Probabilistic Memory Consistency Specifications

Reese Levine
UC Santa Cruz
USA

Tyler Sorensen
UC Santa Cruz
USA

1 INTRODUCTION

Memory consistency specifications (MCS) have been developed as a way to reason about the interactions of shared memory across multiple cores. The simplest and most intuitive specification is sequential consistency (SC) [13], which specifies executions must correspond to some total order which is consistent with per-thread program order.

However, modern architectures do not implement SC by default, with research showing the prohibitive hardware costs of a globally consistent memory store [10]. Thus, architectures have been optimized to include layers of caches and buffers that increase performance and researchers have developed *relaxed* MCSs, such as the x86 [22] and C11 [6] memory models, which formally explain the nuanced behaviors allowed by a relaxed MCS.

Despite the relaxed models of many architectures, many applications require stronger memory orderings. Programs must enforce this using *memory fences*, which are special instructions that enforce the required ordering. While fences are often necessary for full correctness, they impose a performance penalty on applications that use them.

One idea that researchers have built on to increase performance of many applications is to note that in some circumstances, an approximate result may be accurate enough. This paradigm is called *approximate computing*, and includes techniques like approximating load values in hardware [16], discarding some atomic operations in GPU kernels [21], and speeding up critical sections by sometimes avoiding acquiring locks [19]. The idea of probabilistic disruption of thread communication is known as *relaxed synchronization*.

This proposal aims to extend the notion of relaxed synchronization to relaxed memory consistency specifications, introducing rare but possible weak memory behaviors that could lead to some result inaccuracy but increased performance. Our contributions will be:

- Empirically exploring the occurrence of weak behaviors on many different architectures under different system states
- Designing probabilistic memory consistency specifications (P-MCS), specialized to each system
- Using the P-MCS to provide approximate guarantees on different applications that use synchronization

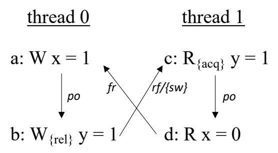
GPUs provide a good target for our work because of the diversity of devices, the high cost of memory fences compared to CPUs, and GPU applications such as graphics and machine learning that may tolerate approximate results. For example, work on fence insertion in CPUs showed conservative, safe strategies led to around a 30% increase in runtime [3], while similar analysis on GPUs showed a median of 174% increase in runtime [23].

1.1 MCS Definitions

An MCS can be illustrated using *litmus tests*, which are small concurrent programs. Figure 1a is an example of a test called Message Passing (MP), where one thread writes a value to a data variable

Initialize: x = 0 && y = 0	
thread 0	thread 1
a: W x = 1	c: $R_{\{acq\}}$ r0 = y
b: $W_{\{rel\}}$ y = 1	d: R r1 = x
Condition: r0 == 1 && r1 == 0	

(a) Pseudocode for the MP litmus test



(b) Candidate execution of MP using notation from axiomatic MCSs

Figure 1: A message passing (MP) litmus test. Making instruction b a *release* write and c an *acquire* read introduces the *synchronizes-with* relation, disallowing the weak behavior.

(a) followed by a flag variable (b), while a second thread first reads the flag (c) and then the data (d). The condition illustrates a weak behavior of this litmus test, where despite the flag variable being read as 1, the data variable is read as 0. Without adding synchronization, this behavior is visible on modern systems, such as ARM CPUs and many GPUs.

This weak behavior can be described using an axiomatic MCS, which represents program behaviors as candidate executions consisting of sets of events and relations between them. Following notation used in prior work [4, 6], the candidate execution that represents the condition `r0 == 1 && r1 == 0` is shown in Fig. 1b. Program-order (po) relates events in the same thread, reads-from (rf) relates reads to the write they read from, and from-reads (fr) is an inferred relation that relates writes to a read that occurs before the write. Many weak behaviors can be characterized by cycles in these relations.

To enforce ordering using fences, high-level languages (e.g. C++) allow a *memory order* to be attached to shared-memory operation, which translates to different types of fences depending on the architecture. Following this notation, we can *strengthen* event b to a *store-release*, and event c to a *load-acquire*, which is shown as the subscript in Fig. 1. Under these semantics, if c reads the value b wrote, we say b synchronizes-with (sw) c. In the C++ MCS, the sw and po relations are subsets of the happens-before (hb) relation, which reflects the order in which events should appear to happen and in this case, enforces that d reads from a.

1.2 Weak Behaviors in Practice

To provide confidence in a given MCS, prior work has explored methods for empirically testing and identifying weak behaviors on CPU-based systems [5, 8]. While it was once thought that weak behaviors were unlikely to be observed on GPUs [9], more recent work ran large experimental campaigns on GPUs from different vendors and revealed that GPUs show many of the same weak behaviors as CPUs [2]. This led to more work that generalized the

Table 1: Percentage of times a weak behavior was observed on GPUs running on Android devices, using Vulkan as the GPU framework.

