A Case for System Support for Concurrency Exceptions

Luis Ceze, Joseph Devietti, Brandon Lucia and Shaz Qadeer[†]

University of Washington
{luisceze, devietti, blucia0a}@cs.washington.edu

[†]Microsoft Research
qadeer@microsoft.com

Abstract

In this position paper we argue that concurrency errors should be fail-stop. We want to put concurrency errors in the same category as division-by-zero, segmentation fault in unmanaged languages and cast exceptions in managed languages. This would make nondeterminism in multithreaded execution be much more manageable. Concurrency exceptions would improve the debugging process during development and make crashes due to concurrency errors that happen in the field be more descriptive. Our goal in this paper is to justify our position, propose a general approach to concurrency exceptions and discuss system requirements and implications. Specifically, we discuss the semantics of concurrency exceptions at the language level, their implications in the compiler and runtime systems, how they should be delivered and, finally, how they are enabled by efficient architecture support.

1 Introduction

The nondeterministic nature of multithreaded execution causes concurrency errors to manifest intermittently. This leads to major difficulties in the debugging process. In addition, state dumps of multithreaded program crashes rarely provide useful information. Concurrency errors are typically inserted when programmers incorrectly use synchronization primitives or simply overlook the need for synchronization due to wrong assumptions about the code and system behavior.

There has been a significant amount of work on tools to detect concurrency bugs such as data-races [14] and atomicity violations [10, 11]. However, while such tools are very useful, most past proposals have false positives and false negatives, and often impose a significant runtime overhead. With parallel programs becoming pervasive, we strongly believe that concurrency errors should be treated as exceptions. In other words, they should be detected and treated like a segmentation fault in unmanaged languages, a division by zero, or cast exceptions in managed languages.

Treating concurrency errors as exceptions requires that the system provide strong guarantees of concurrency error situations, without sacrificing performance. One way to think of it is as a pervasive, always-on concurrency error detector that is 100% accurate and causes negligible performance impact. We envision this being done by clearly specifying conditions that are not allowed and have the system constantly check for those exact conditions (not approximations). This, in turn, requires involvement of all pieces of the system stack, from the specification of the exceptions' semantics in languages, to mechanisms for exception delivery in the runtime system to architecture-level hooks to precisely and efficiently detect the situations considered concurrency errors. In this paper we advocate for concurrency exceptions and discuss their implications and requirements across the system stack.

We discuss exceptions for three types of concurrency errors: (1) sequential consistency violation, (2) locking discipline violation, and (3) atomicity violation. For clarity, we now briefly and informally define each one in the context of this paper. A *sequential consistency violation* happens when reorderings of memory operations with respect to the program order and ambiguities in the global order of writes (non-atomic writes) might change the semantics of the program. For example, two accesses to two different locations in the same thread were executed out of program order and a remote thread accesses the location whose accesses were reordered — i.e., a remote thread observed the reordered accesses. A *locking discipline violation* happens when a piece of code does not hold the appropriate lock while accessing shared data. Finally, an *atomicity violation* occurs when the programmer fails to enclose inside the same critical section all accesses that should be atomic. Note that these categories of concurrency errors are different. For example, atomicity violations do not necessarily imply locking discipline violations.

Why Not Simply a Data-Race Exception? The term data-races is overloaded. It is used to refer to several different forms of concurrency errors. However, it is important to understand what is meant by a data-race because several memory models are defined for "data-

