Leveraging Hardware TM in Haskell

Ryan Yates
Houghton College
Houghton, NY, USA
ryan.yates@houghton.edu

Michael L. Scott
University of Rochester
Rochester, NY, USA
scott@cs.rochester.edu

Abstract

Transactional memory (TM) is heavily used for synchronization in the Haskell programming language, but its performance has historically been poor. We set out to improve this performance using hardware TM (HTM) on Intel processors. This task is complicated by Haskell's `retry` mechanism, which requires information to escape aborted transactions, and by the heavy use of indirection in the Haskell runtime, which means that even small transactions are likely to overflow hardware buffers. It is eased by functional semantics, which preclude irreversible operations; by the static separation of transactional state, which precludes privatization; and by the error containment of strong typing, which enables so-called *lazy subscription* to the lock that protects the "fallback" code path.

We describe a three-level hybrid TM system for the Glasgow Haskell Compiler (GHC). Our system first attempts to perform an entire transaction in hardware. Failing that, it falls back to software tracking of read and write sets combined with a commit-time hardware transaction. If necessary, it employs a global lock to serialize commits (but still not the bodies of transactions). To get good performance from hardware TM while preserving Haskell semantics, we use Bloom filters for read and write set tracking. We also implemented and extended the newly proposed *mutable constructor fields* language feature to significantly reduce indirection. Experimental results with complex data structures show significant improvements in throughput and scalability.

CCS Concepts • **Computing methodologies → Concurrent programming languages**; • **Software and its engineering → Concurrent programming languages**; **Concurrent programming structures**.

Keywords Haskell; transactional memory; synchronization; scalability

PPoPP '19, February 16–20, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-6225-2/19/02…$15.00
https://doi.org/10.1145/3293883.3295711

1 Introduction

Since its introduction in 2005 [6], Haskell's Software Transactional Memory (STM) 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. The implementation of the Glasgow Haskell Compiler (GHC) run-time system is not without compromises, however. Transactions incur significantly more overhead than in 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 abort rates tend to be high for all but the smallest transactions. As a result, while STM is the de facto synchronization mechanism of choice for real-world Haskell, it tends to be used in an idiosyncratic way—with small transactions and with heavy use of the `retry` mechanism, which allows a transaction to wait for a programmer-specified precondition.

We believe that a more performant implementation of STM in Haskell would allow it to be used effectively in a much more natural way—the way it is typically envisioned in the TM literature—namely, as a means of encapsulating complete, atomic updates to complex data structures. In this more natural idiom, uses of `retry` would, we believe, be relatively rare, and extraneous wake-ups (due to cheaper, more approximate read set tracking) could be tolerated.

With this vision in mind, we set out to leverage the hardware transactional memory (HTM) of recent IBM and Intel processors. (We focused on the Intel version [7] because it has considerably more robust support in GHC.) To our surprise, the task turned out to be quite difficult. We expected to face challenges with `retry`, which naturally requires information (details of the awaited condition) to escape an aborted transaction. We soon realized that the representation of read and write sets (required by `retry` even for successful hardware transactions) introduced significant management overhead and tended to bloat the cache footprint, causing transactions to overflow hardware limits on the size and associativity of buffered speculative state. The biggest challenge turned out to be GHC's dependence on multiple levels of indirection in the implementation: given lazy evaluation (evaluated v. unevaluated values), speculation (transactional values), type-safe variants, and boxing of primitive types, a single integer, accessed in a transaction, could easily require four or more cache lines of speculative state. As a result, even small transactions risk exceeding hardware speculation limits.

This paper describes and evaluates the TM system we ultimately developed: a three-level hybrid design that first tries to use HTM to perform an entire transaction, then falls back to validating and committing in hardware (after first devising a plan in software), and finally, when necessary, employs a global lock for the commit phase. We leverage Haskell semantics wherever possible. In particular,

- Due to pure functional semantics, transactions never perform "irreversible" operations and never need to be "inevitable" [17].
- Due to static separation of non-transactional and transactional values, transactions never need to worry about "privatization safety" [11].
- Due to strong typing, errors in "zombie" transactions—those that have read inconsistent state, but do not yet realize they are doomed to abort—can always (with care) be contained, allowing us to safely employ *lazy subscription* [4, 10] to the fallback lock, reducing abort rates.