	Litmus Test					
Device	**MP**	**S**	**R**	**LB**	**SB**	**2+2W**
ARM Mali - G71	0	0	0	0	0	0
ARM Mali - G78	0.23	0	0	0	0	0
Qualcomm Adreno 610	0	0	0	0	0	0
Qualcomm Adreno 640	0	0.13	0	0.09	0.13	0.17
Qualcomm Adreno 642L	0	0.22	0	0.15	0.2	0.27
Qualcomm Adreno 660	0.01	0.59	0	0.28	0.64	0.58
PowerVR GE8320	0	0	0	0	0	0
NVIDIA Tegra X1	0.0006	0.0007	0.0009	0.0009	0.001	0.001

techniques for exposing weak behaviors into *testing environments* that were used to expose weak memory related bugs in existing GPU applications [23] and to find bugs in GPU compilers and architecture [12].

2 A PROBABILISTIC MCS

An MCS specifies exactly the types of behaviors that are allowed by shared-memory accesses and provides safety guarantees when synchronization is used. However, the rates at which weak behaviors actually occur varies across devices and architectures, as well as how *stressful* the test environment is. Some initial data on rates of weak behaviors, collected through testing mobile GPUs running on Android devices using the Vulkan compute framework [11], is shown in Tab. 1. Each row shows a different device, while the columns show the data for six classic weak memory litmus tests described by prior work [4]: message passing (MP), store (S), read (R), load buffer (LB), store buffer (SB), and 2+2 write (2+2W).

The initial data shows that even when running under state-of-the-art test environments, weak behaviors are extremely rare, with the SB test on a Qualcomm Adreno 660 showing the highest percentage at .64%. In contrast, other devices from ARM, Qualcomm, and PowerVR show no weak behaviors. Therefore, if an application can tolerate occasional inaccurate results, it may be possible for them to forego using memory fences that enforce synchronization to increase their performance. For example, Niu et al. found that stochastic gradient descent could achieve convergence from between 3-10x faster when shared memory processors sometimes overwrote each other's results with [18], and Renganarayana et al. use relaxed synchronization to speed up k-means clustering between 2-15x depending on the parameters, with no change in the ultimate convergence of the clustering algorithm [19].

Our plan is to augment an MCS with probabilistic properties, giving estimates around the number of times weak behaviors might occur. The system can be tuned depending on the amount of anticipated load, and numbers will be based on rigorous MCS testing campaigns using litmus tests. Results will be specialized to each different chip and will be evaluated against approximate computing workload, allowing us to document the accuracy/performance trade-offs across different devices.

Our initial work will target an MCS for ISAs, as high level languages must cope with the additional consideration of the compiler introducing weak behaviors. Once we are able to show that our approach provides benefit at the ISA level, we will examine language level MCSs and their additional complexities.

Ultimately, a probabilistic MCS will be beneficial to high level language designers as well, as it could affect compiler development and language semantics. Further work will explore finer-grained probabilistic properties of re-orderings at the micro-architectural level by using the μhb relation defined in [15].

3 EVALUATION

The evaluation of probabilistic MCSs will consist of three parts:

- Collecting data on weak behaviors across a variety of GPUs by running comprehensive litmus test campaigns.
- Synthesizing P-MCSs from empirical testing results.
- Utilizing the P-MCS to make per-device probabilistic guarantees about approximate computing applications, such as prefix scans and k-means. We will explore the unique performance/accuracy trade-off per device.

As part of our prior research [14], we have already built tools for running litmus tests in both WebGPU and Vulkan. Some of the initial results of these experiments on Vulkan are shown in Tab. 1. We plan on running much more detailed experiments and collect data on more GPUs to build confidence in our methodology.

Empirical data collected under highly specialized stress could represent the worst case in terms of the rate of weak behaviors, where empirical data collected under little system stress could represent the best case for a P-MCS. We will innovate approaches for designing P-MCSs that take into account representative system environments when GPU synchronization is actually executed.

4 RELATED WORK

The idea of probabilistic MCSs follows both from work on testing MCSs to reveal weak behaviors and approximate computing. Frameworks like “litmus" [5] focused on exposing weak behaviors on CPUs, while later work developed frameworks to show weak behaviors and find bugs on GPUs [2, 12, 24]. We extend these works and will use the data on weak behaviors to develop new models for approximate computation under weak MCSs. MCSs are often used in high level languages, but research on micro-architectural MCSs have focused on verifying semantics of programs across the full stack [25].

In the field of approximate computing, relaxed synchronization was introduced to avoid some critical sections by skipping acquiring locks [19]. A number of relaxed synchronization techniques have been proposed, with work investigating the effect of injecting concurrency bugs such as removing barriers or changing atomic operations to conventional ones [1]. Data structures that execute without synchronization, increasing performance at the expense of accuracy and introducing data races, have been developed [20], along with compilers that generate approximate parallel programs that may include races [17]. A framework for reasoning about the behavior of programs with relaxed non-deterministic programs, including ones that remove synchronization, has been introduced [7].

While this work is inspired by relaxed synchronization, prior work on it has focused on specific algorithms rather than using empirical data to derive a probabilistic specification that can be used in general application design. Our goal is to augment existing MCSs with probabilistic properties, providing pragmatic information that developers can utilize to implement fast approximate applications across the ever-increasing range of GPU devices.

