# ...and region serializability for all

Jessica Ouyang     Peter M. Chen     Jason Flinn     Satish Narayanasamy

University of Michigan

{jouyang,pmchen,jflinn,nsatish}@umich.edu

## Abstract

A desirable concurrency semantics to provide for programs is *region serializability*. This strong semantics guarantees that all program regions between synchronization operations appear to execute in some global and serial order consistent with program order. Unfortunately, this guarantee is currently provided only to programs that are free of data races. For programs with data races, system designers currently face a difficult trade-off between losing all semantic guarantees and hurting performance. In this paper, we argue that region serializability should be guaranteed for *all* programs, including those with data races. This allows programmers, compilers, and other tools to reason about a program execution as an interleaving of code regions rather than memory instructions. We show one way to provide this guarantee with an execution style called *uniparallelism* and simple compiler support. The cost of the system is a 100% increase in utilization. However, if there are sufficient spare cores on the computer, the system adds a median overhead of only 15% for 4 threads.

## 1. Introduction

The concurrency semantics of a language provide guarantees to the programmer about how threads of a program can or cannot interleave during an execution. A language's concurrency semantics directly impacts programmability as software developers depend on those guarantees to reason about all possible legal behaviors of their programs and ensure software correctness.

The concurrency semantics of modern languages, such as C++ and Java, are specified in the form of a memory model. A memory model defines the set of possible orders in which memory operations from different threads can interleave and the possible values a read can return. Thus, a memory model serves as a language-level contract that specifies what guarantees programmers may assume and what constraints the compiler, runtime, and processor must collectively satisfy.

*Region serializability* is a strong and desirable concurrency semantics that guarantees that all program regions between synchronization operations (*synchronization-free regions*) appear to execute in some global and serial order that is consistent with program order. This guarantee greatly reduces the set of interleavings that programmers, compilers, and verification/testing tools must consider. It allows programmers and tools to consider each synchronization-free region as an atomic block, and it allows compilers to freely reorder instructions within a region without worrying about conflicting accesses from other threads.

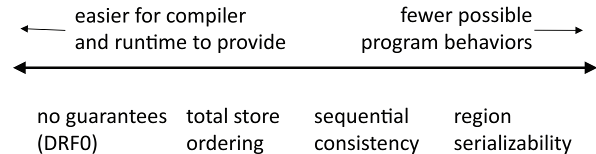


**Figure 1.** Memory models for programs with data races.

Unfortunately, the guarantee of region serializability is currently provided only to programs that are free of data races [6]. For programs with data races, system designers currently face a difficult tradeoff between weaker semantics and lower performance. For example, the DRF0 model [1] provides *no* guarantees to programs with data races, making such programs impossible to reason about and unsafe to run in production [6]. The *sequential consistency* [20] memory model is one of the strongest proposed that provides some guarantees to programs with races, but may be too expensive to provide without hardware support. The Java memory model attempts to define behavior for programs with races, but these semantics are difficult, even for experts, to understand [30, 42].

While it would be nice to assume that programs with data races do not exist, such an assumption is not borne out in practice, as deployed software in common use has been shown to be rife with data races [26]. Thus, in practice, one can neither ignore nor attempt to legislate away programs with data races.

The primary contributions of this paper are as follows:

- We argue that all programs — including those with data races — should benefit from the strong guarantees of region serializability.
- We present the first sytem that provides region serializability without custom hardware support, with a median runtime overhead of 15% for 4 threads.

## 2. A case for region serializability for all programs

### 2.1 Real programs have data races

Real programs have data races, both unintentional and intentional. Unintentional data races are considered bugs by developers, and these are found in many applications that are used in production. For example, a casual perusal of bug reports for four popular open-source applications (MySQL, Apache, Mozilla, and OpenOffice) finds data race bugs reported recently in all four applications. A study of various services in Windows Vista and Internet Explorer found 68 unique data races [36].

Ideally, all unintentional data races would be caught and fixed in development. However, guaranteeing this property requires a sound and complete static data-race detector, and, despite much research and progress on data race detection, such a tool has yet to be developed.

Another class of data races are those added intentionally by developers, usually in an attempt to improve performance. For example, Xiong et al. studied 12 server, desktop, and scientific applications and found many (6-83) ad hoc synchronizations in each program [48]. Programs may introduce data races when updating variables that are not needed for correctness, such as when gathering workload statistics.

While it can be argued that all programs with data races are misguided or wrong [5], it appears that, at least for the foreseeable future, programs used in production will continue to have data races. Unfortunately, classifying such programs as illegal [6] does not make them go away. This paper is concerned with the memory models provided to these programs.

### 2.2 Weak semantics are bad for programs with races

As shown in Figure 1, there is a range of memory models that a compiler and runtime system can provide to programs with data races, ranging from no specific semantics at all to the same semantics that are provided to programs without data races.