To minimize the overhead of write-set tracking (needed in order to tell whether a successful transaction has modified state on which other transactions are waiting), our system employs compact Bloom filters rather than dynamically allocated precise effect maps. To contain any effects of zombie transactions, it interposes validation checks in the garbage collector and scheduler. To reduce indirection overheads, it abandons the natural convention of referencing every transactional value through an indirection object containing metadata; instead, it allows transactional values to be embedded directly in structure fields and array elements. Prior work demonstrated that the resulting reduction in speculative cache footprint could significantly reduce abort rates [20], but only in non-type-safe code. In this current work, we use *mutable constructor fields* [12], a recently proposed but heretofore unimplemented language feature, to restore type safety without indirection. Our implementation of mutable fields was a major engineering effort.

We evaluate our hybrid system on concurrent versions of some of Haskell's most sophisticated data structures: red–black trees, hashed array mapped tries, and treaps. Our results demonstrate significant performance improvements with respect to the best existing implementations, supporting tens of millions of data structure updates per second, and scaling (in many cases) out to the limits of our 72-thread, two-socket machine. We take our success with type-safe indirection elision as a compelling endorsement of mutable fields. We take our lazy subscription results as a similar endorsement of type safety in transactional languages.

2 Background

2.1 Haskell TM

Haskell's TM was introduced by Harris et al. [6]. Its implementation in GHC can be configured to use either of two different locking strategies. The *fine-grain* strategy is based on the OSTM of Fraser [5]; it uses a separate lock for each transactional variable (TVar). The *coarse-grain* strategy uses a single lock to protect the commit phases of transactions. In both strategies, TVar accesses are recorded, provisionally, in a local transactional record (TRec). Reads are then validated, and writes performed, at commit time. We will mostly concern ourselves with the coarse-grain implementation here, because it is more suitable for pairing with hardware TM. Details of the fine-grain implementation as well as a description of *transactional structs* discussed later in this section can be found in prior work [20].

When a transactional variable is accessed for the first time in a transaction, the variable address and its value are stored in a new TRec entry. The entry also has a field for a new value if the transaction writes to that variable. When the body of the transaction completes (in the coarse-grain locking implementation) its thread attempts to commit: it acquires the global commit lock and visits each entry in the TRec, checking that the expected value stored matches the value currently in the TVar. If a value does not match, the TRec is discarded, the lock is released, and the transaction starts again from the beginning. If all the values match, the transaction is valid and the entries are visited again to perform updates to the TVars, after which the global lock is released. The commit lock serializes updates, ensuring atomicity. It does *not* serialize the bodies of transactions; these may see a partial update, as they do not look at the lock except at commit time. We discuss the consequences of such inconsistency further in Section 3.2.1.

At the Haskell language level, transactional variables and the operations on them are explicit. The implementation keeps each TVar in a separate heap object that contains a pointer to the value it holds. Work on *transactional structs* [20] introduced additional heap objects called TStructs with multiple fields and unboxed (in-place) values. This implementation avoided much of the indirection required to access fields of transactional structures, but it was only a run-time level proof of concept: TStructs could be used only in source code that explicitly broke the type system. In Section 4, we describe extensions to the source language and type system that allow us to employ a TStruct-like implementation directly in (type-safe) Haskell code.

The following data declaration is an example of a transactional binary tree. It uses Haskell's *generalized algebraic data types* (GADT) syntax, which directly captures the type signature of the tree's constructor function:

```
1  data Tree where
2    Node :: Key
3         → TVar Node    — left child
4         → TVar Node    — right child
5         → Tree
6    Nil  :: Tree
```

Lines 2–5 indicate that the constructor for the non-empty (Node) tree variant takes a key and two transactional trees (the left and right children) as arguments, and returns a value of type Tree. Line 6 indicates that the constructor for the empty (Nil) tree variant takes no arguments and simply returns a Tree. Each constructor is a pure function without visible side effects. The transactional arguments, however, must be created in a special execution context that accommodates mutability.

Readers familiar with Haskell will recall that while the language is purely functional, it provides a *monad* mechanism that allows nested and tail-recursive functions to be written in an imperative (assignment- and loop-based) style. The type system allows purely functional code to create *actions* that can then be *executed* in an appropriately labeled monadic context. Interaction with the real world is captured as action in a special "IO" context understood by the runtime system. Atomic update of TVars is similarly captured as an action in a special STM context.

GHC provides two functions—newTVar and newTVarIO—to generate actions that will (when executed in an STM or IO context, respectively) in turn create a TVar:

```
1  newTVar :: a → STM (TVar a)
2  newTVarIO :: a → IO (TVar a)
```

Note the notation for the function return type: STM (TVar a) indicates an action that will generate a TVar of type a when executed in an STM context.

