Conflict Exceptions: Providing Simple Parallel Language Semantics with Precise Hardware Exceptions

Brandon Lucia Luis Ceze Karin Strauss Shaz Qadeer

University of Washington Microsoft Research

Hans-J. Boehm

HP Labs

November 19, 2009

Technical Report

MSR-TR-2009-149

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

This page intentionally left blank.

Abstract

We argue in this paper that concurrency errors should be treated as exceptions, *i.e.*, have fail-stop behavior and precise semantics. We propose an exception model based on conflict of synchronization-free regions, which precisely detects a broad class of data-races. We show that our exceptions provide enough guarantees to simplify high-level programming language semantics and debugging, but are significantly cheaper to enforce than traditional data-race detection. To make the performance cost of enforcement negligible, we propose architecture support for accurately detecting and precisely delivering these exceptions. We evaluate the suitability of our model as well as the behavior of our architectural mechanisms using the PARSEC benchmark suite and commercial applications. The results show that the exception model largely reflects how programmers are already writing code and that the main memory, traffic and performance overheads of the enforcement mechanisms we propose are very low.

1 Introduction

As multicores become pervasive, there is a growing need to simplify the process of developing shared memory parallel programs. The nondeterministic nature of shared memory multiprocessing, and data-races in particular, make it very hard to debug and test parallel programs. Moreover, when data-races or similar concurrency bugs manifest themselves, they normally lead to either data corruption or crashes well past the point were the buggy code is actually executed. Finally, data-races have major implications in programming language specifications [9, 23, 31]. Collectively, these issues lead to severe software reliability issues.

We address all these problems with the following approach to data-races: *make them fail-stop and deliver an exception before the race manifests itself* — *i.e.*, before the code with a race is allowed to execute. Treating data-races as exceptions has major implications in debuggability because execution can stop exactly at the point where the race happened. It also improves safety, because the exception will either exit the program and avoid delayed ill-effects or could trigger a recovery action that prevents misbehavior. Most importantly, supporting fail-stop behavior for data-races has major implications in the specification of programming languages [9], as it avoids having to define semantics of data-races and thus eliminates the need for the most complex and least satisfactory piece of, for example, the Java memory model specification [5, 23, 31].

The requirements for supporting data-races as exceptions are: (1) the detection mechanism *cannot* have false positives; (2) exception-free executions must have strong and precisely defined properties; (3) performance degradation needs to be negligible for "always-on" use; and (4) exception delivery needs to be precise, *i.e.*, the instruction that triggered the exception cannot execute and all prior instructions must have completed. We believe hardware support is instrumental in achieving negligible performance cost.

Past proposals on hardware support for concurrency bug detection have taken a best-effort approach, enabling cheaper implementations at the cost of inaccuracies [21, 27, 36]. Unfortunately, these approaches are unsuitable for us because we require precise semantics. On the other hand, implementing precise happens-before race detection [14, 15] (vector clocks) is prohibitively expensive. We address this issue by proposing an exception model that, while still being fully precise, does not incur the high cost of happens-before race detection. The fundamental observation is that some races are sufficiently distant in the execution and therefore can not directly affect it. We leverage that observation by proposing a new property, *conflict-freedom of synchronization-free regions*: if all regions separated by synchronization operations of an execution do not interact with other concurrently executing regions, then the execution either did not have any races or the racy accesses were sufficiently distant from one another. Detecting this property is much cheaper than full happens-before data-race detection because it associates additional access information primarily with cached data, thus adding reasonable overhead to cache area, and main memory overhead is largely negligible.

We make the following contributions: (1) we present the case for fail-stop behavior for data-races and argue that hardware support is an enabler; (2) we propose an exception model that provides most of the benefits of full happens-before race detection but without its prohibitive cost; (3) we propose a reference architecture implementation based on additions to standard coherence protocol and caches; and (4) we evaluate the proposed architecture support, as well as the suitability of our exception model.

In the remainder of the paper, we further explain the problem and provide more background (Section 2). We describe our exception model, discuss its properties, and then formally specify its guarantees (Section 3 and Appendix A). We then describe our architecture support (Section 4) and provide a detailed evaluation of our model and hardware mechanisms (Section 5). Finally, we discuss related work (Section 6), including the relationship of our work to transactional memory, and then conclude (Section 7).

2 Background and Motivation

Memory models from the language down to the architecture. The memory consistency model of a programming language defines the values retrieved by shared memory accesses. It must be obeyed across the entire system stack, including the compiler, runtime system, and hardware.

The simplest memory model is sequential consistency [19], in which execution behaves as if there were some global interleaving of the per-thread memory instructions. Although sequential consistency is simple to define, it leaves little room for the compiler and hardware to perform optimizations. For example, reordering-based compiler transformations designed for single-threaded applications are allowable only under very special conditions.

The restrictions sequential consistency imposes have given rise to a variety of work on "relaxed memory models". Until relatively recently, this was concentrated primarily in the hardware community (*e.g.*, [2]) with an occasional, and largely disconnected, effort on the programming languages side (*e.g.*, in Ada 83 [33]). As multiprocessing became more pervasive, relaxed memory models started to receive more attention from the programming languages community, when a number of problems with existing approaches came to light around 2000 [10, 23, 28]. In many cases these reflected the historical disconnect between hardware and programming language efforts.

Sequential consistency for data-race-free programs. The treatment of shared memory in most programming languages is converging on what is normally termed "sequential consistency for data-race-free programs" [3, 10, 23, 18]. This approach divides memory operations into two categories: *synchronization* operations (*e.g.*, locks, Java `volatile` or C++0x `atomic` variables), and *data* operations. As usual, we define two operations to *conflict* if they access the same memory location and at least one of them is a write. A data-race is defined as two conflicting concurrent accesses, where at least one is a data access.[1] So long as the programmer prevents data-races, the language implementation guarantees sequential consistency. This applies, for example, to the commonly used core spec of both Java and C++0x, the next C++ standard.

Unlike full sequential consistency, this approach allows both hardware *and compiler* to reorder memory accesses in code sections with no synchronization operations. This is possible only because data-race freedom guarantees that the execution of synchronization-free code sections appears indivisible. An obvious benefit of this model is to re-enable single-threaded compiler optimizations. For example, a loop nest with no cross-thread synchronization can be optimized largely as in the single-threaded case. A less obvious benefit of indivisibility is that the execution of a program is not affected by the granularity of data memory accesses (*e.g.*, byte vs. double word), or by intermediate values produced when executing a purely sequential library call. For example, if a synchronization-free library routine stores a password in a global variable and then overwrites it, no other thread in a data-race-free program can observe the password.

Dealing with data-races. While data-race freedom as a contract between the programmer and the system

[1] At the programming language level, each variable or scalar object field is usually viewed as its own memory location. Simultaneous accesses to adjacent byte fields do not constitute a data-race [26].

is easy to understand, it is very difficult to enforce statically and programming mistakes are common. If a programming language leaves the semantics of data-races completely undefined, as in C++0x, we have no way to bound the damage caused by untrusted code with data-races, and we make it difficult to debug the race. While this is somewhat acceptable for C++, languages like Java must guarantee security properties, which would be violated if untrusted, sand-boxed code could somehow conjure up a reference to an object, *e.g.,* representing a password, "out-of-thin-air". Undefined behavior, as in C++0x, does not prevent "out-of-thin-air" values. On the other hand, attempts to preclude them have been elusive, since many seemingly unimportant cases are surprisingly difficult to distinguish from real, and important, compiler transformations [5, 31].

