Valor: Efficient, Software-Only Region Conflict Exceptions

Swarnendu Biswas
PhD Student, Ohio State University
biswass@cse.ohio-state.edu

Abstract

Data races complicate programming language semantics, and a data race is often a bug. Existing techniques detect data races and define their semantics by detecting conflicts between synchronization-free regions. However, such techniques either modify hardware or slow programs dramatically, preventing always-on use today.

This work describes *Valor*, a sound, precise, software-only region conflict detection analysis that achieves high performance by eliminating the costly analysis on each read operation that prior approaches require. Valor instead logs a region's reads and *lazily* detects conflicts for logged reads when the region ends. As a comparison, we have also developed *FastRCD*, a conflict detector that leverages the epoch optimization strategy of the FastTrack data race detector.

We evaluate Valor, FastRCD, and FastTrack, showing that Valor dramatically outperforms both FastRCD and FastTrack. *Valor is the first region conflict detector to provide strong semantic guarantees for racy program executions with under 2X slowdown.* Overall, Valor advances the state of the art in always-on support for strong behavioral guarantees for data races.

This document includes improvements to our work since the PLDI 2015 SRC submission. Since the PLDI 2015 SRC, the work has been published at OOPSLA 2015 [3].

Keywords Conflict exceptions; data races; dynamic analysis; region serializability

1. Problem and Motivation

A *data race* occurs when two accesses to the same memory location are *conflicting*—executed by different threads and at least one is a write—and *concurrent*—not ordered by synchronization operations [8]. Data races are not only indicative of concurrency errors, they also present a fundamental barrier to writing correct shared-memory, multithreaded programs and complicate programming language specifications [13, 14]. Data races can lead to sequential consistency (SC), atomicity, and order violations that may corrupt data, cause a crash, or prevent forward progress. The Therac-25 disaster [12], the Northeastern electricity blackout of 2003 [20], and the mismatched NASDAQ Facebook share prices [16] were all due to race conditions, and are a testament to the danger posed by data races. Data races will only become *more* problematic as software systems become increasingly parallel in order to scale with parallel hardware.

A *memory consistency model* (or simply a memory model) defines the set of possible orders in which memory operations can interleave and the possible values returned by a read. Memory models of modern programming languages such as Java and C++ guarantee strong semantics for *data-race-free* executions—the execution is SC, i.e., the execution is equivalent to another execution where the instructions from different threads interleave according to program order [11]. This property in turn implies a much stronger guarantee where *synchronization-free regions* (SFRs) of code appear to execute atomically, i.e., the execution of SFRs is serializable [13]. The following figure shows an example of SFRs, which are dynamic regions[1] of code that are separated by synchronization operations (lock acquire/release, thread fork/join, volatile read/write, etc.) [13].

```
unlock(...);
...              SFR
unlock(...);
...
...              SFR
lock(...);
...              SFR
lock(...);
...              SFR
unlock(...);
```

However, these language memory models provide few or no guarantees *if* there is a data race [1]. For example, the behavior of a racy C++ program is undefined [6]. A recent study emphasizes the difficulty of reasoning about data races, showing that a C/C++ program with seemingly "benign" data races may behave incorrectly due to compiler transformations or architectural changes [5]. In contrast, Java's memory model preserves memory and type safety despite races, but permits non-SC behaviors [14]. Unfortunately, Java's safety guarantees preclude important compiler optimizations [18].

The complexity and risk associated with data races, and the lack of semantics for racy executions provided by programming languages like Java and C++ motivate this work. Our goal in this work is to develop a mechanism that equips present and future languages with clear, intuitive semantics *even for programs that permit data races*. Recent work gives *fail-stop* semantics to data races, treating a data race as an exception [13, 15]. Our work is motivated by these efforts, and our techniques also give data races fail-stop semantics.

[1] In this work, we use SFRs and regions interchangeably.