Actions can be bound together into larger actions using a familiar looking imperative syntax—namely, Haskell's **do** notation. Each line of a **do** block is an action to execute; the result of the composed action (itself an action) is the result of the last line in the block. For example, the code to insert a value on the left in a binary tree looks like this:

```
1  insert :: Key → Tree → STM ()
2  insert newKey (Node key leftVar rightVar)
3    | newKey < key = do
4        leftTree ← readTVar leftVar
5        case leftTree of
6          Nil → do
7            l ← newTVar Nil
8            r ← newTVar Nil
9            writeTVar leftVar (Node newKey l r)
10         tree → insert newKey tree
11 ... — Other cases follow.
```

Note the uses of newTVar at lines 7 and 8.

Line 1 gives the type of insert as a function that takes a Key and a Tree and produces an STM action with no meaningful result (the unit type, denoted by empty parentheses, and used in an analogous way to C's `void`). Lines 2 and 3 together form the left-hand side of the equals defining one *case* (overload) of our function. Line 3 is a condition that must match for this case. Line 2 gives names to parameters and, in this case, matches only when the second argument (in parentheses) is a Tree constructed with the Node constructor. The fields of that constructor are bound to the variables key, leftVar, and rightVar. Note that leftVar and rightVar are bound to TVars, not to the values the TVars contain. To access a value we use readTVar (Line 4) and writeTVar (Line 9). The left arrow in **do** notation is effectively assignment: it binds a name (in an IO or STM context) to the result value of the action on the right-hand side. On Line 4, readTVar leftVar has the type STM Tree, so leftTree will have the type Tree.

To execute a transaction, an STM action is given to the atomically function, which results in an IO action. Within a transaction, Haskell supports conditional blocking with the retry STM action. A transaction T executes retry to indicate that it has encountered a condition in which it cannot proceed. The runtime attempts to validate T (i.e., to confirm that it has seen consistent state). If validation fails, execution of T restarts immediately. If validation succeeds, execution first blocks until some other thread commits a transaction that has updated (or appears to have updated) one of the TVars in T's read set; then (and only then) does T restart.

Building on retry, Haskell also provides an orElse mechanism that can be used to compose alternative transactions. In the construct atomically (t1 `orElse` t2), if t1 executes retry, all of its effects are discarded (though its read set is retained) and t2 is attempted. While no currently available HTM is able to retain its read set while clearing its writes, we observe that hardware transactions can still be used if the write set is empty when retry is encountered in the first half of an orElse at run time. To exploit this observation, programs may be rewritten to delay their writes, either manually or (in potential future work) as a compiler optimization. Our current implementation supports fully general use of orElse by falling back to software transactions when needed.

2.2 Hardware Transactional Memory

Intel's Transactional Synchronization Extensions (TSX) provides a simple way to speculatively execute code while the hardware looks for conflicting accesses [7]. Hardware transactions are started with `XBEGIN`, which takes the address of an abort handler as an argument, and ended with `XEND`, which attempts to commit the transaction. We also use `XTEST`, which indicates if code is currently executing in a hardware transaction and `XABORT`, which takes an 8-bit value and explicitly aborts the transaction, passing the value as an argument to the abort handler. (Note that in contrast to other hybrid TM systems [3, 15, 16], we do *not* generate entirely different code paths for hardware and software transactions.)

A TSX transaction T will abort if some other thread reads or writes a location that T has already written, or writes a location that T has already read or written. Like most HTM implementations, TSX is "best effort" only: transactions may also abort for a variety of "spurious" reasons. Certain

instructions (e.g., syscalls) will always cause an abort, as will overflow of the associativity or capacity of the L1 cache, where speculative state is maintained.[1] Nondeterministically, certain system events (e.g., device interrupts or timer expiration) may also lead to aborts. Hardware provides the abort handler with an indication of the cause of the abort, but this is only a hint. Conventional practice is to retry a failed hardware transaction a limited number of times, then fall back to a software implementation of atomicity, unless the abort code indicates that failure is likely to be deterministic, in which case immediate software fallback is advisable.

