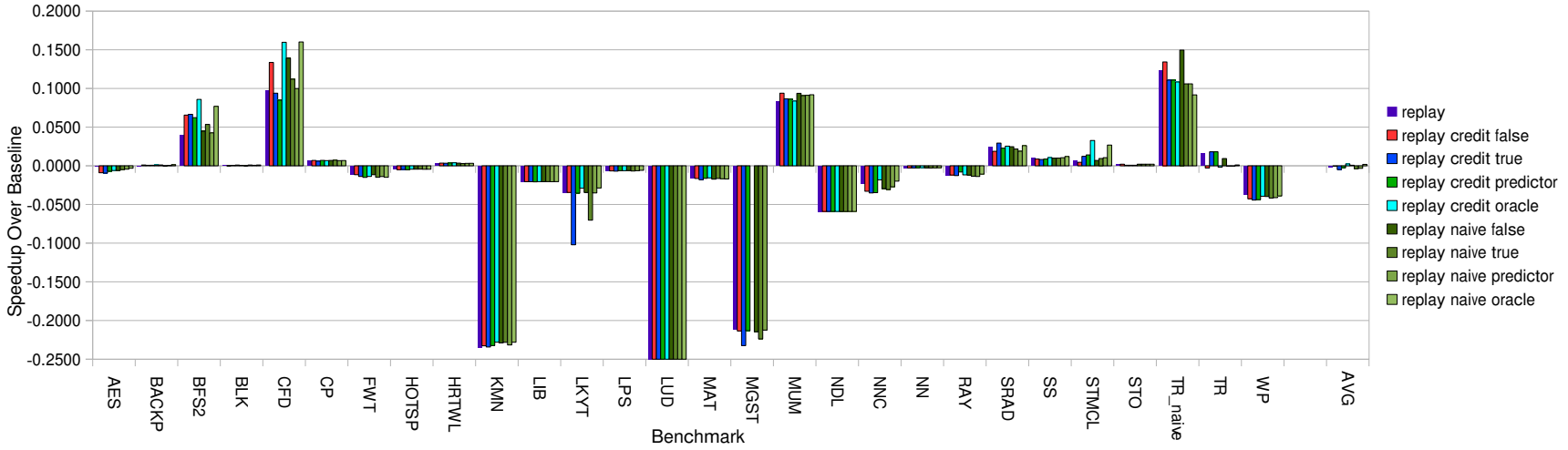

Figure 7.8: Speedup of predicted maximum speedup, replay and various hazard avoidance mechanisms over baseline.

Hazard prediction attempts to limit needless replays by avoiding issuing instructions that will likely result in a hazard and thus another replay. Figure 7.8 shows the results of each type of hazard prediction tested as well as the maximum predicted speedup for comparison. The different predictors are described in Chapter 6. The "predictor" predictor used was a global predictor across all cores, and consisted of saturating counters indexed by the PC of each load. Prediction accuracy was generally very good. The oracle predictor tests directly whether a load instruction will hit in the cache at issue (i.e., this is not physically realizable).

7.2.1 Predictors

All prediction schemes are relatively close in performance in general, although some benchmarks show marked differences. The "predictor" predictor does not have noticeably improved results over the static predictors. Predicting false seems to provide the most consistent performance benefits. This may be because it only limits the scheduling of instructions that have replayed (and are thus known to miss in the cache), thus avoiding the slowdown that would result from not issuing instructions that would hit in the cache. Some benchmarks are sensitive to this, such as CFD, while others are ambivalent. SRAD and TR are unique in having a slight but stronger improvement with the true predictor.

The oracle predictor provides an indication of the limits of the benefits of hazard prediction. The oracle predictor provides the best results for BFS2 and CFD, and these are unmatched by any other predictor, indicating that more sophisticated predictors could achieve this performance. In particular a "predictor" predictor could potentially achieve these benefits. Table 7.1 gives prediction accuracy ratios for each of the four possibilities. Of these, the incorrect predictions (ptf and pft) pose the most danger to performance. The ptf type indicates how often loads are unnecessarily restricted from issue, even though they will hit. This is a relatively minor problem for both benchmarks with less than 2% affected. The pft type incorrectly predicts that a load will hit, when it will not, oversubscribing MSHR demand, and leading to replays. Both benchmarks suffer heavily from this; about 25% of loads are affected. In both benchmarks this poor accuracy results from the most executed load addresses, which themselves have around a 60% hit rate. This poor prediction may be the result of the predictor being shared across cores rather than local to each core. However, it is also possible that more information than just load address is needed to accurately predict whether loads will hit.

