Accelerating Native Calls using Transactional Memory

Dave Dice Victor Luchangco John Rose

Oracle

Abstract

Transitioning between executing managed code and executing native code requires synchronization to ensure that invariants used by the managed runtime are maintained or restored after the execution of the native code. We describe how transactional memory can be used to accelerate the execution of native methods by reducing the need for such synchronization. We also present the results of a simple experiment that, although preliminary, suggests that this approach may have a significant effect on performance. Unlike most of the work on exploiting transactional memory, this approach does not depend on concurrency and improving scalability. Indeed, the experiment presented uses a single-threaded benchmark.

Categories and Subject Descriptors D.1.m [*Programming Techniques*]: Miscellaneous

Keywords transactional memory, managed runtime, native methods, JNI

General Terms Design, Performance

1. Introduction

Managed runtimes such as the Java Virtual Machine (JVM) [5] provide many advantages by abstracting away machine details and providing facilities such as automatic memory management (i.e., garbage collection), just-in-time (JIT) compilation, and various security and sandboxing guarantees. However, sometimes a program being run in a managed runtime environment must execute code written in a language other than that understood by the managed runtime. For example, the code may use legacy code written in another language, or it may require platform-specific features, or it may require more fine-grain control than is possible through the managed runtime. Such *native code* is typically written in a lower-level language such as C or C++ or even assembly, and it is not guaranteed to preserve invariants maintained by the managed runtime. Thus, the runtime must carefully manage transitions between executing managed code and executing native code, to ensure that the required invariants hold when managed code is executed. Also, certain functions of the managed runtime, such as garbage collection, may need to treat threads running native code differently from threads running managed code. For example, the runtime may use safepoints to avoid races between threads executing managed code and the managed runtime itself. But threads executing native code may not respect these safepoints. Thus, native calls are typically "bracketed" by expensive checks to ensure that the assumptions of the managed runtime are not violated. For these reasons, calls to native methods can be expensive.

In this paper, we describe a technique to accelerate native calls by using transactional memory to execute the native code. Unlike uses of transactional memory to improve scalability by enabling greater parallelism, this technique aims at improving performance by eliding some of the expensive bracketing code for transitioning between native and managed modes of execution. Thus, we expect to see benefits even for single-threaded code. We present results of a simple benchmark that suggests that this technique may have a significant impact on performance.

2. Managed Runtimes, Safepoints and Native Code

To provide many of their benefits (e.g., safety and security guarantees, automatic memory management and just-in-time compilation), managed runtimes impose and maintain invariants necessary to ensure correct execution of the managed code. For example, a runtime may detect unreachable objects and reclaim the memory they use (i.e., garbage collection) and also relocate reachable objects. To do this, it must be able to access the threads' stacks to find and redirect managed pointers to the new memory locations. This requires the runtime to synchronize with the threads so that they do not see inconsistent views of the memory.

One common way to ensure that threads do not see inconsistent views is to use *safepoints*. During a safepoint, threads executing managed code are suspended while the runtime does the necessary maintenance. Whenever the runtime needs to do this maintenance, it sets a flag to indicate that it wishes to start a safepoint; we say that the safepoint is pending. Threads executing managed code periodically check for a pending safepoint; that is, the JIT compiler injects safepoint polling points into loops and other places in the managed code to ensure that these threads discover a pending safepoint in a timely fashion. A thread that discovers a pending safepoint suspends until the safepoint com-

pletes. When all threads have suspended, the safepoint is active and the runtime proceeds with the necessary maintenance. Because of all this synchronization, safepoints are expensive, so it is important for performance that they are rare.

Threads executing native methods can present a problem for a managed runtime: When the native method returns, the runtime must ensure that its invariants are maintained. This requires the returning thread to check for a pending safepoint, which can involve a relatively expensive synchronization operation. Also, the runtime typically provides some means for native code to access managed data—for example, Java provides the Java Native Interface (JNI) [4] for C functions to call into the JVM—and such calls may also require synchronization.