There are three ways to deal with these implications of data-races: (1) Attempt to define semantics for programs that actually execute data-races — Java takes this route, but the result is not entirely satisfactory, in that it is overly complicated, and has surprising, ill-understood, and unfortunate implications on compiler optimization [5, 31]; (2) Design the programming language to statically preclude races — over the past 35 years, a variety of static type and effect systems [1, 8, 11] have been proposed to ensure data-race freedom, but these systems have not yet been widely adopted for mainstream applications; and (3) Continuously monitor and dynamically detect that a "problematic" data-race is about to execute and raise an exception, instead of executing the data-race. Since programs are designed to be data-race-free, it is safe to treat any of these as an error that should be reported to the programmer. This greatly simplifies the programmer's job, as subtle failures from accidental data-races are reported directly at the point of occurrence and produce easily describable outcomes. This last alternative (3) is what we explore in this paper.

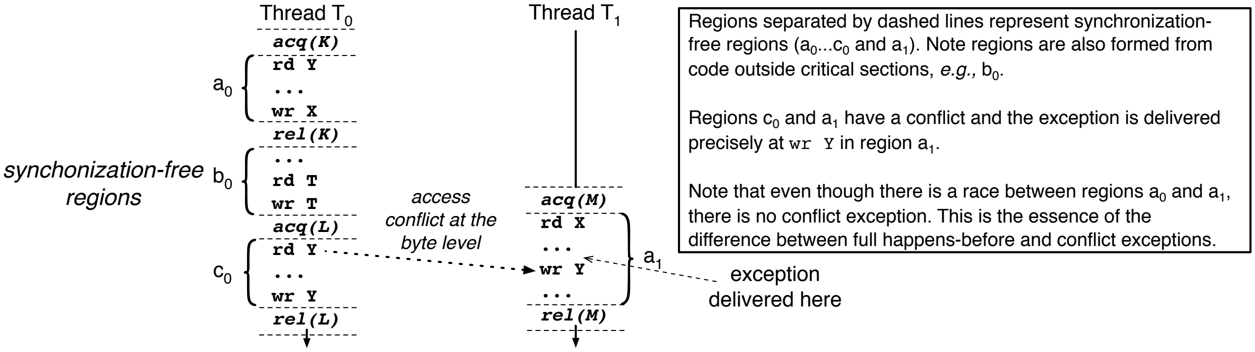
The problem with happens-before race detection. The major obstacle to treating data-races as exceptions is implementation difficulty. We know how to report data-races precisely, even without hardware support [14, 15], but unfortunately at a time and space cost prohibitive for always-on use. Race checks are frequent (potentially on every memory access) and a large amount of state is associated with each object. With vector clock algorithms, we potentially need to remember the time each object was last read by each thread, so that we can determine whether a write was properly ordered with respect to each of those reads. Since objects can be as small as a byte, each byte may require as many time stamps as there are threads in the process, possibly hundreds or thousands. And in a system in which not every pair of threads interacts regularly (think of a background thread that fails to synchronize with other threads for a long time, a case often found in real large systems), this information may need to be maintained essentially indefinitely. While optimizations have been proposed [15], they still do not result in a system that can be mandated by a language specification and used for always-on data-race detection.

To enforce data-race freedom with runtime exceptions *we need a precise detection mechanism that has near-zero time cost and low space overhead*. This is the gap we fill. We propose an exception model and implementation that preserves the utility of precise race detection at a fraction of the cost of prior approaches.

3 Conflict Exceptions

We propose a new property to enforce data-race freedom with runtime exceptions: *conflict-freedom for synchronization-free regions*. Synchronization-free regions of a thread are sections of an execution demarcated by synchronization operations. If a synchronization-free region has a conflict, it means that the region interacted with some other code that was executing concurrently, which *must* have been the result of a data-race. If all synchronization-free regions are free of conflicts, the execution *may* still have data-races, but the conflicting operations must have been separated by some potentially unrelated synchronization operation in that particular execution of the program. A *conflict exception* is delivered right at the point in the execution where the conflict happened (more precisely, right before the instruction that caused a conflict is committed). Figure 1 shows an example. Thread T_0 has three synchronization-free regions, a_0, b_0 and c_0, and Thread T_1 has one, a_1. Note that b_0 is formed of code *outside* a critical section and that synchronization-free regions by construction never nest. c_0 and a_1 have a conflict in memory location `Y`, and the exception is delivered right before `wr Y` in T_1. Also note that `wr X` in T_0 races with `rd X` in T_1 but that does not generate an exception because the regions do not execute concurrently. That illustrates the key difference between conflict exceptions and happens-before data-race detection.

Exception-free executions ensure that no synchronization-free code segment interacts in any way with concurrently executed code. Equivalently, every synchronization-free code section behaves as though it were executed indivisibly, *i.e.*, it is isolated. This is sufficient to cleanly define the semantics of programming languages. Moreover, this maintains a significant fraction of the debugging utility of full race detection. Any data access that observes or interferes with the effect of a concurrently executing synchronization-free region will lead to a conflict exception. Below we articulate in detail the benefits of conflict exceptions.



Regions separated by dashed lines represent synchronization-free regions (a_0...c_0 and a_1). Note regions are also formed from code outside critical sections, *e.g.,* b_0.

Regions c_0 and a_1 have a conflict and the exception is delivered precisely at `wr Y` in region a_1.

Note that even though there is a race between regions a_0 and a_1, there is no conflict exception. This is the essence of the difference between full happens-before and conflict exceptions.

Figure 1: Example of conflict exceptions.

Simpler programming language semantics. Conflict exceptions provide the following nice properties: (1) The compiler or hardware can reorder memory accesses and eliminate redundant loads or stores within a synchronization-free region, without affecting the semantics of the program; (2) Synchronization-free regions always appear atomic, and thus the semantics of the program is independent of the granularity of memory accesses. Moreover, the semantics of synchronization-free library routines do not depend on the ordering of memory accesses inside those routines; (3) An exception-free execution is guaranteed to be sequentially consistent even in the presence of compiler and hardware optimizations; (4) The fundamental programming rules are unchanged: The programmer needs to avoid data-races, but does *not* need to understand memory ordering issues. Data-race-free programs exhibit sequential consistency as before. This largely abstracts the complexity of relaxed memory models; (5) Programs with data-races exhibit much better defined behavior than with the current Java or C++0x definitions. Even malicious sandboxed code that is trying to exploit data-races to introduce a security hole cannot violate atomicity of synchronization-free regions. We expect this will facilitate reasoning about security properties. Similarly, code with data-races cannot violate this property, and cannot expose compiler transformations that might otherwise produce very unexpected results under the manifestation of the race. As a result, it enables more natural reasoning about buggy programs and makes them easier to debug; (6) The need for Java's complex and problematic "causality" rules [23, 31] is eliminated, since only sequentially consistent executions are ever produced — we no longer need to choose between comprehensible specifications and security properties.

Note that having the hardware enforce sequential consistency [12, 16, 35] is not sufficient to offer many of the nice properties of conflict exceptions, such as (1), (2) and (5). Compiler transformations (1) are often critical for performance, and granularity independence (2) is very important for programmability.

