Data-Race Exceptions Have Benefits Beyond the Memory Model

Benjamin P. Wood Luis Ceze Dan Grossman
University of Washington
{bpw,luisceze,djg}@cs.washington.edu

Abstract

Proposals to treat data races as exceptions provide simplified semantics for shared-memory multithreaded programming languages and memory models by guaranteeing that execution remains data-race-free and sequentially consistent *or* an exception is raised. However, the high cost of precise race detection has kept the cost-to-benefit ratio of data-race exceptions too high for widespread adoption. Most research to improve this ratio focuses on lowering performance cost.

In this position paper, we argue that with small changes in how we view data races, data-race exceptions enable a broad class of benefits beyond the memory model, including performance and simplicity in applications at the runtime system level. When attempted (but exception-raising) racy accesses are treated as legal — but exceptional — behavior, applications can exploit the guarantees of the data-race exception mechanism by performing potentially racy accesses and guiding execution based on whether these potential races manifest as exceptions. We apply these insights to concurrent garbage collection, optimistic synchronization elision, and best-effort automatic recovery from exceptions due to sequential-consistency-violating races.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

General Terms Languages, Performance, Reliability

Keywords data-race exception, sequential consistency, concurrent garbage collection

1. Introduction

Among concurrency errors, data races are the most problematic when defining the semantics of programming languages and memory models for shared-memory multithreaded programs, since they may cause sequential consistency violations. Memory models for Java and C++ guarantee sequential consistency for data-race-free programs [5, 18], but semantics are murkier in the presence of races. The Java Memory Model gives racy programs weaker semantics that retain security and safety guarantees, but its complexities have resulted in subtle bugs despite extensive design effort [30]. C++ simply leaves semantics of racy programs undefined.

As consensus that sequential consistency for concurrent programs is highly desirable — if not indispensable — grows, researchers have proposed treating data races as exceptions to simplify memory models [2, 8, 11, 17, 19]. Such exceptions vary in strength, but all guarantee that execution either maintains sequential consistency or raises an exception. However, the cost-to-benefit ratio of precise data-race exception support has been too high for widespread adoption. Most ongoing research to improve this cost-to-benefit ratio focuses on improving performance [12, 17, 19, 28]. *In this position paper, we argue that with small changes in how we view data races, data-race exceptions enable a broad class of benefits beyond the memory model, including performance and simplicity in concurrent runtime systems.*

We propose that executions with data races need not be inherently wrong, so long as these races are never allowed to manifest. Rather, exception support should banish the *harmful effects* of races by raising an exception whenever a racy or sequentially inconsistent access is attempted, while enabling runtime systems or programs to react to *useful and timely information* on an attempted racy access and continue safe, sequentially consistent execution instead of halting. By treating *attempted* data races or sequential consistency violations as exceptional, but legal, behavior, we can implement runtime systems *on top of* data-race exception support, using it for general and precise conflict detection without the need for the cost or complexity of specialized rollback support. We explore this idea as follows:

Section 2 gives a brief overview of data races and what we require of data-race exceptions. We demonstrate the usefulness of data-race exceptions by sketching implementations of concurrent garbage collection (Section 3) and optimistic synchronization (Section 4) that use data-race exception support to implement conflict detection. We also outline extensions to conventional race detection to support these runtime systems. Section 5 describes *conflict races*, a principled sub-

MSPC'11, June 5, 2011, San Jose, California, USA.

set of data races that is cheaper to detect, and outlines uses of conflict-race exceptions as compared to data-race exceptions in runtime systems. Finally, Section 6 concludes.

2. Data Races and Exceptions

A *data race* is a pair of memory accesses to the same memory location by two different threads, where at least one access is a write and neither access *happens before* the other. The *happens-before* relation over operations in a program execution is defined by the transitive closure of the *program order* within each thread and the *synchronization order* across synchronization operations in different threads.

2.1 Properties of Data-Race Exceptions

A *data-race exception* (DRE) is an exception raised when a thread attempts a racy access. DRE support simplifies memory models by guaranteeing:

- **data-race freedom or an exception**, preventing unreported racy executions.

Since modern systems ensure that data-race freedom implies sequential consistency, sound-and-complete DRE support also guarantees "sequential consistency or an exception," which is generally as useful for simplifying memory models.