Figure 7.9 plots the IPC speedups of both the naive and credit trackers on the predicted maximum speedup scatter plot. Some benchmarks with high maximum speedup can obtain larger speedups with replay and hazard prediction combined. HRTWL, BFS2, CFD, and SRAD all benefit drastically from hazard prediction. Figure 7.10 shows the cycle breakdown including cycles where no issue took place due to the hazard prediction mechanism (labeled as "restrict"). BFS2, CFD, and SRAD show the strongest improvements here as well. Figure 7.11 shows the associ-

Classification Label	Predict MISS	MISS?	ratio: BFS2	ratio: CFD
ptt	True	True	0.084525	0.025508
ptf	True	False	0.015640	0.004459
pft	False	True	0.269444	0.257120
pff	False	False	0.630392	0.712913

Table 7.1: "Predictor" predictor accuracy ratios.

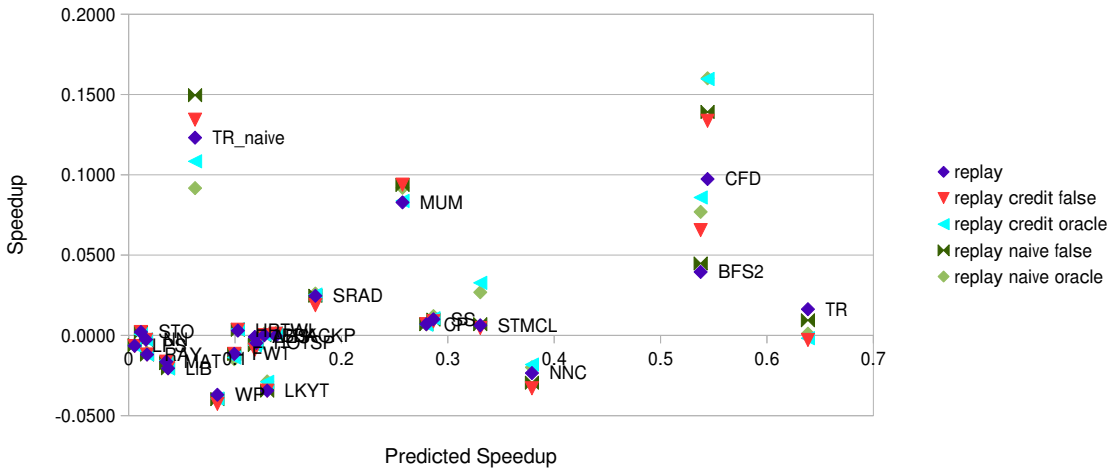

Figure 7.9: Scatter plot showing, replay and two best performing hazard avoidance mechanisms versus predicted maximum speedup.

ated hazard breakdown for each benchmark. Hazard prediction greatly reduces MSHR hazards in benchmarks with MSHR hazards (BFS2, CFD, SRAD, SS, STMCL and WP). The credit tracker is relatively more effective at preventing MSHR hazards than the naive tracker. This is expected given its design. Except for BFS2, this does not translate into any performance advantage.

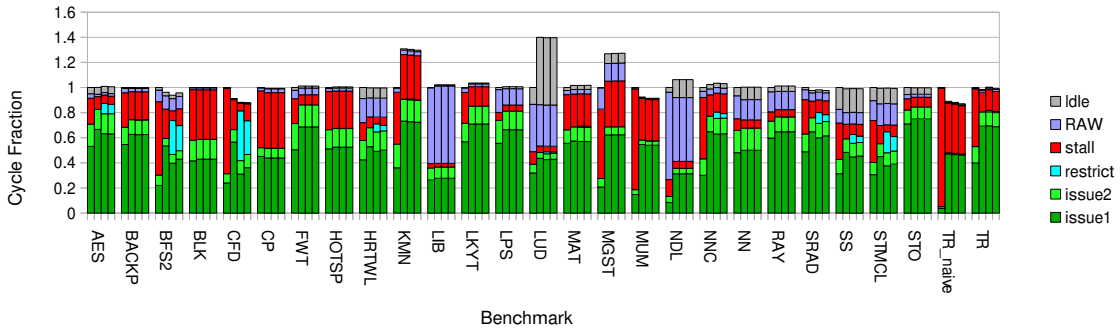

Figure 7.10: Cycle breakdown of (in order of bars) baseline (stalling), replay, replay credit false, replay naive false.

