Write-after-Read Hazard Prevention in GPGPUsim

Michael Mishkin
University of Wisconsin
Madison, WI 53706
mmishkin@wisc.edu

Nam Sung Kim
University of Illinois
Urbana, IL 61801
nskim@illinois.edu

Mikko Lipasti
University of Wisconsin
Madison, WI 53706
mikko@engr.wisc.edu

ABSTRACT

The GPGPUsim simulator has a detailed model of streaming multiprocessor microarchitecture including scheduling logic that facilitates both instruction level parallelism and thread level parallelism. However, the pipeline performance model lacks proper safeguards against write-after-read hazards. The absence of mechanisms to enforce strict ordering of register file accesses permits more instruction level parallelism than would be otherwise achievable. This biases simulation results for some benchmarks and is potentially misleading to users of the simulator. In this paper, a few implementations of hazard handling mechanisms are considered, each with its own implementation complexity and granularity of bookkeeping. Evaluation of each mechanism's impact on performance is included as well as demonstration of the variation on conclusions that may be drawn from experiments depending on the choice of write-after-read hazard handling technique.

1. INTRODUCTION

Architectural simulators that are separated into functional and timing simulation components are useful because features can easily be implemented as modifications to the performance model without disrupting the functional correctness of the simulation. The functional stability can be harnessed to permit simplifications that approximate projected timing behavior and avoid needlessly excessive levels of detail. However, the apparent correctness of the simulation may mask underlying flaws in the performance model. When functional correctness is violated by an oversimplified performance model, simulation may yield results of precarious validity that introduce biases into experiments. Simulator users may include error checking in their code to catch these flaws, but identifying these situations is not always easy. When the functional inconsistency is part of the simulator's base implementation, it is especially unlikely that a user will identify the problem without expertise using the simulator.

GPGPUsim has gained favor among researchers of GPU architecture. As was noted in analysis of a previous version of the simulator [1], the level of detail of the model varies from one architectural component to the next. In general, the performance model of the multistage pipelines within each shader core are more detailed where performance critical contention can occur, such as the register file and the warp scheduler.

GPGPUsim executes CUDA kernels compiled to parallel thread execution (PTX) format. The PTX virtual machine model is an intermediate abstraction supported by many generations of NVIDIA GPU's with an ISA resembling assembly that is easily translated to the native instruction set in CUDA assembly (SASS) format [2]. Programs compiled to PTX have an unlimited set of virtual registers that may be used by PTX instructions. The nvcc compiler rarely rewrites to the same virtual register ID, other than within loops. Register allocation takes place during subsequent compiler passes that translate from PTX to SASS.

In GPGPUsim, the instruction streams executed on each core are derived from functional simulation of the kernel's PTX code without translation to a native ISA and without register allocation. Simulated register identifiers are based on the virtual register identifiers of programs in PTX format, rather than the registers allocated by the compiler. Since virtual register identifiers need not be reused to represent different values, destructive writes are a rare occurrence in PTX code.

GPGPUsim 3.2.2 is packaged with a program to convert PTX code to the PTXplus format that better resembles native SASS code. PTXplus extends the PTX instruction set to include features found in SASS. PTXplus uses the register assignments obtained from the SASS and therefore contains destructive writes to register identifiers which correspond to a finite set of physical registers. Destructive writes are much more frequent within PTXplus code than PTX code. However, some of the more complex instructions in PTXplus are not well defined in the performance model.

The shader core pipeline of GPGPUsim lacks a mechanism for preventing writing to a register while there are pending reads from that register by earlier instructions. Although occurrence of the hazard may be rare, a properly constructed pipeline should be capable of either preventing or handling the hazard. Inclusion of a hazard handling mechanism can have a dramatic impact on performance that differs from one benchmark to the next. Therefore, simulation without precisely defined hazard handling assumptions may yield unintentionally biased simulation results. This paper provides an analysis of the sensitivity of simulation results to different mechanisms for supporting write-after-read hazards in GPGPUsim for a set of GPGPU benchmarks drawn from Rodinia, the CUDA SDK, and those bundled with GPGPUSim.

The following section includes an overview of relevant microarchitectural constructs of the compute unit front-end and a more detailed description of the write-after-read hazard. Section 3 provides a compiler perspective of hazard handling that attempts to allocate registers in a way that prevents the hazard. Section 4 discusses microarchitectural mechanisms for handling the hazard. In section 6 experiments regarding each microarchitectural mechanism and its impact on experiments pertaining to other features are considered.

2. GPGPU BACKGROUND

Kernel invocations distribute thread groups over a set of streaming multiprocessors. The GPU compute unit (SM) maintains many active thread groups at a time. Each thread group (CTA) is subdivided into Warps of 32 threads that execute in unison [3]. In GPGPUsim, decoded instructions are inserted into a set of per warp instruction buffers where instructions are prioritized and scheduled. Scheduled instructions enter the operand collector in program order where many instructions wait for register file accesses to complete. Instructions may be dispatched from the operand collector out of program order if multiple instructions from the same warp reside in the operand collector simultaneously.

