Extending OpenJDK to Support Hybrid STM/HTM

Preliminary Design

Keith Chapman[†] Antony L. Hosking[†*] J. Eliot B. Moss[$]

[†]Purdue University, USA [*]Australian National University / Data61, Australia [$]University of Massachusetts, USA

keith@cs.purdue.edu antony.hosking@anu.edu.au moss@cs.umass.edu

Abstract

We have recently described and evaluated a research prototype system (called XJ, for *transactional* Java) that allows execution of Java programs extended with transactional memory (TM) abstractions (Chapman et al. 2014, 2016). The system allows mixed execution of these abstractions using both software (STM) and hardware (HTM) transactional memory. The prototype system, based on OpenJDK, suffers a number of roadblocks impeding production use, including: (*i*) Per-object metadata in support of manipulation of objects by transactions is inserted via bytecode rewriting at load time, in the form of a new word-sized instance field placed at the beginning of each object; (*ii*) The HotSpot optimizing compilers (C1 and C2) require gentle coaxing to compile both the STM and HTM versions of methods via alternating execution in a warm-up phase. Here we explore possible changes to OpenJDK that would allow for more integrated support for TM in HotSpot, as needed to support hybrid STM/HTM. These changes include encoding per-object transactional metadata in the synchronization word carried by all OpenJDK objects, and integrating more effectively with the profiling and compilation mechanisms of the HotSpot interpreter and compilers. We believe that the proposed changes are incremental, though we also expect that a deeper re-engineering would yield somewhat better ability to tune synchronization hotpaths for performance.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures; D.3.4 [*Programming Languages*]: Processors—code generation, compilers, incremental compilers, run-time environments

Keywords nested transactions, hardware transactional memory, software transactional memory, Java

1. Introduction

We have recently built and evaluated a research prototype system called XJ for execution of Java programs augmented with transactional memory abstractions (Chapman et al. 2016). That system relies on somewhat heavyweight mechanisms to support transactional memory mechanisms on a mostly unmodified OpenJDK platform. The bulk of the effort to make Java code run transactionally in XJ was achieved via bytecode and class rewriting at class load time. The only extension to OpenJDK is to allow injection of Intel's TSX hardware transactional memory (HTM) instructions into execution of interpreted and compiled code via HotSpot intrinsics. Two particular shortcomings of that approach are the addition of a transactional metadata word as an extra instance variable in all objects, and some jumping through hoops to convince the HotSpot optimizing compilers to compile HTM-enabled transactions. We discuss both of these issues with respect to OpenJDK and consider alternative implementations that represent a tighter integration with the OpenJDK implementation for improved performance. We thus hope to obtain feedback and promote useful discussion at the workshop.

2. Locking Protocol

XJ performs conflict detection at the level of objects, and tracks writes at the level of fields using an undo log. In the prototype STM implementation, each object carries an *extra* transactional metadata field, which holds the lock for writes, and otherwise contains a version number for the object, which is incremented upon commit. However, in XJ HTM and STM can safely co-exist and execute concurrently. Thus the two mechanisms need to play well with each other. In general, we adopt pessimistic concurrency control for writes, and optimistic concurrency control for reads. When running under STM, writes acquire a *write lock* on the object, which is noted in the metadata field—only one transaction can write to the object at a time. Readers proceed optimistically under STM, simply logging the value of the metadata field (a version number), and the log is then processed at commit time to validate the transaction (if the logged version number does not match the current value and the owner of a locked object is not the current transaction then the transaction aborts).

When running under HTM, writers commit by incrementing the version number, thus invalidating conflicting STM readers and conflicting with both HTM readers and writers. Reads under HTM perform a check to make sure that the object is not locked by another transaction, explicitly aborting if necessary. In sum, the lock/version word "glues" together the STM and HTM schemes into a coherent (and safe!) hybrid TM. We now present details on how this locking protocol can be integrated into OpenJDK, to be more efficient, rather than having to rely on an extra field added to each object.

