An Update on Haskell H/STM *

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

Abstract

At TRANSACT 2014 we described plans and a preliminary implementation for a hybrid TM for the Glasgow Haskell Compiler. Here we give an update on our work and new performance results on a 72-thread Intel Haswell system showing that hardware transactional memory can improve the performance of Haskell TM at realistic scales. We describe our constant space approach to supporting transaction blocking (`retry`). We also compare performance with a common Haskell idiom for concurrent data structures, showing how TM improves on this as concurrency increases.

1. Introduction

A year ago we introduced our preliminary implementation of a hybrid TM for Haskell and outlined some further steps needed to fully support the blocking (`retry`) and choice (`orElse`) features of Haskell's STM. We have now implemented full run-time support for these features and have explored the performance of our hybrid system on a large Intel machine with Transactional Synchronization Extensions (TSX) [10].

Major changes from a year ago include:

1. Where our previous system required per-transaction logging space linear in the number of accessed transactional variables, we now use constant space, even when blocking on `retry`. This change allows us to accommodate much larger data structures using hardware transactions.
2. To support waking up blocked transactions we introduce a global structure containing approximate read sets for blocked transactions. The semantics of blocked transactions do not demand that wakeup happens atomically with the write that unblocks a transaction, but does demand that no wakeup opportunities are missed. We use a combination of acquiring a global lock for the wakeup structure and lock-elision–based speculation to accommodate both hardware transactions that block by committing an update to the structure (but no updates to anything else) and concurrent transactions that mutate the structure to wake up blocked transactions, while avoiding aborts of the former by the latter in the common case.
3. We compare TM performance to a common alternative Haskell idiom in which a large pure-functional data structure is accessed through a single global mutable reference. Updates in this idiom install a "new" version of the structure that typically shares much of its state with the old. Our experiments show that while this idiom yields good performance up to eight cores, performance quickly degrades as the single reference becomes a significant bottleneck. Our hybrid TM systems as well as the original fine-grain STM system outperform this common technique as the number of cores increases.
4. Running on a large system has exposed an important corner case in the GHC run-time system. GHC's implementation allows STM transactions to continue running after observing inconsistent state. While this is safe to do in the absence of side effects, threads can erroneously enter a non-allocating infinite loop and miss requests to enter stop-the-world garbage collection, which in the default configuration is tied only to allocation. Our experiments show that this corner case is encountered much more often with many threads and when using our hybrid system. A straightforward fix to this problem adds overhead of roughly 5%; we are exploring cheaper alternatives.

We give a brief overview of Haskell's STM implementation in Section 1.1, including a description of the relevant portions of the run-time system in Section 1.1.1. In Section 1.1.2 we elaborate on the limitations this implementation poses—and the challenges it poses—for an H/STM hybrid. We also briefly discuss Intel's HTM support in Section 1.2 and some related work on hybrid TM in Section 1.3. In Section 2 we describe our hybrid TM variants and in Section 3 we discuss performance challenges specific to Haskell's TM. Finally our current performance 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. Since its introduction by Harris et al. in 2005 [8], GHC's STM has grown increasingly popular. Most uses are not performance critical, but rather focus on ensuring correctness in the face of concurrency from user interaction or system events. TM-based concurrent data structures are less common, and little effort has been invested in the sort of performance tuning that has characterized STM work for imperative languages [9, chap. 4].

In comparison to most of those imperative implementations, GHC's TM is unusual in its use of explicit transactional variables called `TVar`s. Haskell has a static separation of effects that prevents inspecting or manipulating these variables outside of the context of a transaction. In fact, there is no special compiler support for STM beyond the existing type system. STM is supported instead by the run-time system. Inside transactions, execution is restricted to operations on `TVar`s and the usual pure functional computations. `TVar` operations consist of creation (with an initial value), reading, and writing.

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 facilitate the construction of a *sandboxed* TM runtime [1], in which "doomed" transactions can safely execute beyond the point at which they first read inconsistent values from memory (we return to this subject in Section 2.4).

