TRANSACT 2014

A Hybrid TM for Haskell *

Ryan Yates Michael L. Scott
Computer Science Department, University of Rochester
{ryates,scott}@cs.rochester.edu

Abstract

Much of the success of Haskell's Software Transactional Memory (STM) can be attributed to the language's type system and purity, which together allow transactions to be expressed safely and precisely. By construction, transactions are free from many of the perils that other systems must work hard to avoid. Users do have to work in a constrained environment as a result, but the popularity of Haskell's STM indicates that this is not a burden too hard to bear. At the same time, the performance of Haskell's STM does not reach the level achieved by recent systems for other languages.

The use of Hardware Transactional Memory (HTM) is just beginning, with Intel's release of the first widely available TM-capable processors. There has been excellent recent work building hybrid transactional memory systems that combine the performance benefits of HTM and the progress guarantees of STM. In this paper we present our ongoing work to build a hybrid transactional memory system for Haskell. Haskell's restricted environment provides an opportunity for us to explore designs that leverage compile-time information to improve performance while preserving safety. At the same time, some of the features of Haskell STM prove to be challenging to support efficiently.

1. Introduction

Since its introduction in 2005 [10], Haskell's Software Transactional Memory (STM) monad has become the preferred means of coordinating concurrent threads in Haskell programs, making Haskell arguably the first successful transactional memory programming language outside the research community. Some of this success is due to an interface that allows a thread to wait for a precondition within a transaction (via the `retry` mechanism), or to choose an alternative implementation of the entire transaction (via the `orElse` mechanism) if a precondition does not hold. The implementation of the Glasgow Haskell Compiler (GHC) run-time system is not without compromises, however. Transactions incur significantly more overhead than in recent STM systems for mainstream imperative languages. Both per-thread latency and scalability are very poor. Fairness in the face of contention is also poor, and significant amounts of per-transactional location metadata lead to high abort rates for large transactions. In consultation with the core Haskell development team, we are attempting to address some of these issues through changes to the run-time system and—more significantly—by using hardware transactional memory to run Haskell transactions in whole or in part. The semantics of `retry` and `orElse` pose challenges for this effort, because they require partial rollbacks: a transaction that aborts explicitly must "leak" information about the cause of the abort, so the surrounding code can tell whether (and when!) to retry or to attempt an alternative implementation. Hardware transactions do, however, offer significantly lower overhead in many common cases.

* This work was supported in part by the National Science Foundation under grants CCR-0963759, CCF-1116055, CNS-1116109, and CCF-1337224.

```
data TVar a = ...

instance Monad STM where ...

newTVar :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()

retry :: STM a
orElse :: STM a -> STM a -> STM a

atomically :: STM a -> IO a
```

Figure 1. Interface for Haskell's STM.

In this paper, we describe ongoing work on a hybrid transactional memory implementation for Haskell. Specifically, we employ Intel's Transactional Synchronization Extensions (TSX) [11] for low overhead hardware transactions within GHC's STM runtime. Sections 1.1 and 1.2 provide background on GHC's current STM implementation and on Intel's TSX, respectively. Section 2 describes our preliminary implementation, its limitations, and the various design choices we see for support `retry` and `orElse`. Preliminary results appear in Section 4.

1.1 GHC's STM

The Haskell programming language as implemented by the Glasgow Haskell Compiler has many innovative features, including a rich run-time system to manage the unique needs of a pure functional language with lazy evaluation. This system has served as a vibrant research laboratory for over twenty years and remains in active development, with over one hundred contributors. GHC's STM was introduced by Harris et al. in 2005 [10]. The implementation in the initial paper predates GHC's multicore run-time system, introduced by Marlow et al. in 2009 [15]. Some details of the initial STM implementation, together with motivations for key design choices, are provided in a companion paper by Harris et al. [9].

The STM API is simple and explicit, as seen in the Haskell code in figure 1. Each transactional variable (`TVar`) is a heap object containing a mutable reference to some other heap object. The type (`a`) of the referenced object is a generic parameter of the `TVar` constructor. The structure of a `TVar` is not exposed to the user but must be accessed through the `readTVar` and `writeTVar` operations. Similarly, new variables are created with `newTVar` and not directly with a data constructor. Each of these operations is a function that returns an "STM action."

Because Haskell is a pure functional language, it has no side effects. STM actions are not statements in the sense of C or Java,

```
deposit :: TVar Int -> Int -> STM ()
deposit account value = do
    balance <- readTVar account
    writeTVar account (balance + value)

deposit' :: TVar Int -> Int -> STM ()
deposit' account value =
    readTVar account >>= \balance ->
        writeTVar account (balance + value)
```

Figure 2. A simple Haskell transaction written with do-notation and the same transaction with an explicit bind operator (>>=).

but rather actions of the STM *monad*, which provides syntactic sugar for "threaded" tail-recursive functions. Semantically, each STM action takes the entire transactional heap (the contents of all TVars) as an implicit hidden argument. In addition to its visible return value, it then returns the (hidden) contents of (a new version of) the transactional heap, which can then be passed to further STM actions.