2.1 Integrating Per-Object Transactional Metadata

The XJ prototype uses byte-code rewriting at load time to make every transactional application class inherit from a new `TransactionalObject` class, which has the transactional metadata word as its only instance field. Adding a field to each transactional object is costly in space and also in time to initialize and access the field. Ideally we would like this word to be a part of the object header. In OpenJDK every object is preceded by a class pointer (the "klass" word, which is native-sized or 32 bits depending on the use of compressed object pointers) and a header word. These are optionally followed by a 32-bit length word (if the object is an array), a 32-bit gap (if required by alignment rules), and then the object itself, comprising zero or more instance fields, array elements, or metadata fields. One option would have been to add another word to the header to store the transactional metadata. This would have worked well and is simple, but we want to do even better in terms of space and performance. Instead, we took a closer look at the format of the header word. Figure 1 shows the layout of the header word and how its contents evolve during the standard locking/unlocking process of Java object synchronization expressed using the **`synchronized`** keyword.

The most significant bits of the header word typically store multiple pieces of information as shown in Figure 1. These bits represent a hash code when the object is hashed, a thread id when the object is biased locked, a pointer to a lightweight lock, or a pointer to a heavyweight lock. The three lowest-order bits of the header word indicate which pieces of information the header holds. When an object is created and initialized it resides in the unlocked state (the most significant bits store no information). From this state an object can either transition to a hashed state or a biased locked state. If an object is hashed and a lock is requested (or vice versa), the object then transitions to a lightweight lock (the hash code and the thread id are moved into a lock record allocated on the stack). Lightweight locked objects that become subject to contention when another thread tries to lock them are "inflated": the object moves into a state where it refers to a heavyweight lock.

Transactions can be seen as an alternative method to achieve the same effect as **`synchronized`**: atomic updates to objects. It is reasonable to assume that any particular object is unlikely to be locked using both mechanisms, at least not at the same time. Thus an object that is participating in a transaction will not typically undergo all the states shown in Figure 1. We took this into account and tried to devise a mechanism to store the transactional meta-data in the existing header word. To do this we need a bit to indicate that the object is locked in transactional mode. We observed that we could accomplish this by enforcing 8-byte alignment on the "pointer to lock record" and the "pointer to heavyweight monitor". This gives us an extra bit to indicate that the object is being manipulated in transactional mode. Figure 2 shows how the contents of the mark word evolve under this scheme.

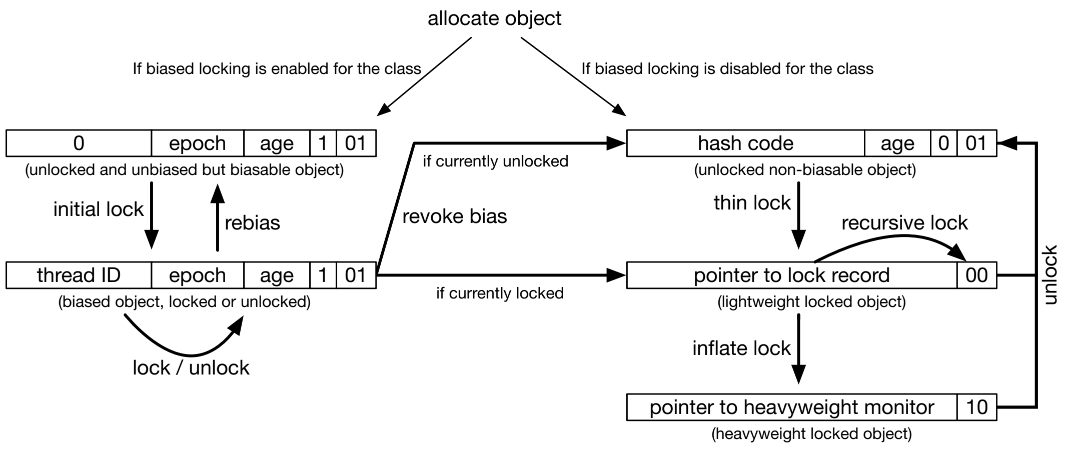
The proposed scheme allows us to store the transactional meta-data in the existing object header word, as long as the object remains unhashed and is not **`synchronized`**. Our approach allows efficient access to the transactional meta-data for the object when it is stored in the header word. This is the most common case, and our proposed scheme is optimized for it. One bit of the transaction meta-data is used to indicate if the value stored is a transaction id (to indicate that the object is write locked) or its transactional version number. If a hash code is requested for a transactional object, or it becomes **`synchronized`**, then the transactional meta-data will be moved to a heavyweight monitor ("fat lock"). The monitor will be augmented with a field to be used as the lock/version for transactions. It need not incur all the overhead of a standard object monitor except when used (in the rare case) for both transactional access and **`synchronized`** manipulations. If an unhashed object is unlocked then the transactional meta-data will be moved back into the header word making it more efficient.