Haskell transactions also support a `retry` operation. Conceptually, the compiler and runtime ensure that each transaction exe-

* This work was funded in part by the National Science Foundation under grants CCR-0963759, CCF-1116055, CCF-1337224, and CCF-1422649, and by support from the IBM Canada Centres for Advanced Studies.

cutes only when it can do so without encountering `retry`. Pragmatically, the implementation aborts (the current execution of) the transaction when `retry` is encountered, and arranges to try again as soon as any `TVar` that was read during the aborted run is updated by some other (subsequent) transaction (suggesting the possibility that a new run might take a different code path). The `retry` operation serves to implement condition synchronization. It can be used, for example, to build a blocking queue: the dequeue operation reads the queue's head variable and, if the queue is empty, executes `retry` rather then returning no results. The transaction's thread is then blocked (atomically with respect to its discovery of emptiness) until another transaction performs an `enqueue` operation.

1.1.1 Run-time Support

GHC serializes transactions using either a coarse-grain lock or per-`TVar` fine-grain locks. 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 an updated value if one was 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 commits 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. This implementation draws heavily on the OSTM system of Fraser and Harris [6], but without its nonblocking progress guarantees.

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 (the portion of the language that interacts with the outside world and with non-transactional mutable state). Separation facilitates 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 [7] 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 `TVars` 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 overhead, 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 (represented in place, rather than via indirection) if the type of the value allows.

1.2 Intel TSX

In our work we use Intel's Transactional Synchronization Extensions (TSX) [10][1] for hardware transactions. We use only the Restricted Transactional Memory (RTM) form, with `XBEGIN` and `XEND` instructions to mark the beginning and end of transactions, `XTEST` to determine at run time if execution is transactional, and `XABORT` to abandon a transaction and return an eight bit reason code. While some of our uses of TSX resemble Hardware Lock Elision (HLE), we do not use the direct support for HLE as we gain flexibility in handling aborts directly with RTM and we have no need for the backward compatibility of HLE. Our work could in principle be ported to IBM's Power or z architectures, which provide HTM capabilities similar to TSX, but GHC support is not as solid on these platforms as it is on x86.

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.

Subsequent 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. 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. [5] and Riegel et al. [20] present variants of this scheme with similar performance.

More recent work by Matveev and Shavit [17, 18] builds on both NOrec and time-based STM, with two levels of fallback from pure

[1] Unfortunately, Errata HSD136 and HSE44 led Intel to disable TSX for all uses except software development [11]. Future versions will not be affected by the issue. Our results are from a machine that has TSX enabled according to instructions given by Intel.

HTM. The first level of fallback uses hardware transactions for part of the work—specifically to validate and write back atomically—allowing it to coexist with mostly uninstrumented hardware transactions. The second level of fallback requires that hardware transactions maintain the metadata used by STM.

2. Hybrid Haskell TM

Our hybrid TM for GHC uses TSX in several ways. In Section 2.1 we look at the simple scheme of eliding the coarse-grain lock during the STM commit phase. This 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. Section 2.3 discusses a preliminary implementation in which we use hardware transactions in the commit phase of the fine-grain locking STM. Trade-offs and interactions among these schemes are discussed in Section 2.4.

2.1 Eliding the Coarse-grain Lock

The simplest way to employ TSX in GHC's STM implementation requires no 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.

While this simple approach will work, we make a few changes that improve performance significantly. If a transaction has written any `TVar`s, it must on commit wake up any blocked transaction with a read set that overlaps the committing write set. We move these wakeups outside the global commit lock and protect a global wakeup structure holding the read sets of blocked transactions with a separate wakeup lock. Details of the wakeup structure appear in Section 2.5. For the results discussed here, our implementation represents read sets with small, 64–bit Bloom filters, and our programs seldom employ `retry`. Future work will explore the impact of such parameters as Bloom filter size, hash function choice, and wakeup structure design.