To amortize the cost of transitioning between native code and managed code, programmers often combine what would naturally be several separate native methods into a single native method. This has the unfortunate consequence of reducing modularity in the program.

3. Hardware Transactional Memory

Several recent systems [2, 3, 6] provide hardware transactional memory (HTM) support, which allows programmers to designate a section of code as a transaction. Such code must appear to execute atomically, in which case we say the transaction *commits*, or else it must *abort* and have no side effects other than an indication that the transaction failed and possibly a condition code suggesting the reason for the failure. For example, Intel's Haswell processor [2] provides Restricted Transactional Memory (RTM), which defines three new instructions: XBEGIN to start a new transaction, XEND to attempt to commit a transaction, and XABORT to abort a transaction. Failed transactions pass control to a fallback code path specified by the XBEGIN instruction with an abort status returned in a special register. Such HTM implementations are typically "best effort": they do not guarantee that any transaction will succeed. However, because they guarantee that the transaction appears atomic if it does commit, they can greatly simplify some concurrent code, and also improve its performance and scalability [1].

One common way to use HTM (and also STM—software implementations of transactional memory) is to do *lock elision*, by replacing critical sections protected by locks with transactions; that is, the lock acquisition and release are elided. This eliminates, or at least reduces, the false conflicts induced by locks that are too coarse-grain. However, because transactions may fail for any reason, it is typically necessary to provide a fallback path that reverts to acquiring the lock when a transaction cannot be committed.

4. Accelerating Native Calls Using HTM

In this paper, we describe an approach that exploits HTM to reduce the cost of transitioning between native and managed modes of execution. The basic idea is similar to that of lock elision: If a native method can be executed in a single transaction, then we can avoid the expensive synchronization bracketing the native call. In particular, because a transaction appears to execute atomically, if there is no safepoint pending when the transaction begins, then there is no need to check again when the native method returns. Also, perhaps more importantly, there is no need for any synchronization (beyond that provided by the transaction) when the native method calls back into the managed runtime.

```
class Box {
  int value;
  native void getFunction();
// declare native method
  native void setFunction();
// declare native method
}

// Test harness

r = new Box();
for 100 iterations {
  t = System.nanoTime();
  for 1000000 iterations
    r.GetFunction();          // native call
  report(System.nanoTime() - t);

  t = System.nanoTime();
  for 1000000 iterations
    r.SetFunction();          // native call
  report(System.nanoTime() - t);
}
```

Figure 1: Pseudocode for benchmark

5. A Proof-of-Concept Experiment

To investigate the viability of this approach, we have implemented a prototype proof-of-concept implementation and carried out a simple (and simplistic) experiment. The results, although preliminary, suggest that using transactional memory to execute native calls may yield significant performance benefits.

Pseudocode for the benchmark in our experiment is shown in Figure 1. The Box class contains a single integer field and declares native getFunction and setFunction methods. These methods are implemented in C: The getFunction method iterates 32 times, calling the JNI function GetIntField on the value field. Similarly, setFunction calls SetIntField 32 times. The benchmark simply reports the time to execute each of the native methods (i.e., getFunction and setFunction) one million times. To give the JVM's just-in-time compiler sufficient time to compile the code, we exe-

	no HTM	with HTM
getFunction	70	74
setFunction	320	204

Table 1: Times in milliseconds for a million native calls with and without exploiting HTM.

cute this code 100 times and use the times reported in the final iteration as the data points. Note that this benchmark is single-threaded: it tests the improvement in performance that we can get by reducing the cost of transitioning between managed and native modes of execution rather than by improving scalability.

We ran this benchmark using both off-the-shelf JDK8 and a JVM modified to accelerate native calls using HTM running on an i7-4770 processor with Ubuntu 14.04. We describe the modifications to the JVM in more detail below. The native C functions were compiled using GCC 4.9.1. The results are shown in Table 1.