The Warp Scheduler is composed of a scoreboard used to determine issue eligibility and a scheduling policy for selecting among ready warps. Some SIMT workloads benefit more than others from the form of instruction level parallelism that scoreboard based scheduling provides. For example, workloads with insufficiently many warps to achieve high utilization may be more susceptible to individual warp latency affecting overall performance. The degree to which instruction level parallelism can be exploited by the hardware is largely dependent upon the warp schedulers and the scoreboard policies.

The most basic of the warp selection policies is round robin (LRR), which fairly selects among the ready warps. This policy has a low hardware overhead but also can result in inefficient execution with high data cache miss rates when fairness results in loss of locality [4].

The Greedy-Then-Oldest (GTO) selection policy favors a single warp until it stalls then selects the oldest available warp [4, 5]. Workloads that exhibit significant intrawarp locality tend to benefit from a GTO policy, while workloads with interwarp locality tend to benefit from a round robin policy [5].

The two-level warp scheduler (TLS) partitions the warps into two fixed size groups of active and pending sets based on whether the warp is waiting for long latency stalls [6]. With a two level policy, cache miss rates can be decreased when compared to a round robin policy because of the locality within a smaller set of warps [4].

The Scoreboard maintains information about the state of execution for each warp. Readiness criteria is evaluated for newly decoded instructions, and these criteria are updated as instructions commit [7]. The scoreboard is used to track the identifiers of registers with pending writes for each warp in the instruction buffer. Each decoded instruction compares its accessed registers with those marked in the scoreboard, and the scheduler stalls any warps with scoreboard conflicts until the conflicts are resolved [7]. Once all operands are ready to be retrieved from the register file, the warp is eligible to be issued to operand collection.

The limited instruction level parallelism enabled by the scoreboard is achieved by guarding against read-after-write and write-after-write hazards. However, since only written registers are tracked by the scoreboard, write-after-read hazards remain unhandled. Scoreboards have been used in single threaded pipelines [8, 9] with additional structures included to handle write-after-read hazards, however these bookkeeping structures generally do not scale well to multi-threaded pipelines.

The Operand Collector allocates a collector unit to each instruction issued by the scheduler [10]. Instructions wait in the operand collector until all register file read requests have been satisfied. The amount of time that an instruction spends waiting in the operand collector is dependent upon several factors including the number of operands required, register file contention, and availability of execution units. Once all operand values for an instruction have been collected, the warp is eligible for dispatch.

The Register File for each SM is large enough to support 32 instances of each register identifier per warp. The register file is composed of multiple single ported register banks that operate in parallel. Register file bank contention occurs when multiple read and/or write requests are simultaneously directed at the same bank. According to GPGPUsim, writes take precedence over reads, which can increase operand collector residency for an instruction that waits for its register read requests to be satisfied.

For load instructions, not all results arrive at the same time and GPGPUsim writes the responses directly to the register file as responses arrive. The register bank written by the load response is reserved across all execution lanes independent of how many registers are being written. The resulting contention can last up to 32 cycles for uncoalesced requests, which could cause instructions to remain in the operand collector for extended periods of time, thus increasing the likelihood of writing to a blocked register.

A write-after-read hazard occurs when an instruction is waiting in the operand collector to read a register while a subsequent instruction from the same warp advances to write back. As the possibility of write-after-read cannot be ruled out, it must be handled in some way. In GPGPUsim 3.2.2, there is no mechanism for handling these hazards. The resulting impact on performance in comparison to ideal handling of the hazard is minimal, however the overhead of ideal hazard handling is significant enough to motivate exploration of realistic implementations and their impact on performance. In section 3, a compilation technique that allocates registers in a manner that prevents the hazard is considered. In section 4 micro-architectural mechanisms to handle the hazard are discussed.

3. HAZARD PREVENTION

If the write-after-read hazard is to be left unhandled by the architecture, then it may be possible for the compiler to prevent the hazard. The compiler replaces PTX virtual register identifiers with architectural register identifiers during register allocation. If registers are allocated in such a way that the scoreboard's read-after-write hazard prevention is har-

nessed to prevent write-after-read hazards, then there may be no need for the architecture to handle the hazard.

The compiler optimization technique of register allocation by graph coloring [11] is the basis of many register allocation algorithms. Its graphical formulation considers live ranges of virtual registers to be nodes and overlapping live ranges to be edges such that the register allocation is the solution to a graph coloring problem, where the architectural registers are considered to be colors. This technique can be adapted to achieve hazard prevention by expanding the live ranges of registers to include at least one instruction that reads each value generated by instructions that access the register to which the live range pertains.

Typically, a value's live range begins when it is written to a register and ends after the last time that the register value is read. Registers may be reused by instructions outside of the live range. Policies that arbitrarily select a free register for an instruction immediately following a live range may permit or even encourage write-after-read hazards to occur. Given such an algorithm, the hazard can be prevented simply be expanding the live ranges.