Another change we make is to elide the usual garbage collection (GC) write barriers within a hardware transaction. When a `TVar` is mutated the GC needs to know to follow the value pointer if that pointer points into a younger generation. Each mutated `TVar` that is not in the youngest generation is put on a mutated list for GC to follow. We can safely move this list maintenance out of the critical section, by ensuring that GC cannot happen between when the hardware transaction commits and when the `TVar`s are added to the mutated list. We discuss this optimization further in Section 3.

Finally, we can avoid a double traversal of the `TRec` when executing in a hardware transaction. The `TRec` stores its read and write sets together in one table. Software transactions traverse the table once to validate that the `TVar`s hold the values seen when the variables were first encountered, and again to perform the updates from the write set. In a hardware transaction we can perform both tasks in a single pass, and use `XABORT` with a reason code indicating a validation failure when any `TVar` holds a different value.

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. [5], 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, but we can make the window of vulnerability quite small by reading the lock only at the end of the hardware transaction. (As discussed in Section 1.1, and examined more closely in Section 2.4, this optimization is safe in Haskell but not in most other languages.)

In the other direction, transactions used for the commit phase of fallback software transactions will be aborted whenever they conflict with an all-hardware transaction, in a manner analogous to the *reduced hardware transactions* of Matveev and Shavit [17]. If we elide the global lock as described in Section 2.1, aborts will not be caused by conflicts on the coarse-grain lock itself: in the absence of (true data) conflicts, software transactions can commit concurrently with a running hardware transaction,

2.3 Fine-grain Locking with Hardware Transaction Commit

A third hybrid strategy starts with the fine-grain locking STM and attempts to perform its commit phase using a hardware transaction. Because the fine-grain STM uses the `TVar`'s value field as a lock, this third strategy ends up being very similar to elision of the global lock in the coarse-grain STM, as described in Section 2.1. We do not need to include a global lock variable in our read set, however, as each `TVar` value read in the hardware transaction *is* a fine-grain lock.

In the commit phase of the coarse-grain hybrid, if we observe that the global lock is held, we use `XABORT` to roll back the transaction, as we cannot commit in the middle of another thread's software commit phase. We use the abort code to indicate that the lock was held; we then spin until the lock is free before attempting to execute the hardware transaction again. In the fine-grain version, a locked `TVar` indicates a validation failure; no further attempts will be made at performing the hardware transaction.

2.4 Interaction Between Transactions

In our previous work we noted that Haskell's STM allows continued execution of transactions that have observed inconsistent state. We argued that these "doomed" transactions do not lead to unsafe executions, so long as the transactions themselves do not contain explicitly labeled *unsafe* operations, which circumvent Haskell's type system. Our experiments have revealed, however, that on larger machines and with the faster execution of hardware transactions, a doomed transaction can easily enter a infinite loop that contains no memory allocation operations. Since allocations provide the hook for initiation of stop-the-world garbage collection, the entire program can hang as a result. The problem is easily fixed by using the `no-omit-yield` compiler flag to ensure that all loops (in Haskell's case, recursive function calls) contain a GC initiation check. This flag is reported to incur a cost in binary size of around 5% while overall performance remains uneffected[21]. We note that infinite loops are less of an issue in hardware transactions, which will always abort eventually, due to scheduler interrupts. In fact, we *prefer* to avoid GC checks inside of hardware transactions.

While it is easy to turn on the `no-omit-yield` flag in our research work, it may be a more significant burden for developers and users as it is unclear which modules will need the extra instru-

mentation. We would also like to avoid wasting time in doomed transactions. More lightweight detection (and earlier termination) of doomed transactions is a subject for future work.

2.5 Supporting Blocking

STM Haskell's `retry` operation poses a challenge for all-hardware transactions. In the original STM, when a transaction executes `retry`, its Haskell thread is suspended and placed on a *watch list* for every `TVar` in its read set. When another transaction commits it must 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 arranged for hardware transactions to record the address of each write in a transactional record. This proved to be too costly to be effective in practice, largely because the write set implementation performs dynamic memory allocation. It is safe, if inefficient, 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 any 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 employ this latter option in our current implementation, with constant-space Bloom filters for approximate read and write sets.