At the left end of this spectrum is the DRF0 model [1], which provides no semantic guarantees at all to programs with data races. The main benefit of this approach is the ease with which it can be provided. Processor architectures need not provide cache coherence to programs with data races, except on explicit synchronization operations. Compilers can assume that programs have no data races — even if this assumption is invalid for any given program, the compiled code still operates within the allowed range of "arbitrary". The main cost of this approach is the implication of losing all semantic guarantees. Since DRF0 allows arbitrary behavior for programs with data races, it is impossible for developers and verification tools to reason about the program (except to conclude that it has a data race). For example, under this model, compiler optimization can transform programs with data races in ways that are very surprising to programmers, such as jumping to arbitrary program locations [6]. To preserve safety guarantees, the Java memory model takes pains to define the semantics of data races [30].

In the middle of this spectrum are the memory models traditionally provided by cache-coherent multiprocessors, such as sequential consistency and total store ordering. These memory models operate at the granularity of instructions. Such fine granularity is well suited for guaranteeing the concurrency semantics of machine-level code. However, consistency models at the level of machine instructions do not provide meaningful guarantees at the source-code level because surprising results can be produced when lines of source code are divided into multiple machine instructions and interleaved with other threads. Providing a meaningful guarantee for the source code in higher-level languages requires additional effort on the part of compilers, and it is not yet known if such a guarantee can be provided without significant performance loss or hardware support [32, 43]. More fundamentally, simply guaranteeing a globally sequential order to all threads is a weaker guarantee than programmers are accustomed to.

### 2.3 We already assume region serializability

At the right end of this spectrum is the memory model provided by most higher-level languages to programs without data races, which we call *region serializability*. In this memory model, the source code between synchronization operations is considered a region (also called a *synchronization-free region* [27]), and the compiler and runtime guarantee that regions appear to execute in a sequential order that is consistent program order. The set of synchronization operations is defined by the language. In this paper, we consider `pthread` functions, system calls, and atomic variable accesses to be synchronization operations.

One indication of the value of stronger memory models, such as region serializability, is how often they are assumed or sought after, even when the intended runtime does not actually guarantee this memory model. We believe most data race bugs are due to programmers who, being used to region serializability for race-free programs, assume this guarantee holds even though their program has races. Similarly, compilers commonly perform sequential optimization techniques (e.g., loop-invariant code motion) on multithreaded code without first proving that the program is free of data races, even though those optimizations are invalid for programs with data races. Software verification tools and model checkers assume sequential consistency by reasoning about *thread interleavings*, each of which is assumed to be seen consistently across all threads. Some proposals for verifying programs seek to shrink the state space by grouping multiple instructions into atomic units, either by proving commutativity, as in Lipton's theory of reduction [21, 23], or by relying on programmer annotations, as in thread-modular verification [16]. Other examples are systematic testing tools that only explore preemptions at syn-

chronization operations [34]; this is exactly the concurrency semantics guaranteed by region serializability.

## 3. Design

If performance were not an issue, a trivial way to guarantee region serializability for all programs would be to actually run regions serially. This could be done by a system with the following three properties:

**Property 1**. *All instructions from a particular address space are executed on a single processor, i.e., only one thread runs at a time.* Limiting execution to a single processor prevents multiple, concurrent processors from interleaving instructions from multiple regions.

**Property 2**. *Preemptions are only allowed at region boundaries.* Even on a single processor, allowing one region to preempt another would violate our goal of running regions serially.

**Property 3**. *Synchronization-free regions correspond to the original source code.* The first two properties guarantee that synchronization-free regions in the program binary are executed serially. To make this guarantee meaningful to programmers, the compiler must not perform any optimizations that violate the ordering or atomicity constraints of regions. For example, it must not move instructions from one region to another. However, because regions in the program binary are guaranteed to execute atomically, the compiler can optimize freely within a region.

By satisfying Properties 1-3, a system would provide all programs with region serializable semantics, even those containing data races.

## 4. Uniparallel execution

Clearly, requiring a program to execute on a uniprocessor (Property 1) is too restrictive given the current architecture trend to provide parallelism through multicore and many-core computing. Our system leverages a style of execution called *uniparallelism*, first used by Veeraraghavan et al. [46]. Uniparallelism allows our system to scale with increasing number of CPUs, while providing the uniprocessor semantics required by Property 1.

In uniparallelism, an application's execution is divided into distinct time intervals, called *epochs*. The goal is to run epochs in a pipelined fashion, allowing later epochs to start before earlier epochs finish. A pipeline consisting of 8 epochs, Ep 0-7, is shown on CPUs 4-11 of figure 2. This execution of the program is called the *epoch-parallel* execution, as it allows epochs to run in parallel on separate cores. Each epoch is isolated in its own address space, and runs on a single CPU. The program's semantics are equivalent to a sequential execution: e.g., writes to local memory or files in one epoch are visible to epochs that logically occur later.