3 Hybrid Haskell TM

In Section 3.1 we describe several implementation strategies for a hybrid TM system. Section 3.2 then considers the ways in which these strategies interact and the circumstances under which hardware transactions can be expected to improve performance. Among other things, we explain why hardware transactions in Haskell can, under reasonable assumptions, safely "subscribe" to the software fallback lock at the end of the transaction rather than the beginning. Finally, Section 3.3 describes our implementation of conditional blocking (`retry`) and its integration with hardware TM.

3.1 Hybrid Options

Our hybrid TM for GHC uses TSX in several ways. In Section 3.1.1 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 3.1.2, as the fallback mechanism for a second scheme, in which we attempt to run Haskell transactions entirely in hardware. Section 3.1.3 discusses a third 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 3.2.

3.1.1 Eliding the Coarse-grain Commit Lock

The coarse-grain lock in GHC's STM serializes transactional commits. Replacing this lock with a hardware transaction (eliding the lock) has the potential to greatly improve scalability by allowing commits to overlap and relying on the hardware for conflict detection. As the commit traverses the `TRec`, it may see a `TVar` holding a value that does not match the expected value. In this case we have it execute an `XABORT` instruction with a value that indicates that validation has failed. This conveniently will discard any updates we have performed so far in the transaction. If fallback is required, we acquire the global lock and perform two traversals of the `TRec`, first to validate the read set and then to perform the updates (the write phase).

[1] Strictly speaking, only writes are constrained by the capacity of the L1; reads that overflow that capacity are tracked, conservatively, by a supplemental summary structure [8, Section 12.2.4.2].

Compared to the work of a full transaction, the commit is compact and focused on the transactional accesses. To improve this focus even more we move wakeup (support for `retry`) and the garbage collection (GC) write barrier to another traversal of the `TRec` after the hardware transaction commits. Details on these improvements are in Section 3.3 and 4.3 respectively. Note that eliding the commit lock does not eliminate the need to maintain the transactional record (`TRec`): to allow transactions to read their own writes and to avoid interfering with peers, instrumentation is still required on transactional loads and stores.

3.1.2 Full Haskell Transactions in HTM

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. To achieve this goal, we use the coarse-grain locking implementation and start a hardware transaction in `atomically`. In read and write operations we then use `XTEST` to see if we are running a hardware transaction and, if so, access fields directly (skipping most of the instrumentation). To support `retry`, we still need to maintain a concise representation of `TRec` read and write sets (as described in Sec. 3.3 below) for wakeup or blocking after the transaction commits.

As noted in Section 2.1, hardware transactions do not support partial aborts. We therefore fall back to software transactions when `orElse` requires discarding writes. If the transaction has not yet performed any writes when it encounters a `retry` in the first half of an `orElse`, it will continue on to the second half in a hardware transaction.

If an all-hardware transaction fails to commit, we fall back to a software transaction with the elision method described in Section 3.1.1. Since an all-hardware transaction could start and commit in the middle of the fallback transaction's write phase, we need to worry about the possibility that the software transaction will see inconsistent state. We address this in the standard way, by including the global lock in each hardware transaction's read set and checking that it is unlocked. 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 [15]. If we elide the global lock, 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.

To understand and tune performance, we used Linux's `perf` tool (which provides access to hardware performance counters) to track transaction starts, commits, total aborts, conflict aborts, and contention aborts. First we focused on single-thread performance, trying to ensure that most transactions would fit in the capacity available to the hardware. Poor performance here led to exploring constant space read and write set tracking (see Section 3.3), to improving the

hardware transaction code path to avoid extra register stores for "foreign" calls (to parts of the runtime written in C), and, eventually, to the mutable constructor field work we describe in Section 4, which enhances both the type system and the runtime in order to reduce indirection.

In addition to `perf` we used Intel's Software Development Emulator (SDE) to diagnose conflicts in multi-threaded executions. SDE outputs a log of the memory locations and accesses most often responsible for transaction aborts. Early in our work, SDE showed a significant (but relatively low) number of aborts in the scheduler code. By detecting and delaying yields when running hardware transactions, we avoid these aborts and the wasted execution they entail. In another case we saw many conflict aborts on the state of the pseudo-random number generator and the counters for our testing framework. We eliminated these aborts by modifying the library to pad the state variables to a full cache line. This change improved peak hybrid performance by 15% without inducing any measurable overhead in the fine-grain STM version.

3.1.3 Fine-grain Locking with HTM 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 each `TVar`'s value field as a lock, this alternative strategy ends up being very similar to elision of the global lock in the coarse-grain STM, as described in Section 3.1.1. 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. A locked `TVar` indicates a validation failure; we abort the hardware transaction with `XABORT` and restart the entire hybrid transaction at the beginning.