Haskell input and output are mediated by the IO monad, which provides the illusion of threading the state of the outside world through tail-recursive functions. Values can be moved from pure functional code into the STM monad by passing them as arguments to writeTVar. STM actions can in turn be promoted to IO actions by using the atomically primitive to identify them as a transaction. Rules on the use of monads ensure that side-effect-like state transformations associated with I/O and concurrency are never visible to purely functional code. But just as GHC doesn't really pass the state of the entire outside world from one IO action to another, neither does it copy the entire transactional heap from one STM action to another. In practice, it mutates values in place, much as STM systems for conventional imperative languages do. The language retains the benefits of purely functional semantics, while the executing program uses impure effects for efficiency.

There are important differences, however. The static separation between transactional and nontransactional values eliminates the concept of privatization and its implementation challenges [14]. More significantly, GHC's strict controls on runtime-level side effects should make it easier to construct a sandboxed TM runtime [1], in which "doomed" transactions can safely be permitted to execute beyond the point at which they first read inconsistent values from memory (we return to this subject in Section 2.3).

STM actions are the building blocks for transactions and can be composed together with the monadic bind operator (>>=), which carries the (conceptually) mutated heap from one action to the next. Haskell simplifies the syntax even further with its do-notation, giving binding a fully imperative appearance. Figure 2 shows a simple transaction written using do-notation. The function deposit takes two parameters, an account represented by a TVar holding an Int and a value to be added. The result of the deposit function is an STM action that produces a "unit" value (similar to the void type in C). The body of the action follows the do keyword and is built out of two smaller STM actions resulting from applying values to readTVar and writeTVar. We compose these actions by binding the name balance to the result of running readTVar and then providing a second STM action that has that name in scope. For comparison, deposit' is the desugared version of the same transaction using the bind operator and a lambda function binding the name balance and resulting in the writeTVar action.

As previously noted, the atomically function identifies an STM action as a transaction and promotes it into an IO action. The entry point to a Haskell program is the IO action named main. An IO action is not restricted in the effects it may have when

```
withdraw :: TVar Int -> Int -> STM ()
withdraw account value = do
    balance <- readTVar account
    writeTVar account (balance - value)

transfer :: TVar Int -> TVar Int -> Int -> STM ()
transfer a b value = do
    withdraw a value
    deposit b value

main = do
    a <- atomically (newTVar 100)
    b <- atomically (newTVar 100)
    c <- atomically (newTVar 0)
    forkIO (atomically (transfer a b 100))
    forkIO (atomically (transfer b c 100))
```

Figure 3. Running a transaction atomically. Three accounts are initialized then two threads run transactions concurrently. The end result must always be $a = 0$, $b = 100$, and $c = 100$

run. Having STM as a separate type from IO statically prevents non-transactional effects inside transactions. Only STM primitive operations and the evaluation of pure functional code are executed when an STM action is run.

1.1.1 Run-time Support

GHC serializes transactions using either a coarse-grain lock or per-TVar fine-grain locks. We focus on the coarse-grain version as it is more easily converted to a hybrid system. In both versions, each transaction is represented by a record called a TRec, with an entry for every transactional variable touched during the execution of the transaction. The entries in the TRec store the value seen when the variable was first encountered and any new value written during the transaction. Values are always pointers to immutable heap objects. A read of a TVar first searches the TRec for a previous entry and uses its value. If no entry exists it reads the value from the TVar itself and adds an appropriate entry to the TRec. Similarly writes look for a previously read entry, adding a new one if the variable has not been seen, and in either case writing the new value into the entry. In the coarse-grain implementation, the global lock is acquired when the transaction attempts to commit. While holding the lock, the transaction validates its state by ensuring that all accessed TVars still hold the expected value recorded in the TRec. If they do, the transaction can commit by writing all the changed values into their respective TVars and releasing the global lock.

The fine-grain implementation is similar, but at commit time it acquires a lock for each TVar. The value of the TVar—the reference it contains—does double duty as the lock. Initial reads spin on locked values. At commit time, validation is done by first checking for consistent values and acquiring locks for any writes, then checking for consistent values again (as in the coarse-grain version) now that the locks are held. Reads from other transactions will be blocked, and other attempts to commit will fail when seeing a lock as a TVar's value. Read-only transactions need not acquire any locks: they commit after the second validation. Writer transactions unlock TVars by overwriting their locks with the corresponding new values. Each TVar also has a version number that is incremented when a new value is written. This version number is what is checked in the second validation. It allows a transaction—call it T—to notice if two other transactions have altered and then restored the value of a TVar while T was acquiring its locks. This implementation draws heavily on the OSTM system of Fraser and Harris [7], but without its nonblocking progress guarantees.