2.2 Handling Statics

Integrating the transactional metadata into the object header word solves the transactional locking issue for instance fields of an object, but it does not address static fields of a class. We need to handle statics separately. In the XJ prototype this was done by moving the static fields into a separate static singleton object, which allowed us to use the same locking scheme used for instance fields on the static fields. For our modified OpenJDK VM we propose to have a distinct static field (a synthetic field) to hold the transaction metadata for the static fields of the object. The proposed scheme can be extended to have a distinct lock word for disjoint subsets of the statics, if that added complexity offers enough performance advantage. This might increase concurrency and could be easily implemented via an annotation, similar to the existing `@Contended` annotation, on a group of static fields.

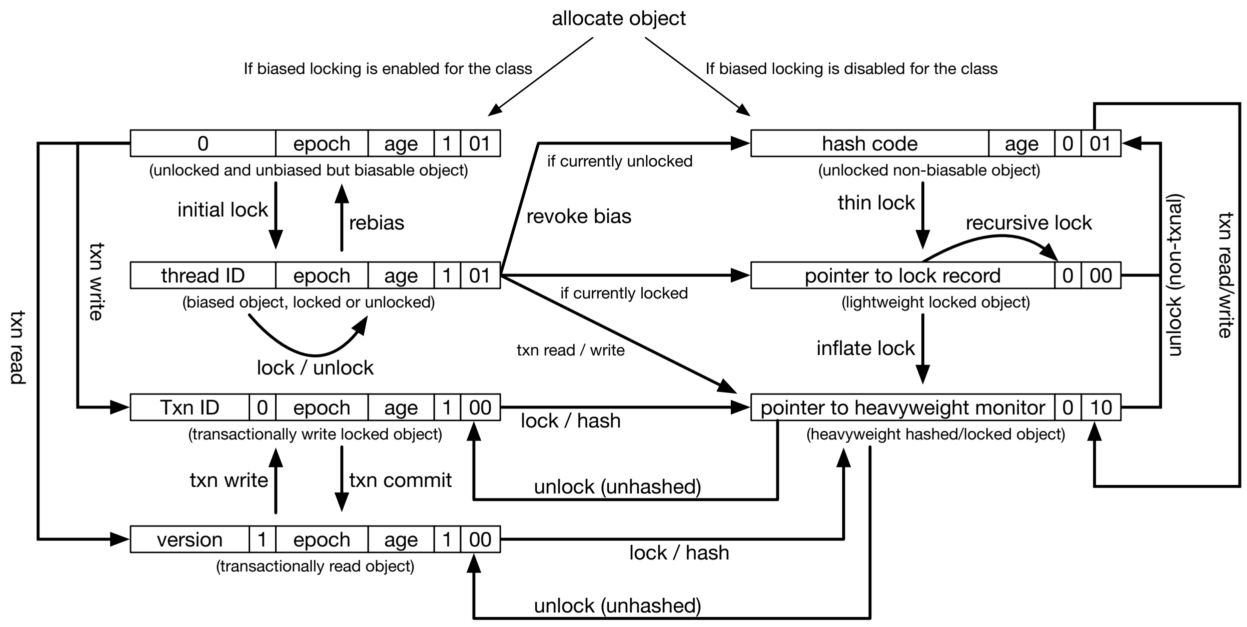
2.3 Handling Arrays

Using a single lock to protect a whole array does not scale in general since it will become a concurrency bottleneck. The XJ prototype injects wrapper classes at class load time for arrays, but we would prefer an integrated solution that allocates arrays as *arraylets*. These have been used to good advantage in real-time Java implementations (Siebert 2000;