A transaction that encounters a `retry` while running entirely in hardware can save its read set and commit the hardware transaction so long as it has not yet updated any `TVar`s. Similarly, if `retry` is encountered while in the first branch of an `orElse`, before any writes, execution can continue directly to the second branch. If a transaction *has* written a `TVar` it must abort. We could potentially employ an abort code that signals the runtime to unblock the thread whenever a writer transaction commits. In principle we could even use the abort code itself as a tiny read-set Bloom filter, but TSX makes only 8 bits available. For now, we simply restart the transaction in STM mode. The ideal solution, we believe, is to develop compiler transformations that delay writes, whenever possible, until after any point at which the transaction might encounter a `retry`; we plan to explore this option in future work.

By varying the size of Bloom filters, we also plan to explore the tradeoff between the accuracy of wakeup and the time and space overhead of instrumentation. 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.

Global Wakeup Structure Committing transactions need to be able to find the read sets of blocked transactions. For this purpose we use a simple global structure we call the *wakeup list*, protected by a global wakeup lock. A committing writer transaction acquires the global wakeup lock, searches for overlapping read sets, and then wakes the threads associated with those sets.

We improve on this simple scheme by buffering the wakeups and performing them after the commit lock is released, and by eliding the global wakeup lock when searching for threads to wake. Specifically, when an all-hardware transaction executes `retry` it acquires the global wakeup lock before committing the hardware transaction (Fig. 1). This avoids any window between the end of the transaction and the insertion into the list, during which a committing transaction might overlook the `retry`-er. We also prevent data conflicts on the wakeup structure itself from needlessly aborting our all-hardware transaction. Speculative elision of the lock is performed only by threads performing wakeup (Fig. 2), so in the event of a conflict the more expensive and important `retry`-ing transaction wins. Because they acquire the lock for real, no two all-hardware transactions can commit a `retry` at the same time. The overlap here is quite small as long as we optimize the wakeup list for quick insertion.

Our wakeup list is implemented as a linked list of chunks. Under protection of the wakeup lock, insertion can be performed by incrementing the chunk's next-free-slot index and storing the read set Bloom filter and thread pointer. If a chunk is full, a new chunk is inserted to the head of the list. When threads are woken, the Bloom filter is overwritten as zero (an empty read set). When searching the list for matches, any chunk found to contain all zero entries is unlinked. Garbage collection time also provides an opportunity to compact the list.

```
top:
  retrying = false
  XBEGIN  // failure path not shown
          // (same as in any all-hardware transaction)
    ...
    // retry with no prior TVar writes:
    if (wakeup_lock == 1)
      XABORT(RESTART)
    wakeup_lock = 1
    retrying = true
  XEND
  if (retrying)
    insert(wakeup_list, read_set)
    release(wakeup_lock)
    wait_for_wakeup
    goto top
```

Figure 1. Handling of the wakeup lock in a hardware transaction committing `retry`.

```
XBEGIN
  ... normal transaction execution with writes ...
XEND
for 0..retry_count do
  XBEGIN(fail)
  if (wakeup_lock == 1)
    XABORT(RESTART)
  wakeup(wakeup_list, write_set)
  XEND
  return                  // exit transaction
 fail:
end
// fallback:
acquire(wakeup_lock)
wakeup(wakeup_list, write_set)
release(wakeup_lock)
```

Figure 2. Speculative search of wakeup list by a transaction that writes.

3. Performance Challenges

Since we introduced our Hybrid TM we have discovered a new challenge to performance in the GC write barrier. While we will only focus on this GC challenge in this section, as discussed in

our previous work, we still suffer from unnecessary indirection in `TVars`; we hope to address this soon with improved transactional array support.