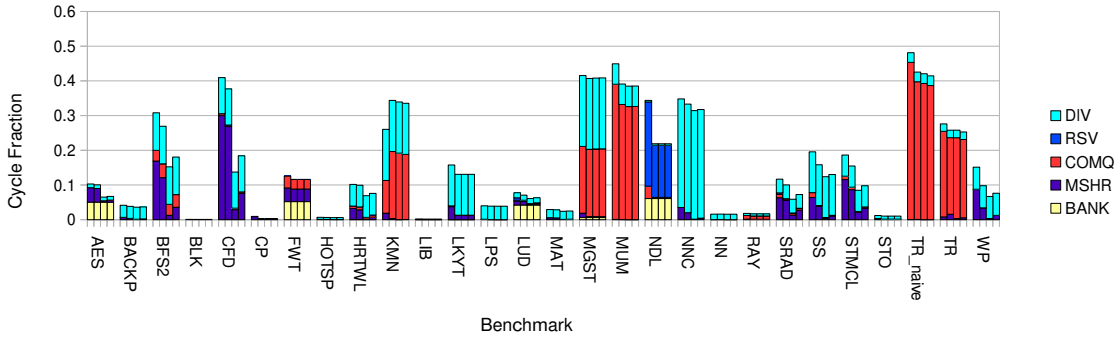

Figure 7.11: Hazard cycle breakdown of (in order of bars) baseline (stalling), replay, replay credit false, replay naive false.

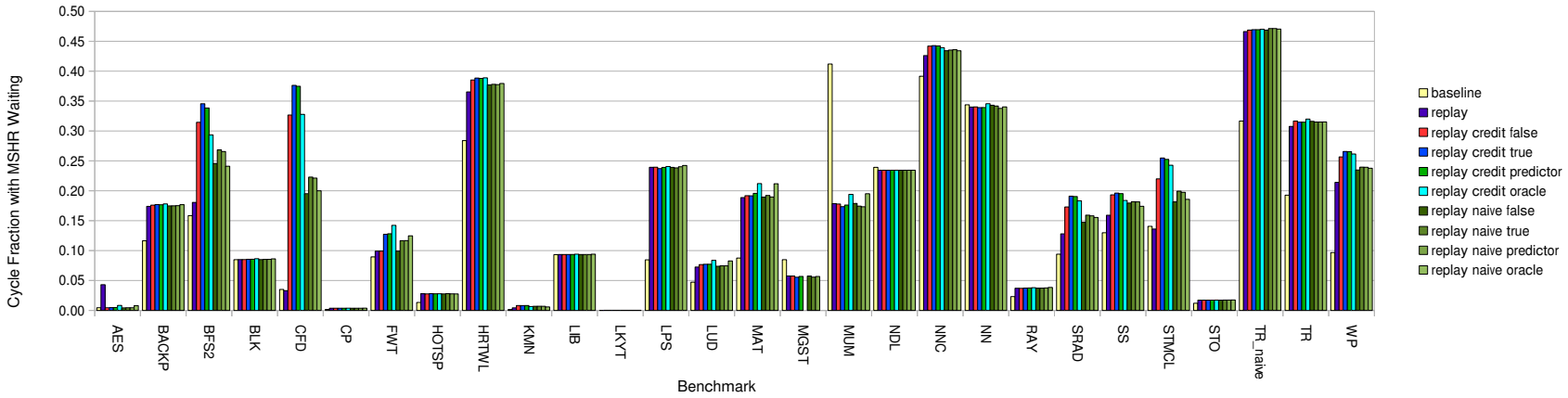

Figure 7.12: Fraction of cycles in which an MSHR using instruction was ready to issue or in the pipeline, and at least one MSHR was available.