The majority of the STM implementation is written in C, with additional portions in Haskell and in `Cmm`, GHC's flavor of the C-- intermediate language. The `Cmm` code provides shims to call into the C library and is exposed to Haskell code as primitive operations. Our work mainly comprises changes to the run-time system. It also incorporates changes to the code generator, to allow a critical path in hardware transactions to avoid calling out to C code.

1.1.2 Limitations

An important design decision in Haskell's STM is the choice to declare separate transactional variables, and to separate STM effects from the IO monad. These separations facilitate programmer reasoning by statically disallowing effects that cannot be rolled back inside transactions. The run-time system strongly mirrors the separation of transactional variables by representing a `TVar` with a heap object containing a pointer to the actual data value. Unfortunately, this indirection significantly increases the memory footprint of each transaction. Harris et al. justify the design [9] by noting that the usual style of transactional programming in Haskell entails a relatively small number of transactional variables. Indeed, Haskell programmers get along well without mutation for the majority of each program. This justification impacts another choice in the STM as well: Each `TRec` is an unordered list of accessed variables, which must be searched on each transactional read, leading to $O(n^2)$ time for a transaction that reads n variables. The cost is reasonable for operations on queues or other simple containers, where only a handful of variables needs to be touched. For larger structures it could quickly become problematic.

To minimize the number of `TVar`s accessed in a transaction, Haskell programmers commonly rely on the language's lazy evaluation and devote a single `TVar` to each large (pure, functional) data structure. A pure map structure with thousands of nodes, for instance, can be the value held by a single `TVar`. Inserting a value into the map can be achieved by pointing the `TVar` at a thunk (an unevaluated function) that represents the computation of the insertion. A later (post-transactional) lookup to the map would then have the effect of forcing the thunk to evaluate. There is no reason this lookup needs to be done inside a transaction—just the mutation at the `TVar` that moves the pointer to the new computation.

Building up delayed computations is not without its downsides, however. The collection of unevaluated computations can sometimes use much more space than the final evaluated structure would occupy. To minimize this sort of "space leak," programmers commonly employ pure data structures that are "spine strict," where pointers to children are always evaluated when the parent is evaluated. Map data structures are usually also "key strict," where the keys are fully evaluated and often unboxed if the representation of the value allows.

1.2 Intel TSX

Intel's recent release of their 4th generation core architecture introduces support for best-effort hardware transactional memory, called the Transactional Synchronization Extensions (TSX) [11]. This support comes in two forms: a backward-compatible hardware lock elision (HLE) mechanism, and a more flexible restricted transactional memory (RTM) mechanism. HLE is invoked by adding `XACQUIRE` and `XRELEASE` prefixes to the instructions used to acquire and release a lock, respectively. (The lock must be written in such a way that the second instruction restores the value overwritten by the first.) With the prefixes in place, the hardware elides the initial write to the lock, and instead begins a transaction. Operations performed inside the transaction remain local until the transaction commits. If another thread performs a conflicting operation on any accessed location, the transaction aborts, rolls back, and starts over, this time actually acquiring the lock and ensuring forward progress.

RTM uses special `XBEGIN` and `XEND` instructions to mark the beginnings and ends of transactions. Additional instructions allow a thread to test if it is currently executing a transaction (`XTEST`) or to cause that transaction to abort (`XABORT`). `XBEGIN` takes as argument the address of a handler, to which execution will jump if the transaction aborts. Hardware transactions can be nested, but an abort will undo all levels. There is no support for nontransactional reads or writes within transactions, and the only information to escape an aborted transaction comes in the form of an 8-bit code.

Certain instructions (e.g., a system call or x87 floating point instruction) will immediately cause a transaction to abort. Even if these instructions are avoided, there is no guarantee of success. TSX in its initial implementation uses the L1 data cache to buffer a transaction's speculative writes. Conflicts are detected at cache line granularity, and the number of written lines is limited by L1D capacity and associativity. Speculative reads may safely overflow the L1, but the summary structure that tracks them when they do may sometimes announce a false conflict. For all these reasons, programs that use hardware transactions must provide a software fallback to ensure progress. At the same time, cache misses are typically a significant part of the cost of a transaction, so a transaction that retries (still in hardware) after a conflict-induced abort can benefit from cache warmup, and may be more likely to succeed the second time around. For this reason, a high abort rate for hardware transactions is not always a clear indicator of poor performance.

1.3 Related Work

Early work on hybrid TM includes the systems of Damron et al. [4] and Kumar et al. [12], both of which add instrumentation to hardware transactions to enable them to interoperate correctly with software transactions. In an attempt to avoid this instrumentation, Lev et al. [13] arrange for the TM system to switch between hardware and software *phases*, with all transactions executing in the same mode within a given phase; the resulting performance is often superior, but brittle.