Appendix A formally articulates the guarantees that conflict exceptions provide from a programming language perspective.

Recovery action support. Although exceptions indicate a data-race, and thus a violation of the programming rules, it might be reasonable to continue execution. It may be possible to shut down only the offending sub-system. Since exceptions are raised only for visible data-races, it may even be feasible to simply retry the access and continue after logging the error.

4 Architectural Support for Conflict Exceptions

Our support for conflict exceptions has three components: (1) hardware/software interface; (2) recording memory accesses inside a region at the granularity of individual bytes; and (3) monitoring for byte-level conflicts (*write-after-write*, *read-after-write*, and *write-after-read*) concurrent regions, and delivering a precise exception when that happens.

A natural implementation of (2) and (3) above is to extend cache lines with meta-data that keeps access information and to leverage existing cache coherence protocols to perform the monitoring. This resembles conflict detection mechanisms in hardware transactional memory systems [25, 29]. However, we cannot afford to monitor and record access at the granularity of lines — this would lead to false, spurious exceptions, which is unacceptable. This is one of the major challenges we face in our design.

4.1 Hardware/Software Interface

Our system divides the execution of threads into *regions* demarcated by special instructions, `beginR` and `endR`, which, in the intended usage scenario, exactly encapsulate a synchronization-free region (*i.e.*, the code executed between any two synchronization operations in a thread). For code outside regions, each instruction is a *singleton region* (*e.g.*, the code between a_0 and b_0 in Figure 1). Synchronization operations are implemented with singleton regions.[2] The execution of regions and singleton regions within a given thread preserves program order, *i.e.*, they are never reordered with respect to each other. We say a region is active if it has begun but not ended yet. An access a raises a conflict exception if and only if: (1) a is a *write* and some other active region has performed a *read* or *write* operation in the same byte address; *or* (2) a is a *read* and some other active region has performed a *write* operation. The exception is delivered precisely *before*

[2] This includes lock-free data-structures routines. Note this represents the desired behavior: singleton regions have sequentially consistent behavior and do not lead to exceptions if facing conflicting accesses from other singleton regions.

a is actually committed, and after all previous instructions in the same thread have committed. Instructions inside singleton regions do not raise exceptions unless they conflict with explicit regions.

4.2 Protocol State and Invariants

Without loss of generality, we describe our protocol as an extension to a baseline directory-based MOESI protocol that maintains cache coherence within private L1 caches.

State. Each cache line is associated with four bit vectors, *access bits*, each containing as many bits as bytes in the cache line. Access bits keep track of bytes read/written within the active region of the local thread, as well as active regions of remote threads. As Table 1 describes, local bits are set as the local processor accesses data. Remote bits are set by piggy-backing on coherence messages: responses to coherence requests carry the local bits of suppliers and other caches that have access bits set for the line.

Name	What it records	Set on	Cleared on
Local read bits	Bytes read by the local thread during active region	Local read access (hit/miss)	End of region in local thread
Local write bits	Bytes written by the local thread during active region	Local write access (hit/miss)	
Remote read bits	Bytes read by other threads in their active regions	Local read or write miss	End of region in remote thread that originally set the bit
Remote write bits	Bytes written by other threads in their active regions	Local read or write miss	

Table 1: Access bit vectors associated to a cache line and their purposes.

High-level description of protocol operation. The protocol will be explained in more detail in Sections 4.3.1 and 4.3.2, but a basic description follows. When a region starts, all local bits are clear. On any access performed within a region, a local bit is set accordingly. A cache that suffers a miss receives the local bits of other threads and accumulates them into the corresponding remote bits with a logical OR operation. When a region ends, the local cache sends a message to other caches that might have access information in their remote bits, so that they can be cleared accordingly. The local cache then clears its local access bits. Regions do not nest, so there can only be one active region per thread. Before performing any access, the cache checks for exception conditions. An exception is thrown when a cache detects a byte-level conflict between a local access and a remote active region, *e.g.*, a local read accesses a byte with the remote write bit set. The exception is thrown precisely before the operation is performed. Table 2 enumerates all possible conflicts and how they are detected using access bits.

Conflict type	Local access	Remote access	Condition 1	Condition 2
read-after-write	Read	Write	Thread is about to read byte	Remote write bit set and local write bit clear
write-after-write	Write	Write	Thread is about to write byte	Remote write bit set and local write bit clear
write-after-read	Write	Read	Thread is about to write byte	Remote read bit set

Table 2: Conflict conditions between active regions and how to detect them (both conditions 1 and 2 need to be true for an exception to be thrown).

Invariants. Table 3 shows the invariants guaranteed by our protocol. Each invariant concerns a particular combination of coherence state and access bits. The second column shows the local state of the cache line, the third column indicates to which type of access bit set the invariant applies, and the last column states the invariant. For example, consider invariant 3: when a cache line is in state Modified or Exclusive, a remote read bit is set *if and only if* at least one thread other than the local one has read the corresponding byte within its active region. Invariant 3 says that, when a write access is about to complete (requires Modified or Exclusive state), the cache's remote read bits reflect exactly which bytes were read by other threads within their active regions. This guarantees that these remote read bits are up-to-date at the time an exception check is performed for this write access. In Section 4.4 we explain in detail why these invariants hold.

#	State	Type of bit set	Invariant
1	Any	Local read bits	A local read bit is set if and only if the local thread has read the corresponding byte within its active region.
2	Any	Local write bits	A local write bit is set if and only if the local thread has written the corresponding byte within its active region.
3	M or E	Remote read bits	A remote read bit is set in a cache if and only if at least one thread other than the local one has read the corresponding byte within its active region.
4	O or S	Remote read bits	A remote read bit is set in a cache *only if* at least one thread other than the local one has read the corresponding byte within its active region.
5	Any	Remote write bits	A remote write bit is set in a cache if and only if at least one thread other than the local one has written the corresponding byte within its active region.

Table 3: Invariants guaranteed by our extended protocol.

4.3 Adding Support to the Coherence Protocol

In addition to access bits, we add a *supplied bit* to each cache line, which indicates whether the line was supplied to any other cache during the active region. We also add three other bits to each *cache*: (1) an *in-region bit* to indicate whether the local thread is currently in an active region; (2) a *cache-level supplied bit*, used to summarize whether any cache line accessed within the active region was supplied to another cache; and (3) an *out-of-cache bit*, used to indicate whether any line with non-null access bits was evicted.

All the state used to record information about cache lines is stored into a separate state table that has the same basic geometry (same number of sets and ways) as the cache itself, as shown in Figure 2. This table is backed by main memory, as will be explained in Section 4.3.2.