Recoverability To be more broadly useful, DREs must also be *recoverable*, meaning they are:

- **precise**, suspending execution of the thread immediately on the racy access, without allowing the thread to complete the racy access or any subsequent operation; and
- **handleable**, taking the form of a trap that supplies an exception handler with information about the race.

Recoverable DREs can be caught and execution can continue without breaking sequential consistency.

Atomicity Data-race detection providing "data-race freedom or a precise DRE" guarantees that exception-free regions of a thread's execution containing no synchronization (and, in fact, larger regions) are atomic. This is the key property that makes DRE support useful for conflict detection.

Implementations Static data-race detectors (*e.g.*, [1, 6, 21]) guarantee data-race-freedom at compile time, but they are too conservative for some reasonable data-race-free programs, and our goal is to exploit races not present in the original program. Goldilocks [11] was the first system to propose and provide DREs. It implements *sound* (catch all races) and *complete* (catch only real races) dynamic race detection in a JVM, but its performance is limiting. Even state-of-the-art precise software race detection costs roughly 10x overhead [12]. Existing hardware race detectors (*e.g.*, [25, 31]) are unsound, lacking the guarantees memory models require.

Beyond the requirements of precise and recoverable exceptions, the exact implementation of data-race exception support is not important in this paper. For presentation purposes we assume it involves hardware support.

3. Concurrent Garbage Collection

Concurrent garbage collectors address the scalability problems of stop-the-world GC pauses by collecting concurrently with the execution of mutator threads. The challenge for concurrent GCs is to obtain a consistent view of the heap while mutators update it. Concurrent GCs insert *read barriers* or *write barriers* before each mutator access to keep the GC's view of the heap sufficiently consistent with the actual heap. This approach works, but requires extra work on every mutator access, at least during collection. Furthermore, it requires careful reasoning about the memory ordering semantics of the architecture, since adding full synchronization in mutators is to be avoided at all costs. Concurrent compaction, where objects are moved to reduce fragmentation while mutators continue to execute, is an even harder problem.

When mutators *and* the GC run on top of DRE support, exceptions on mutator-GC races capture those accesses where barriers actually perform useful work. Moving barrier work to exception handlers, we can reduce mutator performance overheads and reason less about memory ordering.

In the rest of this section we sketch implementations of concurrent mark-sweep heap tracing and concurrent object copying using DRE support. Then, we outline extensions to DREs support to enable these implementations. Finally, we discuss issues in using DRE support to detect program races in the presence of GC.

3.1 Concurrent Mark-Sweep

Mark-sweep GCs walk the heap to find all reachable objects, generally using *tri-color marking* [10]. Objects are initially marked white, meaning they have not been shown to be reachable. Objects marked gray are reachable, but objects they reference may not be marked yet. Objects marked black are reachable and all objects they reference have been marked. Collection starts by marking the roots gray. Marking proceeds until every object is either black or white, once all reachable objects have been visited. When a gray object is visited, it is marked black and any white objects to which it refers are marked gray. When no gray objects remain, all white objects are unreachable and may be collected. To prevent collection of reachable objects, we must ensure that no black-to-white references ever exist.

Figure 1 shows how concurrent heap updates by mutators could cause a naive mark-sweep GC to collect a reachable object. At (0) the collector visits node a, the head of a linked list, marking it black and marking b, referenced by a.`next`, gray. The third node in the list, c, remains white. Next, a mutator interleaves at (1), removing node b from the linked list, and breaking the no black-to-white references invariant. At (2), the collector dequeues b and, finding that b holds no non-null references, marks b black, and marking is complete. In the sweep phase, the collector frees the white c object at (3), even though it is reachable. Additionally, b, now unreachable, is not collected.