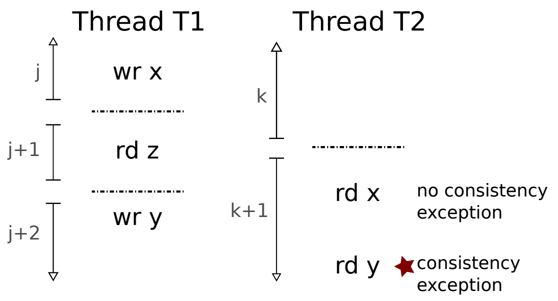


Figure 1: Under SFRSx, an execution generates an exception only for a data race that *may* violate SFR serializability.

2. Background and Related Work

There is a long history of research into detecting data races [7, 8]. These prior efforts are limited by a fundamental tradeoff between coverage (detecting as many data races as possible), precision (no false positives), and efficiency. Dynamic analyses that track the happens-before relation are precise and find all data races in an *observed* execution [8]. But even with clever optimizations [8], software happens-before data race detection imposes a high run time cost (e.g., 8.5X slowdown [8]) because it must track (1) the "last access" information at every memory access and (2) the happens-before relation at every synchronization operation. The cost of maintaining information about prior accesses is especially *high for mostly read-shared data*; updating metadata at concurrent reads trigger expensive 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.

Detecting and fixing all data races seem intractable, and eliminating them entirely from current programming languages presents other impediments. A promising, more recent approach to providing strong semantics for racy program executions is to detect and throw so-called *data race exceptions* when a data race occurs [13, 15]. Prior work called *Conflict Exceptions* (CE) avoids the expense of detecting all happens-before races by instead detecting conflicts between SFRs [13]. Every SFR conflict is a true data race, but not every data race is a conflict. CE executes a program and either reports an exception on a SFR conflict or ensures serializability of SFRs. We call this memory model *SFRSx*. Figure 1 shows an execution with data races on two shared variables, x and y. The dashed lines in Figure 1 indicate SFR boundaries, and the labels j, j+1, etc. indicate a per-thread SFR id which is incremented at each region boundary. An implementation of SFRSx does not generate a consistency exception at the read of x because the SFRs accessing x do not overlap. In contrast, SFRSx generates an exception at the read of y (because the SFRs accessing y overlap), to avoid violating SFR serializability (e.g., suppose Thread T1's SFR later writes to x or y). As long as regions are SFRs or larger [3, 7, 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 but *only* for data-race-free executions [1]. However, existing region conflict detectors are impractical: they either rely on custom hardware support [13] or slow programs substantially [7].

3. Efficient Software-Only Region Conflict Detection

The goal of this work is to develop a region conflict detection mechanism that is useful for providing guarantees to a programming language implementation, and is efficient enough for always-on use. Our work introduces a *novel, efficient region conflict detection technique called Valor*. For the purposes of comparison, we also propose an alternative conflict detector called FastRCD. Both FastRCD and Valor provide SFRSx.

3.1 FastRCD: Detecting Conflicts Eagerly

FastRCD is a new software-only dynamic analysis for detecting region conflicts. FastRCD reports a conflict when a memory access executed by one thread conflicts with a memory access that was executed by another thread in a region that is *ongoing*. The FastRCD algorithm extends the state-of-art sound and precise dynamic data race detection analysis called FastTrack [8]. In FastRCD, each thread keeps track of a clock that is incremented at every region boundary. This clock is analogous to the logical clocks maintained by FastTrack to track the happens-before relation. FastRCD uses epoch optimizations similar to FastTrack's optimizations for efficiently tracking read and write metadata. FastRCD keeps track of the last region to write each shared variable, and the last region or regions to read each shared variable.

FastRCD provides SFRSx, similar to CE [13] which needs custom hardware support: it either throws a serializability exception at the precise point when a racy access is about to happen, or the execution guarantees serializability of SFRs. But despite FastRCD being a natural extension of FastTrack, Section 4 experimentally shows that FastRCD's need to track last readers imposes overheads that are similar to FastTrack's and are *too high for always-on use*.