4.3.1 Basic In-Cache Operation

We now explain the protocol assuming in-cache operation, *i.e.*, there is no eviction of line with access bits and misses are always serviced by a peer cache. We cover out-of-cache operation in Section 4.3.2. We have written a model of the in-cache operation of our protocol and checked that the invariants introduced in

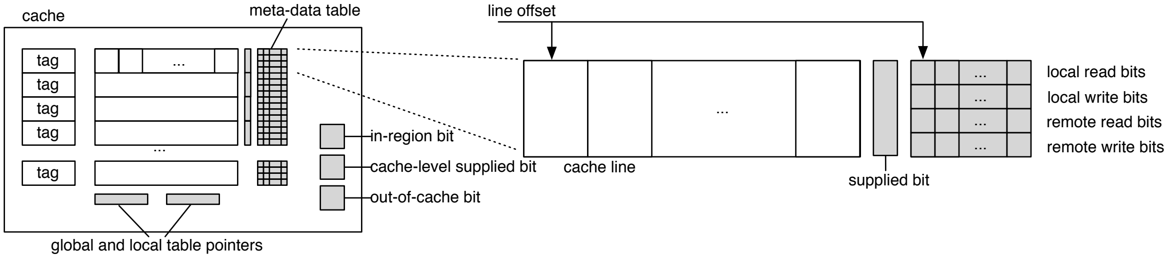


Figure 2: Cache modifications for conflict exception detection. New structures are shown in grey.

Section 4.2 indeed hold. We provide more detail on how we checked it in Section 5.1.

Starting a region. When a processor executes a `beginR` instruction, it sets the in-region bit. The `beginR` instruction has the effect of a full fence instruction.

Exception check. Before any memory access within an active region is allowed to proceed, the cache performs an exception check. The exception conditions are described in Table 2.

Cache hits. After the exception check, the cache sets the corresponding local (read or write) bit.

Cache misses. When a miss is serviced and the line arrives in the local cache, the access bits of the incoming line are accumulated into the remote access bits with a bitwise OR. On global read misses, if any other cache indicates it has local read bits set for the requested line, the line is brought from memory in shared state, instead of exclusive. This is necessary to keep invariant 3 in Table 3.

Servicing a remote read miss request. When a cache supplies data for a read miss, it appends to the reply message: one bit indicating whether the line has any local read bit set and the local write bits OR-combined with the remote write bits. The line-level and cache-level supplied bits are then set.

Servicing a remote write or invalidate miss request. When a cache receives a write or invalidate request, it sends both its local read and write bits to the requester (if they are set), even if the line is in invalid state, and independent of being able to supply the data. Then, the cache sets its line-level and cache-level supplied bits and invalidates its cache line, but preserves the supplied and access bits associated with the cache line.

Ending a region. When the `endR` instruction is executed and the cache-level supplied bit is clear, the in-region bit is cleared together with each of line-level supplied bits and local access bits[3]. In addition, if both local write bit and remote write bit are set for the same byte in a cache line, the corresponding remote write bit of the cache line is also cleared. This is necessary to keep invariant 5 in Table 3. If the cache-

[3] This can be done efficiently with gang-clearing [24].

level supplied bit is set, the cache iterates over the line-level supplied bits, appending the following to an *end-of-region message*: the line address; and the corresponding local read and write bits of cache lines for which the supplied bit is set. The message is then sent to other caches that may have received access bits from the ending region.[4] When all acknowledgments are received, the bits are cleared as above. The `endR` instruction acts as a full fence.

Processing an incoming end-of-region message. When a cache receives an end-of-region message, it checks for the presence of access bits for each address in the message. If a remote bit is present and set, the cache checks if the corresponding bit in the message is also set. If so, the cache clears its remote bit. In addition, if the cleared bit is a remote read bit, and the cache line is in a state that allows a write hit (M or E), this line is downgraded to a state that disallows it (O or S). This is necessary to guarantee remote read bits are refetched when the processor is about to write the cache line and avoid problems when multiple processors read the same byte within active regions (invariant 3 in Table 3).

Singleton regions. Memory accesses in singleton regions do not set access bits. Therefore, conflicts between accesses of two singleton regions do not lead to exceptions. However, if a singleton region conflicts with an explicit region, an exception is delivered to the instruction in the singleton region.

4.3.2 Out-of-Cache Operation

In-cache operation is the common case, but caches may have to occasionally evict cache lines with access bits set. When a line is evicted, if any supplied or local access bits are set, they need to be saved. They will be used in three situations: (1) when the cache that evicted the line suffers another miss on the line; (2) when the cache that evicted the line ends its region, and (3) when another cache suffers a cache miss on the line.

We chose to save evicted state in two distinct structures in memory. The *local table* is a per-thread table that stores a list of addresses accessed and evicted within the active region, used in cases (1) and (2). The *global table* is a per-process table that individually saves supplied and local bits for all threads that had to evict them, used in all three cases above. The global table is organized hierarchically (like page tables) and is indexed using the physical address of a line. We augment each cache with two memory pointers, one for each table (see Figure 2). We also augment the directory state for each cache line with an *in-memory bit* that

[4] One way of doing this distribution is to leverage the directory. Instead of indicating sharers of the data, the directory could indicate which caches share the data *or* have been supplied with any access bits that have not been cleared yet. Clearing directory bits can be done lazily at invalidates in which no caches respond with access bits to avoid extra complexity.

indicates whether any thread has saved access bits for that line in memory. We extend the in-cache operation described earlier to save, search and restore access and supplied bits as below.

Cache evictions. If a cache line being evicted has a supplied or local bit set, the line address is saved in the local table and local access/supplied bits are saved in the global table. The out-of-cache bit is then set and the directory is notified to set the in-memory bit for the corresponding line.

Cache misses. During a miss, if the in-memory bit in the directory is set, other threads' local bits are retrieved from memory by accessing the global table and the corresponding supplied bit is set. The bits retrieved from memory are OR-combined into the remote bits, together with local bits received from other caches (if any). If the out-of-cache bit is set, the cache also searches the global table for its supplied, local read and local write bits in case they have been previously evicted, and restores them.

Ending a region. If the out-of-cache bit is set, even if the cache-level supplied bit is not set, the cache retrieves evicted line addresses from its local table and supplied and local bits from the global table, and appends to the end-of-region message those that have their supplied bit set. The cache clears its in-memory local table in the process, as well as the corresponding entries in the global table.

4.3.3 Examples

Figure 3(a) shows an example of in-cache operation. For simpiclity, all operations involve the same cache line *x* (2 bytes long) and all caches are in an active region. Initially, cache A writes byte 0 and sets local write bit 0 (1). Then, cache B suffers a miss when attempting to write byte 1 and sends an invalidation to cache A (2), which invalidates its line and sends data, its local read bits and OR-combined local and remote write bits to cache B (3). Cache B then OR-combines the received bits into its remote read and write bits, performs an exception check (no exception is thrown because it is attempting to write byte 1), and completes the write operation by setting local write bit 1. Later, cache C attempts to read byte 0, suffers a miss and sends a read request to cache B (4). B responds with the data, a single bit indicating that all its read bits are clear and its OR-combined local and remote write bits (5). Finally, cache C receives the response from B. It ignores the read bit because it is transitioning to a shared state, OR-combines the received write bits into its remote write bits and is ready to perform an exception check (6). An exception is thrown because cache C is attempting to read byte 0, but it finds its remote write bit 0 set, indicating that another cache has written this byte in an active region.