More recent work builds on the NOrec system of Dalessandro et al. [2], which uses value-based validation, and serializes transaction write-back with a global lock. The subsequent Hybrid NOrec [3] leverages this design to allow uninstrumented hardware transactions to run concurrently with everything except the commit phase of software transactions. Performance in this system is best when hardware transactions are able to perform non-transactional reads of the software commit-phase lock. Felber et al. [6] and Riegel et al. [**?**] present variants of this scheme with similar performance.

Recent work by Matveev and Shavit [16] also builds on Hybrid NOrec, but employs two levels of fallback. The first uses hardware transactions for part of the work, allowing it to coexist with commits of hardware transactions. With the second level of fallback, hardware transactions are at risk of observing inconsistent partial writes—a problem we address in Section 2.3.

Throughout the current paper, we emphasize issues unique to Haskell's STM. In particular, we leverage the benefits of pureness (side effect freedom) and of strict separation between transactional and nontransactional code and data. At the same time, we must address the challenges of `retry` and `orElse`. In all cases, we restrict ourselves to the hardware support available with TSX.

2. Hybrid TM

We have a preliminary implementation of a hybrid transactional memory for GHC that uses Intel's TSX in several forms. In Section 2.1 we look at the simple scheme of eliding the coarse-grain lock during the STM commit phase. This simple scheme has some benefits over the fine-grain implementation and serves, in Section 2.2, as the fallback mechanism for a second scheme, in which we attempt to run Haskell transactions entirely in hardware. Trade-

offs and interactions among these schemes are discussed in Section 2.3. Possible mechanisms to support `retry` and `orElse` are discussed in Sections 2.4 and 2.5, respectively.

2.1 Eliding the Coarse-grain Lock

There is a simple way to apply TSX in GHC's STM implementation, without any significant changes to the run-time system. The coarse-grain version of the system has a global lock protecting the commit phase. We can apply hardware lock elision to the global lock and, in the best case, this will allow independent software transactions to perform their commits concurrently. In comparison to the fine-grain version of the system, this strategy avoids both the overhead of acquiring and releasing multiple locks and the overhead incurred on the first read of any `TVar` to check that the variable is unlocked. It does not, however, address the overhead of maintaining the transactional record: to allow transactions to read their own writes and to avoid interfering with peers, instrumentation is still required on transactional loads and stores.

If an HLE-enabled commit aborts for any reason, it will retry after acquiring the global lock "for real." This acquisition will cause any concurrent hardware transactions to abort. Elision will resume once the system passes through even a brief period of quiescence, with no transaction actively committing.

2.2 Executing Haskell Transactions in Hardware Transactions

The goal of executing an entire Haskell transaction within a hardware transaction is to avoid the overhead of STM `TRec` maintenance, relying instead on hardware conflict detection. In the coarse-grain implementation, when running in a hardware transaction, we simply read and write directly to the referenced `TVar`s. Implemented naively, we avoid the need for a `TRec`, but our transaction could start and commit in the middle of a software transaction's write phase, seeing inconsistent state. Since our fallback is the coarse-grain lock, we can fix this problem in a manner analogous to that of Dalessandro et al. [3] and Felber et al. [6], by including the global lock in our hardware transaction's read set and checking that it is unlocked. Our hardware transactions will be aborted by *any* committing software transaction, as every software transaction acquires the lock at commit time. When we elide the lock as described above, the speculative commit of the fallback can be rolled back when the hardware detects conflicts with committing hardware transactions. In the absence of such (true data) conflicts, software transactions can commit concurrently with a running hardware transaction—aborts will not be caused by conflicts on the coarse-grain lock itself.

2.3 Interaction Between Transactions

Now consider more closely how our layers of fallback interact. When the fallback is speculating on its commit (via HLE), no other transaction can see a partial commit. If, however, the fallback is fully in software, a concurrent transaction *can* see a partial commit. Reading the global lock and using `XABORT` if it is held fixes this issue as mentioned above. The question is *when* to do this read. If we read the lock at the beginning of an all-hardware transaction, then we will abort if we overlap with any part of a software transaction's commit phase—even when the software transaction touches a disjoint set of data locations. If we read the lock late—at the end of the transaction, just before committing—an overlapping software commit may be able to complete and release the lock before our read. The hardware transaction will still see any true conflicts, but the window of false conflicts is greatly narrowed.

There is a serious problem, however, with reading the lock late. The hardware transaction executes based on the data it reads, and this data may be inconsistent. Application-level invariants, which the user is explicitly using transactional memory to protect, are not guaranteed to hold. Assumptions based on those invariants could lead the code to jump to some harmful instruction. In a hardware transaction, perhaps the most harmful situation would be to jump to an `XEND` instruction, committing the transaction before it has read the lock and realized it needs to abort.