7.2.2 Trackers

Both the credit and naive trackers impose a delay between when an MSHR returns and when the issue stage is allowed to issue MSHR requiring instructions. This leads to at least the pipeline latency from the issue stage to the memory stage of time when an MSHR that could be used sits idle. This is in addition to scheduling delays (the scheduler is issuing non MSHR using instructions) or stalling delays in the baseline architecture. The data in Figure 7.12 attempts to measure the number of cycles where this additional latency occurs. The plot shows the fraction of execution cycles in which at least one instruction that could use an MSHR was ready to issue or in the pipeline, and there was at least one MSHR available of the correct type. Larger fractions indicates more cycles where an MSHR could be used. Surprisingly, the baseline architecture (stalling) shows less penalty than with replay. This is possibly due to memory instructions waiting directly in the memory stage for resources (using them up quickly), whereas in the replay configuration, instructions must wait to be rescheduled and proceed through the pipeline. For benchmarks with large numbers of MSHR hazards (BFS2, CFD, SRAD, SS, STMCL, and WP), the credit tracker displays higher levels than the naive tracker. This could be because the naive tracker issues extra MSHR requiring loads (i.e., it only stops issuing when the MSHRs are actually used, so oversubscribes due to pipeline delay), and these extra loads in the pipeline use up MSHRs quickly as they become free to use. It is not clear how much of an impact this has on performance since MSHR use may not need to be maximal due to other limitations (e.g., the memory system and interconnect could be saturated at a lower MSHR usage level), however the naive tracker does perform slightly better for CFD.

7.2.3 Hazard Prediction Without Replay

Hazard Prediction does not rely on replay to work. Indeed in a stalling architecture, not issuing instructions which will stall in the pipeline should also provide a speedup. These results are shown in Figure 7.13. Note that credit false and naive false show no improvement because the false predictor never restrict issue of non replaying instructions.

The results are very different from those with replay enabled, but generally not exceptional. LUD for example benefits slightly from any sort of predictor (whereas it does poorly under all forms of replay). These differences are likely caused by a few reasons. For the benchmarks that do poorly on replay, these benchmarks likely suffer from the pathologies inherent in replay, such as low available instructions, restricted issue window, and poor tolerance to memory reordering. LUD and MGST fall into this category, however, as predicted in the scatterplots and shown in these results, they have MEM/ALU interference that can be exploited. Other benchmarks such as CFD and BFS2 perform inconsistently with some hazard predictors as opposed to with replay where they do well generally. This is likely due to how replay and hazard prediction work together constructively. Mispredictions in the replay architecture do not lead to stalls, just excess issue bandwidth. In the stalling architecture, however, a misprediction leads to a stall so it is more costly to be optimistic

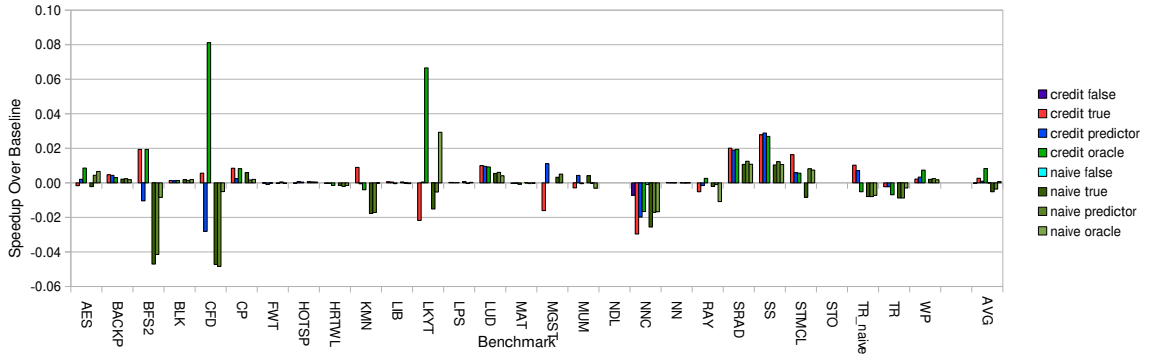

Figure 7.13: Speedup of hazard prediction without replay.

in issuing instructions. Thus, the most conservative hazard predictor (credit true) is seen to do well in hazard intensive benchmarks in the stalling architecture: CFD, CP, FWT, LPS, SRAD, SS and STMCL. The credit true hazard predictor is the most conservative because it assumes all loads will miss, and never issues more loads than it knows for sure will have an available MSHR. This most conservative configuration can be overly conservative, and for a few benchmarks is surpassed by the credit predictor configuration, for example, MGST. Allowing loads which often hit in the cache to issue is of some benefit for this benchmark.