The disparity between the times for getFunction and setFunction is due to a special optimistic fast path implemented in the JVM: There is a counter that is incremented whenever a safepoint begins or ends, so that it is odd whenever there is an active safepoint. The JNI Get accessors read this counter, fetch a tentative value from the heap, and then read the counter again. If the counter changed or was odd, then they revert to the regular slow path. However, if the value is even (i.e., there is no active safepoint) and it did not change between the two times it was read, then the fetched value is valid and we can return it. There is no corresponding fast path for SetIntField because it must mark the thread as a mutator to safely interact with potential safepoints.

To accelerate native calls, we modified the JVM so that it first tries to execute native methods using RTM: Whenever a native method is called, we execute the XBEGIN instruction to start a transaction and then check the counter used by the JNI Get functions (i.e., the one incremented whenever a safepoint begins or ends). If the counter is odd (i.e., there is an active safepoint), or if the transaction fails for any reason, we simply revert to the slow path. If the counter is even (i.e., no active safepoint), then we set a flag in the thread's structure to indicate that we are executing in a transaction and then call the native method. If a JNI function is called from a native method executed within a transaction, we can elide the synchronization ordinarily required when calling into the JVM because the transaction guarantees that there is no active safepoint. If a safepoint starts while the transaction is active, it will write the counter and cause the transaction to fail (so the implementation will revert to the usual path). If we successfully complete the native method (i.e., return without aborting the transaction), we clear the flag in the thread's structure and then execute the XEND instruction to commit the transaction.

As Table 1 shows, this optimization reduces the latency of setFunction by about 35%. It has no significant effect on the latency of getFunction because the optimistic fast path has much the same effect as using a transaction. We also experimented with running the native call as a transaction but still using the original bracketing transition code, and the times are about the same as the no-HTM version, which suggests that the performance boost is entirely due to eliding the transition code.

Our prototype implementation is, obviously, very simple, so we should not draw too much from it. However, it does suggest that this is a worthwhile direction to pursue.

6. Discussion

We have described a technique for using transactional memory to reduce the cost of transitioning between native and managed modes of execution. We have shown results from a simple benchmark that suggest that this technique may yield significant improvements, but we have not yet attempted to characterize this benefit over a range of applications. Naturally, applications that frequently transition between native and managed modes stand to benefit the most.

As mentioned earlier, the cost of transitioning between native and managed modes of execution also induces programmers to coalesce several conceptually distinct native methods into a single method. This practice is contrary to good software engineering, but is a pragmatic concession. Unfortunately, it also hurts the viability of the technique described here because large native methods are less likely to be successfully committed in a transaction. There is an interesting synergy here: using HTM to accelerate native calls may actually encourage programmers to write smaller native methods, improving the modularity of their programs in addition to their performance.

Also note that this technique can also be applied to calls from native code into the managed runtime (e.g., when calling JNI functions). This may be useful for cases in which the native method is too large to commit within a transaction, but the JNI calls are not.

There are several directions to extend this line of inquiry. First, and most obviously, additional more realistic benchmarks should be run. Also, because our benchmark is single-threaded, it does not give any insight about the impact this may have on concurrent code. That said, we do not expect any problems to arise combining this technique with other techniques that exploit transactional memory for concurrency control or scalability. Indeed, there may also be some synergy there: transactions that call native methods may have had to abort because of the expensive bracketing code. It may be possible that such transactions could be committed when using our technique to elide this bracketing code. However, this is mere conjecture as we have not explored this direction.

References

[1] Dave Dice, Yossi Lev, Mark Moir, Daniel Nussbaum, and Marek Olszewski. Early Experience with a Commercial Hardware Transactional Memory Implementation, 2009. Sun Labs Technical Report SMLI TR–2009–180.

[2] Intel Corporation. Transactional Synchronization in Haswell, 2012. [online; retrieved 2015].

[3] Christian Jacobi, Timothy Slegel, and Dan Greiner. Transactional memory architecture and implementation for IBM System z. In *Proceedings of the 45th International Symposium on Microarchitecture*, 2012.

[4] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.

[5] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification*. Addison Wesley, Java SE7 edition, 2013.

[6] Amy Wang, Matthew Guadet, Peng Wu, Jose Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of Blue Gene/Q hardware support for transactional memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, 2012.