Consider, for example, a program that maintains the application-level invariant that `TVar`s a and b hold arrays that are always the same length. Consider then a transaction that reads the length of the array in a and uses that length to calculate an index at which to access the array. Knowing that the arrays are of equal length, the programmer might be tempted to use the built-in `unsafeAt` operation to read the array in b at the same index. If the application-level invariant does not hold because of concurrent activity in another transaction, we can imagine a scenario in which an invalid read off the end of b causes the transaction to jump to an `XEND`.

Fortunately, so far as we have been able to determine, *every* possible scenario in which a "doomed" transaction does something that might be visible to the rest of the program requires that the programmer have employed (directly or indirectly) a function explicitly marked as `unsafe`. Correctness can be maintained, we believe—even for an implementation that delays reading the global lock until the end of all-hardware transactions—by prohibiting the use within transactions of `unsafe` operations that read from more than one `TVar`.

It is important to note that similar problems already exist in GHC STM today. Validation occurs only at commit time; in both the fine and coarse-grain systems, a transaction may read—and compute on—inconsistent state. Jumping to an `XEND` is clearly not an issue, but segmentation faults can arise in the presence of `unsafe` operations. (It is also possible, in the current implementation, to fall into an infinite loop, even without the use of `unsafe`. If the loop contains memory allocation, the garbage collector will eventually force a validation and terminate the loop. Non-allocating loops need to force a similar validation; the fact that they do not is a known bug, which will eventually be fixed.) In other words, restrictions that are already required for correctness in STM Haskell suffice to enable late reading of the global lock in hardware transactions. Removing the restrictions would require that both the coarse- and fine-grain STM systems employ incremental validation.

Interestingly, our preliminary results do not show a performance difference between early and late reading of the global lock in hardware transactions. We expect the issue to become important as we implement further optimizations.

2.4 Supporting Blocking

STM Haskell's `retry` operation poses a challenge for all-hardware transactions. In our preliminary implementation, when we encounter a `retry`, we simply abort and have the fallback handler try again immediately or switch to the STM code path. A more difficult issue arises when we have a transaction that executed `retry` under the STM. In this situation, the transaction's Haskell thread is suspended and placed on a *watch list* for every `TVar` in its read set. When another STM transaction commits it will wake up every thread in the watch lists of all the `TVar`s in its write set. A hardware transaction must somehow provide analogous functionality.

In our preliminary implementation, we arrange for hardware transactions to record the address of each write in a transactional record. This proves to be too costly to be effective in practice, largely because the write set implementation performs dynamic memory allocation. In ongoing work, we are exploring solutions that require constant space. It is safe, but less efficient, for a transaction to be woken when it still cannot make progress: it will simply block again. Given this fact, the simplest option is to wake up all blocked transactions when a hardware transaction commits. Given

one bit of space we can distinguish read-only transactions, which need not perform any wakeups. With more bits we can create a summary structure that conservatively approximates the write set, with no false negatives. We have room then to tune the accuracy of wakeup by adjusting the amount of space we reserve for each write set. For the benchmark results in Section 4, which do not require `retry`, we approximate the performance of a constant-space write set by eliding the wakeup code in writer transactions: any transaction that encounters a `retry` in our current system restarts immediately, as if it had suffered a conflict abort.

The constant-space approximation of write sets suggests a similar mechanism to support `retry` in hardware transactions. The difficulty here is that we need to discard the effects of an aborting transaction while recording the locations that it read—and that if written should cause it to retry. Using `XABORT` to discard the effects also discards any record of the read set. If nontransactional stores were available in TSX they would allow us to "leak" a representation of the read set. In their absence, TSX does allow a small amount of information to leave an explicitly aborted transaction—namely the 8-bit abort code. If we reserve one bit to indicate whether a transaction aborted due to `retry`, the other 7 could be used to encode the read set as a miniature Bloom filter. An additional improvement would be to detect read-only transactions, which could record a more precise read set, commit instead of aborting, and then wait for the usual wakeup. Compiler passes similar to those discussed in Section 2.5 could move or buffer writes to just beyond the decision that leads to a `retry`; this would make a transaction read-only up to the `retry`. Unfortunately, all these mechanisms require instrumentation on reads as well as writes.

It seems feasible to support different granularity in what the STM and HTM can track. The two would perform wakeups at whatever precision was available. Even false negatives (such as those induced by an inappropriately empty read set) might be tolerable if we provided a mechanism to periodically wake all suspended transactions. Certain programming idioms, however, might lead to unreasonable delays. Some Haskell programs, for example, use `retry` as a barrier, to signal the next phase of a computation; these might experience a significant pause between phases. Other programs, including those based on the popular `async` library, create suspended transactions to represent a large number of very rare conditions. Waking all of these periodically might lead to unacceptable amounts of wasted work.