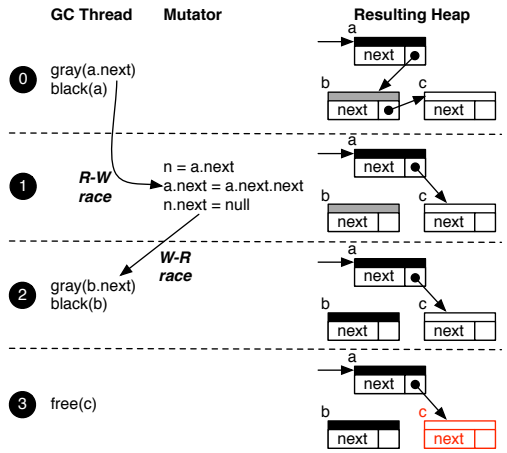

Figure 1. A naive mark-sweep collector collects reachable objects when mutators modify the heap during tracing.

3.1.1 Related Work

A conventional concurrent mark-sweep GC prevents this black-to-white reference problem with a write barrier that marks the target of the new reference gray to ensure the collector sees it is reachable [10]. An alternative is to mark the source, a, gray if it is already black [29], ensuring that the collector revisits it and finds the new reference to c. Other systems improve barrier efficiency by combining GC and STM barriers [20] (similar in spirit to our approach) or using specialized hardware support for barriers [9].

3.1.2 Implementation with DRE Support

Note that the update in Figure 1 that caused the GC to collect a reachable object includes a read-write data race between the GC and the mutator. Any potential black-to-white reference written during marking will raise an exception, since all reference fields of black objects, such as o.x, have been *read* by the GC thread and there is no synchronization between the GC and mutators during marking. Mutator reads do not raise exceptions due to the GC reads, since races include at least one write. Thus the check and marking performed by a conventional write barrier can be moved to DRE handlers for mutator updates and GC reads.

Exception in mutator: A mutator raises an exception on a write that races with a previous read by the GC, as in the race between (0) and (1) in Figure 1. The mutator marks the reference target gray if it raced with the GC. Races between mutators are treated normally. Next, the mutator should perform the access that raised the exception. However, it will still race with the GC's read until the next synchronization from the GC to the mutator (after the end of the marking phase), likely defeating the no-pause goal of concurrent mark-sweep, so it must interact closely with the race detector to allow the update. In Figure 1, when the mutator thread, which we name m_1, raises an exception attempting to update a.next at (1), it first marks c gray. Before attempting the update again, it erases the race detector's record of the GC's last read of a.next. It then updates a.next and replaces the GC's last read in the race detector, so subsequent mutator updates raise exceptions.

While the GC's last read is missing during m_1's handling of the exception, another mutator m_2 may write a new black-to-white reference in a.next. In this case, m_2 must have a real program race with m_1, since m_1 has not synchronized with m_2 since initially attempting the write. If m_2 writes *after* m_1 has successfully written its new value, but before the GC's last read is reinstalled, then m_2 will raise an exception on its potential black-to-white write. Otherwise, if m_2 writes *before* m_1, then m_1 will raise an exception on its write, which we handle by first reinstalling the GC's last read in the race detector and marking a (the source, not the target, of the reference in question) gray before raising the exception to the program. Since arbitrarily many mutators may interleave writes here, it is much simpler to mark a to be revisited than to determine the new target of a.next correctly in m_1.

Finally, once marking is done, exceptions raised due to mutator writes racing with GC reads are no longer useful. The GC should proceed to clear all its last reads so that subsequent mutator updates to that location will no longer race with the GC. This could be accelerated by support in the race detector to clear all last reads of a given thread (*i.e.*, the GC thread).

Exception in collector: When the GC raises an exception, it is on a write-read race with a mutator update, as in the data race between (1) and (2) in Figure 1. The GC must ensure that it sees the new value written by the mutator, but it does not need to perform any special marking. If write-read DREs also deliver the value written by the racing write, this is simple. Otherwise, the GC must wait until the mutator update has been published to do its read. In either case, we need the detector to allow and record the GC's read *even if it races with the last write* so that future mutator updates to this location will raise exceptions and perform marking.

3.2 Concurrent Object Copying

Copying and compacting GCs help reduce heap fragmentation, but concurrent copying is challenging: only a handful of fully concurrent copying GCs have been developed (*e.g.*, [20, 23, 24]). Copying GCs typically use a *forwarding pointer* in the header of each object to point to the current version of that object. Mutators always read the forwarding pointer in an object to redirect their accesses to the right location: reading o.x becomes a read of o.forward.x. The forwarding pointer in object o points back to o initially; after the GC copies o to o', it points o's forwarding pointer at o'.