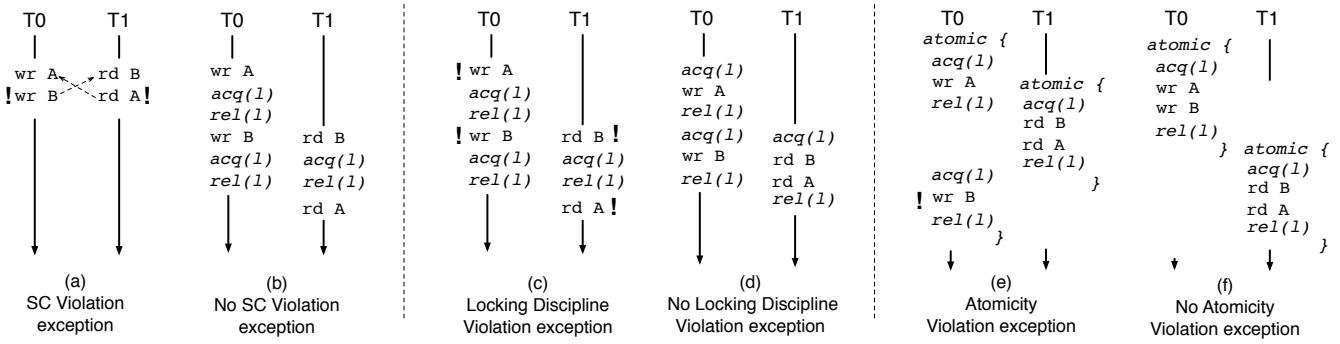

Figure 1: Examples of the concurrency exceptions discussed in this paper. In (a) the dashed arrow denotes a happens-before relationship. In (c) and (d) the locking discipline maps data items A and B to lock l. In (e) `atomic` specifies the region intended as atomic by the programmer — i.e., it is not a transaction. The exclamation point (!) indicates where the exception should be delivered. Notice the overlap between the examples, no SC violation (b) does not imply that there are no locking discipline violations (c), and no locking discipline violation (d) does not imply that there are no atomicity violations (e).

race-free" programs. For example, memory models such as the Java Memory Model [12] and the upcoming proposal for the C++ memory model [1] specify sequential consistency behavior at the language level for data-race-free programs. We believe that lack of data-races is important only to yield sequentially consistent executions and therefore not affect the operational semantics of programs. Given that the exact definition of what is a data-race varies with the memory model specification of the language, it becomes very hard to specify a generic "data-race" exception. Our goal is to provide end-to-end guarantees about concurrency exceptions, from the language to the hardware. Therefore, we choose to advocate a sequential-consistency violation exception as opposed to a data-race exception. In addition, we argue that data-races per se are not very useful indicators of synchronization defects. Both locking discipline violations and atomicity violations can occur in "data-race-free" executions and these two defects are much more meaningful to the programmer. Hence, in this paper we also advocate exceptions for locking discipline and atomicity violations, which can happen even in sequentially consistent executions.

2 Support for Concurrency Exceptions

Broadly, to support concurrency exceptions we need exactness in the definition of concurrency error conditions, absolute precision in detecting these conditions, and absolute efficiency, since this feature will be enabled continuously. Given these requirements, there are virtually no current mechanisms for concurrency error debugging that could be used to support concurrency exceptions.

We divide the support for concurrency exceptions into three components: (1) specification of exceptional conditions, (2) detection of these conditions as a program executes, and (3) exception delivery. Table 1 provides an overview of the responsibilities of each level the system stack. The main trade-off in supporting concurrency exceptions is one of complexity versus performance cost. Concurrency exceptions can be provided with software-only support via either a managed runtime, compiler or binary instrumentation. This would not involve the complexity of supporting it in hardware but would likely cause a significant impact in performance due to dynamic checks [4, 5]. We believe that the best option is to support the condition checks in hardware, the same way it is done for division by zero and segmentation faults.

2.1 Specifying the Exception Conditions (Programming Language)

Sequential consistency violation exceptions require the language to convey information about synchronization operations (i.e., anything that results in fence operations) to the detection mechanisms in the lower levels of the system stack. In addition, while not required, conveying what pieces of data are thread-private also aids the detection mechanism, since violations of sequential consistency involve only shared accesses. Based on this information, the system determines whether the execution is *guaranteed* to be sequentially consistent — i.e., the system checks for sufficient conditions to guarantee SC. Figure 1(a) shows an example of such a violation: writes to A and B by thread $T0$ were observed out of order by thread $T1$.

Programming Language	• Specifying exception conditions • Exception event semantics
Compiler	• Check as much as it can statically • Map the necessary dynamic checks to the lower level
Runtime System/OS	• Dynamic checks in software • Exception event delivery • Determinism (reproducibility)
HW/SW Interface and Architecture	• Mechanisms for fast dynamic checks of exceptions • Semantics of the low-level exception events (state) • Define the interaction with the HW memory model • Determinism (reproducibility)

Table 1: The responsibility of each level of the system stack in supporting concurrency exceptions.

Locking discipline exceptions require language support to declare shared data, declare locks (or other synchronization objects) and the association between locks and data items. Whenever a data item is accessed and the thread does not hold the appropriate lock, an exception is raised (Figure 1(c)). To make this compatible with languages that support transactional memory, we envision making transactions implicitly acquire a virtual lock TL, which is necessarily treated as a different lock than any other lock in the program.