3.2 Interaction Among Transactions

To understand how the coarse-grain hybrid design of Section 3.1.2 can improve performance, it helps to consider the ways in which non-conflicting transactions can interact with each other. Specifically, consider the four modes in which a transaction may execute: all hardware, software fallback (of the body of the transaction), hardware commit (of a software fallback), and software commit (of a software fallback).

First, consider concurrent all-hardware transactions. If there are no conflicts at cache line granularity, we do not expect transactions to cause each other to abort. Similarly, all-hardware transactions and software fallback, in the absence of cache line conflicts, should cause each other no problems.

Next consider an all-hardware transaction alongside a software commit. As soon as the software commit is reached it acquires the fallback lock. If the full hardware transaction has this lock in its read set, it will abort even if there is no conflict with transactional variables. In the other direction, we can consider an all-hardware transaction that starts while the fallback lock is held. Before the transaction ends it will observe the lock variable and, if it isn't free, abort the transaction (more on this in Section 3.2.1).

In the case of an all-hardware transaction alongside a hardware commit, the fallback lock is read, but not written. The hardware commit can be thought of as performing all the effects of a full hardware transaction without the interleaving computation. The resulting interactions are thus the same as between two full hardware transactions. Similarly, a hardware commit and a software commit interact in the same way as an all-hardware transaction and a software commit.

3.2.1 Lock Subscription

In the original Haskell STM work, Harris et al. [6] note that their implementation allows continued execution of transactions ("zombies") that have observed inconsistent state, arguing that the allowed effects of Haskell execution limit the ill effects of such executions. In particular, stores through uninitialized data pointers and jumps through uninitialized code pointers are guaranteed by the type system never to occur—everything is always initialized before it can be accessed. The two effects that must be considered are allocation (e.g., an attempt to create an enormous object) and non-termination (i.e., an infinite loop). In the case of allocation, a problematic request will trigger garbage collection (GC), which gives the runtime the opportunity to validate the current transaction (if any) before continuing execution. Similarly, non-termination can be handled by returning to the scheduler periodically and performing validation. The mechanism to trigger this return is the same as for GC: another thread or interrupt handler can set the transaction thread's heap limit variable to zero, causing execution to return to the scheduler at the next safe point.

Over time, changes to GHC have undermined this scheme, prompting us to develop further mitigation mechanisms. For instance, GHC is capable of producing non-allocating loops that do not check if they have reached the heap limit. This problem is easily fixed by using the `no-omit-yield` compiler flag to ensure that all loops (in GHC'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 unaffected[18].

Allocations have also become problematic (https://ghc.haskell.org/trac/ghc/ticket/12607): some large allocations may request memory directly from the OS, without triggering GC, and potentially trigger an out-of-memory error. Though we have yet to implement the fix, it would suffice to validate before large allocations and to ensure they happen in a place where the thread can return to the scheduler.

3.2.2 Unsafe Operations

So far we have assumed that only type-safe operations are executed in transactions. In practice many Haskell library operations deliberately escape the type system in order to implement optimizations that can be proven safe by hand but

are not guaranteed so by the compiler. By convention these operations are labeled with the unsafe prefix. For example the **array** library provides an unsafeRead operation, which refrains from checking the bounds on the index. This operation allows the programmer to improve performance by reducing the number of bounds checks when an index has already been checked or can be proved to be in bounds. Unfortunately, reading beyond the bounds of an array could easily lead to execution of arbitrary code in a zombie transaction—even an XEND instruction without a read of the fallback lock!

To safely use unsafe operations directly in transactions, the programmer must devise a correctness proof that encompasses the possibility of transactions executing with an inconsistent view of memory. Fortunately, unsafe operations are typically not used directly, but rather inside library routines that can be proven (manually) to be type safe for all possible combinations of parameters. If we imagine a transaction that calls such a routine with incorrect parameters (by virtue of having read inconsistent state) and that subsequently experiences an error (e.g., due to out-of-bounds indexing) there must exist another (non-transactional) program that calls the same library routine with the same parameters and thus must experience the same error—contradicting the assumption that the library routine is type safe for all parameters.

We can use similar reasoning to argue that the run-time system itself is type safe, for all parameters, allowing transactional support routines (e.g., for reads and writes), to be safely employed within transactions. Other parts of Haskell execution rely on similar safety guarantees. Parallel execution of pure functional code, for example, requires that objects and lazy updates are all initialized in the right order to prevent seeing uninitialized fields. This requires a correct implementation of the run-time system, code generator, and any compilation passes that might move accesses relative to one another.