7.2.4 Power

Hazard prediction has the potential to reduce power consumption by avoiding issue of instructions that will needlessly replay. Hazard prediction might even reduce power over stalling if unused pipeline stages could be turned off (as opposed to maintaining state for a stalled instruction). Every replay that is prevented from issuing results in saved pipeline stage power for each stage between the Scheduler and the Memory Stage. In particular, operand loads from the register file, address calculation, and a cache check are avoided for each avoided replay.

It is beyond the scope of this work to estimate the power savings, but reduction in instruction issues can be quite dramatic. Figure 7.14 shows the fractional reduction of cycles where instructions are issued for both the credit false and credit oracle hazard predictors. Issue cycles in CFD are reduced by 50%. In terms of restricting instruction issues, the credit false predictor achieves most of the benefit of the credit oracle predictor. Whatever power saving is achieved is in addition to power savings due to faster execution.

The power requirements of credit tracking are minimal, amounting to maintaining a few bits of state for each memory instruction, and a small amount of logic. The false predictor uses no special resources. The “predictor” predictor would consume more resources which would depend on the size of the saturating counter table and the logic and signals to query and update it.

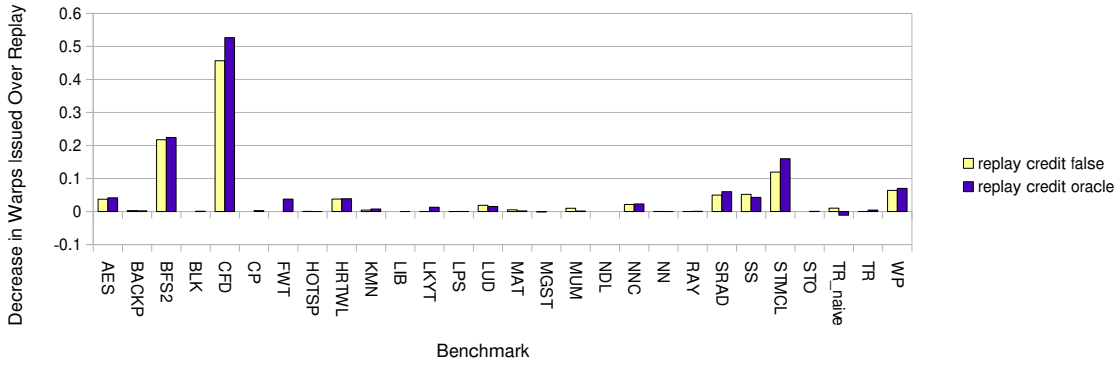

Figure 7.14: Reduction in cycles in which instructions are issued due to hazard prediction.

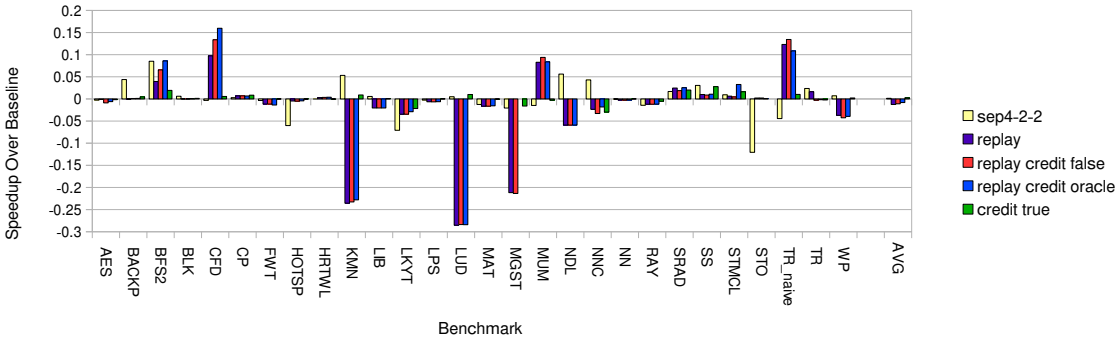

Figure 7.15: Comparison of various hazard management techniques.