While the epoch-parallel execution satisfies Property 1 and, additionally, allows us to scale uniprocessor execution, we must still address how to start later epochs before prior ones complete. Because of program dependencies, we must *predict* the starting state of later epochs (for example, in figure 2, the starting states of Ep 1-7). To do this, uniparallelism executes a second copy of the application. This copy of the program produces predictions of starting state, called *checkpoints*, that include relevant program state, such as the address space and thread registers, that are necessary to start later epochs.

In order for uniparallelism to provide scalable uniprocessor execution, three properties must be true. First, the checkpoints must be generated by the second execution of the program before the epoch completes in the epoch-parallel execution. In our system, the execution generating checkpoints runs ahead of the epoch-parallel execution because its threads execute simultaneously on multiple cores. For that reason, the second execution is called the *thread-parallel* execution. (Figure 2 shows the thread-parallel execution of an application with 4 threads on CPUs 0-3.) The thread-parallel execution produces a checkpoint at the end of an epoch that is used to start the next epoch of the epoch-parallel execution.

Second, the system must detect and recover when the state predicted by the thread-parallel execution is incorrect. This is done by comparing the memory and register state of the two executions at epoch boundaries. If the predicted ending state for an epoch (generated by the thread-parallel execution) does not match the actual ending state (generated by the epoch-parallel execution), the thread-parallel execution has generated a checkpoint that exposes non-region serializable behavior. The system must flush the pipeline of any epochs that depend on the incorrect prediction, and the thread-parallel execution must restart from the last committed state (this is called a *rollback*). If the checkpoint and ending state match, the execution up to that epoch is a valid region-serializable execution of the program.

Third, the state predictions generated by the thread-parallel execution must *usually* be correct. While incorrect predictions do not affect correctness, they do reduce performance. Each rollback causes the pipeline to be flushed, which reduces parallelism and wastes the CPU time spent on the flushed epochs. To ensure the accuracy of the predicted state, uniparallelism uses *online replay* [22] to keep the thread- and epoch-parallel executions as similar as possible. The thread-parallel execution logs most sources of nondeterminism, and the logged values are replayed during the epoch-parallel execution. Logged events include all synchronization operations (e.g., all `pthread` operations) and input received from system calls (e.g., disk and network I/O). All synchronization operations are replayed in a way that obeys the happens-before partial order of synchronization objects observed during recording. Because both executions use the same happens-before order of synchronization objects, shared-memory accesses can cause the executions

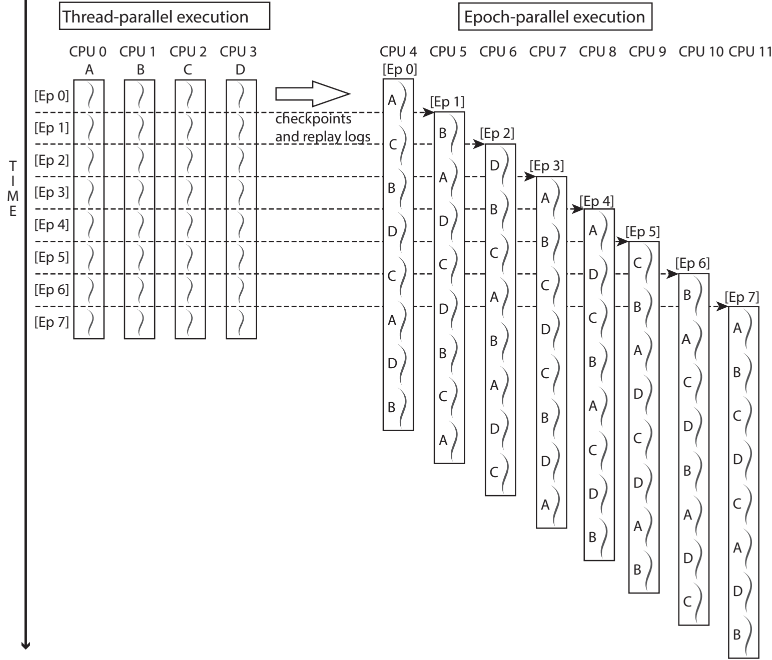


**Figure 2.** Uniparallelism overview.

to diverge only when there is a data race [41] that exposes non-RS state.

Thus, as shown in Figure 2, uniparallelism runs the application twice. The epoch-parallel execution guarantees region-serializable semantics by time-slicing threads within each epoch on a uniprocessor, while the thread-parallel execution runs ahead to predict future epoch states, thus enabling multiple epochs in the epoch-parallel execution to run in parallel.