Any design that employs a summary structure based on hashed `TVar` addresses will also need to consider the impact of relocating garbage collection. The simple option is to wake all waiting transactions after a GC pass, letting them either make progress or wait on a newly computed read set. Another option would be to include in each `TVar` a stable identifier not affected by GC. This could even be a hash of the address at the time of allocation; the uniqueness of the identifier is not important for correctness.

2.5 Supporting Choice

A transaction can be built out of a choice between two transactions by using the `orElse` function. It takes two transactions as arguments, running the first one and, if it blocks, discarding the writes and then running the second transaction. This composed transaction's read set always includes the read set of the first transaction. That is, if the second branch of the `orElse` commits, it must do so in a state where the first branch *could not* do so. This means we cannot abort the first branch before we try the second, because global state could change in-between. If the first branch is to execute within a hardware transaction, it must buffer its writes so they can be rolled back (without aborting the hardware transaction) in the event of a `retry`, before attempting the second half.

```
tA = do
    a <- t1         -- t1 may write.
    if p a          -- p is some predicate.
      then retry
      else t2 a

tB = tA `orElse` tC

----------

tB' = do
    (a, t1w) <- t1 -- t1 is now read only and
                   -- produces an action t1w
                   -- to perform its writes.
    if p a
      then tC      -- Directly jump to the
                   -- alternate transaction.
      else do
        t1w        -- Perform the writes of t1.
        t2 a
```

Figure 4. Transactions such as `tB` can be converted into transactions such as `tB'`, by having `t1` delay writes into an STM action `t1w` that will perform the writes after the choice leading to a `retry` is resolved.

The overhead of buffering leads us toward using GHC's Rule system [17] and compiler passes [5] to improve the performance of common scenarios. A tree of nested `orElse`s, for example, can be refactored so all left branches are leaves. This leads to only a single level of write buffering, rather than an arbitrary number. Other code could be rewritten to explicitly delay writes until after the blocking choice, avoiding buffering in the run-time system by making it explicit in the transaction's (intermediate-level) code. For an example, see Figure 4. This sort of transformation would work well when the choice of a `retry` can be resolved with only reads and when analysis can fully identify reads after writes. It also avoids the overhead of checking at each read to see if the read should come from a buffered write. Several common scenarios should be able to benefit from such analysis. One particularly important instance appears in the `TChan` structure of the `stm` package, which attempts to dequeue an item from any of several queues.

Given the potential of this approach and the overhead of buffering, it makes sense to make hardware transactions execute the first branch of an `orElse` and fall back to STM when a `retry` is executed and that branch has performed a write. Repeatedly trying the first branch could result in too much bias towards that branch and should be avoided. This means we should indicate in the abort that the fallback path is required due to an `orElse`.

3. Performance Challenges

Several aspects of the standard STM Haskell runtime pose significant barriers to low per-thread latency and reasonable scalability. We have addressed the first of these (Section 3.1) in our current implementation; the others are work in progress.

In all our efforts, we have benefited from the richness of applications that already use the STM monad (a richness not yet found in any other language). We are continuing to import and develop further benchmarks to identify the strengths and weaknesses of alternative hybrid strategies, especially with regard to `retry` and `orElse`. We are also exploring modifications to the STM runtime to improve the performance of both STM and HTM transactions.

For the record, we do not currently plan to support the data invariants described by Harris and Peyton Jones [8]. While these are

a feature of GHC's STM, they have, to our knowledge, never been used. Their implementation has significant performance implications when any data invariants are active. For instance, when any affected variables are touched by a transaction, locks must be taken for the entire read set. In addition, the semantics of data invariants with regard to `retry` and `orElse` remain unclear and the current implementation can lose invariants at run time. These issues can be addressed, but as the original intent was simply for debugging, we see little need to support invariants in our hybrid system.

3.1 The Haskell-to-C Impedance Barrier

In GHC's current design, each read and each write in a transaction must perform an external, "foreign" function call to the STM runtime system, written in C. This call is handled by a `Cmm` primitive operation. This mechanism is quite unfortunate for hardware transactions: the difference in calling conventions requires registers to be stashed and a C stack put in place for the foreign call. The resulting memory accesses pollute the read and write sets of the transaction, which logically only needs to read or write directly to the `TVar`. The extra overhead quickly leads to transaction capacity aborts on TSX. We solve the problem by modifying GHC's code generator to issue an `XTEST` instruction and perform the `TVar` access in-line in the HTM case.

Initial experiments indicate that a transaction with in-lined accesses can read thousands of `TVars` and still commit successfully in hardware. Without inlining, the largest successful single-threaded commit is around fifty `TVars`.