Figure 1 shows a SASS code snippet obtained from the heartwall benchmark that contains a write-after-read hazard. The IADD instruction following the IMAD instruction reuses the R0 register immediately after the completion of the live range that began with the IADD32I instruction. If the IADD instruction were to advance to writeback before register R0 were read for the IMAD instruction, then an incorrect value would be retrieved.

```
MVC R5, c [0x0] [0x47];
IADD32I R0, R0, 0xffffffff;          }
IMUL.U16.U16 R18, R5L, R0H;          }
IMAD.U16 R18, R5H,R0L, R18;          }   Extended
SHL R18, R18, 0x10;                  }   Live Range
IMAD.U16 R5, R5L, R0L, R18;          }   for R0
IADD R0, R16, c [0x0] [0x4f]         }
IADD R0, R5, R0;                     }
SHL R0, R0, 0x2;
IADD R0, R15, R0;
IADD32I R0, R0, 0xfffffffc;
```

Figure 1: SASS code snippet from the heartwall benchmark with extended live range containing write-after-read hazard

Register allocation that prevents the hazard needs to guarantee that a register is not reused until all instructions that access the register have completed. If the architecture contains a scoreboard to prevent read-after-write hazards, an instruction can be guaranteed to be completed once its written register is accessed by at least one other instruction. Therefore, extending the live ranges to include the first instruction that accesses each register written to by an instruction that accesses a given value does guarantee that reuse of the physical register holding the value outside of the extended live range does not result in a hazard.

Reduction of the size of the extended live range may be achieved by insertion of fence-like instructions that read a register value for the sole purpose of guaranteeing that the end of the extended live range has been reached. The fence can be any instruction that reads the register, although a designated instruction that waits for scoreboard checks to pass but does not actually access the register file may be preferable for this purpose.

The consequence of register allocation based on extended live ranges is an increase in conflicts between extended live ranges and in turn an increase in the minimum number of registers required for execution of a given kernel without increasing register spills. The number of registers required by a kernel directly impacts the number of threads that can be scheduled to a compute unit per kernel invocation. This limitation on the GPU's ability to exploit thread level parallelism could hinder the GPU's achievable performance.

This compiler based approach to preventing destructive writes to active registers is only effective for handling instructions that write to a register. For instructions that do not write to a registers, such as store instructions, an additional mechanism is still necessary to prevent the WAR hazard. The following section describes a hardware based mechanisms for preventing the hazard that is appropriate for supporting these instructions as well as preventing the hazard without any compiler based hazard prevention.

4. HAZARD HANDLING

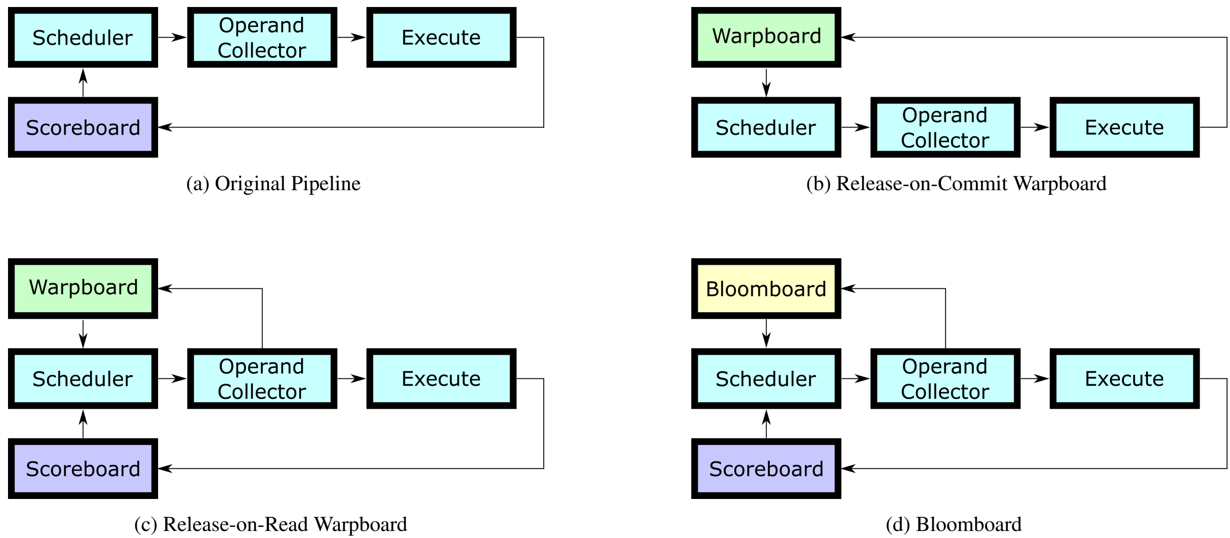
In lieu of guaranteed prevention of write-after-read hazards in the instruction stream, the hazard must be handled within the SM pipeline. Techniques for safeguarding against write-after-read hazards in single thread pipelines, such as renaming [8] and write-back buffering [8], are too cumbersome to support multithreaded pipelines. We propose and analyze three light-weight mechanisms with coarse grained register tracking to mitigate the write-after-read hazard, as shown in figure 2.

