Efficient Software-Based Detection of Region Conflicts

Swarnendu Biswas
PhD Student, Department of Computer Science and Engineering
Ohio State University
biswass@cse.ohio-state.edu

1. Problem and Motivation

Data races are not only indicative of concurrency errors [12], they also present a fundamental barrier to writing correct shared-memory, multithreaded programs and complicate programming language specifications [13, 14]. Modern languages such as Java and C++ provide strong semantics—serializability of *synchronization-free regions* (SFRs)—in the absence of data races [5, 14]. However, they provide few or no guarantees if there is a data race [1, 4, 17]. The void between these two extremes motivates the need for strong semantics even for programs with data races. Providing strong semantic guarantees for data races will only become more important as software becomes more parallel to exploit the proliferation of increasingly parallel hardware. The complexity and risk associated with data races provide two main motivations for this work: (1) systems should provide strong data race semantics that programming languages can rely on, and (2) developers require efficient tools for detecting data races.

2. Background and Related Work

There is a long history of research into detecting data races [6, 8–10, 15]. These prior efforts have demonstrated a fundamental tradeoff between coverage (detecting as many data races as possible), precision (producing no false positives), and efficiency. Dynamic analyses that track the happens-before relation are precise and find all data races in an *observed* execution [6, 10, 15]. But even with clever optimizations [10], software happens-before data race detection imposes a high run time cost (e.g., 8.5X slowdown [10]) because it must track (1) "last access" information at every memory access and (2) the happens-before relation at every synchronization access. The cost of maintaining information about prior accesses is especially *expensive for mostly read-shared data*; updating metadata at concurrent reads trigger remote cache misses. Furthermore, the analysis must perform synchronization to ensure that happens-before race checks and metadata updates are atomic. Such high overheads of happens-before checking prohibit their use as a basis for programming language semantics and limit their use as a debugging tool [3, 15].

Detecting and fixing all data races seem intractable, and eliminating them entirely from current programming languages presents other impediments. A promising, recent approach to providing strong semantics for program executions with data races is to detect and throw so-called *data race exceptions* when a data race occurs [9, 13, 16]. Prior work avoids the expense of detecting all happens-before races by instead detecting conflicts between SFRs [11, 13, 16]. Every SFR conflict is a true data race, but not every data race is a conflict. As long as regions are full SFRs or larger [8, 13], these approaches provide a strong guarantee: if they do not detect a conflict, the execution is guaranteed to achieve serializability of its SFRs—the same guarantee provided by the DRF0 memory model for *data-race-free* executions [1, 5, 14]. However, existing region conflict detectors are impractical: they either rely on custom hardware support or slow programs substantially [8, 13].

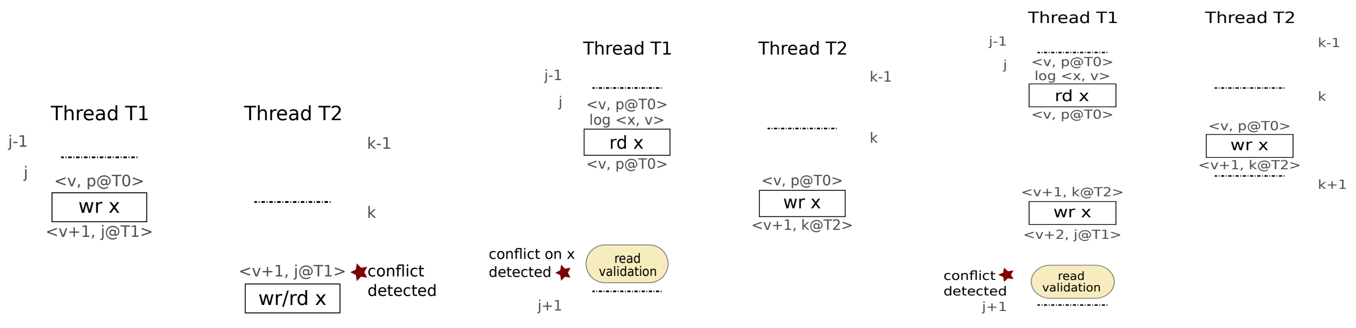
3. Valor: Mixing Eager and Lazy Conflict Detection

We have adapted prior work called *Conflict Exceptions* [13] to design a software-only conflict detector called *FastRCD* that eagerly detects conflicts among overlapping SFRs. FastRCD obviates the need for custom hardware, and reduces analysis overheads by using FastTrack's epoch optimizations for tracking read and write metadata. FastRCD provides strong guarantees similar to prior work [13]: it either throws a data-race exception at the precise point when a racy access is about to happen, or the execution guarantees serializability of SFRs. However, FastRCD continues to suffer from high overheads for similar reasons as sound happens-before detectors (Section 2) since it needs to perform expensive analysis at reads.

We now briefly overview our proposed software-based region-conflict detector called *Valor*.[1] Valor soundly and precisely detects each access that conflicts with another access in an ongoing region, that correspond to true data races. The key insight of Valor is to *elide* tracking of each shared variable's last reads, thus avoiding the high cost, incurred by existing analyses, of maintaining last reader information. This allows Valor to achieve high performance unlike FastTrack and FastRCD.

Valor *only* keeps track of each shared variable's last writer in the form of a tuple $\langle \mathtt{v}, \mathtt{c@t} \rangle$ that includes the epoch `c@t` of

[1] Valor is an acronym for Validating Anti-dependences Lazily On Release.



