Controlling Non-determinism for Semantic Guarantees

Sriram Rajamani G. Ramalingam
Venkatesh-Prasad Ranganath Kapil Vaswani

Microsoft Research India
{sriram,grama,rvprasad,kapilv}@microsoft.com

Concurrent programs are hard to design, develop, and debug. It is widely accepted that we lack good abstractions to design and reason about concurrent programs, and good tools to debug concurrent programs. Recent technology trends, such as the increasing prevalence of multicore processors, make concurrent programming more important than ever.

Non-determinism arises in concurrent programs when the order in which threads can execute is unconstrained. While executions of concurrent programs on multiprocessors inherently exhibit non-determinism, the executions on uniprocessors exhibit non-determinism due to the choices of the thread scheduler in the underlying O/S. Undesired non-determinism is a major cause of errors in concurrent programs. Nevertheless, we believe that non-determinism can be *safely, permissively, and automatically controlled* to tolerate runtime errors in concurrent programs and to provide various desirable semantic guarantees.

In this position paper, we sketch some recent work we have done in this direction and outline our longer term goals along the same direction.

From Tolerating Races to Isolation Guarantees

Numerous efforts in the recent past observed that race conditions represent an important class of errors in concurrent programs and proposed various techniques to detect races. Nagpal et.al. [2] proposed TOLERACE, a runtime technique for *tolerating asymmetric races.* TOLERACE is a *fault tolerance* technique that attempts to ensure that a concurrent program does not exhibit undesirable behavior even in the presence of races.

Inspired by this effort, we recently proposed a technique [3] called ISOLATOR that provides stronger semantic guarantees than those provided by TOLERACE. ISOLATOR guarantees *isolation*, a fundamental property in concurrent programs. In a concurrent program, a thread T may read and/or update certain shared variables in a critical section of code and it may be necessary to ensure that other threads do not *interfere* with T in the critical section. Specifically, a critial section executes in *isolation* if the threads executing outside the critical section should neither observe values of nor update these shared variables. Isolation helps avoid undesirable outcomes arising out of unexpected interactions between different threads and it enables local reasoning (that is, it enables programmers to reason locally about each thread, without worrying about interactions from other threads).

Today, *locking* is the most commonly used technique to achieve isolation. Every shared variable is protected by a lock. A *locking discipline* requires that every thread hold the (corresponding) protecting lock while accessing a shared variable. We say that a thread is *well-behaved* if it follows such a discipline. If all threads are well-behaved, then the thread T holding the locks corresponding to a set of shared variables $\mathcal{V}$ will be isolated from any accesses to $\mathcal{V}$ from all other threads.

However, prevalent programming languages do not support mechanisms to ensure that such locking disciplines are indeed followed by all threads in a program. Thus, even when a thread T_{well} holds a lock ℓ protecting a shared variable g, nothing prevents another (*ill-behaved*) thread T_{ill} from directly accessing g without acquiring lock ℓ, either due to programmer error or malice. Such an access to g violates the isolation property expected by thread T_{well} and makes it impossible to reason locally about the program. Such interferences leads to well-known problems such as *non-repeatable reads*, *lost updates*, and *dirty reads*.

In recent work, we proposed a runtime scheme called Isolator that guarantees isolation (by detecting and preventing isolation violation) for parts of a program that follow the locking discipline, even when other parts of the program fail to follow the locking discipline [3]. Isolator exploits the available non-determinism in a concurrent program, and "steers" the program toward an execution that satisfy isolation.

The Idea. Isolator employs a custom memory allocator to ensure that all variables protected by a lock ℓ are allocated in the same page. Then, it exploits page protection to guarantee isolation. Specifically, when a well-behaved thread T acquires a lock ℓ, Isolator makes a local copy of the page, and turns on protection for the page. All further accesses to the page from thread $T$are redirected by Isolator to the local copy of the page. If an ill-behaved thread now tries to access a variable in the page without acquiring the lock ℓ, a page protection exception is raised, and is caught by an exception handler registered by Isolator. The exception handler code just yields control and retries the offending access later. The access succeeds only after the thread Thas released lock ℓ, at which point Isolator copies the local page back to the global copy and releases page protection on the global copy.

Isolation via Other Language Constructs

From a different perspective, Isolator assigns a different semantics for the locking primitives. The usual semantics of locking primitives is *operational*: if thread T_1 acquires a lock ℓ, then another thread T_2 cannot acquire ℓ until T_1 releases ℓ. However, Isolator reinterprets the primitives to guarantee a stronger property: if thread T_1 acquires a lock ℓ, and g is a variable protected by ℓ, then Isolator guarantees that other threads cannot read or write g until T_1 releases ℓ.

