

Verifying Correct Usage of Atomic Blocks Using Access Permissions

[Extended Abstract]

Nels E. Beckman
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
nbeckman@cs.cmu.edu

1. INTRODUCTION

It is now widely accepted that the age of parallelism has arrived. Thanks mostly to heat density, it has become increasingly difficult for computer architects to improve the clock speed of modern CPUs. Instead, the steady flow of transistors given to us by Moore's law are now being used to add an increasing number of processing units, or cores, to CPUs. While this fact is widely understood, what is well not understood is what effect this fact will have on modern software development.

The question is, how will developers write software that can take advantage of multiple cores, given the difficulty of concurrent programming? One idea that seems to have gained a fair amount of traction is transactional memory [9]. Transactional memory is a means of simplifying mutual exclusion in shared-memory applications. While it is a subtle and complex topic, at its core, transactional memory provides a simple language primitive to programmers, the atomic block. The idea is that code executing inside of the atomic block will execute as if no other threads were executing at the same time. Transactional memory is so-named because this primitive is often implemented as a transaction that will execute optimistically and then abort and roll back its memory effects if the thread observes memory values inconsistent with its promised semantics.

While the semantics of atomic blocks are quite simple, especially compared with locks, it is not the case that writing concurrent code becomes simple with its mere presence. Unfortunately, while atomic blocks simplify the creation of critical regions, correctly determining which parts of code must execute atomically in order to ensure behavioral correctness remains difficult. Specify too much atomicity, and the performance of your application decreases, specify too little and your application may no longer behave correctly. The addition of blocking primitives such as the 'retry' operator [8], often needed for thread communication protocols, further complicates matters, as over-specification of atomicity can potentially lead to incorrectness [12].

At a high-level, the purpose of my thesis is to address the interplay between behavioral verification of object-oriented software and transactional memory. I will show how access permissions [3], a mechanism for controlling aliasing in object-oriented software, can also be used to specify thread

sharing, and that this information can in turn be used to verify type-state [13] annotations, simple behavioral annotations describing object protocols. The verification of these specifications helps programmers to determine whether or not atomic blocks are being used appropriately in order to ensure application correctness. Furthermore, while transactional memory helps to simplify the verification of object-oriented code, the same verification can help reduce the large overhead typically associated with transactional memory implementations. This extended abstract briefly describes the proposed thesis work.

2. PROBLEM

At a high level, the problem we are attempting to address is the difficulty in writing correct, multi-threaded code. Probably the most well-known problem encountered when writing multi-threaded software is the potential for race conditions. Consider the Java source excerpt seen in Figure 1.

In this program, a GUI for a network chat program uses an object of the Connection class as an abstraction of a network connection. In the trySendMsg of the GUI class, the GUI checks to see if its network connection is open before sending a message across the network. In this application, remote user could potentially close the network connection at any time. Therefore, imagine a second thread, the network call-back thread, also has a pointer to the same Connection object and may call disconnect in response to a remote disconnect. Now the code has a race condition that could potentially result in a null pointer dereference. This could happen if the GUI thread calls the isConnected method, it returns true, indicating that the connection is open, and at that precise moment, the network call-back thread closes the connection in response to a message over the network. The interesting thing to note in this example is that all accesses of thread-shared memory occur inside of critical regions. The race condition exists solely because another thread can potentially invalidate an implicit precondition of the send method.

Similarly, it is possible to use blocking primitives in a manner such that deadlocks are introduced into an application. In the world of transactional memory, the most commonly proposed blocking primitive is the 'retry' primitive [8]. The behavior of retry is as follows: when a thread executes the retry command, the thread aborts the currently executing

```

class Connection {
    void disconnect() {
        this.socket.close();
        this.socket = null;
    }

    boolean isConnected() {
        atomic: {
            return (this.socket != null);
        }
    }

    void send(String msg) {
        atomic: {
            this.socket.write(msg);
            this.counter.increment();
        }
    }
    ...
}

class GUI {
    boolean trySendMsg(String msg) {
        if ( this.myConnection.isConnected() ) {
            this.myConnection.send(msg);
            return true;
        }
        return false;
    }
    ...
}

```

Figure 1: An example where a race condition could occur.

transaction and returns to the beginning of the transaction. This primitive allows programmers to code up idioms that would normally use wait/notify. Consider the code excerpt shown in Figure 2.