## 5. Evaluation

We built a system that provides region serializability using the above design.

To provide Property 1, we modified the Linux scheduler to block all-but-one thread for each address space. On a page fault or other exception in a synchronization-free region, the kernel may choose to run a thread from a different address space, but it cannot run another thread from the same address space.

To provide Property 2, we modified the Linux scheduler so it can preempt threads only at boundaries of synchronization-free regions (i.e., at system calls and synchronization operations).

To provide Property 3, We used an LLVM compiler that has been explicitly modified to support DRF0 [31]. To prevent the compiler from reordering across region boundaries, the initial compiler pass inserts explicitly memory fences before and after every `atomic` and `volatile` memory access and `pthread` operation. Because the LLVM compiler already does not optimize across system calls, these fences are sufficient to preserve source-level regions.

To provide scalability, we used the DoublePlay infrastructure [46], which splits a process into epochs and runs those epochs on multiple processors using uniparallelism.

### 5.1 Methodology

For our evaluation, we used an 8-core Intel Xeon processor running CentOS Linux version 5.3. Our system used modified versions of the Linux 2.6.26 kernel, the GNU glibc library version 2.5.1, and the LLVM compiler. We evaluated our system on 3 SPLASH-2 [47] benchmarks (water-nsquared, ocean, and lu), two parallel desktop applications (pfscan and pbzip2), and Apache v2.2.11. For pfscan, we performed repeated searched for a query string on files with a total size of 983 MB. For pbzip2, we compressed a 522 MB file. Finally, we evaluated our system with Apache version 2.2.11. We used the Apache benchmarking tool, ab, to request a 17MB file 100 times over a local network. To measure the runtime of each application, we ran each experiment a total of five times and report the mean of these values in figures 3 and 4.

### 5.2 Scalability

Figures 3 and 4 show the runtime cost of providing region serializability with and without spare cores, respectively. Each bar in the figures shows the normalized execution time of the program, running with the specified number of worker threads (we do not count control threads that perform little computation). The baseline is the execution time of the same

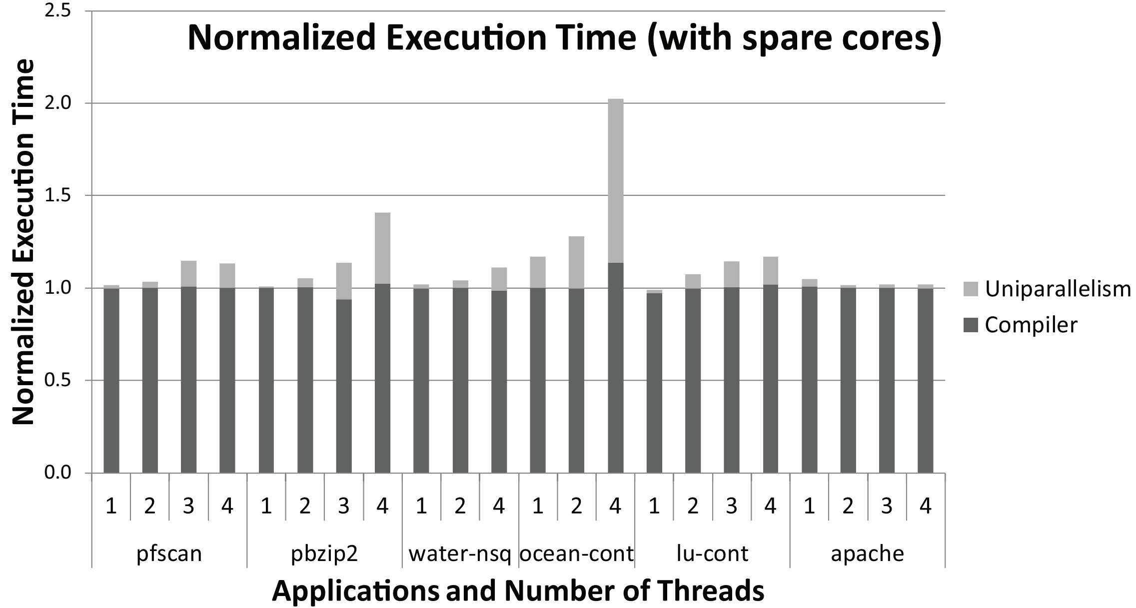


**Figure 3.** Performance with spare cores. This graph shows the cost of providing region serializable semantics when there are spare cores available. Each bar represents the normalized runtime of the application with the specified number of worker threads, relative to the runtime of the application compiled with an unmodified LLVM compiler, with all optimizations enabled, using the same number of worker threads. The darker portion of the bar shows the performance impact the compiler, which was modified to prevent optimizations across `pthread` operations. The lighter bar shows the additional cost of our uniparallel runtime. We show the mean of 5 trials for each of the experiments. The average overhead is 4%, 8%, 11%, and 31% for 1, 2, 3, and 4 threads, when there are spare cores available.

application compiled with an unmodified LLVM compiler and all optimization enabled, running the same number of threads, without our uniparallel runtime. The darker bottom portion of each bar shows the performance impact of our modified compiler. The lighter top portion of each bar shows the overhead of our uniparallel runtime.