Thus, Isolator treats locking primitives more as a *specification* mechanism rather than as an implementation mechanism and steers program execution toward runs that satisfy the specification. We believe that this semantics, in fact, very naturally captures a programmer's intent in using locks.

From this perspective, ISOLATOR is close in spirit to language constructs that have been recently proposed, such as atomicity (often realized via software transactional memory (STM) [4, 1]). (However, most existing STMs support only *weak atomicity*, which can suffer from the same problem of isolation violation as manual locking [5].) We think that the implementation mechanism used by ISOLATOR may be useful for implementing features such as strong atomicity as well, though this remains as future work. However, ISOLATOR differs from such constructs in proposing a *semantic strengthening* of existing language features and idioms. As such, it may be easier to deploy and use a scheme like ISOLATOR in existing systems, where locking disciplines, designed and implemented by programmers, continue to be a predominant programming paradigm for concurrency (due to reasons such as the need to interoperate with legacy programs written in legacy languages as well as efficiency concerns).

Towards Semantic Isolation

We now describe our long term vision. Isolation, as provided by ISOLATOR or STMs, is a very desirable property. However, in general, that is not sufficient. The programmer still decides the *boundaries* of isolation: the code segment that is guaranteed isolation. Programmers can still make mistakes with regards to the boundaries leading to concurrency related errors. So, *can we improve the mechanism described earlier to provide even stronger semantic guarantees?*

Often, the ultimate goal of concurrrency control mechanisms (such as locking) is to ensure that certain desired invariants are maintained, which allow modular reasoning about the code. Code that reads a shared data-structure typically assumes that the data-structure satisfies its invariant. Code that updates the shared data-structure may temporarily break the invariants before reestablishing it. In such cases, the updating code locks the shared data-structure to ensure that no other code observes the data-structure in an inconsistent state (a state that violates its intended invariants).

Manually ensuring the above methodology is fraught with difficulties, involving the usual safety-performance tradeoff. On one hand, an emphasis on safety can lead to code where data-structures are locked even when not required, reducing concurrency and performance. On the other hand, aggressive optimization of the locking discipline can lead to incorrect code. *Note that holding a lock while accessing the data it protects is insufficient to guarantee that a thread will not observe data in an inconsistent state.*

Thus, our long term goal is to develop techniques that can *semantically guarantee isolation*: ensure that when one thread breaks invariants of a data-structure, no other thread can observe the inconsistent data-structure until the original thread reestablishes the invariants. We are pursuing runtime techniques, similar to ISOLATOR, that guarantee invariants at runtime. The idea is that the programmer specifies invariants that the shared data-structures are required to satisfy. An invariant-preserving runtime uses the invariants to “steer” the program toward an execution that satisfy the invariant. We are also investigating both the independent and the complementary use of static techniques, that utilize static program analysis, to meet the same goal.

We note that there are two possible modes for such mechanisms. The less ambitious possibility is to use the mechanism as a *fault tolerance* scheme. Here, the programmer still makes manual choices, e.g., in terms of acquiring and releasing locks. The runtime mechanism is used solely to work robustly even when the programmmer makes concurrency control mistakes. The more ambitious possibility is that the programmer does not worry about concurrency control. Instead, the system automatically identifies when to isolate data-structures using the invariants. (While the latter possibility is convenient, it requires programmers to write complete specifications to ensure correct operation. The former possibility, however, can be used even if the programmer specifies only a few of the desired invariants.)

Conclusion

Non-determinism is a source of trouble for concurrent programs. It forces people to reason about large number of interleavings. Also, testing becomes hard since it is hard to cover all the interleavings. Further, debugging becomes hard since it is hard to reproduce a failure due to a specific interleaving.

However, non-determinism can be our friend if we want to tolerate faults in concurrent programs. We can build static techniques and runtime systems that exploit non-determinism and steer the program towards executions that satisfy desired semantic properties.

References

1. T. Harris and S. P. Jones. Transactional memory with data invariants. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*, 2006.
2. R. Nagpal, K. Pattabiraman, D. Kirovski, and B. Zorn. ToleRace: Tolerating and detecting races. In *Proceedings of the Second Workshop on Software Tools for Multi-Core Systems (STMCS)*, 2007.
3. S. Rajamani, G. Ramalingam, V. P. Ranganath, and K. Vaswani. Isolator: Dynamically ensuring isolation in concurrent programs, 2008. *In Submission.*
4. N. Shavit and D. Toutiou. Software transactional memory. In *14th ACM Symposium on Principles of Distributed Computing*, 1995.
5. T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in STM. In *Proceedings of the Conference on Programming language design and implementation (PLDI)*, pages 78–88, 2007.