7.3 Comparing Across All ALU / MEM Hazard Management Techniques

Figure 7.15 shows the speedup over the baseline architecture achieved by the best of the techniques examined. A separated operand collector, replay, replay with hazard prediction, and stalling with hazard prediction are compared.

Chapter 8

Conclusions

This work focuses on the pathological interaction of ALU and MEM instructions in a Compute Accelerator architecture. The implications of Replay in terms of this interaction are examined. There is some success in predicting which benchmarks will benefit from replay. In addition, a method for predicting hazards is proposed as a way to improve performance of replay where excessive MSHR hazard replays lead to slowdowns. The method is successful, and provides a small speedup (up to 3.3%) over replay in the two benchmarks with a high number of MSHR replays.

Although replay performance is somewhat predictable based on MEM / ALU balance, number of hazards, and average thread activity, some benchmarks perform very poorly (for example LUD, MGST). LUD has a relatively small number of active threads, and most of the performance loss comes from the reduced instruction window to execute required by the replay implementation. If it is found that this problem is more common, as an extention a more complicated Warp Buffer could allow unlimited instructions to fill the warp buffer, even while replayable instructions are retained.

There is no clear reason why MGST performs so badly with replay. It is possible that MGST (and to a lesser extent other benchmarks such as LYCT) is susceptible to increased latencies from replays even though there is a moderate availability of threads. Other reasons may yet still be discovered.

Hazard prediction is successful for benchmarks dominated by many MSHR hazards. Of the predictors and trackers, the credit tracker, with a tracker that only throttles replays (which are known to miss in the cache) performs the best. Other hazard types, such as COMQ hazards, could also be tracked and influence warp issue. This could be a credit system which specifically tracks the queues in the caches, or could more generally monitor the interconnect load. Tracking the queues themselves could be inaccurate since queues remain full for only a short time.

Hazard prediction is particularly effective at decreasing excessive replays and thus the total number of instructions issued. For CFD, hazard prediction reduces the number of instructions issued by nearly 50%. These savings in instructions issued likely lead to dramatic power savings.

An oracle predictor performed better than the most successful predictor, indicating that there is room for improved prediction schemes. The "predictor" predictor, that based prediction of whether a load would miss based on the load address did not perform adequately, indicating that more information might be required to get closer to oracle predictor performance.

The credit MSHR tracker could also be improved. The tracker does not allow instructions requiring an MSHR to issue unless an MSHR is actually ready. This causes at least the pipeline delay in latency before newly freed MSHRs can be used. This may hurt performance by under utilizing MSHR resources. A future tracker version could anticipate MSHR freeing, perhaps through an advance signal through the interconnect.

8.1 Future Work

The research in this work suggests avenues for future exploration. This work covered varying operand collector organization, number of schedulers, and hazard prediction schemes. Many other architectural variables may have large effects on performance and replay performance. Aside from basic changes such as cache sizing, clock speeds, interconnect type, etc., architectural latencies within the core might significantly affect performance. For example, a larger number of pipeline stages between the issue stage and the operand collector could cause stalling to be more detrimental, and replay to delay instruction execution by longer, which may affect the tradeoffs between the two schemes. Further investigation is required.

Signal latencies could also be examined. For example the naive predictor as implemented updates its MSHR use measure instantly, but in reality, information at issue would be some number of cycles out of date.

Bibliography

[1] Kunal Agrawal, Anne Benoit, and Yves Robert. Mapping linear workflows with computation/communication overlap. In *Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems*, ICPADS '08, pages 195–202, Washington, DC, USA, 2008. IEEE Computer Society.

[2] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 145–149, New York, NY, USA, 2009. ACM.

[3] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software(ISPASS 2009)*, pages 163–174, April 2009.

[4] Ronald D. Barnes, Erik M. Nystrom, John W. Sias, Sanjay J. Patel, Nacho Navarro, and Wen-mei W. Hwu. Beating in-order stalls with "flea-flicker" two-pass pipelining. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 387–, Washington, DC, USA, 2003. IEEE Computer Society.