4.1 Release-on-Commit Warpboard

One way to guarantee operand consistency is to require that a warp may have only one outstanding instruction at a time. This type of book keeping only requires one bit per warp to be set when an instruction is issued and reset when the instruction commits. While the bit is set, no instructions from the warp are permitted to be scheduled. The warp tracking seamlessly integrates into the front end issue logic and actually obviates the need for a scoreboard.

4.2 Release-on-Read Warpboard

Since the scoreboard handles read-after-write and write-after-write hazards, the warpboard only needs to safeguard against write-after-read hazards. This can be accomplished by releasing the warpboard once all register reads have occurred. Reads are guaranteed to be completed once the instruction has left operand collection, so the warp's warpboard tracking bit can be reset when an instruction is dispatched from the operand collector. This allows scheduling for the warp to continue as usual much earlier than the release-on-commit warpboard. The consequence is that only one instruction from a given warp can be collecting operands at a time. While this restricts instruction level parallelism it is much less restrictive than imposing in-order execution by releasing the warpboard entry only after the previously issued instruction has committed.



(a) Original Pipeline
(b) Release-on-Commit Warpboard
(c) Release-on-Read Warpboard
(d) Bloomboard

Figure 2: Four Pipeline Architectures Under Consideration

4.3 Bloomboard

To achieve a finer granularity of instruction bookkeeping, a larger storage structure containing bins mapped to by a hash function applied to the register ids can be used to track sets of registers for each warp. The hash function applied to the register identifier could be the same function that is used to determine which register bank a given register id corresponds to.

The amount of storage required depends on how many bits are used for each bin. For a 1 bit bin, a bin is marked when an instruction is issued that accesses any register ids that hash to that bin. Subsequent instructions are allowed to be scheduled as long as they do not access a register identifier with a reserved bin. These bins may also be implemented as multi-bit counters to allow multiple reads per register identifier while preventing writes to registers that map to the same bin until the counter value decrements to zero. This increases the size of the structure since more logic is required for a multi-bin finite state machine than a bit array. Analysis with the GPGPUsim performance model provides insight into the value of the different stalling conditions as the size of bins is adjusted.

Abbr	Description	SM Capacity Limitation
AES	Cryptography	Max Threads
BIN	Binomial Options	Max Threads
CP	Coulombic Potential	Max CTAs
KMN	Kmeans Clustering	Partial Load
LPS	Laplace Solver	Partial Load
LUD	LU Decomposition	Partial Load / Max CTAs
MC	Monte Carlo	Registers
MUM	DNA Sequencing	Registers
NN	Neural Network	Max CTAs
NW	Needleman-Wunsch	Partial Load / Max CTAs
PFR	Pathfinder	Max Threads

Table 1: Benchmarks Considered in Experiments

5. EVALUATION METHODOLOGY

The experiments are conducted using GPGPUsim 3.2.2 and its GTX480 configuration with modifications to model each of the proposed scoreboard policies. Experiments are run for benchmarks from Rodinia 3.0 [12, 13], CUDA SDK 4.0 [14], and GPGPU-sim [15]. The list of benchmarks can be found in table 1. All benchmarks included in the results section are run as both PTX and PTXplus. Experiments are run for combinations of three scheduling policies and six scoreboard policies. The warp schedulers include round robin (LRR), greedy-then-oldest (GTO), and two level scheduling (TLS). The scoreboard policies include the gpgpusim default, the release-on-commit (RoC) warpboard, the release-on-read (RoR) warpboard, and bloomboards with 1 bit bins, 2 bit bins, and unbounded counters. The hash function used for the bloomboards is the register bank corresponding to the register identifier used by gpgpusim.

6. RESULTS

The experiments are analyzed from two perspectives. The first perspective compares the performance impact of the scoreboard policies for architectures with each type of warp scheduler. The measurements from this perspective are reported as clock cycles normalized to the gpgpusim default scoreboard policy and the same warp scheduler. These results are analyzed to understand the efficacy of each scoreboard policy.

Comparison of results for PTX and PTXplus are included here as well. For many of the benchmarks the results obtained from PTX simulation differ from PTXplus simulation. This is to be expected since although both represent the same program, the PTXplus kernels have undergone compiler optimization and better resemble what would be run on an actual GPU. The compiler optimization may expose more instruction level parallelism, which would be restricted

by these scoreboard policies. The increased register reuse in PTXplus kernels leads to an increased likelihood of the write-after-read hazard occurring. Neither the warpboard nor the bloomboard are directly based on register identifiers so the performance impact for both PTX and PTXplus kernels are similar.