Figure 2 shows a type of schedule that concurrent object copying must avoid. After the GC copies o.x in the old object o to o'.x in the new copy o' at (2), but before it installs the forwarding pointer from o to o' at (4), the mutator writes

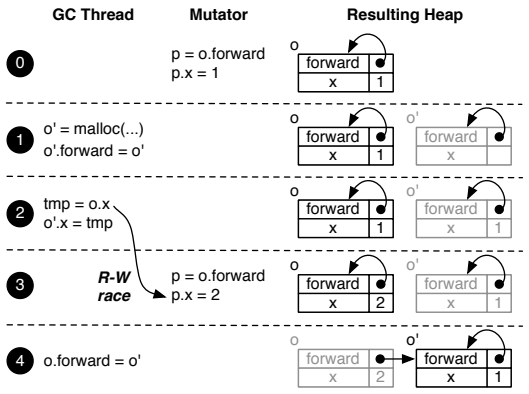

Figure 2. A naive copying collector causes lost updates when mutators modify the heap during copying.

to $o.\mathtt{x}$ again at (3). This new update is never copied to o', so subsequent reads follow the forwarding pointer in o to o', missing the update to x.

3.2.1 Related Work

To avoid missed updates, existing concurrent copying collectors use approaches such as copying only immutable objects concurrently [14], integration with transactional memory [20], batch page-granularity copying using virtual memory page protection (effectively reintroducing some pauses) [15], an intricate lock-free algorithm to copy requested objects at all costs (including nontermination in pathological cases) [23], and a simple lock-free algorithm that eagerly abandons copies on missed updates [24].

3.2.2 Implementation with DREs

In the example in Figure 2, observe that the mutator's lost update to $o.\mathtt{x}$ at (3) races with the GC's previous read of $o.\mathtt{x}$ when copying $o.\mathtt{x}$ to $o'.\mathtt{x}$ at (2). This will raise an exception in the mutator, which we handle by spinning on the forwarding pointer until it changes. This may introduce pauses in mutators, especially for large objects. However, this scenario is relatively unlikely in practice, occurring on less than 6% of copies in the worst case, and on less than 1% on average [24]. Pauses in these rare cases seem acceptable.

If the race in Figure 2 happened in the opposite order, then the GC would raise an exception reading $o.\mathtt{x}$. We could abandon the copy, but the chance of raising this exception is somewhat higher than the first possibility. Working around this requires deep cooperation with the race detector, to allow unchecked accesses in the GC to interact with checked access in mutators.

Once the GC has finished copying an object, it executes a fence, to ensure all the object's fields are visible, before installing the forwarding pointer. Proper copying also requires synchronizing the race-detection state of the old and new object, but this is a complex endeavor (see Section 3.4).

3.3 Extensions to DRE Support to Improve GC

Running a concurrent GC on top of low-level DRE support has clear benefits for low-overhead conflict detection, but to fully exploit the DRE support, some extensions are required.

The main requirements for the interface DRE support provides to a GC running on top of it are the extensions mentioned in our mark-sweep and copying algorithms. The DRE system must support removal and replacement of the record of a thread's last read or last write to a given location. Furthermore, the DRE system must support recording and executing a read that races with a previous write, while checking future accesses against it. Both of these features break assumptions of existing race detectors, but seem to require only small changes to support.

Another concern is support for data-carrying exceptions, where the DRE system supplies the up-to-date value of the last write to a location when raising an exception that races with that write. This is clearly an unsafe tool, as it can leak sequentially inconsistent values, but for the GC it appears necessary. Certainly, programs must not see such values.

3.4 Issues in Running GC on Top of DRE Support

Despite the potential benefits of a DRE-exploiting GC, it is not immediately obvious whether DRE semantics can be preserved *for the program* when garbage collection is run between the DRE system and the program, especially if the GC performs copying or moving. The chief complicating factor is that GC accesses are not explicitly part of the program; they must remain transparent to both the DRE system and the program.