3.2 Transactional Arrays

One aspect of Haskell's STM to which we are devoting particular attention is the implementation of transactional arrays. The current implementation provides an inefficient `TArray` class, implemented as an array of `TVars`. The values held are references to heap objects rather then direct data values—even when the values are of types that would normally not be boxed. As an example of the resulting inefficiency, consider a program in which transactions manipulate values in a large array of integers. If we implement this manipulation in Haskell using the current `TArray`, each integer will be represented by an immutable heap object with a pointer-sized header and an integer payload. Each `TVar` array cell will in turn be at least the size of two pointers and an integer. Every write to a cell is likely to force the allocation of a new heap object for the value.

We are in the process of constructing an alternative implementation in which arrays of unboxed values can be accessed transactionally, giving the benefit of many transactional variables with the overhead of a single `TVar`. This implementation will allow hardware transactions to directly read and write to a compact array of values, leveraging the hardware's natural fine-grain conflict detection. The design may sacrifice granularity for the STM—detecting conflicts at a coarser granularity than hardware transactions, with the expectation that most transactions will commit in hardware. The simplest strategy treats the whole array as a single `TVar` when in the STM. More complicated strategies "chunk" the array for finer STM granularity.

3.3 Transactional Structures

A related performance optimization would allow values with a fixed representation to live directly in a `TVar`. The implementation could mirror that of transactional arrays, but need not be exposed to the user in the same form. We imagine the compiler inferring when direct representation could be applied: GHC already performs some "unboxing" analysis. The motivation for this feature is to reduce the indirection overhead when the transactional variables store values with fixed representations, with a nicer interface than transactional arrays.

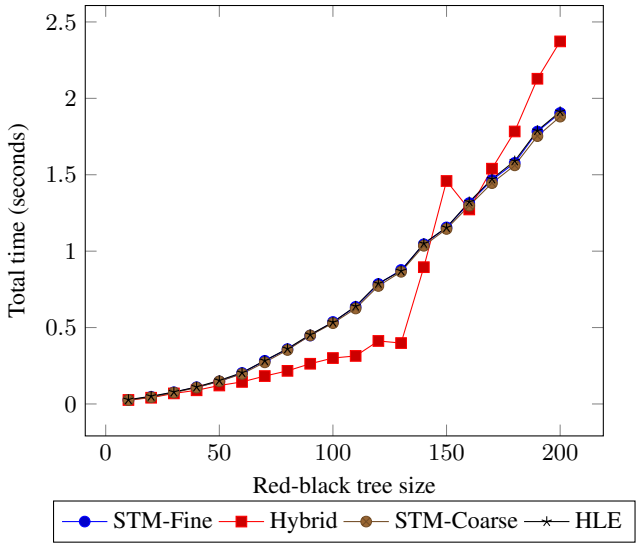

Figure 5. Reads from a red-black tree. At each size tree, each of 4 threads performs 5,000 transactions of 40 lookups each.

3.4 Full Inlining

It is not yet clear what the impact will be of widespread use of `XTEST` in our code. We plan to explore more complete separation of the code paths of all-hardware and fallback transactions. This might be accomplished using GHC's existing type class dictionary inlining mechanisms. It would entail a change to the user-facing API, potentially also allowing users to indicate which transactions should attempt to execute in hardware and which should always execute in software.

4. Preliminary Results

We have implemented our hybrid TM system in GHC by augmenting the existing STM support in the run-time system. For comparison we show results from the existing STM under fine-grain locks and under coarse-grain locks. In addition we also show the performance of the coarse-grain version with hardware lock elision applied to the global lock. We have not yet implemented full support for `retry` and `orElse`. The benchmarks on which we report do not directly require these operations; if they did, our code would simply fall back to STM. Minimal support is in place as the run-time system internally uses STM for some synchronization.

Results were obtained on a 4-core, 8-thread 4th generation Intel Core processor (Haswell) system. Each data point is the average of several runs. Figure 5 shows the performance of non-conflicting read-only transactions. The benchmark first builds a red-black tree and then spawns four worker threads. Each thread executes 5,000 transactions, each of which performs 40 lookup operations on the tree. In Figure 6 we show the performance of a write-heavy workload, which replaces all the nodes in a red-black tree, in parallel, with new nodes, repeating the process 10 times.

In the read-only benchmark our hybrid TM performs better up to a size where the transactions start to rely on the fallback due to conflict aborts. We suspect that these conflicts stem from the metadata writes inherent in executing Haskell code. Performance of the hybrid system was dramatically worse before we altered the hardware transaction code path to avoid calling out to a foreign C function on every load and store.

The write-heavy benchmark shows our hybrid performing roughly 30% slower than coarse-grain locking or the HLE fallback.