Our system is based on the DoublePlay infrastructure and our performance costs are largely the same. For a detailed discussion of specific application performance, we refer the reader to [46]. Additionally, the impact of the LLVM compiler was negligible, and manual inspection of the LLVM byte-code indicated that LLVM does not perform cross-boundary optimizations for these applications, which is not surprising, since LLVM currently performs few optimizations across function calls, due to the difficulty of validating the correctness of interprocedural analyses.

As shown in figure 3, when spare cores are available on our system, the mean overhead of providing region serializability is 4%, 8%, 11%, and 31% for 1, 2, 3, and 4 cores respectively.

Most applications scale well with our system. In addition to the the 2x utilization cost, the uniparallel runtime incurs overhead to keep its two executions in-sync. The runtime must record and replay non-deterministic events, such as system calls and synchronization operations. Additionally, the runtime must perform a memory comparison at the end of each epoch, to check whether a data race has caused a memory divergence. Of all the applications, `ocean` has highest overhead at 4 threads. This behavior has been observed in other systems using uniparallelism [46] and is due to the application touching a large amount of memory, all of which must be compared at the end of each epoch, to check for divergences caused by data races. (`pbzip2` also has higher overhead due to data races that cause memory divergences, which we discuss in section 5.3.)

The availability of spare cores greatly impacts the performance of our system. When the application can scale to use all of the available cores, our system incurs at least a 100% runtime overhead due to the utilization costs of uniparallelism. Because uniparallelism requires a second execution, we are effectively halving the CPU resources available to an application that can scale to all available CPUs. Figure 4 shows that without spare cores, providing region serializable semantics has an average 134% overhead. `Apache` continues to scale, even at 8 threads, because the application is network-bound.

### 5.3 Strong semantics for programs with races

The goal of our system is to provide region serializability to all programs, including those with data races. If a data race

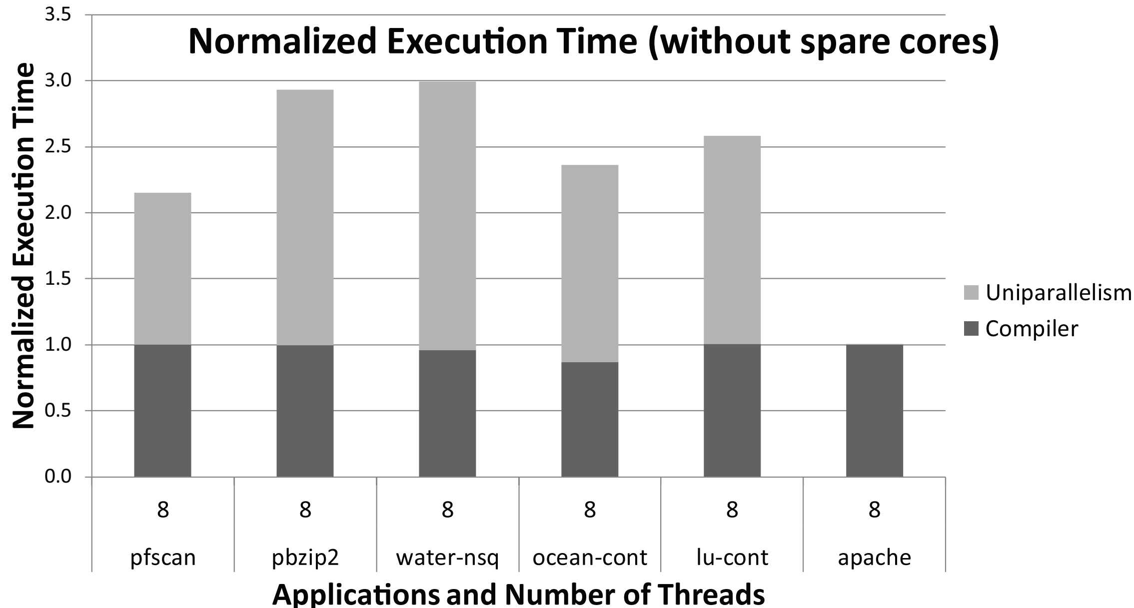


**Figure 4.** Performance overhead without spare cores. This graph shows the cost of providing region serializability when applications can scale to use all 8 cores. Each bar represents the normalized runtime of the application with the 8 worker threads, relative to the runtime of the application compiled with an unmodified LLVM compiler, with all optimizations enabled, also running with 8 worker threads. The darker portion of the bar shows the performance impact the compiler, which was modified to prevent optimizations across `pthread` operations. The lighter bar shows the additional cost of our uniparallel runtime. We show the mean of 5 trials for each of the experiments. The average overhead, when no spare cores are available is 134%.

occurs, the state predicted by the thread-parallel execution may be wrong, and this will lead to a rollback and slower performance. Note that rollbacks only occur when the data race occurs and results in a state that differs from the epoch-parallel execution. No overhead is incurred if the race is never triggered, or if the race does not affect program state.