Every GC must obey the following rules to avoid false or missed DREs *for the program* when running on top of DRE support, regardless of whether it uses DREs *internally* as proposed earlier in this section. First, transitive ordering between mutators via synchronization with the GC must not break program DRE semantics by hiding program-level races. Second, object copying must be transparent. Only program writes in the heap may be recorded by the race detector; GC writes should be ignored. Similarly, the GC and race detector must cooperate to ensure that when an object is moved, its race-detection state follows it, continuing to represent one logical object. Ensuring this is atomic in a concurrent copying GC is nontrivial. Finally, when an object is allocated, its race-detection state must be fresh, to avoid false races due to detection state reincarnated from a previous allocation of the same memory. This may be accomplished by clearing race detection state on collection or on allocation.

4. Optimistic Synchronization

In this section, we show how to exploit the atomicity of DRE-free regions to transform some lock-protected critical sections to lock-free versions, reducing the cost of synchronization and eliding unnecessary mutual exclusion, all without the need for rollback support.

```
try {  HB.in(lock);
       a.x = b.y + c.z;  // <-- optimistic
       HB.out(lock);
} catch (DataRaceException e) {
  do {  try {  synchronized(lock) {
                 HB.in(lock);
                 a.x = b.y + c.z;  // <-- pessimistic
                 HB.out(lock);    }
               break;
        } catch (DataRaceException e1) { }
  } while (true);
}
```

Figure 3. Pseudocode for optimistic execution of the critical section `synchronized(lock){a.x = b.y + c.z;}`

The basic idea is to retain the ordering information provided by the race detector's locking instrumentation, but elide the mutual exclusion provided by the lock itself. Figure 3 shows Java pseudocode for the optimistic version of `synchronized(lock){a.x = b.y + c.z;}` with DREs. The detector instruments the `synchronized` block to report happens-before ordering induced by lock acquires (`HB.in(lock)`) and releases (`HB.out(lock)`). This essentially transforms the lock-based critical section to a transaction. DRE support suffices to provide eager conflict detection, and we avoid rollback entirely by allowing the transformation only on a limited class of critical sections.

When the optimistically synchronized critical section executes without raising any DREs, it is effectively mutually exclusive. If two threads enter critical sections originally guarded by `lock` concurrently and perform conflicting accesses, then one will raise a DRE. At this point, the handler in Figure 3 begins trying to reexecute the fully synchronized version. To distinguish a program data race from one caused by lock elision, a thread in a retry loop checks if all other looping threads are also trying fully synchronized critical sections. If so, the race is a program data race.

Restrictions Within each optimistic critical section and *all* critical sections using the same data (they may also be aborted and retried due to a race), we allow any number of reads followed by at most one write (and no subsequent reads) and disallow updates to initially live local variables.

By eliding lock operations, we make two important assumptions about the DRE mechanism. First, instrumentation of synchronization operations must not rely on those operations to ensure its own consistency. Second, DRE support must not assume a total order on releases of the same lock. Optimistic synchronization breaks this total order, allowing two threads into an optimistic critical section concurrently and causing unordered logical releases. Subsequent acquires must be ordered after both logical releases.

4.1 Performance

Optimistic synchronization is best for frequently executed, low-contention critical sections. Profile- or feedback-guided selection of full or optimistic synchronization can help avoid excessive conflicts and repeated reexecution under high contention. Nested optimistic critical sections further complicate contention, as it is not obvious what level of nesting needs to be retried. A conservative approach retries the outermost level with full synchronization.

Previous work to elide synchronization in read-only critical sections showed that read-only critical sections do account for a small, but targetable, fraction of all critical sections. Though the cost of memory ordering via CAS and fences was substantial, lock elision achieved performance gains of up to 5% [22]. In comparison, DRE-based optimistic synchronization is safe for a larger class of critical sections and memory ordering costs may be absorbed by data-race detection cost. With careful contention management, optimistic synchronization could yield similar gains.

4.2 Related Work

Modern JVM implementations use techniques like thin locks [4] and biased locking [27] to make unnecessary or uncontended lock operations cheap. Our orthogonal proposal takes into account the cost of instrumenting synchronization operations for race detection, and allows elision of lock operations except in contended cases, relying on the race detector to identify these cases.