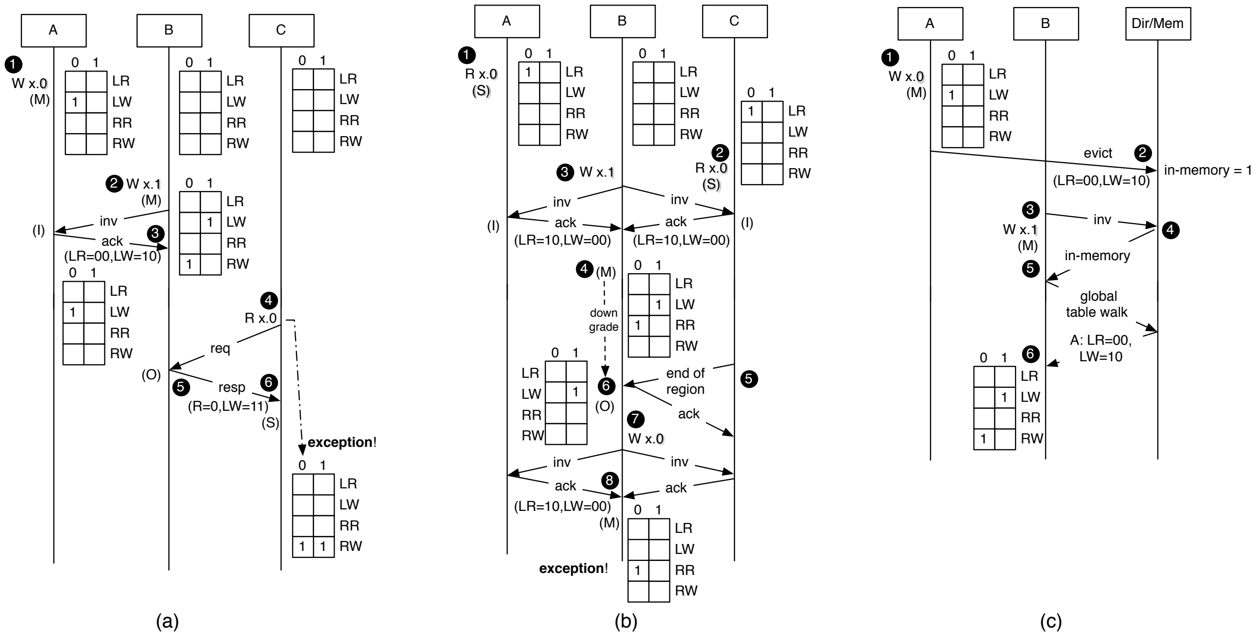


Figure 3: Examples: (a) In-cache operation; (b) In-cache operation with downgrade; and (c) Out-of-cache operation. Caches A, B and C access the same cache line x, which consists of two bytes.

Figure 3(b) shows another example. A reads byte 0 and sets its local read bit 0 (1). C then reads byte 0 and sets its local read bit 0 (2). Next, B writes byte 1 (3), collects local read bits from A and C, and sets its remote read bit 0 (4). C then ends its region and sends an end-of-region message (5). When the message arrives at cache B (6), B clears its remote read bit 0 and downgrades the line. When B suffers a write miss for byte 0 (7), it sends invalidates, collects C's local read bit, sets its remote read bit 0, and finally throws an exception (8). Note that if the line had not been downgraded, B would not have detected the conflict.

Figure 3(c) shows an out-of-cache operation example. Again, for simplicity, assume a single cache line *x* and all caches are in active regions. First, cache A writes byte 0 and sets its local write bit 0 (1). Then, cache A evicts line x by sending its supplied, local read and local write bits to memory (2). At this point, the directory sets the in-memory bit for the corresponding line. Later, cache B suffers a write miss when attempting to write byte 1 and sends a write request to the directory (3), which responds notifying B that another cache has local bits in memory (4). B then performs a global table walk to retrieve those local bits and set the corresponding supplied bit (5). B then OR-combines the local bits it retrieved into its remote bits, checks for exceptions and continues (6). Notice that, had B attempted to write byte 0, it would have correctly thrown an exception because it would have its remote write bit 0 set, even though it did not retrieve this information directly from cache A, but from the in-memory global table.

4.4 Why Invariants Hold

We now justify why the proposed invariants hold and why they are sufficient to guarantee our exception specification. Exception checks are done when a memory operation is about to complete. For a memory operation to reach this point, the cache controller must have already performed all related coherence actions, so the line must be in a stable state (any valid state for a read, M or E states for a write). The invariants guarantee that the remote bits needed at exception check time accurately reflect whether any other thread has touched that line within their currently active region.

Table 4 shows the invariants and the protocol mechanisms that guarantee them. First, consider invariants 1-2: both local read bits and local write bits are set and cleared only based on local events, so this invariant follows directly from the local cache operation. Next, consider invariants 3-5 (Column 4): they say that a remote read bit set for a thread implies that some other thread has accessed the corresponding byte within its (still active) region. This is guaranteed by our protocol because remote bits are always OR-combinations of local bits of other threads, and the protocol guarantees they are cleared when the region in which they were set ends. Now, consider invariant 3 (Column 5). It says that, if the state of a line is Modified or Exclusive, remote read bits accurately represent read accesses of other threads within active regions. The protocol guarantees this invariant by collecting all local read bits from other threads on write misses and by only transitioning to Exclusive state if no other thread has any local read bit set. Notice that invariant 4 does not guarantee an access within an active region in one thread implies a set remote read bit for another thread when the cache line state is owned or shared. This is not an issue because these states do not allow write hits, and read-read conflicts do not throw exceptions.

#	State	Type of bit set	Reason why a set bit implies an access within an active region	Reason why an access within an active region implies a bit is set
1	Any	Local read bits	Follows directly from local access actions. Bits are set on access and cleared when a region ends.	
2		Local write bits		
3	M or E	Remote read bits	Remote bits are a combination of local bits from other threads. end-of-region message clears all remote bits previously set by data supplied to other caches, including the local cache.	Read misses can only result in exclusive state if no local read bits for the line are set for other threads. Transitions to M state require acknowledgments with local read and write bits from other threads.
4	O or S			Not enforced.
5	Any	Remote write bits		If an access happens after an in-region remote write, it will be satisfied with data supplied by the cache that wrote that byte or by a cache that wrote it later. The combination of local and remote write bits guarantees they are properly propagated.

Table 4: Reasons why invariants hold on exception checks.

Finally, consider the last column of invariant 5. This invariant concerns remote write bits and says that they accurately represent in-region write accesses of other threads. The protocol guarantees this invariant because it enforces exclusiveness and serialization of write operations to the same cache line (coherence) and ensures that all write bits set within active regions are communicated to caches being supplied with the cache line by OR-combining its local write bits and its remote write bits. This guarantees that any writer has the most up-to-date set of remote write bits for the line and propagates those to any subsequent readers.

4.5 System Issues

Exception cleanup. When an exception is raised, the system still leaves the current region active. To support this, the access bits need to be kept throughout the execution of the handler, however support must be provided to prevent the handler code from affecting the access bits.

Virtualization. Access bits are associated with a specific thread and not with a physical cache, therefore, architecture resources need to be virtualized. Context switches are done by extending the thread context to contain the local and global table pointers, the cache-level supplied bit, the in-region bit, and the out-of-cache bit, which are all saved when a thread is switched out. Moreover, all cached lines that have non-null access bits need to be sent to the in-memory log and the access bits are reset — note that this potential cost can be mitigated if context switches tend to happen at synchronization points, which is often the case. When memory is paged out to disk, the OS needs to associate the corresponding access bits in the local and global tables (if any) with the virtual address of the page being sent to disk. When the page is brought back, the OS need to remap the old entries to the new physical address.