Atomicity violation exceptions require language support to allow the programmer to express atomicity specifications. We define *atomic block* as a specification of the intention of the programmer to have a block of code be atomic. If any execution violates the atomicity of a block of code declared as atomic, an exception should be raised (Figure 1(e)). In contrast, we see *transactions* as implementation of the enforcement of atomic blocks of code.

All these exceptions discussed above can co-exist but do not depend on each other. Also note that all cases discussed in this section might also apply to languages such as C/C++, where threads are supported as libraries. However, this might require lightweight program annotations and needs to be integrated into the exception model of the language.

2.2 Detecting Exception Conditions (Compiler/Architecture)

Detecting exception conditions involves the compiler and the hardware. Once the programmer provides a specification of the desired synchronization, locking discipline and the atomicity properties, the compiler should convey this information to the runtime system and the hardware. Ideally, if the languages allows, the compiler should attempt to statically determine whether exceptions are guaranteed to occur. If so, it should generate a compilation error. However, given limited knowledge at compile time, it is likely that a majority of checks for exceptions are going to happen dynamically, i.e., by the runtime system or the hardware.

The role of the architecture is to provide the necessary mechanisms to make detection of concurrency exceptions essentially free, i.e., no performance cost. Consequently, we need to define the hardware/software interface for these mechanisms and the semantics of their use.

The key challenge in detecting exception conditions for concurrency errors is that it needs to be 100% accurate. Below we describe how each type of exception can be detected and the challenges involved in doing so.

SC Violations. Precisely detecting violation of SC in executions of arbitrary programs is an NP-complete problem [8]. Past work proposed to check for sufficient conditions for SC [7] — i.e., these conditions guarantee that the execution is sequentially consistent, however lack of these conditions do not necessarily imply a violation of sequential consistency. The approach we chose for this position paper was proposed by Gharachorloo and Gibbons in [7]. Their mechanism detects when two or more memory operations are in flight in a processor and there are remote coherence events that conflicts with any in-flight access. Memory operations are considered in flight when they have not been globally performed yet — i.e., not all processors are guaranteed to have observed the effect of these memory operations. This is sufficient for SC because if this does not happen, it is guaranteed that no remote processors have observed the possible reordering of in-flight memory operations. Synchronization operations do not participate in this detection.

Figure 1(b) shows an example of an execution that does not violate SC. The synchronization operations in this example act as fences and therefore prevent two accesses to shared memory to be in-flight simultaneously and reordered, preventing an SC violation. However, as discussed later in this section, this execution has a concur-

rency error that will result in a locking discipline violation exception.

The mechanism just described detects possible violations of sequential consistency on the binary code generated by the compiler when running on systems weaker than sequential consistency. However, the memory consistency model of a language needs to be obeyed across the system stack; the compiler and the hardware together need to make sure that the programmer gets the specified behavior. Broadly, this is implemented by: (1) the compiler not reordering memory accesses across fences and (2) and the compiler conveying information about synchronization to the processor, generally in the form of fences. The first criterion guarantees that any execution violating SC in the binary program is possible even in the source program, thereby avoiding the compiler being a possible source of false SC violation exceptions. The second criterion prevents harmful dynamic reordering of memory operations by the hardware.

Note that the mechanism described earlier *may* detect violations of sequential consistency even if they were caused by the compiler reordering memory operations. However, unless the compiler conveys to the detection mechanism information about the order of operations in the original program (what the programmer actually wrote), it is not possible to pinpoint the exact cause of the possible violations of sequential consistency. In other words, since violations of sequential consistency need to be detected with respect to the original program, if compiler reorderings cause violations of sequential consistency, it can only be detected if information of the original ordering is available.

Locking discipline violation. The programmer specifies either directly or indirectly the lock objects associated with each piece of shared data. Note that depending on the synchronization model of a language, some of this information might be implicit — e.g., synchronized methods in Java implicitly use a lock. The compiler then conveys this information about shared data, locks and their association down to the runtime system and the hardware. The system keeps track of, for each thread, which locks are held. When a thread accesses a piece of shared data, the system determines whether the thread holds the appropriate lock. The actual hardware detection mechanism for this exception would be very similar to support for data-centric synchronization, as described in Colorama [2].

One complicating factor of specifying a locking discipline is that it might change dynamically. For example, consider a data hand-off scenario, when a thread hands ownership of a data object to another thread. This might imply that the locking discipline changes. Therefore, we need a way of specifying locking discipline as a dynamic property. A straightforward way of doing this is using annotations that allow the programmer to convey when there is a remapping between the data-structure and the associated lock.