This listing shows the partial implementation of a one-shot thread barrier. When a thread calls the `waitUntilFinished` method, it will not return until `NUM_THREADS` threads have also called this method. This functionality is achieved by having each thread increment a counter, `numWaiting`. The first threads to arrive will not return until the `stopWaiting` flag has been set, using the `retry` primitive to block until this time. When the last thread arrives, it sees it is the last thread because the `numWaiting` counter has reached `NUM_THREADS`, at which point it sets the `stopWaiting` flag and returns.

This code is correct, but its correctness depends on a subtle requirement: the `waitUntilFinished` method must not be called inside of a transaction. This is because the incrementation of the `numWaiting` variable performed by the first threads must become visible to the last thread so that it recognizes that it is in fact the last thread. (The `numWaiting` variable is a shared variable used for inter-thread communication.) If each thread were to call this method inside of a

```

void waitUntilFinished() {
    atomic: {
        numWaiting++;
        if ( numWaiting == NUM_THREADS ) {
            stopWaiting = true;
            return;
        }
    }

    atomic: {
        if ( stopWaiting ) return;
        else retry;
    }
}

```

Figure 2: A single-use barrier. This method cannot be called within a transaction.

transaction, then whichever thread were to arrive last would not see the results of the other threads' memory writes, due to the isolation property of memory transactions. While this example may seem trivial, due to the dynamic nesting of atomic blocks, it is possible to accidentally call a method inside of a transaction that should not be. This example is particularly troublesome because it affects the overall ability to compose code written using atomic blocks. Users of library code that internally makes use of blocking primitives should not necessarily have to be aware of this fact.

Both these examples serve merely to show that while transactional memory primitives can significantly ease concurrent programming, they can still be used incorrectly.

3. PROPOSED THESIS PLAN

For my thesis, I plan to develop an analysis based on access permissions [3] to solve some of the problems typically associated with multi-threaded programming.

Hypothesis. The goal of this thesis is to show that access permissions, which statically describe the aliasing behavior of program references in object-oriented programs, provide a good basis for the verification of the implementation and usage of object protocols in concurrent systems, allowing us to verify real programs and provide optimizations of the underlying runtime system.

3.1 Approach

Concurrent Typestate Verification with Atomic Blocks. In work that I have already begun [2], we have discovered that by recasting the access permissions of Bierhoff and Aldrich [3], which were originally developed as a means of statically controlling aliasing patterns, into a concurrent context, we were able to soundly verify the object protocol implementation and usage of multi-threaded code. This verification helps developers ensure the absence of "application-level" race conditions, race conditions that exist at the level of program logic. This work uses the atomic block, a primitive provided by transactional memory systems, as its mutual exclusion primitive. We have developed a type system,

proven sound, that formalizes our verification methodology. We have also developed a modular, branch sensitive data-flow analysis for the static verification of Java source code based on the formal rules of this system.

While a full description of access permissions is outside the scope of this abstract, it is quite helpful to understand the five basic permissions.

Unique A unique reference points to an object that is not referenced by any other variable in the program. In a multi-threaded context this permission indicates that only one thread can read and modify the object.

Full A full reference is a reference that can be used to read and modify an object that can only be read by other references in the system. In a multi-threaded context, this permission allows one thread to modify an object that several others can concurrently read.

Immutable An immutable reference points to an object that can be aliased but not modified by any number of references. In a multi-threaded context, all threads can read the object, but none can modify it.

Share A share reference points to an object that can be used to read and modify an object that any number of other references might also modify. In a multi-threaded context, all threads can read and modify the object.

Pure A pure reference is a reference that can be used to read but not modify the object that it points to. Moreover, other references (specifically full and share ones) may exist and have the ability to modify the object. In a multi-threaded context, this permission allows a thread to read but not modify an object that can be modified by other threads.