3.2 Valor: Mixing Eager and Lazy Conflict Detection

In response to FastRCD's high overhead, we develop ***Valor***,[2] *which is the main contribution of this work*. Valor is a novel, software-only region conflict detector that eliminates the costly analysis on read operations that afflicts FastRCD (and FastTrack). Like FastRCD, Valor reports a conflict when a memory access executed by one thread conflicts with a memory access previously executed by another thread in a region that is ongoing. Valor detects write–write and write–read conflicts *eagerly* as in FastRCD. The key insight in Valor is to *elide* tracking of each shared variable's last reads, thus *avoiding* the high cost, incurred by existing analyses,

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

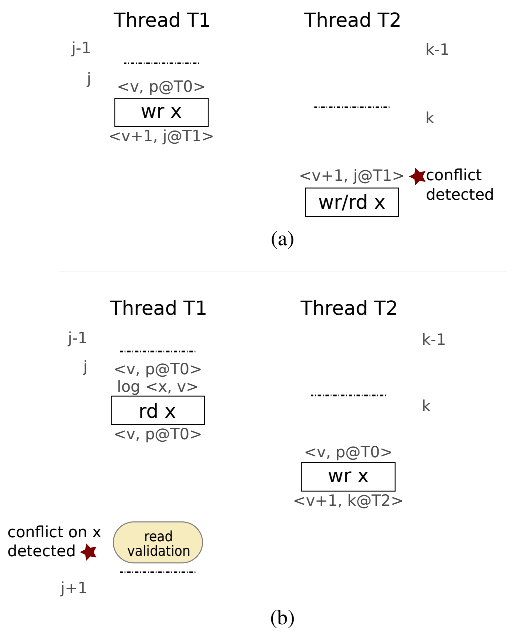


Figure 2: (a) Like FastRCD, Valor eagerly detects a conflict at T2's access because the last region to write x is ongoing. (b) *Unlike* FastRCD, Valor detects read–write conflicts lazily. During read validation, T1 detects a write to x since T1's read of x.

of maintaining last reader information. This allows Valor to achieve high performance unlike FastTrack and FastRCD. Eliding tracking of readers imply that Valor has to detect read–write conflicts *lazily*.

In Valor, each thread logs read operations locally. At the end of a region, the thread *validates* its read log, checking for read–write conflicts between its reads and any writes in other threads' ongoing regions. By *lazily* checking for these conflicts, Valor can provide fail-stop SFRSx semantics without hardware support and with overheads far lower than even our optimized FastRCD implementation.

Valor *only* keeps track of each shared variable's last writer in the form of a tuple ⟨v, c@t⟩ that includes the "epoch" c@t of 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 2(a). Similar to Figure 1, the labels j-1, j, etc. indicate a thread's clock that is incremented at each region boundary. The grey text above and below each program memory access (e.g., ⟨v, p@T0⟩) shows the shared variable x's last writer metadata before and after the access. Since Valor does not track each shared variable's last readers, it cannot detect read–write conflicts *at* the conflicting write (shown in Figure 2(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 ⟨x, v⟩ in T1's read log with x's current version, in order to detect conflicts. In Figure 2(b), x's version has changed to v+1, so the analysis detects a read–write conflict.