Speculative Loads. Compilers will have to avoid "control speculative" loads, *i.e.*, performing loads that might not be requested by the programmer on that control path. These can introduce data-races. These issues did not appear in our experiments. Moreover, they can typically be replaced by prefetch instructions.

5 Evaluation

The goals of this evaluation are to: (1) understand the costly events in the protocol; (2) assess space and traffic overhead of access bits storage and transfers; (3) assess the suitability of our exception model.

5.1 Experimental Setup

Model checking. We ensured the correctness of the protocol with model-checking of its in-cache operation using Zing [4]. We used configurations with at most three caches and two bytes per cache line, and executions of at most four requests. We could not check larger configurations due to the space explosion problem inherent to model checking, but we believe the largest configuration we successfully checked is sufficient to ensure the protocol invariants indeed hold.

Simulation. We evaluate performance using a simulator based on Pin [22] and SESC [30]. The simulator faithfully models the exception detection and the cache coherence protocol, including cache-to-cache and off-cache operation. Given that our goal is to understand the protocol itself, the simulator does not include a timing model. We model a multiprocessor with 8 cores, with 8-way 32KB L1 caches and 32-byte lines.

Benchmarks. There are no programs written from scratch assuming our exception model yet. Nevertheless, the model is intuitive enough that legacy programs tend to conform to the model. We leverage this fact and use existing parallel C/C++ programs to evaluate our architecture support. Sections of execution between calls to pthreads are considered synchronization-free regions — *i.e.*, between the *return* from any pthread function call and the next *call* to a pthread function. This includes code outside critical sections. Instructions inside the pthreads library are considered singleton regions. We used the PARSEC benchmark suite [6] (`simsmall` input set), MySQL (sysbench OLTP benchmark), and Apache (apache-bench utility).

5.2 Performance

Table 5 characterizes dynamic region size and frequency (Columns 2 and 3) and false sharing between active regions (Columns 4-6). There is a large variance of region size among applications. For example, `fluidanimate` has many (3M) small regions (< 1K memory operations), whereas `freqmine` has very few (96) large regions (>50M memory operations). False sharing also varies significantly among applications.

App.	Regions		False Sharing / 1B Mem. Ops			EOR Messages			Mem. M-D Lkup / 100K Mem. Ops		Mem. Ovhd	Traffic Ovhd (B/MB)			
	Total #	Mem. Ops / Region	WaW	WaR	RaW	% Reg. w/ Msg.	Avg. # Lines	% to Mem.	Rem. Bits	Loc. Bits	(% Ftpt)	Total	Rd Reply	Inv Ack	EOR Msgs
blackscholes	50	12M	0.0	0.0	11.5	2.00	1	100	0.01	70.81	0.42	3.55	2.55	0.00	1.00
bodytrack	37K	70K	8.0	3.4	15.6	0.02	3	100	0.01	58.41	2.29	1.95	0.63	0.46	0.87
facesim	35K	2M	487.0	0.1	2251.5	0.05	297	100	0.14	285.09	8.58	79.61	62.49	13.52	3.60
ferret	87K	30K	0.0	0.0	0.0	0.00	0	0	0.00	769.20	27.36	0.00	0.00	0.00	0.00
fluidanimate	3M	996	0.0	0.0	1.4	0.00	1	100	0.01	9.60	1.58	0.36	0.23	0.00	0.13
freqmine	96	54M	9.7	0.8	1048.0	5.21	112	100	0.05	213.29	17.09	75.99	63.89	0.64	11.47
swaptions	96	20M	112.2	31.1	460.4	8.33	21	100	0.67	492.53	0.46	45.69	27.76	8.64	9.29
vips	17K	252K	2.1	0.7	2.3	0.09	1	100	0.00	163.78	1.32	0.42	0.06	0.07	0.28
x264	1K	1M	0.0	0.0	0.0	0.00	0	0	0.00	474.93	6.89	0.00	0.00	0.00	0.00
canneal	118	39M	1.5	0.0	2.8	1.69	7	100	0.00	683.44	5.09	0.20	0.06	0.03	0.11
dedup	18K	204K	68.8	28.7	96.5	1.92	1	100	0.07	104.96	1.01	19.16	3.63	3.67	11.85
streamcluster	1K	1M	0.0	0.0	2.8	0.08	2	100	0.01	1465.45	7.10	0.05	0.03	0.00	0.02
MySQL	1M	7K	0.0	0.0	0.1	0.00	1	100	0.00	24.15	0.24	0.03	0.01	0.00	0.02
Apache	62K	27K	5.9	0.6	2.3	0.01	1	100	0.01	153.86	6.13	0.72	0.07	0.20	0.45
Mean	304K	9M	49.7	4.7	278.2	1.39	32	86	0.07	354.96	6.11	16.27	11.53	1.95	2.79

Table 5: Protocol events in detecting conflict exceptions. “Mem. Ops” refer to issued load and store instructions.

The most significant events for performance evaluation purposes are end-of-region messages and access bit lookups in memory that occur on cache misses. We characterize end-of-region messages in Columns 7-9 of Table 5. First, note that less than 2% of regions send end-of-region messages for most benchmarks, many times close to zero (Column 7). Moreover, end-of-region messages typically contain access bits for just a handful of lines (Column 8), many times a single line (*e.g.,*`vips`). However, they often must get this information from memory (Column 9). We find a couple of outliers, both in terms of percentage of regions with end-of-region messages (`freqmine` and `swaptions`) and in terms of average number of lines per end-of-region message (`freqmine` and `facesim`). Even then, the fraction of regions with messages is no higher than 9% and at worst, messages contain fewer than 300 lines. Note that the size of regions for these benchmarks is large enough to amortize the cost of end-of-region messages. As expected, benchmarks with more false sharing also have more end-of-region messages, and they tend to be larger. This shows an opportunity to reorganize the code and reduce false sharing, potentially eliminating end-of-region messages and reducing the number of lines in an end-of-region message, among other desirable performance benefits. Overall, end-of-region messages are unlikely to be a performance issue.

We now analyze the other main source of overheads, the lookup of access bits in memory. Columns 10 and 11 in Table 5 show the frequency of lookups. Remote access bit lookups (Column 10) happen when a thread needs to fetch access bits from remote threads. Generally, costly remote bit lookups are infrequent (never more than 7 per million memory operations, which is very low). In `swaptions`, the benchmark with highest cost, frequent false sharing and evictions lead to an increase in remote bit lookups. Local bit lookups (Column 10) occur whenever a line with access bits is evicted and accessed again in the same region. We observe they are correlated with cache miss rates. The rate of local bit lookups is modest — from less than 10 every 100K memory operations (`fluidanimate`) to around 1 every 100 (`streamcluster`, which has high cache miss rates). Note that both remote and local access bit lookups only happen on cache misses, and only on a fraction of them, so they are also unlikely to lead to significant performance degradation. They can also be reduced with an "access bit victim buffer", which we do not explore in this work.

5.3 Overheads: Space, Traffic

Columns 12-16 in Table 5 characterize storage and traffic overheads. We report the maximum memory overhead to keep the local and global tables of access bits as a fraction of the maximum application footprint (Column 12). The memory overhead is less than 2.5% for most benchmarks and often less than 1% (`blackscholes`, `MySQL`). `freqmine` and `ferret` have the highest overhead, due to frequent eviction of lines with access bits, and in the case of `freqmine`, extremely long regions.