Software annotated with these permissions can be checked for consistency, at which point object protocol specifications can be modularly verified. Briefly, object protocols are abstract descriptions of the states that an object can be in, and determine the methods that can legally be called when objects are in a given state. The idea of statically checking object protocol conformance is known as ‘type-state checking [13].’ As an example, consider the following type-state specification for the `send`, and `isConnected` methods of Figure 1:

```
@Pure
@TrueIndicates("connected")
@FalseIndicates("disconnected")
boolean isConnected() { ... }

@Share(requires="connected", ensures="connected")
void send(String str) { ... }
```

The first specification states that, in order call the `isConnected` method, the caller must at least have pure (non-modifying) permission to the receiver, but that object can be in any state. Depending on whether the method returns true or false, at the return the receiver will be known to be in either the connected or disconnected state. The second specification states that, in order to call the `send` method, the caller must have a share (modifying) permission to the

receiver and that object must be known to be in the connected state. Putting it all together, the analysis itself works by tracking these states but forgetting any known state information for objects whose permissions indicate they could potentially be modified by other threads, that is pure and share permissions. At the moment, I have formalized and proven sound this analysis, and we are in the early stages of implementing a prototype checker.

Making Atomic Blocks Composable. We would like to add additional features to our analysis that will help to make atomic blocks more composable. Atomic blocks themselves have very straightforward semantics. Unfortunately, the combination of atomic blocks with blocking primitives such as ‘retry’ allows developers to inadvertently write code that is not correct in all contexts. This usually occurs when a piece of code must commit a transaction in order that the effects of that transaction become visible to other threads in the system. We have seen an example of this problem in Figure 2. One existing solution to this problem is the use of an effect system which requires certain methods to be called outside of a transaction [12]. Any method that calls that method, in turn, must be typed in the same way. Unfortunately, this restriction prevents libraries from developing their own threading architecture that can then be called by any client in any context. We wish to add a type-and-effect system that prevents certain methods from being called inside of atomic blocks until the point in the call hierarchy where threads are created. Because memory effects would be visible to these threads, code higher up in the call stack now is allowed to open atomic blocks. In this manner, libraries will be allowed to create their own threads and write code that requires blocking behavior, while at the same time remain call-able from transactional contexts.

Using Permissions to Optimize STM. It turns out that the access permission information that is specified as part of the verification process can be immensely useful for the underlying implementation of software transactional memory systems. Transactional memory systems achieve the apparent atomicity of the atomic block by undo-ing the memory effects of threads executing inside atomic blocks that see a memory state that is inconsistent with atomicity. In order to do this, a large amount of synchronization and logging of memory effects is required, which can impose a significant overhead. Using access permission annotations, some amount of logging and synchronization can be eliminated due to the static knowledge of the way in which a memory reference will be used. For example, if a reference is associated with an immutable permission, the memory reads of that object’s fields do not need to be logged, and no synchronization must be performed, because we know statically that they object cannot ever be modified by any thread. We have developed a source-to-source translator for the Java programming language that incorporates optimizations based on permission annotations, and we have seen improved performance in a small suite of examples.

3.2 Evaluation

At a high level, the goal for evaluation of our specification system is to ensure that it is expressive enough to be able

to describe most common patterns of multi-threaded object-oriented code while maintaining a relatively low annotation burden. To this end, the specification and verification of real programs is an important goal. I plan to perform at least five case studies using the tool on reasonably sized open-source Java programs, as well as many smaller programs. The expressiveness of our system will then be determined by the number and significance of the false-positives that are reported by our analysis, since these false-positives are typically the result of a systems' inability to precisely describe the thread-sharing and aliasing patterns that are used in an application. Across this set of applications, I will measure and report the average number of annotations per line of source code, and compare this result with the well established OO behavioral verification systems JML [11] and Spec# [10]. One risk is that there may not be many multi-threaded OO applications that have a need for tpestate verification due to their design. In order to address this risk, I intend to immediately begin the search for appropriate Java programs. The desire to specify and check real programs has made the development of an automated checker a high priority. Currently development is underway, and the tool is mature enough to allow me to check some small examples. At this point I have specified and verified around ten toy programs, but this early work has already been helpful in suggesting improvements for the analysis