allocate object
If biased locking is enabled for the class
If biased locking is disabled for the class
0 | epoch | age | 1 | 01
(unlocked and unbiased but biasable object)
initial lock
rebias
thread ID | epoch | age | 1 | 01
(biased object, locked or unlocked)
lock / unlock
if currently unlocked
revoke bias
if currently locked
hash code | age | 0 | 01
(unlocked non-biasable object)
thin lock
recursive lock
pointer to lock record | 00
(lightweight locked object)
inflate lock
pointer to heavyweight monitor | 10
(heavyweight locked object)
unlock

Figure 1. HotSpot standard synchronization
(reproduction under GPLv2 license of a figure appearing in Kotzmann and Wimmer (2008)).



allocate object
If biased locking is enabled for the class
If biased locking is disabled for the class
0 | epoch | age | 1 | 01
(unlocked and unbiased but biasable object)
initial lock
rebias
thread ID | epoch | age | 1 | 01
(biased object, locked or unlocked)
lock / unlock
txn read
txn write
Txn ID | 0 | epoch | age | 1 | 00
(transactionally write locked object)
txn write
txn commit
version | 1 | epoch | age | 1 | 00
(transactionally read object)
if currently unlocked
revoke bias
if currently locked
txn read / write
lock / hash
unlock (unhashed)
lock / hash
unlock (unhashed)
hash code | age | 0 | 01
(unlocked non-biasable object)
thin lock
recursive lock
pointer to lock record | 0 | 00
(lightweight locked object)
inflate lock
pointer to heavyweight monitor | 0 | 10
(heavyweight hashed/locked object)
unlock (non-txnal)
txn read/write

Figure 2. Proposed extension to the object header mark word

Bacon et al. 2003; Sartor et al. 2010; Pizlo et al. 2010). The size of these segments could be specified by the user. The integrated solution would allocate a transactional metadata word for each arraylet, solving the concurrency bottleneck issue for large arrays.

3. Interpreter and Compiler Concerns

As is well known, HotSpot has a byte-code interpreter as well as two levels of optimizing just-in-time (JIT) compilers (C1 and C2) that produce native code. Given the amount of work that the interpreter does, the data structures it touches and updates, etc., HTM will not work when interpreting byte-codes. This is because Intel's TSX hardware piggybacks on caching protocols and thus has a limited buffer size causing interpreted HTM transactions to fail due to buffer overflow. However, STM transactions can execute in the interpreter. HotSpot already uses reasonable heuristics to decide when it might be profitable to generate and execute native code. For transactional code, it might be useful to adjust those heuristics a bit since, once code is JIT compiled, HTM may be useful and HTM appears to run 5-10 times faster than STM for successful HTM transactions. But our main point is that HTM becomes interesting only for compiled code.

One of the main issues we encountered early on with using HTM was that many transactions failed with result code

0 (i.e., no specific reason given). Using the Intel Software Development Emulator, we found these aborts to be caused by execution of instructions that are incompatible with TSX—`FXRSTOR` and `FXSAVE` (perhaps among others)—and which are compiled into HotSpot's run-time stubs that control dynamic optimization and linking, and to resolve static and virtual method calls. By design, the HotSpot compilers patch these call sites at run time (Paleczny et al. 2001). Thus our hardware transactions always failed, *and* those failures prevented triggering of the patching mechanism. Our workaround was to devise a mechanism to "warm" the system up in STM mode before attempting any hardware transactions. However, so that the compiler's optimizations will be triggered appropriately, and so that linking/patching will occur, these STM transactions had to follow the same code path (except for not using the TSX instructions) as HTM transactions did. We used a global flag to indicate whether we were in the software-only warm up phase, "weaving" together STM and HTM in the same code sequence, with if-then-else structure for each operation that HTM and STM handle differently.