(a) (b) (c)

Figure 1: (a) Valor eagerly detects a conflict at T2's access because the last region to write x is ongoing. (b) Valor detects read–write conflicts lazily. During read validation, T1 detects that at least one write conflicts with x since T1 read x.

the last region c from thread t to write x, and a version v that the analysis increments whenever a new region writes to x. Tracking last writer allows Valor to detect write–write and write–read conflicts eagerly, as shown in Figure 1(a). The dashed lines in Figure 1 indicate region boundaries, and the labels j-1, j, etc. indicate a thread's clocks incremented at each region boundary. The grey text above and below each program memory access (e.g., $\langle$v, p@T0$\rangle$) shows shared variable x's last writer metadata. Since Valor does not track each shared variable's last readers, it cannot detect read–write conflicts *at* the conflicting write (shown in Figure 1(b)). Instead, each *region* maintains information about its reads in a read validation log, and detects read–write conflicts lazily when it *ends*. When a region ends, read validation compares each entry $\langle$x, v$\rangle$ in T1's read log with x's current version, in order to detect conflicts. In Figure 1(b), x's version has changed to v+1, so the analysis detects a read–write conflict. Figure 1(c) motivates the need for Valor to track *both* epoch and version to soundly and precisely detect read–write conflicts when there are remote write(s) interleaved before a write by the current thread.

Guarantees. If regions are at least as large as SFRs, then Valor provides a strong execution model that *either* reports a true data race or guarantees serializability of SFRs [13]. Since Valor detects read–write conflicts lazily it cannot provide precise exceptions, which is acceptable as long as the effects of potentially conflicting regions do not become externally visible. Thus, before a thread performs sensitive operations (for e.g., perform a system call), Valor validates the current region's reads so far. Other conflict and data race detectors have detected conflicts asynchronously and have provided similar guarantees [7, 16].

Detecting data races. Prior work has proposed detecting conflicts among SFRs to catch true data races [13]. Valor detects conflicts among *release-free* regions (RFRs)—regions which are bounded only by synchronization release operations. Since RFRs are always at least as large as an SFR, this design allows Valor to potentially detect more conflicts and thus true data races.

4. Results

We have implemented a prototype of Valor in Jikes RVM 3.1.3 [2]. For comparison purposes, we have implemented current state-of-art happens-before analysis called *FastTrack* [10] and FastRCD. For our evaluation, we used benchmarks from the DaCapo 2006 and 9.12-bach suite, and fixed-workload versions of SPECjbb2000 and SPECjbb2005.

We evaluate the data race coverage of Valor and compared with FastTrack and FastRCD, by collecting data races reported *at least once* in ten trials. Overall FastRCD and Valor detect fewer races than FastTrack, but can still expose 67% of the known data races. We and others [8, 13, 16] argue that region-conflict detection techniques detect the *more important* races, i.e., data races that can violate sequential consistency. We also evaluate the performance of the three implementations. FastTrack adds 340% overhead, whereas FastRCD introduces an overhead of 270%. In contrast, Valor incurs an overhead of only 108%. Thus, the overhead added by Valor is 3.2X less than FastTrack and 2.5X less than FastRCD, respectively. The overhead added by Valor is potentially low enough to enable its use in different alpha, beta and in-house testing environments and potentially to even certain production settings, thus enabling a wider outreach of using precise dynamic analysis tools to deal with data races.

5. Contributions

This work proposes the first software-based region conflict detector, called Valor, that has overheads low enough to be considered practical for providing programming language guarantees. The key insight behind Valor is that region conflict detectors need not track "last reader(s)" of shared variables. Tracking only the last write information and using lazy validation techniques for reads allows Valor to still soundly and precisely detect region conflicts. Valor guarantees that every execution will either report a data race or is region-serializable. We apply Valor to detect conflicts among release-free regions, which naturally extend to true data races. Our performance experiments show that Valor substantially outperforms current state-of-art.

References

[1] S. V. Adve and H.-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *CACM*, 53:90–101, 2010.

[2] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes Research Virtual Machine Project: Building an Open-Source Research Community. *IBM Systems Journal*, 44:399–417, 2005.

[3] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *CACM*, 53(2):66–75, 2010.

[4] H.-J. Boehm. Position paper: Nondeterminism is Unavoidable, but Data Races are Pure Evil. In *RACES*, pages 9–14, 2012.

[5] H.-J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. In *PLDI*, pages 68–78, 2008.

[6] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional Detection of Data Races. In *PLDI*, pages 255–268, 2010.

[7] J. Devietti, B. P. Wood, K. Strauss, L. Ceze, D. Grossman, and S. Qadeer. RADISH: Always-On Sound and Complete Race Detection in Software and Hardware. In *ISCA*, pages 201–212, 2012.

[8] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H.-J. Boehm. IFRit: Interference-Free Regions for Dynamic Data-Race Detection. In *OOPSLA*, pages 467–484, 2012.

[9] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A Race and Transaction-Aware Java Runtime. In *PLDI*, pages 245–255, 2007.

[10] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, pages 121–133, 2009.

[11] K. Gharachorloo and P. B. Gibbons. Detecting Violations of Sequential Consistency. In *SPAA*, pages 316–326, 1991.

[12] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *ASPLOS*, pages 329–339, 2008.

[13] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-J. Boehm. Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-Races. In *ISCA*, pages 210–221, 2010.

[14] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *POPL*, pages 378–391, 2005.

[15] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective Sampling for Lightweight Data-Race Detection. In *PLDI*, pages 134–143, 2009.

[16] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. DRFx: A Simple and Efficient Memory Model for Concurrent Programming Languages. In *PLDI*, pages 351–362, 2010.

[17] J. Ševčík and D. Aspinall. On Validity of Program Transformations in the Java Memory Model. In *ECOOP*, pages 27–51, 2008.