Systems for safe optimistic concurrency, such as software transactional memory (*e.g.*, [13]) or Speculative Lock Elision [26] generally require specialized support for speculation or rollback. Our transformation is similar to eager-update/eager-conflict-detection TM [16]; we avoid problems of escaping and rollback by restricting the transformation to limited critical sections. DRE-based optimistic synchronization makes a compromise, supporting more (*but not all*) critical sections than software lock elision [22], with no specialized support beyond DREs.

5. Conflict-Race Exceptions

Given the expense of sound-and-complete dynamic data-race detection, recent research has explored detection of a principled subset of data races: those that cause sequential consistency violations. Detecting this subset of data races is sufficient to provide simple memory models and can be significantly cheaper than detecting all data races.

A *conflict race* is a data race whose accesses are in two synchronization-free regions executing concurrently in real time. A *synchronization-free region* (SFR) is a region of a thread's execution that contains no synchronization. Conflict races differ from general data races in that the potential for an access to be involved in a conflict race is not arbitrarily long-lived. If no access in a synchronization-free region is involved in a conflict race before that SFR completes, none of its accesses will *ever* be involved in a conflict race. As such, conflict races are a much closer — but still conserva-

tive — approximation of sequential consistency violations than general data races.

5.1 Properties of Conflict-Race Exceptions

Conflict-race exceptions (CREs) are similar to DREs, but instead of using data-race-freedom, they support the memory model by guaranteeing:

- **sequential consistency or an exception**, preventing unreported sequentially inconsistent executions.

CREs guarantee that exception-free SFRs are atomic. Like DREs, they must be *recoverable* to be useful for our goals.

Implementations Conflict Exceptions [17] provides recoverable CREs, but has nontrivial complexity costs. DRFx [19, 28] achieves low overheads for conflict-race detection by sacrificing precise exceptions, and thus recoverability, guaranteeing only that an exception will be raised eventually (*e.g.*, before a system call) if a conflict race occurs. DRFx is thus unsuitable for the applications in this paper.

5.2 Applications

With small changes to the concurrent GC in Section 3, CRE support is a sufficient replacement for DRE support, and perhaps a more natural choice. Space limits preclude a discussion of these changes.

Despite their similarities, CREs cannot be treated as DREs in the general case, nor vice-versa, since the absence of an exception is as meaningful as its presence. For example, CREs are a poor fit for the optimistic synchronization algorithms in Section 4, since their atomicity guarantees are restricted to single SFRs, whereas nested critical sections expect atomicity across multiple SFRs.

Best-Effort Automatic Recovery from CREs CREs are well-suited for driving execution away from sequential-consistency-violating schedules (*i.e.*, CRE-raising schedules) in programs where some schedules are sequentially consistent and others are not. We can exploit this to reduce (but not eliminate) the probability that an execution terminates with an unhandled CRE. This behavior is desirable for deploying a program with known problematic races: a successful non-buggy execution is highly preferable to termination with an exception. Unlike our proposals for concurrent GC and synchronization elision, this technique reacts to races in the original program, rather than inducing new ones.

We exploit the fact that conflict races are defined in terms of real-time overlap of SFRs to delay accesses until they no longer cause conflict races.[1] Consider the execution in Figure 4. Boxes represent SFRs in each thread; time flows down. Assume that Thread 2 has accessed neither x nor y in its current SFR before reading x at (1). When Thread 2 tries to read x at (1), it conflicts with the previous write to x by Thread 1, because Thread 1 is still executing the SFR in which this write occurred. Allowing the read may violate sequential consistency. Thread 2 handles the CRE with a brief delay and a second attempt to read x. The delay lets Thread 1 finish and retire its SFR, so there is no longer a conflict race on x when Thread 2 retries the read at (2), and execution continues safely.

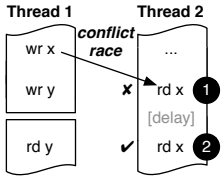

Figure 4. Some conflict races are avoidable: the read at (1) raises a CRE, but retrying at (2), it succeeds.

Hardware can offer optimal-length delays. When retiring an SFR, Conflict Exceptions sends an end-of-region message to other caches, listing locations it is relinquishing with the SFR [17]. A message relinquishing x can also serve as notification to retry pending delayed accesses to x.