This "weaving" strategy allowed us to executed HTM versions of methods in software to "snap links," etc., as we say. For example, a transaction might call some method `m` of the application where `m` is not yet JIT compiled. The HotSpot JIT compilers will insert a call to a *stub* routine that triggers compilation of the target method `m` if it is called, or, if by that time `m` has been compiled, will patch the stub to call the compiled code for `m`. Both behaviors of a stub cause an HTM transaction to fail, and unwind, thus not actually triggering the compilation or link-snapping behavior. We thus needed a way to execute the *same stub* under STM. Once the stub's behavior had been appropriately triggered, HTM would no longer fail going through that code path. The stubs of which we speak are examples of *guards*. We say a guard *succeeds* if it follows a path where no special condition needs fixing up; this will be a fast path. We say a guard *fails* when it follows a path for a fix up; this will be a slow path.

Weaving together HTM and STM versions leads to code that is probably slower than it can be, because of all the extra if-then-else blocks. Granted, good branch prediction reduces their cost some, but they still need to be executed and they may stress the branch predictor. It would be better to generate HTM code without these branches. We propose two ways to do this: (*i*) returning some information from a failing HTM transaction, and (*ii*) maintaining correlated HTM and STM versions of the code.

3.1 Using HTM Failure Codes

As previously mentioned, failing guards will cause HTM transactions to fail. This has the side-effect that the run-time system then does not know a guard failed and thus cannot fix it up. However, it turns out that an *explicit* abort of an Intel HTM transaction with the `XABORT` instruction can pass 8 bits of information back in the `EAX` register.[1] So, if a piece of code running under HTM has no more than 256 guards, the compiler could use the 8-bit code in the `XABORT` instruction to indicate which guard failed. This may allow a future execution of the transaction to succeed. (We say "may" because a future execution is not guaranteed to follow the same path through the code.)

But what if the HTM code region has more than 256 guards? This could happen in the presence of calls, etc. Here is a scheme to exploit multiple transaction attempts to extract more bits from the failing transactions and narrow down the set of failing guards to the one on which the system should act. First, we assume that there is a per-thread location (it could even be a register) that will indicate which retry of an HTM transaction with a failing guard we are on, and some previously returned information. The attempt number will initially be 0, will be 1 on the first retry, etc. The essence of the scheme is this. We assign each guard a unique number. We develop k hash functions (k is likely 4, given the particulars of our scheme), h_0 through h_{k-1}. On attempt j of a failing guard in a hardware transaction, we return $h_j(i)$ where i is the unique id of the failing guard. These hash function return a seven bit value. The eighth bit we use to indicate whether we are continuing or starting over. On attempts after the first, we check a failing guard's previous hash values against those noted as being returned by previous attempts. If they match, we indicate that and return the hash value for the current attempt. If they don't match, we indicate that and return $h_0(i)$. If we get through four attempts with matches, we will have 28 bits to identify a particular guard. In many cases we might need even fewer, but the scheme generalizes to extract any number of bits, at the cost of additional retries and the increasing risk that we may go down a different code path (depending on the nature of the transaction and of the guard). Notice that the hash codes can simply be groups of seven bits from the guard's unique number, which probably makes for simple code.

This scheme assumes that all we need to know is which guard failed. When updating a polymorphic inline cache (for example), we may desire to know which class was presented that was not in the cache. The same approach can be taken to extract more bits. An alternative would be to have code that would figure out which object's class was being dispatched on, etc. This could get complicated, so returning the information directly (if incrementally) may be simpler. It is certainly more general.