When hardware features are simplified or overlooked by the performance model, simulation results can be skewed by factors that are not entirely understood. A scoreboard policy that overlooks write-after-read hazards exploits instruction level parallelism more aggressively than it would be in practice. Simulation results for susceptible benchmarks can therefore be skewed and conclusions drawn from experiments may be biased. To address this, a second experimental perspective is taken, modeled as a hypothetical warp scheduler experiment, where each scoreboard policy is considered as a baseline and the performance of the warp schedulers is compared. The measurements for these experiments are reported as a speedup of one warp scheduler over another. Observations are made regarding the variability of conclusions drawn using different baselines.

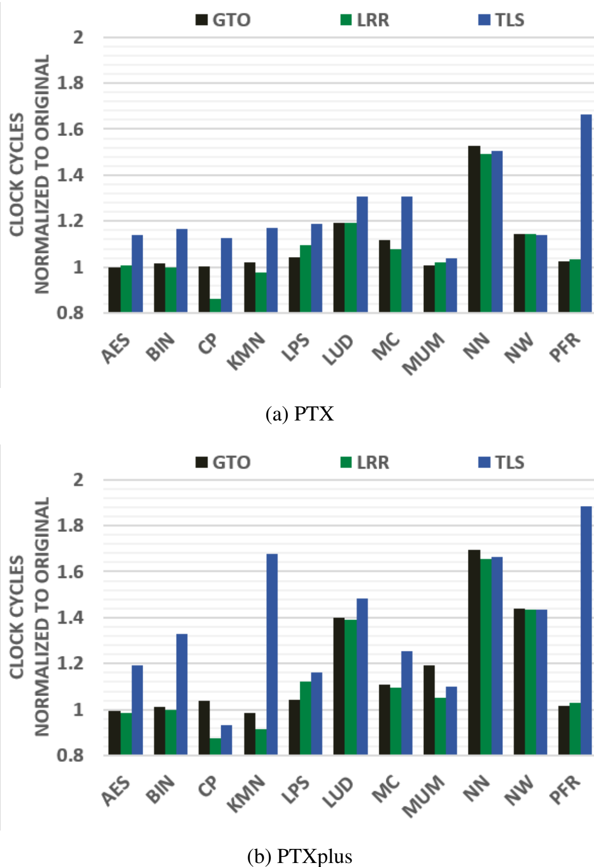
6.1 Comparison of Scoreboard Policies

The results in figure 3 show the effect of using the release-on-commit warpboard depicted in figure 2b. For most of the benchmarks the results show a slowdown. Benchmarks like NN and NW show slowdowns that are more or less independent of the scheduling policy. One thing that these benchmarks have in common is that their warp occupancy is below 17% for some kernel invocations due to the low block size and few threads per CTA. This limits the thread level parallelism that can be achieved and exposes more of the instruction level parallelism. The result is that the overall performance is more sensitive to changes in the per warp execution latency. Other benchmarks appear to be more severely impacted with the two level scheduler than other schedulers. This may be a symptom of warps remaining in the active scheduling window while waiting for release of the warpboard.

The release-on-commit warpboard actually improves performance of some benchmarks. For CP, the memory system response time degrades performance. KMN performance with a round robin scheduling policy also appears to benefit from reduction of instruction level parallelism.

For many of the benchmarks, the slowdown using PTXplus kernels is greater than using PTX kernels. This includes the benchmarks LUD, NN, and NW. Other benchmarks including AES, BIN, KMN, MC, and PFR all show increased slowdowns when using both PTXplus and the two level scheduling policy.

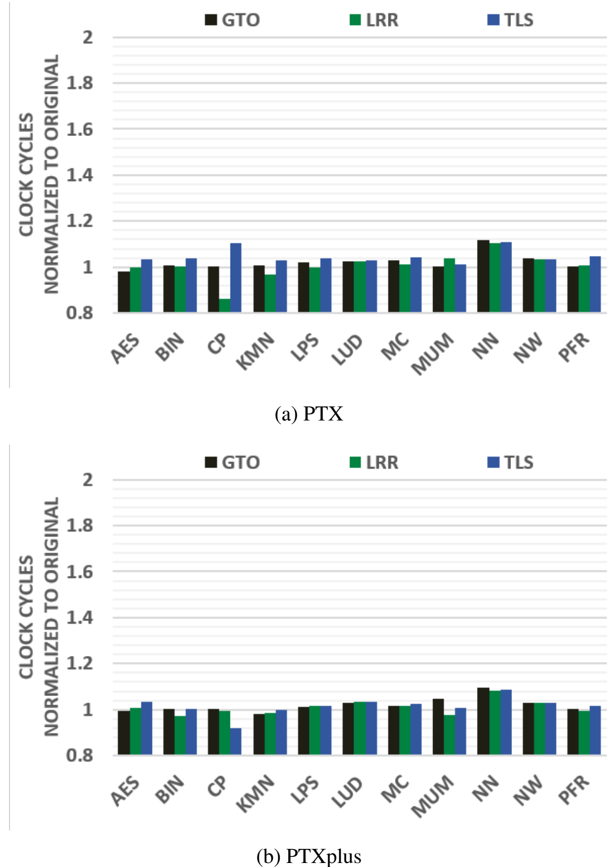
The results in figure 4 show the effect of using the release-on-read warpboard depicted in figure 2c. Better performance