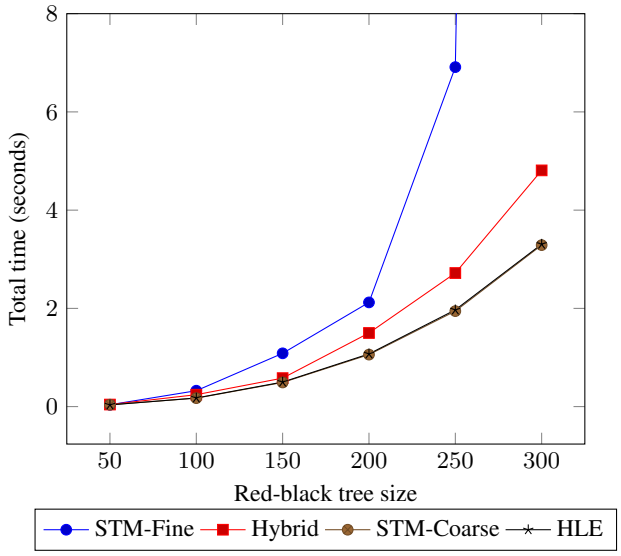

Figure 6. Writes to a red-black tree. At each size tree, actions of 4 parallel threads serve to replace every node in the tree 10 times, by atomically deleting one node and inserting another.

The default fine-grain locking performs very poorly; on larger trees it appears to live-lock, as the commit phases of software transactions cause mutual aborts.

5. Conclusion

Our preliminary implementation of a hybrid TM for GHC Haskell shows promise and performs better then the existing fine-grain STM implementation on microbenchmarks that exercise the parts of our system that we have fully implemented. At the same time, transaction throughput remains quite low compared to TM systems for other languages. We hope in future work to achieve significantly better performance—among other things, by reducing the inherent overhead of `TVars` and the indirection to heap objects, and by fully inlining the hardware transaction code path whenever possible. With these improvements in place, we can explore using different TM systems that could offer better performance.

We hope to also improve performance for code that relies heavily on `retry` and `orElse`, by using the designs described in this paper to allow such transactions to execute fully in hardware in common circumstances.

References

[1] L. Dalessandro and M. L. Scott. Sandboxing transactional memory. In *Proc. of the 21st Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 171–180, Minneapolis, MN, Sept. 2012.

[2] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *Proc. of the 15th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 67–78, Bangalore, India, Jan. 2010.

[3] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory. In *Proc. of the 16th Intl. Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 39–52, Newport Beach, CA, Mar. 2011.

[4] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proc. of the 12th Intl. Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 336–346, San Jose, CA, Oct. 2006.

[5] A. Farmer, A. Gill, E. Komp, and N. Sculthorpe. The HERMIT in the machine: A plugin for the interactive transformation of GHC core language programs. In *Proc. of the 2012 ACM SIGPLAN Symp. on Haskell*, pages 1–12, Copenhagen, Denmark, Sept. 2012.

[6] P. Felber, C. Fetzer, P. Marlier, M. Nowack, and T. Riegel. Brief announcement: Hybrid time-based transactional memory. In *Proc. of the 24th Intl. Conf. on Distributed Computing*, pages 124–126, Cambridge, MA, Sept. 2010.

[7] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25(2):article 5, May 2007.

[8] T. Harris and S. Peyton Jones. Transactional memory with data invariants. In *TRANSACT '06: 1st Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Ottowa, jun 2006.

[9] T. Harris, S. Marlow, and S. Peyton Jones. Haskell on a shared-memory multiprocessor. In *Proc. of the 2005 ACM SIGPLAN Workshop on Haskell*, pages 49–61, Tallinn, Estonia, Sept. 2005.

[10] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *Proc. of the 10th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 48–60, Chicago, IL, June 2005.

[11] Intel. *Intel Architecture Instruction Set Extensions Programming Reference*. Intel Corporation, Feb. 2012. Publication #319433-012. Available as `software.intel.com/file/41417`.

[12] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proc. of the 11th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 209–220, New York, NY, Mar. 2006.

[13] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *Second ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, Portland, OR, Aug. 2007.

[14] V. J. Marathe, M. F. Spear, and M. L. Scott. Scalable techniques for transparent privatization in software transactional memory. In *Proc. of the Intl. Conf. on Parallel Processing (ICPP)*, pages 67–74, Portland, OR, Sept. 2008.

[15] S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore Haskell. In *Proc. of the 14th ACM SIGPLAN Intl. Conf. on Functional Programming (ICFP)*, pages 65–78, Edinburgh, Scotland, Aug.–Sept. 2009.

[16] A. Matveev and N. Shavit. Reduced hardware NOrec. In *5th Workshop on the Theory of Transactional Memory (WTTM)*, Jerusalem, Israel, Sept. 2013.

[17] S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In *Proc. of the 2001 ACM SIGPLAN Workshop on Haskell*, pages 203–233, Firenze, Italy, Sept. 2001.