The total traffic overhead, reported in bytes per megabyte (Column 13) for all benchmarks is low, many times less than 1 byte per megabyte sent. Recall that there are two sources of traffic overhead: the addition of access bits to coherence messages (Columns 14 and 15) and sending end-of-region messages (Column 16). Both are a function of false sharing, so applications with higher false sharing have higher overheads. This is most apparent in `freqmine` and `facesim`, with high traffic overhead compared to others, but still low overall (<< 1%).

5.4 Suitability Analysis

It is important that programmers using a system with support for conflict exceptions can write programs in a familiar way. We show that with few or no changes, our benchmarks can be run with no exceptions. In Table 6, we show the code size for each benchmark (Column 2) and the fraction of regions experiencing exceptions (Column 3), as well as the number of unique instructions (Column 4), lines of code (Column 5), and functions (Column 6) involved in exceptions. Several benchmarks (*e.g.,* `blackscholes`) run unchanged with no exceptions at all. For most that do, exceptions involve under 50 lines of code. Typically, these lines are co-located in just a few functions (11.5, on average). The clumping of code with exceptions suggests that making changes to eliminate these data-races would not be difficult.

App.	KLOC	% Vio. Regions	# Vio PCs	# Vio Lines	# Vio Fns
blackscholes	0.49	0	0	0	0
bodytrack	5.85	5.64	161	22	16
facesim	22.6	6.81	101	21	13
ferret	9.20	1.37	1080	248	50
ferret†	9.20	0	0	0	0
fluidanimate	0.86	0	0	0	0
freqmine	2.71	0	0	0	0
swaptions	1.31	0	0	0	0
vips	109.34	2.59	234	133	41
x264	40.40	7.27	6	6	4
canneal	3.36	4.52	1	1	1
dedup	3.69	35.57	104	58	12
streamcluster	1.26	44.44	147	52	6
MySQL	1600.00	0.05	92	76	41
Apache	602.00	30.10	68	54	17
Mean	171.65	9.88	142.43	47.93	14.36

Table 6: Number of exceptions in our benchmarks and how spread out in the code they are.

Examining code that led to exceptions, we found that hand-crafted synchronization was a common culprit. For instance, streamcluster uses a flag variable in the master thread to coordinate workers. In canneal, a similar mechanism is used to communicate a loop termination condition between threads. Both cases implemented synchronization incorrectly, using a non-atomic data variable. We eliminated the exceptions thrown in canneal by synchronizing around accesses to the flag. Changes affected only 3 lines, and were straightforward.

In ferret, there were many exceptions. These were thrown because ferret employs pipeline parallelism, and pipeline stages' task queue operations were not properly synchronized. Doing so eliminated all exceptions thrown (see ferret† in Table 6). In MySQL, exceptions were caused by true data-race errors. These races had no ill effect on our experimental executions, but could result in unexpected behavior. In this case, exceptions aid in debugging by guiding developers to racy code. Note that although Posix is not completely clear on certain aspects of data-races, the races we found are clearly disallowed.[5] C++0x and Java provide alternative facilities to correctly write such code enabling automatic insertion of beginR and endR, eliminating the exceptions.

6 Related Work

We discussed accurate software race detection in Section 2. In addition, the desire for fail-stop behavior of data-races has been discussed recently in informal forums [9, 13], but this is the first concrete proposal.

High-performance enforcement of sequential consistency (SC) has been actively investigated [12, 16, 34, 35]. SC at the hardware level is definitely valuable but does not provide many crucial guarantees for debugging and simple language semantics, *e.g.*, precise race detection, atomicity for synchronization-free regions and memory access granularity independence. Compiler-based techniques for enforcing sequential consistency [20, 32] share similar properties with the hardware approaches.

Conflict exceptions fundamentally solve a different problem than TM, namely providing fail-stop behavior for concurrency errors that result from missing or incorrect synchronization, as opposed to providing a better synchronization primitive, but they are related in several ways. The two features may prove to be synergistic. Our precise byte-level conflict detection mechanism could be used in TM systems, as several current proposals [7, 29] provide only block-level conflict detection, which can result in spurious transaction

[5] [18] Base definitions, 4.10: "Applications shall ensure that access to any memory location by more than one thread of control (threads or processes) is restricted such that no thread of control can read or modify a memory location while another thread of control may be modifying it."

aborts. Also, data-races are as much a problem in emerging transactional memory specifications [17] as they are in lock-based systems. The detailed implementation considerations and language semantics for such a combined system are beyond the scope of this paper.

One could argue that synchronization-free regions could be converted into transactions and the guarantees would be similar to conflict exceptions. Although conceptually correct, this provides inferior functionality at substantially increased cost. Using transactions in this way could mask some errors, but not all. For example, consider an identity function `id` that happens to acquire and release a lock unrelated to `x`; if we forget a lock around `x=id(x+1)`, we do not atomically increment `x` in either approach. Putting transactions around synchronization-free regions would eliminate any hope of detection while still producing incorrect output. Conflict exceptions would correctly identify the problem of missing synchronization. Also, the cost of the TM approach is higher because transactions formed out of regions can be expected to be quite long and would have to buffer a significant amount of data.

7 Conclusions

In this paper we argued that data-races should have fail-stop behavior. However, providing precise race detection at a low enough cost to be always-on is challenging. We address this challenge with conflict exceptions, which provide enough guarantees to significantly improve debugging and enable the design of simple parallel language semantics. Our evaluation showed that conflict exceptions can be enforced at a low cost. Our suitability study showed that the exception model in fact largely reflects how programmers have already been writing multithreaded programs, which supports the intuitiveness of our model.

Going forward, we advocate that hardware resources should be spent to improve the programmability of multiprocessor systems. We believe that not only hardware should help with debugging, but should also provide enough *guarantees* such that even the programming language can rely on it.

References

[1] M. Abadi et al. Types for Safe Locking: Static Race Detection for Java. *TOPLAS*, 2006.
[2] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12), 1996.
[3] S. V. Adve and M. D. Hill. Weak Ordering—A New Definition. In *ISCA-17*, 1990.
[4] T. Andrews et al. Zing: Exploiting Program Structure for Model Checking Concurrent Software. In *CONCUR*, 2003.
[5] D. Aspinall and J. Sevcik. Java Memory Model Examples: Good, Bad, and Ugly. In *VAMP*, 2007.
[6] C. Bienia et al. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, 2008.
[7] J. Bobba et al. TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory. In *ISCA*, 2008.
[8] R. Bocchino et al. A type and effect system for deterministic parallel Java. In *OOPSLA*, October 2009.
[9] H. Boehm. Simple Thread Semantics Require Race Detection. PLDI FIT, 2009.
[10] H.-J. Boehm and S. Adve. Foundations of the C++ Concurrency Memory Model. In *PLDI*, 2008.
[11] P. Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, June 1975.
[12] L. Ceze et al. BulkSC: Bulk Enforcement of Sequential Consistency. In *ISCA-34*, 2007.
[13] L. Ceze et al. A Case for System Support for Concurrency Exceptions. In *USENIX HotPar*, 2009.

[14] T. Elmas et al. Goldilocks: a Race and Transaction-aware Java Runtime. In *PLDI*, 2007.