(a) PTX

(b) PTXplus

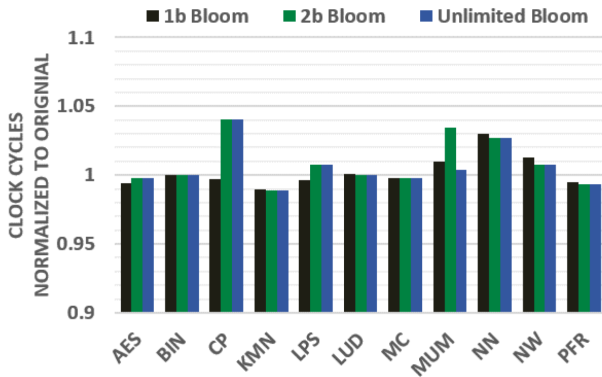
Figure 3: Performance Impact of the Release-on-Commit Warpboard Compared to the GPGPUsim Default.



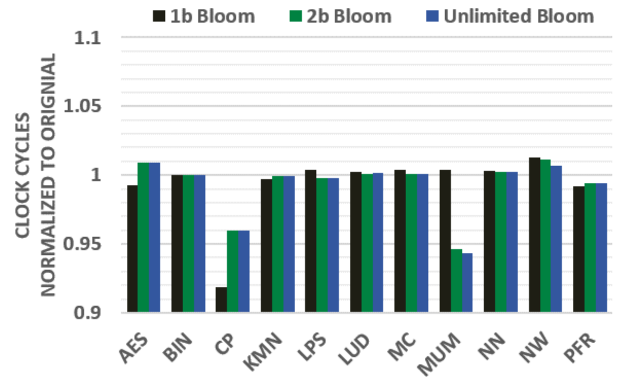
(a) PTX

(b) PTXplus

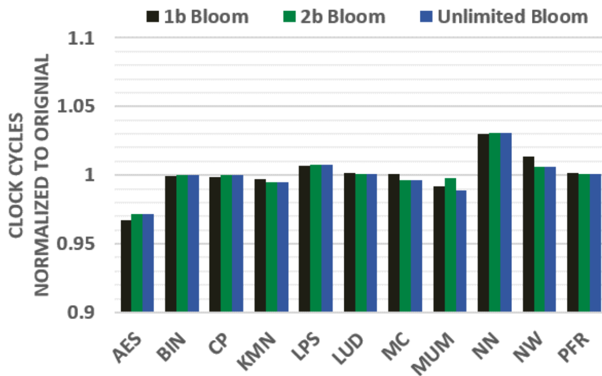
Figure 4: Performance Impact of the Release-on-Read Warpboard Policy Compared to the GPGPUsim Default.



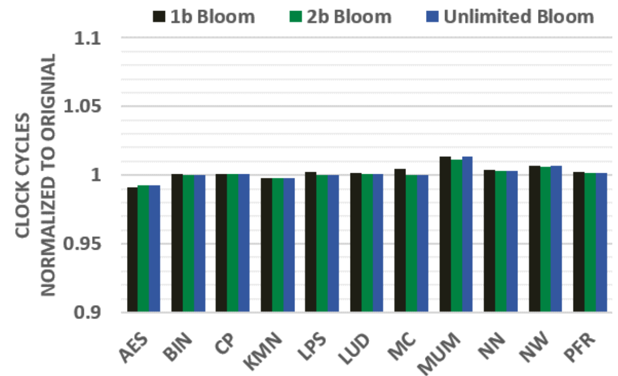
(a) LRR scheduler baseline with PTX kernels



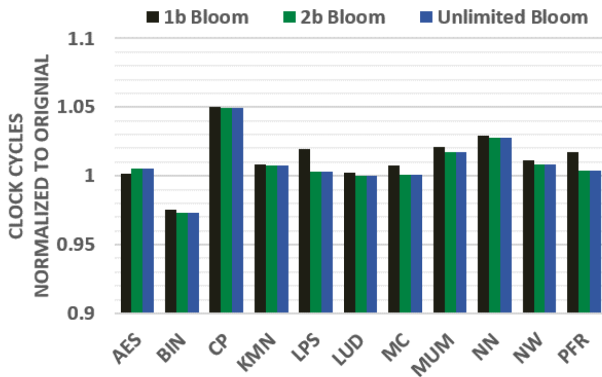
(b) LRR scheduler baseline with PTXplus kernels