Atomicity Violations. Based on the atomicity specification provided by the language, the detection mechanism determines if there is any interleaving that might violate the atomicity of the specified region. This can be done by determining the serializability of the remote accesses that touch any of the same data objects as the region specified as atomic. The actual detection mechanism in this case cannot directly use typical eager conflict-detection techniques used in transactional memory systems. Conflict serializability is a conservative approximation of true serializability, and of course for exceptions we need true serializability. In other words, the point in the execution where the exception should be delivered might be long after a conflict was detected. For additional information, see next section on exception delivery.

Discussion. Data object granularity is an important concern in the detection of all exception conditions described above. For example, SC violation detection cannot be blindly done at a cache-line granularity — if the data object is smaller than a cache line, it might yield false positives and if the data object is larger than a cache line, it might be yield false negatives. Supporting data objects at a byte granularity is the right choice. This implies that the languages/compiler should convey information about data objects boundaries to the runtime/OS/architecture. The hardware, in turn, needs to provide primitives for the detection of coherence events in objects that might be smaller or larger than a cache line. Proposals like Mondrian [16] for fine-grain memory protection are a good starting point for such support.

Also note that it is not necessary to have a sequentially consistent execution to enforce atomicity violation or locking discipline exceptions. One can think of it as the compiler and processor providing an execution that will be subject to the detection of exceptions. This makes it easier to reason about cases where the presence of data-races does not yield a sequentially consistent execution at the language level.

2.3 Exception Delivery (OS/Architecture)

Concurrency exception conditions involve memory operations from multiple threads. This creates two challenges for exception delivery: (1) which thread should receive the exception and (2) what are the guarantees of process state at the time the exception is delivered.

A concurrency exception should be delivered when the event that would make an execution incorrect is *about* to be executed but has not been executed yet. In addition, it should be delivered to the thread that is about to perform the memory operation that triggers the detection mechanism. For example, as Figure 1(e) illustrates, consider an atomicity violation where thread $T0$ performs operation wrA followed by wrB. These operations were interleaved by rdB and rdA from thread $T1$, which makes this interleaving unserializable and therefore characterizes an atomicity violation. We claim that the ideal place for the delivery of the exception is at wrB. This is because this is the memory operation that distinguishes a correct execution from an incorrect execution. If wrB had not been performed, the execution would not lead to an atomicity violation. Analogously, an SC violation exception should be delivered to the thread that issued the access that was reordered (e.g., wrB or rdA in Figure 1(a)) and led to the potential violation of SC. In summary, we believe the exception should be delivered at the event in the execution that distinguishes a correct execution from an incorrect execution.

The state of a multithreaded program includes the state of all its threads. Therefore, once an exception is detected, all threads in the program should be stopped. However, given the nondeterministic nature of multiprocessor systems, it is hard to provide a notion of precise state. There are at least two options to solve this problem. First, and most ambitious, is to make multiprocessor execution fully deterministic [3]. This would provide a deterministic total order of state transitions in the execution of a multithreaded program and therefore provide true notion of precise concurrency exceptions. Second, the system can choose to offer precise state only for the thread to whom the exception is delivered. We believe the former is a longer term and cleaner solution and discuss it later in this section. However, the latter is less complex.

The system should also allow the programmer to specify a handler for the exception. This would enable the use of a data-collection or "fix-up" handler and allow the system to collect state or repair the situation and proceed with the execution (i.e., survive the error [11]). One possibility is to enable user-level handling [15] of concurrency exceptions, which has much lower overhead. This might be important in cases when there are frequent benign exceptions. In addition, the system could provide ways to selectively disable exception delivery either based on code or data regions.

Deterministic Shared Memory Multiprocessing. Deterministic Shared Memory Multiprocessing (DMP) [3] is a family of execution models that remove all internal nondeterminism from multithreaded-execution. The execution of a multithreaded program on a DMP system is only function of its inputs. The key to providing determinism is making inter-thread communication via shared memory be deterministic. The easiest way to accomplish this is to allow only one processor at a time to access memory in a deterministic order. However, this completely serializes execution and removes the performance benefits of parallelism. DMP provides two strategies to recover parallelism, one that does not employ speculative execution and one that does. The first strategy leverages the fact that threads do not communicate all the time: threads run concurrently as long as they are not communicating; as soon as they attempt to communicate, communication is deterministically serialized. The second strategy speculate that periods of a thread execution do not communicate and rolls back and re-executes in a deterministic order if they do communicate. Simulations show that a hardware implementation of a DMP system can have negligible performance degradation over nondeterministic systems. This demonstrates that expecting deterministic execution from the ground-up is reasonable and would significantly increase the value of concurrency exceptions by making them reproducible.