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

[16] C. Gniady et al. Is SC + ILP = RC? In *ISCA-26*, May 1999.

[17] IBM, Intel, and Sun. Draft specification of transactional language constructs for C++, version 1.0. http://software.intel.com/file/21569, August 2009.

[18] IEEE and The Open Group. *IEEE Standard 1003.1-2001*, 2001.

[19] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. on Comp.*, 1979.

[20] J. Lee and D. Padua. Hiding Relaxed Memory Consistency with Compilers. In *PACT*, 2000.

[21] S. Lu et al. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *ASPLOS-XII*, 2006.

[22] C. K. Luk et al. PIN: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, 2005.

[23] J. Manson et al. The Java Memory Model. In *POPL*, 2005.

[24] J. F. Martínez et al. Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors. In *MICRO-35*, November 2002.

[25] K. Moore et al. LogTM: Log-based Transactional Memory. In *HPCA-13*, 2006.

[26] C. Nelson and H.-J. Boehm. Concurrency Memory Model (final revision). C++ standards committee paper WG21/N2429=J16/07-0299, `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2429.htm`, October 2007.

[27] M. Prvulovic and J. Torrellas. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. In *ISCA-30*, 2003.

[28] W. Pugh. The Java Memory Model is Fatally Flawed. *Concurrency - Practice and Experience*, 12(6), 2000.

[29] R. Rajwar et al. Virtualizing Transactional Memory. In *ISCA-32*, 2005.

[30] J. Renau et al. SESC Simulator, January 2005. http://sesc.sourceforge.net.

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

[32] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *TOPLAS*, April 1988.

[33] United States Department of Defense. *Reference Manual for the Ada Programming Language: ANSI/MIL-STD-1815A-1983 Standard 1003.1-2001*, 1983. Springer.

[34] E. Vallejo et al. Implementing Kilo-Instruction Multiprocessors. In *Intl. Conf. on Pervasive Services*, 2005.

[35] T. F. Wenisch et al. Mechanisms for Store-wait-free Multiprocessors. In *ISCA-34*, 2007.

[36] P. Zhou et al. HARD: Hardware-Assisted Lockset-based Race Detection. In *HPCA-13*, 2007.

A Formal Specification of Conflict Exceptions

$$\begin{array}{rcr|rcl}
e, f, r, w & \in & \mathit{Events} & \mathit{SyncOps} & & \\
E & \in & 2^{\mathit{Events}} & \mathit{LoadOps} & = & \{\mathit{Load}\} \times \mathit{ByteAddrs} \\
t & \in & \mathit{ThreadIds} & \mathit{StoreOps} & = & \{\mathit{Store}\} \times \mathit{ByteAddrs} \\
a & \in & \mathit{ByteAddrs} & \mathit{DataOps} & = & \mathit{LoadOps} \cup \mathit{StoreOps} \\
 & & & \mathit{Ops} & = & \mathit{SyncOps} \cup \mathit{DataOps} \\
 & & & \mathit{Tid} & \in & \mathit{Events} \rightarrow \mathit{ThreadIds} \\
 & & & \mathit{Op} & \in & \mathit{Events} \rightarrow \mathit{Ops}
\end{array}$$

We define the following subsets of a set of events E: $\mathit{SyncEvents}(E)$ is the subset of synchronization events in E; $\mathit{ThreadEvents}(E, t)$ is the subset of events by thread t; $\mathit{LoadEvents}(E, a)$ is the subset of load events to address a; $\mathit{AllLoadEvents}(E)$ is the subset of all load events; $\mathit{StoreEvents}(E, a)$ is the subset of store events to address a; and $\mathit{AllStoreEvents}(E)$ is the subset of all store events.

An execution $(E, \mathit{SyncOrder}, \mathit{ThreadOrder}, \mathit{StoreOrder}, \mathit{Source})$ is a tuple satisfying the following conditions: (1) E is a set of events. (2) $\mathit{SyncOrder}$ is a partial order on $\mathit{SyncEvents}(E)$. Hence, synchronization events are independent of all other events. (3) $\mathit{ThreadOrder}$ maps each $t \in \mathit{ThreadIds}$ to a total order $\mathit{ThreadOrder}(t)$ on $\mathit{ThreadEvents}(E, t)$. We overload $\mathit{ThreadOrder}$ to refer to the partial order $\bigcup_{t \in \mathit{ThreadIds}} \mathit{ThreadOrder}(t)$. (4) $\mathit{StoreOrder}$ maps each $a \in \mathit{ByteAddrs}$ to a total order $\mathit{StoreOrder}(t)$ on $\mathit{StoreEvents}(E, a)$. We overload $\mathit{StoreOrder}$ to refer to the partial order $\bigcup_{a \in \mathit{ByteAddrs}} \mathit{StoreOrder}(a)$. (5) Source is a map from $\mathit{AllLoadEvents}(E)$ to $\mathit{AllStoreEvents}(E)$. For all $a \in \mathit{ByteAddrs}$, if $e \in$

$\mathit{LoadEvents}(E, a)$ then $\mathit{Source}(e) \in \mathit{StoreEvents}(E, a)$. Thus, Source provides for each load event the corresponding store event that supplied the data for it.

Given an execution, we define two relations on the set of events E. The causal relation Causal orders a store event w before every load event r that sees the value written by w. The anti-causal relation $\mathit{AntiCausal}$ orders a load event r before every store event w that comes later than the source of r according to $\mathit{StoreOrder}$.

$$\begin{aligned}(w, r) \in \mathit{Causal} &\overset{def}{=} w = \mathit{Source}(r) \\ (r, w) \in \mathit{AntiCausal} &\overset{def}{=} r \in \mathit{AllLoadEvents}(E) \wedge w \in \mathit{AllStoreEvents}(E) \wedge (\mathit{Source}(r), w) \in \mathit{StoreOrder}\end{aligned}$$

An execution has a *data-race* on address a if and only if there are two events $e, f \in E$ such that: (1) e and f are unordered by the transitive closure of $\mathit{ThreadOrder} \cup \mathit{SyncOrder}$; (2) $\mathit{Op}(e) = (\mathit{Store}, a)$; and (3) $\mathit{Op}(f) = (\mathit{Store}, a)$ or $\mathit{Op}(f) = (\mathit{Load}, a)$.

The set of events in an execution can be partitioned into a collection of regions, each being one of three kinds: (1) the set of all data events in a thread before its first synchronization event; (2) the set of all data events in a thread between two synchronization events; (3) the set of all data events in a thread after its last synchronization event.

The constraint graph of an execution is a graph (R, N), where R is the set of regions in the execution and N is the set of edges defined as follows: $(p, q) \in N$ iff p is different from q and there exists $e \in p$ and $f \in q$ such that e is ordered before f by the transitive closure of $\mathit{ThreadOrder} \cup \mathit{StoreOrder} \cup \mathit{Causal} \cup \mathit{AntiCausal}$. An execution is *isolated* iff its constraint graph is acyclic.

Our exception mechanism provides the following two guarantees: (1) If an execution is exception-free, then it is isolated. (2) If an execution throws an exception, then it has a data-race.

These guarantees translate directly to programming language specifications: We can define data-races as we do now and specify that the program is *always* executed as an interleaving of regions, with the one simple adjustment that the second of two accesses that race in the execution may raise an exception.