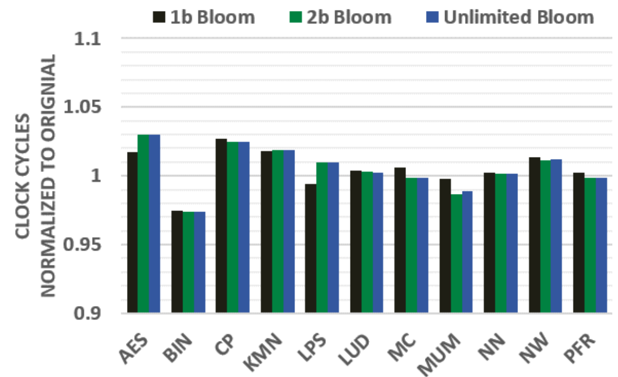
(c) GTO scheduler baseline with PTX kernels



(d) GTO scheduler baseline with PTXplus kernels



(e) TLS scheduler baseline with PTX kernels

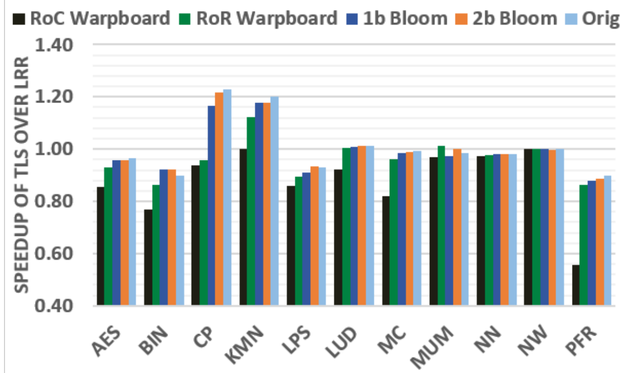


(f) TLS scheduler baseline with PTXplus kernels

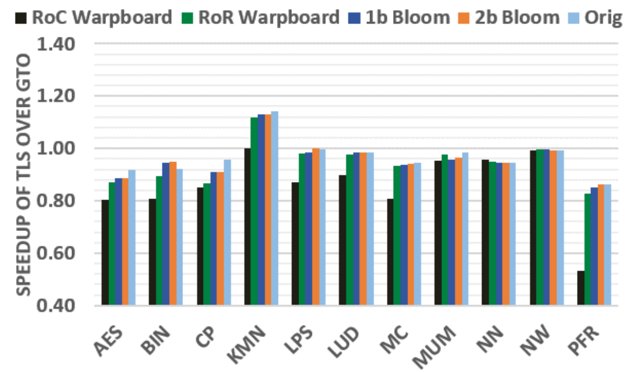
Figure 5: Results of Bloomboard Experiments

is achieved for many benchmarks that had poor performance using release-on-commit warpboards, with maximum slowdown of about 12% compared to the default model, as opposed to 88% with release-on-commit. Benchmarks that improved with release-on-commit have worse performance with release-on-read, while some do still have better performance than the baseline.

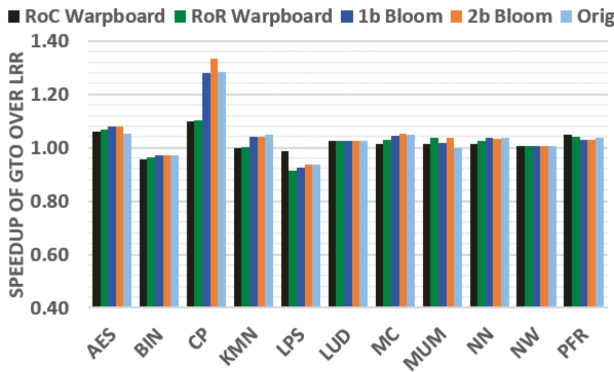
The normalized cycle counts for the bloomboard experiments are given in figure 5. Using a bloomboard yields results closer to those of the unmodified GPGPUsim. However many benchmarks do experience slowdowns or speedups of a few percentage points. For most benchmarks, there is no difference in performance between the bloomboard with a 2 bit counter and an unbounded counter. For benchmarks where there is a difference it is usually less than 1%, except for MUM. The 1 bit bin often achieves similar performance to the 2 bit bin, but not in all cases. Differences between the two tend to be within 5%.



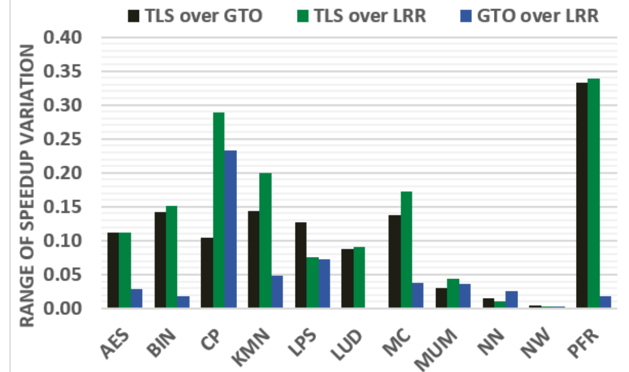
(a) Speedup of the two level scheduler scheduler over the round robin scheduler from baseline architectures with different scoreboard policies.



(b) Speedup of the two level scheduler over the greedy then oldest scheduler from baseline architectures with different scoreboard policies.