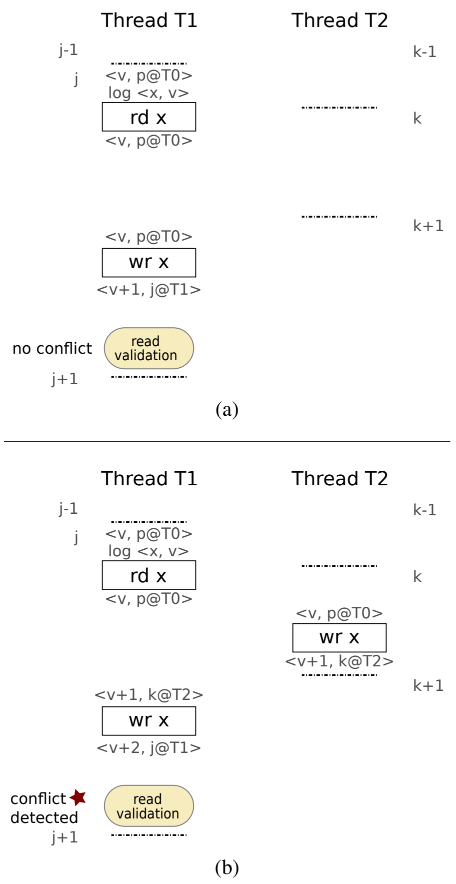
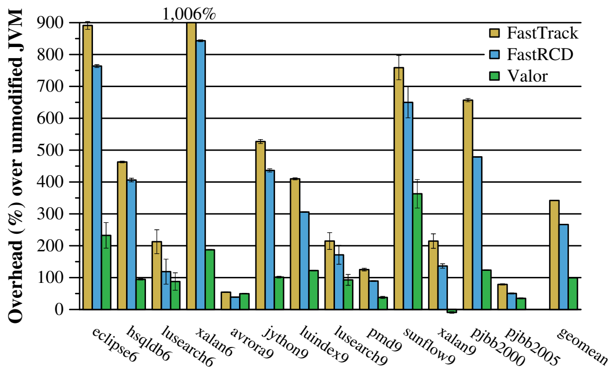


Figure 3: Valor relies on versions to detect conflicts soundly and precisely.

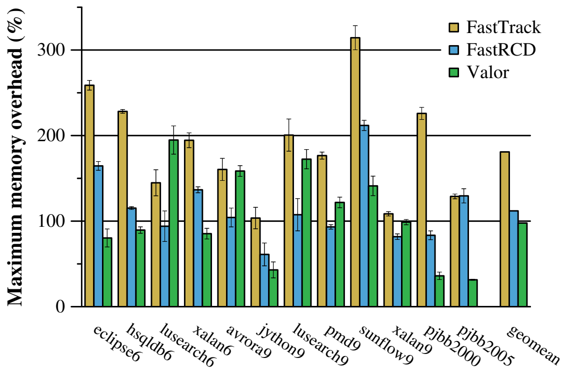
Figure 3 motivates the need for Valor to track *both* epoch and version information to soundly and precisely detect read–write conflicts when there are remote write(s) interleaved before a write by the current thread. Without tracking versions, it is challenging for T1 to infer during read validation whether there were any remote writes during the region (Figures 3(a) and 3(b)).

Similar to Valor, a few software transactional memory (STM) systems have used mixed strategies for detecting conflicts. In particular, McRT-STM [17] and Bartok-STM [10] detect write–write and write–read conflicts eagerly and read–write conflicts lazily. However, these STMs use more expensive techniques to validate reads differently from Valor. Another difference is that Valor must detect conflicts precisely, whereas STMs do not (a false conflict triggers an *unnecessary* abort and retry). As a result, STMs typically track conflicts at the granularity of objects or cache lines. More generally, STMs have not introduced designs that target region conflict detection or precise exceptions. In some sense, our work applies insights from STMs to the context of data race exceptions.

Implications of lazy conflict detection. 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.



(a) Run-time overhead



(b) Space overhead

Figure 4: Run-time and space overhead added to unmodified Jikes RVM by our implementations of FastTrack, FastRCD, and Valor.