Many of the applications used in our evaluation contain data races [45], but only `pbzip2` experiences rollbacks. `pbzip2` experiences 1, 3, and 6 memory divergences on average for 2, 3, and 4 threads, which results in an overhead of 5%, 14%, and 41%. As always, our system guarantees that the racy execution is still well-behaved with region serializable semantics.

### 5.4 Region sizes

One of the benefits of region serializability is that it allows programmers and tools to reason about the interleavings of atomic code regions, instead of individual memory instructions. To quantify this benefit, we measured dynamic size of each synchronization-free region. We created a Pin tool [29] to intercept all system calls and `pthread` library calls and count the number of user-level machine instructions in each region as the programs execute. (We did not count instructions within `pthread` functions.) As shown in Table 1, the average size of a synchronization-free region ranges from 50 to 200 instructions, allowing programmers and tools to reason about program behavior at a granularity that is at least an order of magnitude larger than that of underlying machine instructions.

**Dynamic Region Sizes**

| Application | Mean (instructions) | Median (instructions) |
|---|---|---|
| pfscan | 212 | 129 |
| pbzip2 | 104 | 81 |
| water-nsq | 138 | 111 |
| ocean-cont | 134 | 51 |
| lu | 55 | 11 |

**Table 1.** Dynamic region sizes

## 6. Related Work

To the best of our knowledge, this paper describes the first software system to guarantee region serializability (RS) at the language level for all multithreaded programs (including programs with data races) on commodity hardware. In section, we discuss work related to providing language-level guarantees for concurrent programs.

### 6.1 Sequential consistency

Language-level sequential consistency (SC), which is weaker than RS, has long been accepted as a desirable semantics for concurrency languages, but is also widely believed to be practically impossible to support with reasonable efficiency. Marino et al. [32, 43] showed that a compiler could preserve SC for a low performance cost. However, the end-to-end semantics would still depend on the weakest memory model supported by hardware, and no modern processors provides SC guarantees today.

### 6.2 DRF0 with data race detection

The most popular and widely adopted concurrency semantics is the DRF0 memory model. DRF0 and its derivatives [1, 6, 17, 30] provide RS semantics for data-race-free programs, but provide no [6] or extremely complicated semantics [30] for programs with data races. If a sound static data race detector could be developed, then the compiler could reject programs with data races and enforce the race-free discipline assumed by the DRF0 memory model. While many static type systems have been proposed (e.g., [7, 8, 15, 39]), static solutions must be conservative in their analysis and can impose severe restrictions on programming styles, as well as reject valid programs due to imprecise analysis, such as pointer analysis. Another approach argues for always-on dynamic data race detection that halts program execution when a race is detected [4, 13, 27, 28, 31]. However, detecting races at runtime can incur more than an 8x performance penalty [14], or require custom hardware support [2, 27, 31, 35, 40]. Further, legacy software contains a number of data races that are deliberately used by programmers to achieve high performance [36].

### 6.3 Transactional memory

Perhaps the most closely related approaches to our work are the studies on transactional memory (TM) systems [19, 25]. One way of viewing region serializability is as a transaction memory system that provides strong isolation [18, 33], where all code belongs to a transaction. The strongest guarantee discussed in the context of a TM system is Transactional Sequential Consistency (TSC) [10]. TSC guarantees that all transactions and all memory operations outside transactions appear to have executed in a global order that is consistent with per-thread semantics. In practice, however, TSC is believed to be too expensive to support for all programs [18].

### 6.4 Speculative parallelism

Another category of related work are runtime systems that use speculative parallelism. This style of execution was first proposed by Zilles and Sohi [49], who called it *master-slave speculative parallelization*, and has since been used in other research projects [37, 44, 49]. Uniparallelism was first proposed by Veeraraghavan [46] and combines a speculatively parallel execution style with online deterministic replay [22] and timesharing of multiple threads on a uniprocessor [9]. The uniparallel execution style has also been used for the race detection system, Frost [45].