[5] Darrell Boggs, Aravindh Baktha, Jason Hawkins, Deborah T. Marr, J. Alan Miller, Patrice Roussel, Ronak Singhal, Bret Toll, and K.S. Venkatraman. The Microarchitecture of the Intel® Pentium® 4 Processor on 90nm Technology. *Intel® Technology Journal*, 8(1), 2004.

[6] Wayne P. Burleson, Maciej Ciesielski, Fabian Klass, Associate Member, Wentai Liu, Senior Member, and Senior Member. Wave-pipelining: A tutorial and research survey. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 6:464–474, 1998.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC: In Proceedings of the IEEE International Symposium on Workload Characterization*, pages 44–54, 2009.

[8] Yuan Chou, Brian Fahs, and Santosh Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. *SIGARCH Comput. Archit. News*, 32(2):76–, March 2004.

[9] Brett W. Coon and Erik John Lindholm. US Patent 7,353,369: System and Method for Managing Divergent Threads in a SIMD Architecture, 2008.

[10] Dally, Bill. The end of denial architecture and the rise of throughput computing. http://asyncsymposium.org/async2009/slides/dally-async2009.pdf. Accessed 7/2012.

[11] Dan Ernst, Andrew Hamel, and Todd Austin. Cyclone: a broadcast-free dynamic instruction scheduler with selective replay. *SIGARCH Comput. Archit. News*, 31(2):253–263, 2003.

[12] P. Bose H. M. Jacobson, Z. Hu, A. Buyuktosunoglu, V. V. Zyuban, R. Eickemeyer, L. Eisen, J. Griswell, D. Logan, B. Sinharoy, and J. M. Tendler. Stretching the limits of clock-gating efficiency in server-class processors. In *Stretching the limits of clock-gating efficiency in server-class processors*, pages 238 – 242, 2005.

[13] Ziyad S. Hakura and Anoop Gupta. The design and analysis of a cache architecture for texture mapping. In *Proc. 24th Int'l Symp. on Computer Architecture*, pages 108–120, 1997.

[14] John L. Hennessy and David A. Patterson. *Computer architecture - a quantitative approach, 3rd Edition*. Morgan Kaufmann, 2003.

[15] Hans M. Jacobson, Prabhakar N. Kudva, Pradip Bose, Peter W. Cook, and Stanley E. Schuster. Synchronous interlocked pipelines. In *In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 3–12. IEEE Computer Society Press, 2002.

[16] Khronos Group. OpenCL - The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/. Accessed 7/2012.

[17] Ilhyun Kim and Mikko H. Lipasti. Understanding scheduling replay schemes. In *In International Symposium on High-Performance Computer Architecture*, pages 198–209, 2004.

[18] D. Koufaty and D.T. Marr. Hyperthreading technology in the netburst microarchitecture. *Micro, IEEE*, 23(2):56 – 65, march-april 2003.

[19] David Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *Proc. 8th Int'l Symp. Computer Architecture*, pages 81–87, 1981.

[20] Yongxiang Liu, Anahita Shayesteh, Gokhan Memik, and Glenn Reinman. Tornado warning: the perils of selective replay in multithreaded processors. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 51–60, New York, NY, USA, 2005. ACM.

[21] S. A. Manavski. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *ICSPC 2007: Proc. of IEEE Int'l Conf. on Signal Processing and Communication*, pages 65–68, 2007.

[22] Michael F. Miller, Kennneth J. Janik, and Shih lien Lu. Non-stalling counterflow architecture. In *4th Symp. on High-Performance Computer Architecture*, pages 334–341, 1998.

[23] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. Improving gpu performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 308–317, New York, NY, USA, 2011. ACM.

[24] NVIDIA. CUDA 4.0 Profiler F1 Help Menu. http://www.nvidia.com/content/cuda/cuda-toolkit.html. Accessed 7/2012.

[25] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi, September 2009.

[26] NVIDIA Corporation. NVIDIA CUDA SDK code samples. http://developer.download.nvidia.com/compute/cuda/ sdk/website/samples.html. Accessed 7/2012.

[27] NVIDIA Corporation. *NVIDIA CUDA Programming Guide*, 4.2 edition, 2012.

[28] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *ASPLOS-VII: Proc. seventh Int'l Conf. on Architectural support for programming languages and operating systems*, pages 2–11, 1996.

[29] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.