Delay is sufficient to avoid many — but not all — conflict races. For example, no delay on a CRE in one of two concurrent nonserializable SFRs can yield a conflict-race-free execution. Hence we cannot guarantee sequential consistency on all executions, but we can improve the chances of preserving it and lessen the chances of an unavoidable conflict race exception. Systems such as BulkCompiler [3] or BulkSC [7] do guarantee sequential consistency in all cases, but via more specialized mechanisms.

6. Conclusions

We have outlined uses of data-race exceptions and conflict-race exceptions as general primitives for implementing conflict detection in concurrent garbage collectors, as well as two simpler applications. We hope future research will tip the cost-to-benefit ratio in favor of data-race exception support by combining ongoing advances in the performance of data-race detection, research to harness data-race exceptions for performance, simplicity, and power in new applications, and the programmability benefits of sequential consistency.

Future Work The viability of GC integration with DRE or CRE support merits careful consideration, especially for concurrent copying GC. Also interesting to consider are application-specific responses to data races, such as checkpointing and recovery or application semantics-level retry, and potential supporting programming models.

[1] Note that DREs are poorly suited for automatic recovery since, by definition, no delay can make a previously racy access data-race-free.

References

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for Safe Locking: Static Race Detection for Java. *TOPLAS*, 28(2), 2006.

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

[3] W. Ahn, S. Qi, M. Nicolaides, J. Torrellas, J.-W. Lee, X. Fang, S. Midkiff, and D. Wong. BulkCompiler: High-Performance Sequential Consistency Through Cooperative Compiler and Hardware Support. In *MICRO*, 2009.

[4] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin Locks: Featherweight Synchronization for Java. In *PLDI*, 1998.

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

[6] C. Boyapati and M. Rinard. A Parameterized Type System for Race-Free Java Programs. In *OOPSLA*, 2001.

[7] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *ISCA*, 2007.

[8] L. Ceze, J. Devietti, B. Lucia, and S. Qadeer. A Case for System Support for Concurrency Exceptions. In *HotPar*, 2009.

[9] C. Click, G. Tene, and M. Wolf. The pauseless GC algorithm. In *VEE*, 2005.

[10] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *CACM*, 21, November 1978.

[11] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A Race and Transaction-Aware Java Runtime. In *PLDI*, 2007.

[12] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, 2009.

[13] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *OOPSLA*, 2003.

[14] L. Huelsbergen and J. R. Larus. A Concurrent Copying Garbage Collector for Languages that Distinguish (Im)mutable Data. In *PPoPP*, 1993.

[15] H. Kermany and E. Petrank. The Compressor: Concurrent, Incremental, and Parallel Compaction. In *PLDI*, 2006.

[16] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[17] 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*, 2010.

[18] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *POPL*, 2005.

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

[20] P. McGachey, A.-R. Adl-Tabatabai, R. L. Hudson, V. Menon, B. Saha, and T. Shpeisman. Concurrent GC Leveraging Transactional Memory. In *PPoPP*, 2008.

[21] M. Naik, A. Aiken, and J. Whaley. Effective Static Race Detection for Java. In *PLDI*, 2006.

[22] T. Nakaike and M. M. Michael. Lock Elision for Read-Only Critical Sections in Java. In *PLDI*, 2010.

[23] F. Pizlo, D. Frampton, E. Petrank, and B. Steensgaard. Stopless: A Real-Time Garbage Collector for Multiprocessors. In *ISMM*, 2007.

[24] F. Pizlo, E. Petrank, and B. Steensgaard. A Study of Concurrent Real-Time Garbage Collectors. In *PLDI*, 2008.

[25] M. Prvulovic. CORD: Cost-effective (and nearly overhead-free) Order-Recording and Data race detection. In *HPCA*, 2006.

[26] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *MICRO*, 2001.

[27] K. Russell and D. Detlefs. Eliminating Synchronization-Related Atomic Operations with Biased Locking and Bulk Rebiasing. In *OOPSLA*, 2006.

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

[29] G. L. Steele, Jr. Multiprocessing Compactifying Garbage Collection. *CACM*, 18, September 1975.

[30] J. Ševčík and D. Aspinall. On Validity of Program Transformations in the Java Memory Model. In *ECOOP*, 2008.

[31] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-Assisted Lockset-based Race Detection. In *HPCA*, 2007.