(c) Speedup of the greedy then oldest scheduler over the round robin scheduler from baseline architectures with different scoreboard policies.



(d) Variation in projected speedup for three different comparisons of scheduling policies.

Figure 6: Speedup results from five baseline architectures with different warp scheduling policies using kernels in PTX format.

6.2 Impact on Experiments

A comparison of different scheduling policies for baseline configurations having each scoreboard policy are given in figure 6. This comparison is intended to demonstrate how a typical experiment not pertaining to the scoreboard may be affected by the selection of scoreboard policy. Each comparison is between performance results of two architectures that are identical in all ways other than their warp scheduler. Comparisons are considered for baseline architectures with each scoreboard policy, except for the unbounded bloomboard because it was shown to be nearly equivalent to the 2 bit bloomboard in most cases. The discrepancy between the results of the comparisons can be seen in figure 6. Speedups of the two level scheduler over the greedy then oldest scheduler are given in figure 6b. Speedups of the two level scheduler over the round robin scheduler are given in figure 6a. Speedups of the greedy then oldest scheduler over the round robin scheduler are given in figure 6c.

In figure 6a, the KMN benchmark shows a speedup that appears to be negligible with one hazard handling technique while demonstrating up to 20% speedups with a different configuration. CP, LUD, and MUM all have speedups with one configuration and slowdowns with another. Similar inconclusive discrepancies can be seen for LPS and KMN in figure 6b and for KMN and MUM in figure 6c.

The variation in speedups across all baseline configurations for each scheduler comparison and each benchmark is given in figure 6d. AES, BIN, LUD, KMN, MC, and PFN have high variation for comparisons involving the two level scheduler. CP and LPS have high variation for all scheduler comparisons. MUM, NN, and NW all demonstrate low variation in comparisons of scheduling policies despite the high performance impact that the different scoreboard policies have on these benchmarks.

7. CONCLUSION

The GPGPUsim performance model's simplified write-after-read handling has the potential to skew experimental results by allowing more instruction level parallelism than would otherwise be possible. An ideal mechanism for preventing the write-after-read hazard is not easily implemented in the SM's multithread pipelines without incurring high area overheads. We have discussed four low overhead implementations of this logic that trade-off the granularity of bookkeeping for implementation complexity. The finest granularity considered shows variations of no more than 5% about

the baseline. This margin of error may be sufficiently low to be considered insignificant, however the scoreboard design that achieves this low discrepancy is not trivial.

Oversimplification of abstractions is a common pitfall in architectural simulators [16]. Unfortunately, there is no single method for identifying all flaws in a performance model. Determining the appropriate level of detail for the performance model of a particular feature could require experimental evaluation of various options to assess their impact. Based on the analysis in this paper, it would seem that the write-after-read hazard prevention model is an important factor to consider when using GPGPUsim for certain types of experiments regarding the SM pipeline.

8. REFERENCES

[1] T. Nowatzki, J. Menon, C. H. Ho, and K. Sankaralingam, "gem5, gpgpusim, mcpat, gpuwattch, "your favorite simulator here" considered harmful," in *11th Workshop on Duplicating, Deconstructing and Debunking*, (Minneapolis, MN), June 2014.

[2] NVIDIA, *NVIDIA Compute PTX: Parallel Thread Execution*. NVIDIA Corporation, Santa Clara, California, March 2009.

[3] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *Micro, IEEE*, vol. 28, pp. 39–55, March 2008.

[4] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, (Washington, DC, USA), pp. 72–83, IEEE Computer Society, 2012.

[5] M. Lee, G. Kim, J. Kim, W. Seo, Y. Cho, and S. Ryu, "ipaws: Instruction-issue pattern-based adaptive warp scheduling for gpgpus," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 370–381, March 2016.

[6] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (New York, NY, USA), pp. 235–246, ACM, 2011.

[7] B. Coon, P. Mills, S. Oberman, and M. Siu, "Tracking register usage during multithreaded processing using a scoreboard having separate memory regions and storing sequential register size indicators," Oct. 7 2008. US Patent 7,434,032.

[8] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[9] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout, "The load slice core microarchitecture," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 272–284, ACM, 2015.

[10] S. Liu, J. Lindholm, M. Siu, B. Coon, and S. Oberman, "Operand collector architecture," Nov. 16 2010. US Patent 7,834,881.

[11] G. Chaitin, "Register allocation and spilling via graph coloring," *SIGPLAN Not.*, vol. 39, pp. 66–74, Apr. 2004.

[12] 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 *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp. 44–54, Oct 2009.

[13] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pp. 1–11, Dec 2010.

[14] NVIDIA, *CUDA Samples*. NVIDIA Corporation, July 2013.

[15] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pp. 163–174, April 2009.

[16] T. Nowatzki, J. Menon, C. H. Ho, and K. Sankaralingam, "Architectural simulators considered harmful," *IEEE Micro*, vol. 35, pp. 4–12, Nov 2015.