At a high level, the goal for evaluation of our optimizations is to ensure that they have a significant improvement on the runtime performance of multi-threaded object oriented code. To this end I plan to specify real Java programs and benchmark their performance before and after the optimizations. While we would hope for improvement in the performance of every program we benchmark, the nature of our optimizations suggests that performance is more likely to improve in applications with infrequent memory contention. Therefore, our evaluation will attempt to characterize the relationship between performance and contention with respect to our optimizations. In order to perform these evaluations, I have extended a source-to-source implementation of software transactional memory to use access permission-based optimizations. For the few examples that have been bench-marked, performance gains have been modest, but statistically significant.

4. RELATED WORK

There is a large amount of related work in the area, but our work provides enough improvement over the state-of-the-art to be compelling. When compared with the most similar work [10, 11], systems for specifying and verifying the behavior of lock-based object oriented code, our work is different in that it allows for much richer aliasing contexts. Many works in the area impose an ownership discipline, which does not allow certain real program to be specified. Our focus on atomic blocks simplifies our system by reducing annotation burden (locksets no longer must be specified), but it also allows for some interesting interplays such as our STM optimizations. Other logics for specifying the behavior, such as rely-guarantee [14] and concurrent separation logic [4] allow much more expressive behavioral specifications, but do not allow as many thread-sharing patterns and are not as amenable to automated analysis.

A large amount of work has also been done in the area of static race detection [5] and race-free type systems [7]. While these tools work well at statically prevent data races, they do not handle the sort of race condition seen in Figure 1, and cannot because they do not allow for behavioral specifications. Open nested transactions [1] have been proposed to provide programmers with a kind of "back door" to transactions that allow memory modifications to become visible to other threads before the entire transaction has committed. This solution can be used in examples similar to the one shown in Figure 2 but is generally considered to be extremely error-prone, a solution for experts only [12]. Finally atomicity checkers [6] can verify that the appropriate locks are acquired in order that a method body execute as if it were an atomic operation. However, these systems require the developer to know which methods must execute as if atomic to ensure the correctness of the application, whereas our approach can will tell a programmer which sections of code must execute atomically in order to preserve behavioral specifications.

5. REFERENCES

- [1] K. Agrawal, C. E. Leiserson, and J. Sukha. Memory models for open-nested transactions. In *MSPC*, pages 70–81, New York, NY, USA, 2006. ACM.
- [2] N. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of atomic blocks with tpestate. In *OOPSLA*, 2008. To Appear.
- [3] K. Bierhoff and J. Aldrich. Modular tpestate checking of aliased objects. In *OOPSLA*, 2007.
- [4] S. Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, 2007.
- [5] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. *SIGOPS Oper. Syst. Rev.*, 37(5):237–252, 2003.
- [6] C. Flanagan and S. Qadeer. A type and effect system for atomicity. *SIGPLAN Not.*, 38(5):338–349, 2003.
- [7] D. Grossman. Type-safe multithreading in cyclone. In *TLDI*, pages 13–25, New York, NY, USA, 2003. ACM.
- [8] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05*, pages 48–60, New York, NY, USA, 2005. ACM.
- [9] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, 1993.
- [10] B. Jacobs, K. Rustan, M. Leino, and F. Piessens. Safe concurrency for aggregate objects with invariants. In *SEFM*, 2005.
- [11] E. Rodriguez, M. Dwyer, C. Flanagan, J. Hatcliff, G. T. Leavens, and Robby. Extending sequential specification techniques for modular specification and verification of multithreaded programs. In *ECOOP*, 2005.
- [12] Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. Transactions with isolation and cooperation. *OOPSLA*, 2008.
- [13] R. E. Strom and S. Yemini. Tpestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1), 1986.
- [14] Q. Xu, W. P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2), 1997.