3.2 Maintaining Correlated Code Versions

An alternative to using the HTM failure codes is to maintain STM and HTM versions that have the same guards. This is like taking XJ's code and pulling out a version with all

[1] As an aside to designers of future hardware, we observe that it appears useful to be able to return more bits, and possibly even to have a memory region not subject to HTM semantics in which one could store "side" results of a failing transaction.

the “then” clauses of the HTM-STM if-then-else blocks, and another version with all the “else” clauses. Whenever an action is taken on a guard in one version of the code, we force the same action to occur on the other version. Thus, if the HTM code fails in a guard—a fact that can be indicated with just one distinguished result code value—we can run the STM version and if a guard fails, it will be fixed in both versions and we can try HTM the next time. If the HTM version can usefully indicate which guard failed (i.e., there are not too many guards, and the particular one does not require additional information), then the result code can be used to fix the guard in both versions and HTM retried. However, handling a failing guard is probably quite costly compared with the work that can succeed in an HTM transaction, and even compared with an STM version of that same work, so always running the STM version to trigger guard fixing is a reasonable strategy.

3.3 Further Optimizations

In XJ we supported hand annotation of various actions in a transaction, to enable us to elide locking or logging work. This is particularly applicable to STM code, since HTM inherently avoids some of the work, but it is also relevant to HTM code. We envisioned a byte-code optimizer that would perform the needed data flow analyses and then rewrite the byte-code (or insert the annotations). This could be done as an additional optimization pass in HotSpot, particular to transaction code.

4. Conclusions

The changes to OpenJDK that we have proposed here are intended to be non-intrusive, though we recognize several opportunities for deeper re-engineering. First, our proposal to overload the existing object header word also to include transactional metadata attempts to leave the code sequences for other uses of that word unchanged. We introduce only one case-split on the extra bit that distinguishes transactional from other metadata. However, we expect that the trade-offs inherent in the current design may change as developers more regularly exploit transactional memory. There are alternatives to the existing biased locking design that may spare some bits in the header and allow a more compact representation of both transactional and synchronization metadata (Pizlo et al. 2011). Second, the arraylet scheme would be more effective for concurrency on arrays, and could be useful even in the non-transactional case.

We also show how the HotSpot optimizing compilers can be modified to be aware of transactions, such that HTM can be used in production. We propose two complementary modifications to the compiler that avoid having to warm the system up prior to using HTM. We also discuss other optimizations that the compiler could perform on HTM methods.

We hope that the thought experiment represented by this position paper will provoke discussion and feedback, and look forward to the workshop accordingly.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants CCF-1408896, CCF-1409284, CNS-1405939, CNS-1161237, and CNS-1162246.

References

D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 285–298, New Orleans, Louisiana, 2003. doi: 10.1145/604131.604155.

K. Chapman, A. L. Hosking, J. E. B. Moss, and T. Richards. Closed and open nested atomic actions for Java: Language design and prototype implementation. In *International Conference on Principles and Practice of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 169–180, Cracow, Poland, Oct. 2014. doi: 10.1145/2647508.2647525.

K. Chapman, A. L. Hosking, and J. E. B. Moss. Hybrid STM/HTM for nested transactions on OpenJDK. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Amsterdam, The Netherlands, Nov. 2016. doi: 10.1145/2983990.2984029.

T. Kotzmann and C. Wimmer. *OpenJDK Wiki: Synchronization and Object Locking*, 2008. URL https://wiki.openjdk.java.net/display/HotSpot/Synchronization.

M. Paleczny, C. Vick, and C. Click. The Java Hotspot server compiler. In *USENIX Java Virtual Machine Research and Technology Symposium*, Monterey, California, Apr. 2001. URL https://www.usenix.org/legacy/events/jvm01/full_papers/paleczny/paleczny.pdf.

F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: Fragmentation-tolerant real-time garbage collection. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 146–159, Toronto, Canada, 2010. doi: 10.1145/1806596.1806615.

F. Pizlo, D. Frampton, and A. L. Hosking. Fine-grained adaptive biased locking. In *International Conference on the Principles and Practice of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ, pages 171–181, Kongens Lyngby, Denmark, Aug. 2011. doi: 10.1145/2093157.2093184.

J. B. Sartor, S. M. Blackburn, D. Frampton, M. Hirzel, and K. S. McKinley. Z-rays: Divide arrays and conquer speed and flexibility. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 471–482, Toronto, Canada, 2010. doi: 10.1145/1806596.1806649.

F. Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES, pages 9–17, San Jose, California, 2000. doi: 10.1145/354880.354883.