3 Discussion

While we think the categories of exceptions we chose provides good coverage of concurrency bugs, we do not claim to have necessarily covered a canonical set. One category of bugs we have not addressed is ordering violations [9], which can happen in programs free of any of the exceptions we discussed in this paper. However, they can be difficult to specify. Another possibility we are investigating is detecting unspecified inter-thread *communication* and treat that as exceptions.

Concurrency exceptions might not be desired in lock-free code, as the programmer might intentionally allow situations that would normally be treated as exceptions. For that reason, it will likely be useful to allow concurrency exceptions to be disabled for periods of a program execution.

One could question why repeatability of exceptions (determinism) is useful if the system is able to accurately detect precise conditions for exceptions. That question is especially relevant if the system can collect enough information about the state of the program when the exception was delivered. We believe that being able to reproduce an exception significantly improves the ability of a programmer to fix the code to avoid the defect that lead to the exception. In other words, exceptions with precise state and repeatability of errors are useful by themselves but they are even more useful if offered together.

4 Related Work

To the best of our knowledge, Goldilocks [4] is the first paper in the literature to advocate that data races should be exceptions and to propose a way of providing such functionality. However, Goldilocks cause significant performance degradation which prevents it from being a viable option for deployment. In addition, Goldilocks was tailored to the Java programming language.

Eraser [14] is a tool to detect locking discipline violations. It includes the lockset algorithm, whose goal is to infer a locking policy based on the locks held when pieces of data are accessed. Flanagan and Qadeer [6] proposed a type system for Java to verify the atomicity of java methods. The analysis determines, based on the synchronized blocks and the atomic annotations whether the code might lead to a violation of atomicity. Velodrome [5] is a dynamic atomicity checker that is both sound and complete. While these tools and systems provide strong guarantees for concurrency error detection, they either don't offer soundness and completeness or cost too much performance.

ReEnact [13] is a hardware proposal for race detection based on vector clocks and on using support for speculative execution. ReEnact aims to be an always-on race detection mechanism suitable for production runs. Finally, HARD [17] is a proposal for a hardware implementation of the lockset algorithm.

Acknowledgments

The ideas and opinions on this paper have been evolving based on discussions with many people. We especially thank Karin Strauss and Dan Grossman for valuable comments and feedback.

References

[1] H. Boehm and S. Adve. Foundations of the C++ Concurrency Memory Model. In *Conference on Programming Language Design and Implementation*, 2008.

[2] L. Ceze, P. Montesinos, C. von Praun, and J. Torrellas. Colorama: Architectural Support for Data-Centric Synchronization. In *International Symposium on High-Performance Computer Architecture*, February 2007.

[3] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: Deterministic shared memory multiprocessing. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2009.

[4] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A Race and Transaction-aware Java Runtime. In *Conference on Programming Language Design and Implementation*, 2007.

[5] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *Conference on Programming Language Design and Implementation*, 2008.

[6] C. Flanagan and S. Qadeer. A Type and Effect System for Atomicity. In *Conference on Programming Language Design and Implementation*, 2003.

[7] K. Gharachorloo and P. B. Gibbons. Detecting violations of sequential consistency. In *Symposium on Parallel Algorithms and Architectures*, 1991.

[8] P. B. Gibbons and E. Korach. Testing shared memories. *SIAM Journal on Computing*, 1997.

[9] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from Mistakes - A Comprehensive Study on Real World Concurrency Bug Characteristics. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[10] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[11] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and Surviving Atomicity Violations. In *International Symposium on Computer Architecture*, 2008.

[12] J. Mason, W. Pugh, and S. Adve. The Java Memory Model. In *Symposium on Principles of Programming Languages*, 2005.

[13] M. Prvulovic and J. Torrellas. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. In *International Symposium on Computer Architecture*, 2003.

[14] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. *ACM Transactions on Computer Systems*, November 1997.

[15] C. Thekkath and H. Levy. Hardware and Software Support for Efficient Exception Handling. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.

[16] E. Witchel, J. Cates, and K. Asanović. Mondrian Memory Protection. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[17] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-Assisted Lockset-based Race Detection. In *International Symposium on High-Performance Computer Architecture*, 2007.