A region that performs a read that conflicts with a later write can behave in a way that would be impossible in any unserializable execution. We refer to such regions as "zombie" regions, borrowing terminology from STM systems that experience similar issues by detecting conflicts lazily [9]. To prevent external visibility, Valor must validate a region's reads before all sensitive operations, such as system calls and I/O. Similarly, a zombie region might never end (e.g., might get stuck in an infinite loop), even if such behavior is impossible under any region serializable execution. To account for this possibility, Valor must periodically validate reads in a long-running region. Other conflict and data race detectors have detected conflicts asynchronously [19], providing imprecise exceptions and similar guarantees.

In an implementation for a memory- and type-unsafe language such as C or C++, a zombie region can corrupt memory and types arbitrarily. The situation is not so dire for Valor, which detects region conflicts in order to throw conflict exceptions, rather than to preserve region serializability. As long as a zombie region does not actually corrupt Valor's analysis state, read validation will be able to detect the conflict when it eventually executes—either when the region ends, at a system call, or periodically (in case of an infinite loop). Our implementation targets a safe language (Java), so a zombie region's possible effects are safely limited.

Extending regions. Valor extends regions to detect 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 and also amortizes the cost of read validation performed at region boundaries. We prove in our OOPSLA 2015 paper that RFR conflicts are true data races, and that Valor is a sound and precise region conflict detector [3].

There are useful analogies between RFR conflict detection and prior work. Happens-before data race detectors increment their epochs at release operations only [8], and some prior work extends redundant instrumentation analysis past acquire, but not release, operations [7].

4. Results

We have implemented a prototype of Valor in Jikes RVM 3.1.3 [2], a high-performance Java virtual machine (JVM) [3]. For comparison purposes, we have implemented current state-of-art happens-before analysis called *FastTrack* [8] and FastRCD. For our evaluation, we used benchmarks from the DaCapo 2006 and 9.12-bach suite [4], and fixed-workload versions of SPECjbb2000 and SPECjbb2005. We did our experiments on an AMD Opteron 6272 system with eight 8-core 2.0-GHz processors (64 cores total), running RedHat Enterprise Linux 6.6, kernel 2.6.32

For our experiments, both the conflict detectors use RFRs as regions. We evaluate different metrics to compare the three analyses: performance and space overhead, scalability, and data race coverage. Figure 4(a) shows the runtime overhead added over unmodified Jikes RVM by the three implementations. Each bar is the average of ten trials and has a 95% confidence interval centered at the mean. FastTrack adds 342% overhead, whereas FastRCD introduces an overhead of 267%. The primary result of our work is that *Valor incurs only 99% overhead on average, far exceeding the performance of any prior software-based conflict detection technique.*

Figure 4(b) shows the space overhead, relative to baseline (unmodified JVM) execution for the same configurations as in Figure 4(a). We measure an execution's space usage as the maximum memory used after any full-heap garbage collection (GC). We omit luindex9 since the unmodified JVM triggers no full-heap GCs, although each of the three analyses does. Unsurprisingly, FastTrack uses more space than FastRCD since it maintains more metadata. Valor sometimes adds less space than FastRCD; other times it adds more. This result is due to the analyses' different approaches for maintaining read information: FastRCD uses per-variable shared metadata, whereas Valor logs reads in per-thread buffers. On average, Valor uses less memory than FastRCD and a little more than half as much memory as FastTrack.

Our full paper includes additional performance and scalability overhead results [3]. For example, to measure the sen-

	FastTrack		**FastRCD**		**Valor**	
eclipse6	37	(46)	3	(7)	4	(21)
hsqldb6	10	(10)	10	(10)	9	(9)
lusearch6	0	(0)	0	(0)	0	(0)
xalan6	12	(16)	11	(15)	12	(16)
avrora9	7	(7)	7	(7)	7	(8)
jython9	0	(0)	0	(0)	0	(0)
luindex9	1	(1)	0	(0)	0	(0)
lusearch9	3	(4)	3	(5)	4	(5)
pmd9	96	(108)	43	(56)	50	(67)
sunflow9	10	(10)	2	(2)	2	(2)
xalan9	33	(39)	32	(40)	20	(39)
pjbb2000	7	(7)	0	(1)	1	(4)
pjbb2005	28	(28)	30	(30)	31	(31)

Table 1: Data races reported by FastTrack, FastRCD, and Valor.

sitivity to system architecture, we also evaluated the performance on a 32-core Intel Xeon E5-4620 system. The relative performance of the three techniques remain comparable. Valor continues to substantially outperform the other techniques. For the benchmarks that permit a variable number of threads, our scalability results show that all three techniques scale with an increasing number of threads [3].