GHC's garbage collector separates the heap into generations. New allocations occur in the youngest generation and as values live longer they may be promoted to an older generation. As in all generational collectors, the goal is to allow more frequent, but short collections of the younger generation. (Concurrent collection could even be allowed [16], though the current implementation does not employ it.) As noted in Section 2.1, one challenge of generational collection is the need to track any reference in an older generation that is mutated to point to a value in a younger generation. If a collection of the young generation is not aware of this reference it could collect the young value and leave the reference in the older generation dangling.

In GHC 7.8 a new mechanism is used to track the mutations of `TVars` in older generations. Every mutation invokes the GC write barrier, which adds the address of the mutated heap object to a mutation list specific to the execution context and object generation. When the collection of a young generation happens, objects on the mutation lists for each generation are followed as roots. Only when the transitive closure of the object ends up being promoted to the older generation will the object be removed from the mutation list. This GC strategy affects the design of our hybrid TM, as we wish to avoid tracking a precise write set, preferring instead to use constant space. But the list of mutated values *is* the write set. If we must keep this set around anyway (in anticipation of performing the GC barriers after completing the HW transaction), we might need unbounded space. Fortunately, it is not expensive to identify the generation of a given heap object, and to remember it only in the (relatively rare) case that the mutation is in an older generation.

We have not yet determined what role the GC write barrier is playing in performance, but there are some approaches we could explore to handle it differently. For instance, we could additionally check on each `TVar` write if the value being written is in a younger generation—not just if the `TVar` is in the youngest generation. Another approach would be to have a pre-determined number of writes that we would support in a fully HTM transaction. These writes would be recorded in the constant space it takes to hold them. If more writes are demanded we would either fall back to STM or execute the GC write barrier in the hardware transaction. If we end up with precise information on all the writes this could also have the beneficial side effect of avoiding unnecessary wakeups.

4. Performance 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. We also show the performance of the coarse-grain version with hardware lock elision applied to the global lock, and the fine-grain version with a commit that uses a hardware transaction.

Results were obtained on a 2-socket, 36-core, 72-thread Intel Xeon E5-2699 v3 system. This system is much larger than in our previous work, which was limited to four cores and eight threads. We can clearly observe the costs of communicating across socket, and even across rings within a chip.[2] Hardware transactions succeed in reducing the amount of communication needed between cores, allowing for a significant increase in performance simply by eliding a coarse grain lock. To achieve consistent results we augmented GHC's thread pinning mechanism to allow assignment of Haskell execution contexts to specific cores, and experimented

[2] The internal network of the E5-2699 consists of two communication rings, with 10 cores on one and 8 on the other.

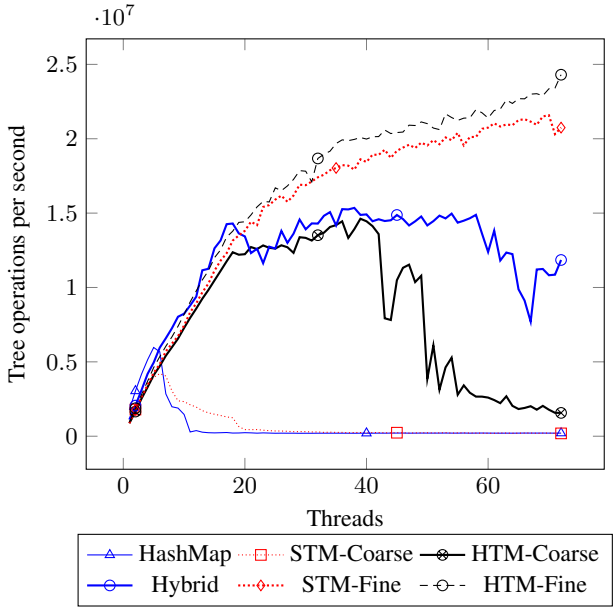

Figure 3. Operations on a red-black tree of roughly 50,000 nodes, where 90% of the operations are lookups and the rest are split between insertions and deletions.

with different assignments as we increased the number of threads used in our experiments. The best performance was achieved by first filling all the cores on one chip then moving to the second chip and finally using the SMT threads. While we didn't exhaustively explore all the options, and the results are dependent to a large degree on the particular workload, we did observe that coarse-grain techniques sometimes benefited from assigning SMT threads before cores on the second socket. In these cases better locality for the global lock was of more benefit than the larger cache space obtained by distributing across chips.

