# The Case for Region Serializability

*Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy*
*University of Michigan*

## 1 Benefits of Region Serializability

It is difficult to write correct multithreaded code. This difficulty is compounded by the weak memory model [1] provided to multithreaded applications running on commodity multicore hardware, where there is not an easily understood semantics for applications containing data races. For example, the DRF0 memory model only guarantees sequential consistency to data-race free programs [2], and while the Java memory model does specify behavior for racy programs, these semantics are difficult to understand [4]. Without clear semantics, it is difficult to write, test, and debug multithreaded code running on commodity multicore hardware.

While sequential consistency is often regarded as the preferred semantics for buggy lock-based programs, it only guarantees a global ordering at the granularity of instructions, which is too fine a granularity to reason about thread interleavings – especially if there are data races. To reduce the burden of multithreaded programmer, the runtime system should serialize larger regions of code to reduce the number of thread interleavings and the compiler should guarantee it does not optimize across these region boundaries. Such a system provides *region serializable* semantics with respect to the original source code.

Increasing the granularity of serialization benefits multithreaded programming in multiple ways: it makes it easier for programmers and software testing tools to understand multithreaded code because there are fewer interleavings and it can make software more robust in production. Moreover, these benefits extend to *all* multithreaded code, including those containing data races.

Provided region serializability, programmers can think about multithreaded programs as large code regions executing serially, even if a program contains a data race. This makes it easier to understand buggy program behavior, since there are fewer possible interleavings of larger code regions than single instructions. Moreover, most programmers who write multithreaded applications naturally assume that their program behavior corresponds to some global order of thread interleavings. Region serializability makes racy programs that assume serial semantics more robust on commodity multicore hardware.

The regions provided by region serializability benefit not only programmer reasoning, but software testing and verification tools as well. While software testing and verficiation is hard even for sequential code, the state space explosion is further exacerbated when tools must undestand the underlying weak memory model and explore different thread interleavings in addition to different paths taken in a single thread. For a given multithreaded program program, as the sizes of code regions increase, the number of possible interleavings decreases. By providing region serializability, we allow tools to exercise fewer preemptions during testing and explore fewer states during verification.

During production, it is possible to prevent some concurrency bugs from manifesting by controlling the thread schedule and limiting the number of preemptions [6]. Because the region-serializable runtime executes large regions of code non-preemptively, it is less likely that an atomicity violation will manifest.

## 2 Providing Region Serializability

To provide region serializability, the runtime system can only preempt a thread at the beginning or end of a region and the compiler must not optimize across these region boundaries. This guarantees that the regions will correspond to constructs within the source code and that all updates performed by a region will appear atomic to all other threads.

### 2.1 Constructing Regions

In our discussion of region serializability, we have not defined precisely what constitutes a region. We observe there are multiple ways to define regions, illustrated in figure 1. For example, in sequential consistency, a region is a single instruction. Serialization can occur at a larger granularity, such as for *synchronization-free regions*. Prior systems [3, 5] have serialized synchronization-free regions and these are the largest possible regions for which it is always possible to find a valid serial ordering of regions.

However, the most desirable semantics would serialize the largest code region possible, beyond synchronization-free regions. *Critical sections* that span the outer-most lock/unlock pair in the code could be serialized, but it is easy to introduce deadlocks that did not exist in the original application when serializing critical sections, as shown in 2. Therefore, we must constrain the size of regions to the largest section of code that can execute without blocking (for a lock, I/O, etc.) to guarantee liveness. We can expand a region until we detect a cyclic

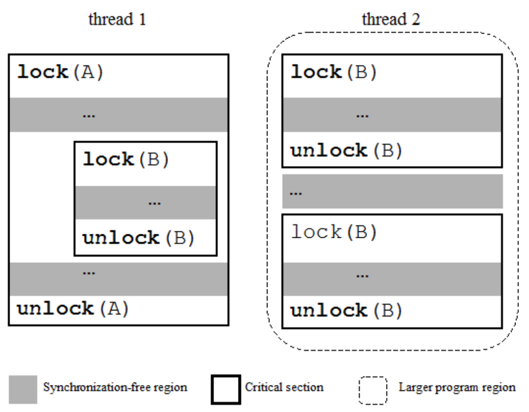

**Figure 1: Regions in Source Code.** Here, we illustrate 3 different types of regions that can be identified in source code. A synchronization-free region consists of code between synchronization operations. For example, in thread 1, the region of code between lock(A) and lock(B). A critical section is a region that begins with a lock acquisition and ends when all locks acquired since that point are released. In thread 1, a critical section begins when A is acquired and ends only when both A and B have been released. A code region larger than those defined solely by synchronization operations is shown in thread 2: it may be beneficial to programmers, analysis tools, and production systems to execute even larger code regions non-preemptively on a uniprocessor.

dependence between this and any other region, which effectively forces a preemption to prevent deadlock. We only end a region when we detect a cyclic dependence.

## 2.2 Executing Regions

Once the largest serializable regions are identified, the compiler cannot reorder instructions across these region boundaries and the runtime can only preempt a thread at a region boundary. One way to provide sequential semantics in the presence of potential data races is to run all threads on a single processor. Unfortunately, this reduces throughput to that of a uniprocessor and prevents the multithreaded application from scaling on multicore machines. To improve performance, we employ *uniparallel execution*, where the epoch-parallel execution running on a single core guarantees region serializable semantics and the thread-parallel execution speculatively parallelizes epochs to allow performance to scale with increasing number of cores [7]. Because we cannot guarantee region serializable semantics in the thread-parallel execution, we can only use the state it generates to speculatively start subsequent epochs before prior ones complete. We are able to detect if when the thread-parallel execution is incorrect using the state generated by the epoch-parallel execution once it finishes.

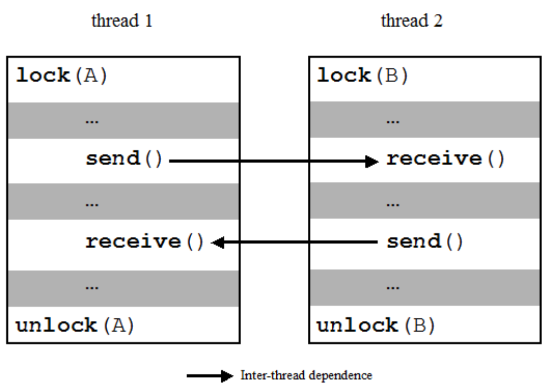

**Figure 2: Cyclic Dependency between Critical Sections.** Here, we show a cyclic dependency within a critical section caused by send() and receive() that prevents us from serializing the critical sections without introduction a deadlock. Note, however, that the synchronization-free regions can be serialized.

## References

[1] ADVE, S. V., AND GHARACHORLOO, K. Shared memory consistency models: A tutorial. *Computer 29*, 12 (1996), 66–76.

[2] BOEHM, H. J., AND ADVE, S. Foundations of the c++ concurrency memory model. In *Proceedings of PLDI* (2008), pp. 68–78.

[3] 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 2010 International Symposium on Computer Architecture* (June 2010), pp. 210–221.

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

[5] 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 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)* (June 2010), pp. 351–362.

[6] VEERARAGHAVAN, K., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. Surviving and detecting data races using complementary schedules. In *Proceedings of the 2011 Symposium on Operating Systems Principles (SOSP)* (October 2011).

[7] 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).