FastTrack detects every data race in an execution. In contrast, Valor and FastRCD *focus* on supporting the SFRSx memory model, so they detect only region conflicts, not all data races. That said, an interesting question is how many data races manifest as region conflicts in typical executions, and how does FastRCD and Valor fare in terms of *coverage* compared to FastTrack? Table 1 shows the data race coverage of Valor and FastRCD and compares with FastTrack, by collecting data races reported *at least once* in ten trials. If the same race is detected multiple times in an execution, we count it only once. The first number for each detector is the average number of races (rounded to the nearest whole number) reported across 10 trials. The number in parentheses is the count of races reported at least once across all 10 trials. Overall, FastRCD and Valor detect fewer races than FastTrack. On average across the programs, one run of either FastRCD or Valor detects 58% of the true data races. Counting data races reported at least once across 10 trials, the percentage increases to 63% for FastRCD and 73% for Valor, respectively. Compared to FastTrack, FastRCD and Valor represent lower coverage, higher performance points in the performance–coverage tradeoff space. We note that FastRCD and Valor are able to detect *any* data race, because any data race can manifest as a region conflict [7]. We emphasize that although FastRCD and Valor miss some data races, the reported races involve accesses that are dynamically "close enough" together to jeopardize region serializability. We (and others [7, 13, 15]) argue that region conflicts are therefore more harmful than other data races, and it is more important to fix them.

5. Contributions

This work introduces two new software-based region conflict detectors, FastRCD and Valor. Valor substantially outperforms prior software-based conflict detection techniques. The key insight behind Valor is that detecting read–write conflicts lazily retains necessary correctness guarantees and has better performance than eager conflict detection. Valor's overhead is potentially low enough to use all-the-time conflict exceptions in various settings, from in-house testing to alpha and beta testing to potentially even some production systems. Overall, Valor represents an advance in the state of the art for providing strong semantic guarantees for racy executions.

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 et al. The Jikes Research Virtual Machine Project: Building an Open-Source Research Community. *IBM Systems Journal*, 44:399–417, 2005.

[3] S. Biswas, M. Zhang, M. D. Bond, and B. Lucia. Valor: Efficient, Software-Only Region Conflict Exceptions. In *OOPSLA*, pages 241–259, 2015.

[4] S. M. Blackburn et al. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*, pages 169–190, 2006.

[5] H.-J. Boehm. How to miscompile programs with "benign" data races. In *HotPar*, 2011.

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

[7] 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.

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

[9] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. Morgan and Claypool Publishers, 2nd edition, 2010.

[10] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing Memory Transactions. In *PLDI*, pages 14–25, 2006.

[11] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Computer*, 28:690–691, 1979.

[12] N. G. Leveson and C. S. Turner. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18–41, 1993.

[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, 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.

[16] PCWorld. Nasdaq's facebook glitch came from race conditions, 2012. `http://www.pcworld.com/article/255911/nasdaqs_facebook_glitch_came_from_race_conditions.html`.

[17] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime. In *PPoPP*, pages 187–197, 2006.

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

[19] A. Singh, D. Marino, S. Narayanasamy, T. Millstein, and M. Musuvathi. Efficient Processor Support for DRFx, a Memory Model with Exceptions. In *ASPLOS*, pages 53–66, 2011.

[20] U.S.–Canada Power System Outage Task Force. Final Report on the August 14th Blackout in the United States and Canada. Technical report, Department of Energy, 2004.