While we have attempted to benchmark some existing STM-based Haskell applications, most current Haskell programs use STM for correctness and expressiveness (via `retry` and `orElse`), not for high performance. So far, our TM enhancements do not translate into performance improvements for such programs. As has commonly been the case in imperative languages, high performance Haskell applications tend to use locks (`MVars`) or CAS (`atomicModifyIORef'`). We will be looking to convert some of these applications into useful benchmarks. In addition to benchmarks from the Haskell world we are also looking to import standard TM benchmarks from imperative languages. We hope soon to have results that include the `vacation` and `kmeans` benchmarks from STAMP [19]. Unfortunately, porting is made difficult by the significant difference in style between C and Haskell programming. We expect this effort to be an opportunity to further explore the performance implications of these differences.

4.1 Data Structure Throughput

Our benchmarking work has focused on red-black tree performance. Figure 3 shows the throughput of a mix of operations on a red-black tree which initially has 50,000 nodes—significantly larger than the very small trees in our previous work. When the benchmark runs, each thread performs a transaction which will search for a random key 90% of the time, insert a random key 5% of the time, and delete a random key the remaining 5% of the time.

In addition to the coarse- and fine-grain variants of the original Haskell STM, and our hybrid and hardware commit versions, we

also show results for a common Haskell idiom in which a set is kept in an immutable tree, and the `atomicModifyIORef'` operation is used to swing a root pointer to a new tree that shares much of its structure with the old. This idiom is often offered as an alternative to concurrency for data structures with internal mutable variables [15]. When using it, we obtained the best performance not with a red-black tree, but with the key–value pair interface of the *hash array mapped trie* found in the `unordered-containers` package. This code is labeled "HashMap" in the figures.

HashMap performs well for small thread counts, but soon succumbs to the bottleneck of a single mutable reference. Similarly, the coarse-grain STM keeps pace with the other TM implementations until around 6 threads. This is most of the cores on a single ring. Hardware commit nicely extends the coarse-grained STM, continuing to improve throughput across cores on a single chip. The fine-grain TM with and without the hardware commit performs the best out of all the variations on two sockets with the hardware commit version (HTM-Fine) consistently performing slightly better. Our hybrid slightly edges out the other TM variants on a single chip without hyperthreads. All options have diminishing returns as both sockets are filled and hyperthreads are a net loss for both our hybrid and HTM-Coarse.

4.2 Retry Overhead

Given the heavy use of `retry` in existing Haskell code, we have also measured the overhead of thread blocking and wakeup mechanisms. Our focus has been on applications in which `retry` is relatively rare, and in which transactions remain blocked for relatively short amounts of time. In future work we hope to support other use cases efficiently as well. When a blocked thread is woken and, after running again, still executes `retry`, we call this a *failed wakeup*. The frequency of failed wakeups is application dependent, but we expect it to increase as a result of our constant-space read sets.

One very common use of TM in Haskell is to multiplex or manage queues. As a simple benchmark we built a ring of threads each with a queue. Each thread begins with some initial number of items in its queue. During execution, each thread repeatedly takes an item from its first neighbor and gives and item to its second neighbor in a transaction. When a queue is empty any attempt to read will execute `retry`. The average throughput of five runs is given In Figure 4. When there are only two threads retries are rare (around 0.05% of transactions) and the blocked thread should wake up on *any* progress by the other thread. As we add more threads there is more room for a delay in the flow of values around the ring, leading to more blocked threads and, in the case of approximate read and write sets, failed wakeups. Performance drops for all variants as blocking becomes more common, but we can see that our hybrid and the fine-grain version with HTM commit perform well as we fill one socket without hyperthreads. Both of these versions use Bloom filters for read and write sets. At the hybrid's peak performance at 18 threads, around 9% of the transactions are failed wakeups. The fine-grain STM gains some performance back as hyperthreads are added. For this particular benchmark, accurate tracking of the read and write sets in the two STM variants means that every wakeup will be successful.