REFERENCES

[1] Ismail Akturk, Riad Akram, Mohammad Majharul Islam, Abdullah Muzahid, and Ulya R. Karpuzcu. 2016. Accuracy Bugs: A New Class of Concurrency Bugs to Exploit Algorithmic Noise Tolerance. *ACM Trans. Archit. Code Optim.* 13, 4, Article 48 (dec 2016), 24 pages. https://doi.org/10.1145/3017991

[2] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU Concurrency: Weak Behaviours and Programming Assumptions. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 577–591. https://doi.org/10.1145/2694344.2694391

[3] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. 2017. Don't Sit on the Fence: A Static Analysis Approach to Automatic Fence Insertion. *ACM Trans. Program. Lang. Syst.* 39, 2, Article 6 (may 2017), 38 pages. https://doi.org/10.1145/2994593

[4] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2010. Fences in Weak Memory Models. In *Proceedings of the 22nd International Conference on Computer Aided Verification* (Edinburgh, UK) *(CAV'10)*. Springer-Verlag, Berlin, Heidelberg, 258–272. https://doi.org/10.1007/978-3-642-14295-6_25

[5] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2011. Litmus: Running Tests Against Hardware, Vol. 6605. 41–44. https://doi.org/10.1007/978-3-642-19835-9_5

[6] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 55–66. https://doi.org/10.1145/1926385.1926394

[7] Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard. 2012. Proving Acceptability Properties of Relaxed Nondeterministic Approximate Programs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 169–180. https://doi.org/10.1145/2254064.2254086

[8] William W. Collier. 1992. *Reasoning about Parallel Architectures.* Prentice-Hall, Inc., USA.

[9] Wu-chun Feng and Shucai Xiao. 2010. To GPU synchronize or not GPU synchronize?. In *2010 IEEE International Symposium on Circuits and Systems (ISCAS)*. 3801–3804. https://doi.org/10.1109/ISCAS.2010.5537722

[10] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. 1991. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, California, USA) *(ASPLOS IV)*. Association for Computing Machinery, New York, NY, USA, 245–257. https://doi.org/10.1145/106972.106997

[11] Khronos Group. 2022. Vulkan 1.3 Core API.

[12] Jake Kirkham, Tyler Sorensen, Esin Tureci, and Margaret Martonosi. 2020. Foundations of Empirical Memory Consistency Testing. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 226 (Nov. 2020), 29 pages. https://doi.org/10.1145/3428294

[13] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[14] Reese Levine, Tianhao Guo, Mingun Cho, Alan Baker, Raph Levien, David Neto, Andrew Quinn, and Tyler Sorensen. 2023. MC Mutants: Evaluating and Improving Testing for Memory Consistency Specifications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 473–488. https://doi.org/10.1145/3575693.3575750

[15] Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2015. CCICheck: Using μhb graphs to verify the coherence-consistency interface. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 26–37. https://doi.org/10.1145/2830772.2830782

[16] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. 2014. Load Value Approximation. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 127–139. https://doi.org/10.1109/MICRO.2014.22

[17] Sasa Misailovic, Deokhwan Kim, and Martin Rinard. 2013. Parallelizing Sequential Programs with Statistical Accuracy Tests. *ACM Trans. Embed. Comput. Syst.* 12, 2s, Article 88 (may 2013), 26 pages. https://doi.org/10.1145/2465787.2465790

[18] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. 2011. HOGWILD! A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *Proceedings of the 24th International Conference on Neural Information Processing Systems* (Granada, Spain) *(NIPS'11)*. Curran Associates Inc., Red Hook, NY, USA, 693–701.

[19] Lakshminarayanan Renganarayana, Vijayalakshmi Srinivasan, Ravi Nair, and Daniel Prener. 2012. Programming with Relaxed Synchronization. In *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability* (Tucson, Arizona, USA) *(RACES '12)*. Association for Computing Machinery, New York, NY, USA, 41–50. https://doi.org/10.1145/2414729.2414737

[20] Martin Rinard. 2013. Parallel Synchronization-Free Approximate Data Structure Construction. In *Proceedings of the 5th USENIX Conference on Hot Topics in Parallelism* (San Jose, CA) *(HotPar'13)*. USENIX Association, USA, 6.

[21] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. 2013. SAGE: Self-tuning approximation for graphics engines. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 13–24.

[22] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. 2009. The Semantics of X86-CC Multiprocessor Machine Code. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 379–391. https://doi.org/10.1145/1480881.1480929

[23] Tyler Sorensen and Alastair F. Donaldson. 2016. Exposing Errors Related to Weak Memory in GPU Applications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 100–113. https://doi.org/10.1145/2908080.2908114

[24] Tuan Ta, Xianwei Zhang, Anthony Gutierrez, and Bradford M. Beckmann. 2019. Autonomous Data-Race-Free GPU Testing. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*. 81–92. https://doi.org/10.1109/IISWC47752.2019.9042019

[25] Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2017. TriCheck: Memory Model Verification at the Trisection of Software, Hardware, and ISA. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 119–133. https://doi.org/10.1145/3037697.3037719