## 7. Future Work

We plan to apply and extend region serializability in several ways, including increasing the size of regions, program verification, deterministic execution, and transactional memory. Currently, regions in our system are bounded by synchronization operations and system calls. It may be possible to increase the size of a region to span multiple synchronization-free regions [12], which would allow programmers and tools to reason about even larger atomic sections of code. Program verification tools try to limit the space of states they must consider by assuming knowledge of atomic regions. These tools can take advantage of the region serializability provided by our system to further limit their state space and reason further about the properties of the program. Deterministic execution of multithreaded programs is a challenging and important feature [3, 11, 24, 38]. In addition to providing serializability for regions, we can also provide deterministic execution of regions by controlling the order of execution between regions and the behavior of synchronization operations. While we have applied region serializability only to programs based on locks, we also hope to experiment with how well our system can support programs based on transactional memory, which provides similar guarantees to region serializability.

## 8. Conclusion

Current systems do not provide meaningful concurrency semantics to programs with data races. The absence of concurrency semantics makes it impossible for programmers and software tools to reason about the behavior of programs with data races and it admits the possibility of arbitrary behavior for the numerous programs used in production that contain data races. In this paper, we argued that strong semantics, namely *region serializability*, should be provided for all programs, and we showed one way of providing it at reasonable performance overhead with a custom runtime system and compiler.

## References


[1] Adve, S. V., and Hill, M. D. Weak ordering—a new definition. In *Proceedings of the 17th International Symposium on Computer Architecture* (1990), pp. 2–14.

[2] Adve, S. V., Hill, M. D., Miller, B. P., and Netzer, R. H. B. Detecting data races on weak memory systems. In *Proceedings of the 18th International Symposium on Computer Architecture* (1991), pp. 234–243.

[3] Berger, E. D., Yang, T., Liu, T., and Novark, G. Grace: Safe multithreaded programming for C/C++. In *Proceedings of the International Conference on Object Oriented*

*Programming Systems, Languages, and Applications* (Orlando, FL, October 2009), pp. 81–96.

[4] BOEHM, H.-J. Simple thread semantics require race detection. In *FIT session at PLDI* (2009).

[5] BOEHM, H.-J. How to miscompile programs with "benign" data races. In *Proceedings of the 2011 USENIX Workshop on Hot Topics in Parallelism* (May 2011).

[6] BOEHM, H.-J., AND ADVE, S. Foundations of the c++ concurrency memory model. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation* (2008), pp. 68–78.

[7] BOYAPATI, C., LEE, R., AND RINARD, M. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of OOPSLA* (2002).

[8] BOYAPATI, C., AND RINARD, M. A parameterized type system for race-free Java programs. In *Proceedings of OOPSLA* (2001), ACM Press, pp. 56–69.

[9] CODD, E. F., LOWRY, E. S., MCDONOUGH, E., AND SCALZI, C. A. Multiprogramming STRETCH: feasibility considerations. *Communications of the ACM 2*, 11 (November 1959), 13–17.

[10] DALESSANDRO, L., AND SCOTT, M. L. Strong isolation is a weak idea. In *TRANSACT '09: 4th Workshop on Transactional Computing* (feb 2009).

[11] DEVIETTI, J., LUCIA, B., CEZE, L., AND OSKIN, M. DMP: Deterministic shared memory multiprocessing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (March 2009), pp. 85–96.

[12] EFFINGER-DEAN, L., BOEHM, H.-J., CHAKRABARTI, D., AND JOISHA, P. Extended sequential reasoning for data-race-free programs. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*.

[13] ELMAS, T., QADEER, S., AND TASIRAN, S. Goldilocks: A race and transaction-aware Java runtime. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation* (2007), pp. 245–255.

[14] FLANAGAN, C., AND FREUND, S. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation* (Dublin, Ireland, June 2009), pp. 121–133.

[15] FLANAGAN, C., AND FREUND, S. N. Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (2000), pp. 219–232.

[16] FLANAGAN, C., FREUND, S. N., AND QADEER, S. Thread-Modular Verification for Shared-Memory Programs. In *Proceeding of the 2002 European Symposium on Programming Languages and Systems (ESOP)* (2002).

[17] GHARACHORLOO, K., LENOSKI, D., LAUDON, J., GIBBONS, P., GUPTA, A., AND HENNESSY, J. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture* (1990), pp. 15–26.

[18] HARRIS, T., LARUS, J. R., AND RAJWAR, R. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[19] HERLIHY, M., AND MOSS, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture* (May 1993), pp. 289–300.

[20] LAMPORT, L. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers C-28*, 9 (September 1979), 690–691.

[21] LAMPORT, L., AND SCHNEIDER, F. B. Pretending atomicity. Tech. Rep. DEC SRC 44, Digital Equipment Corporation, May 1989.