While there is still much to be explored in `retry` performance, this benchmark shows that despite failed wakeups, our hybrid systems continue to provide good performance in at least one common use case.

5. Conclusion

We remain optimistic that Haskell TM can provide both rich semantics and good performance, and that TM hardware can help. Our work to date supports this expectation: throughput on data structure microbenchmarks is now within a modest constant factor of similar experiments in C, and our Hybrid, HTM-Coarse, and HTM-Fine implementations all exhibit scenarios in which they outperform pure STM implementations.

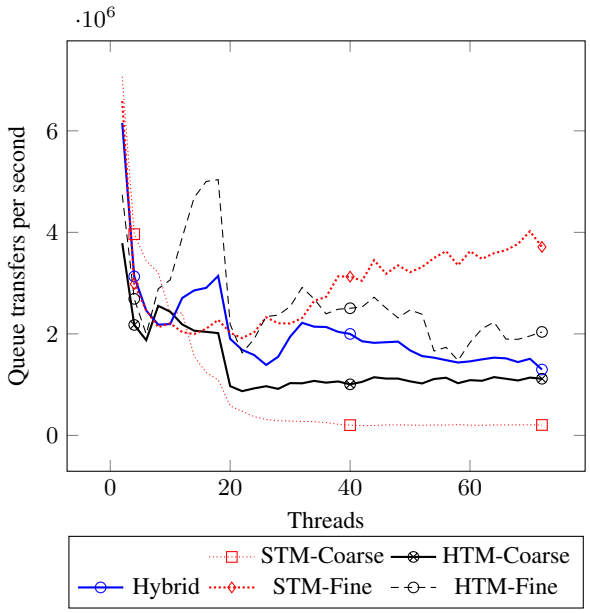

Figure 4. Throughput of transfers around a ring of threads with 1000 initial values each.

The next step in our work will be to implement improved `TArray` support that allows hardware transactions to take advantage of reduced indirection to fit more reads and writes into the L1 cache. As it is now, users simply cannot express a TM data structure in Haskell that has the same overhead as a comparable TM system in C or C++. Our hybrid's best performance on the red-black tree benchmark has a throughput that is around 68% of the performance of NOrec in C on the same machine. We will not be able to start closing that gap until we reduce the overhead of the structures involved.

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] 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.

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

[7] 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.

[8] 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.

[9] T. Harris, J. Larus, and R. Rajwar. Transactional memory. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, 2010.

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

[11] Intel. *Intel Xeon Processor E5 v3 Product Family Processor Specification Update*. Intel Coroporation, Jan. 2015. Publication #330785-005.

[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 *2nd 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. *Parallel and Concurrent Programming in Haskell*. O'Reilly Media, Inc., 2013.

[16] S. Marlow and S. Peyton Jones. Multicore garbage collection with local heaps. In *ACM SIGPLAN Notices*, pages 21–32. ACM, 2011.

[17] A. Matveev and N. Shavit. Reduced hardware transactions: A new approach to hybrid transactional memory. In *Proc. of the 25th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, Montréal, PQ, Canada, July 2013.

[18] A. Matveev and N. Shavit. Reduced hardware NOrec: A safe and scalable hybrid transactional memory. In *Proc. of the 20th Intl. Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Istanbul, Turkey, Mar. 2015.

[19] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35–46. IEEE, 2008.

[20] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing hybrid transactional memory: The importance of nonspeculative operations. In *Proc. of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 53–64, San Jose, CA, June 2011.

[21] T. G. Team. *The Glorious Glasgow Haskell Compilation System User's Guide*. The GHC Team, 2012. URL `downloads.haskell.org/~ghc/7.8.4/docs/html/users_guide/`.