[22] LEE, D., WESTER, B., VEERARAGHAVAN, K., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. Respec: Efficient online multiprocessor replay via speculation and external determinism. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, PA, March 2010), pp. 77–89.

[23] LIPTON, R. J. Reduction: a method of proving properties of parallel programs. *Communications of the ACM 18*, 12 (December 1975), 717–721.

[24] LIU, T., CURTSINGER, C., AND BERGER, E. D. Dthreads: efficient deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011), pp. 327–336.

[25] LOMET, D. B. Process structuring, synchronization, and recovery using atomic actions. In *Proceedings of an ACM conference on Language design for reliable software* (1977), pp. 128–137.

[26] LU, S., PARK, S., SEO, E., AND ZHOU, Y. Learning from mistakes — a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (2008), pp. 329–339.

[27] LUCIA, B., CEZE, L., AND STRAUSS, K. Colorsafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *Proceedings of the 37th International Symposium on Computer Architecture* (Saint-Malo, France, 2010), pp. 222–233.

[28] LUCIA, B., CEZE, L., STRAUSS, K., QADEER, S., AND BOEHM, H.-J. Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-Races. In *Proceedings of the 37th International Symposium on Computer Architecture* (June 2010), pp. 210–221.

[29] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation* (Chicago, IL, June 2005), pp. 190–200.

[30] MANSON, J., PUGH, W., AND ADVE, S. The Java memory model. In *Proceedings of POPL* (2005), pp. 378–391.

[31] Marino, D., Singh, A., Millstein, T., Musuvathi, M., and Narayanasamy, S. DRFx: A simple and efficient memory model for concurrent programming languages. In *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation* (2010), ACM, pp. 351–362.

[32] Marino, D., Singh, A., Millstein, T. D., Musuvathi, M., and Narayanasamy, S. A case for an sc-preserving compiler. In *Proceedings of the ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation* (2011), pp. 199–210.

[33] Martin, M. M. K., Blundell, C., and Lewis, E. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters 5*, 2 (2006).

[34] Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P. A., and Neamtiu, I. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation* (San Diego, CA, December 2008), pp. 267–280.

[35] Muzahid, A., Suarez, D., Qi, S., and Torrellas, J. SigRace: Signature-based data race detection. In *Proceedings of the 36th International Symposium on Computer Architecture* (2009).

[36] Narayanasamy, S., Wang, Z., Tigani, J., Edwards, A., and Calder, B. Automatically classifying benign and harmful data races using replay analysis. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation* (San Diego, CA, June 2007).

[37] Nightingale, E. B., Peek, D., Chen, P. M., and Flinn, J. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, March 2008), pp. 308–318.

[38] Olszewski, M., Ansel, J., and Amarasinghe, S. Kendo: efficient deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (March 2009), pp. 97–108.

[39] Pratikakis, P., Foster, J. S., and Hicks, M. LOCKSMITH: Context-sensitive correlation analysis for race detection. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation* (2006), pp. 320–331.

[40] Prvulovic, M., and Torrelas, J. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings of the 30th International Symposium on Computer Architecture* (San Diego, CA, June 2003).

[41] Ronsse, M., and De Bosschere, K. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems 17*, 2 (May 1999), 133–152.

[42] Sevcik, J., and Aspinall, D. On Validity of Program Transformations in the Java Memory Model. In *Proceedings of the 2008 European conference on Object-Oriented Programming (ECOOP)*.

[43] Singh, A., Narayanasamy, S., Marino, D., Millstein, T., and Musuvathi, M. End-to-End Sequential Consistency . In *Proceedings of the 2012 International Symposium on Computer Architecture* (June 2012).

[44] Süsskraut, M., Knauth, T., Weigert, S., Schiffel, U., Meinhold, M., Fetzer, C., Bai, T., Ding, C., and Zhang, C. Prospect: A compiler framework for speculative parallelization. In *Proceedings of the 2010 IEEE/ACM International Symposium on Code Generation and Optimization* (April 2010), pp. 131–140.

[45] Veeraraghavan, K., Chen, P. M., Flinn, J., and Narayanasamy, S. Detecting and surviving data races using complementary schedules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (Cascais, Portugal, October 2011).

[46] Veeraraghavan, K., Lee, D., Wester, B., Ouyang, J., Chen, P. M., Flinn, J., and Narayanasamy, S. DoublePlay: Parallelizing sequential logging and replay. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (Long Beach, CA, March 2011).

[47] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture* (June 1995), pp. 24–36.

[48] Xiong, W., Park, S., Zhang, J., Zhou, Y., and Ma, Z. Ad hoc synchronization considered harmful. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation* (2010), pp. 163–176.

[49] Zilles, C., and Sohi, G. Master/slave speculative parallelization. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)* (2002), pp. 85–96.