3.3 Supporting Blocking

In the original GHC 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 TVars in its write set.

This scheme poses a challenge for all-hardware transactions: if such a transaction encounters a retry and aborts, any record of its read set will be lost. To address this issue, we track whether a transaction has, as yet, written any TVars; if not, we execute XEND instead of XABORT, and thus preserve the read set. If the transaction *has* performed shared writes, we execute an XABORT and restart the transaction from the beginning on the software fallback path (at which point, if execution follows the same path, it will block in software).

The maintenance of precise read and write sets also imposes significant overhead. Given that the complex data structures on which our work is focused make little use of retry, we have opted—in all our TM systems, including GHC's original STM—to represent sets imprecisely using small, 64-bit Bloom filters. These require no dynamic memory management, and inserts, lookups, and intersections are all extremely fast. (In the STM case, of course, TRecs still contain precise information to allow the commit phase to perform its validation and updates, and to allow transactions to read their own writes.) Future work will explore the impact of such parameters as Bloom filter size, hash function choice, and wakeup structure design.

Bloom filters can of course give false positives, leading to superfluous wakeups. Other designs might allow for false negatives and periodically ensure that all blocked transactions are woken. Certain programming idioms may tolerate one approach or the other better. A barrier implementation, for example, that uses retry to wait for the next phase of computation might experience intolerable delays between phases with false negatives. The popular async library, by contrast, uses suspended transactions to represent a large number of very rare conditions; false positives in this case might result in substantial wasted work.

In place of per-TVar watch lists, our Bloom filter-based retry implementation employs 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.

To minimize critical path length, we buffer wakeups and performing them after the commit lock is released. We also elide the global wakeup lock when searching for threads to wake. When an all-hardware but (so far) read-only transaction executes retry, it acquires the global wakeup lock before committing the hardware transaction. 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. It also prevents data conflicts on the wakeup structure itself from needlessly aborting the all-hardware transaction. Speculative elision of the lock is performed only by threads performing wakeup, so in the event of a conflict the more expensive and important retrying 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.

4 Mutable Constructor Fields

Hardware performance is highly sensitive to the memory footprint of transactions. Minimizing the footprint, especially of locations that are written, increases the chances that a transaction will commit and decreases the chances that it will cause a concurrent transaction to abort. Previous work with TStruct [20] reduced the indirections and the number of accesses by combining multiple transactional fields into a single heap object. While this improves performance, it comes at the cost of type safety and source-code clarity: as a practical matter, the TStruct work served only as a means of assessing the potential benefit of language extensions that might enable a similar—but type-safe—combining of fields.

In this section we describe such a language extension and its implementation, based on the (heretofore unimplemented) *mutable constructor fields* language proposal [12]. We provide a brief introduction to mutable fields in Section 4.1. We describe our extension to TM in Section 4.2 and outline our implementation (of both the original proposal and our extension) in Section 4.3. The extension fixes most of the type-safety issues with TStruct while providing even greater performance benefits.

4.1 Language Level Changes

The mutable constructor fields proposal uses the expressiveness of Haskell's *generalized algebraic data types* (GADT) syntax to declare data structures in which certain constructors return actions instead of values, and in which certain fields are identified as **mutable** (other fields are purely functional). The actions returned by such constructors must then be executed in the appropriate monadic context to produce the desired value. As a result, in-place mutation can safely be controlled without the need for indirection.

Constructors with a monadic context and mutable fields can be mixed together with pure functional constructors in a single data structure, allowing full support for Haskell's powerful pattern matching facilities and enabling the use of *pointer tagging*, an implementation mechanism that uses the low order bits of pointers to indicate the constructor of an evaluated object. Marlow et al. [13] found such tagging to improve the performance of typical programs by 10–15%.

As an example, consider a (non-transactional) variant of the binary tree introduced in Section 2.1. This structure has a Node constructor for a nontrivial tree, which holds a key and a left and right subtree, and a Nil constructor for an empty tree:

```
1  data Tree where
2    Node :: Key
3         → mutable Node — left child
4         → mutable Node — right child
5         → IO Tree
6    Nil  :: Tree
```

Where the Node constructor of Section 2.1 returned a Tree value, the version here (lines 2–5) returns an *action* that can be executed to generate the Tree, but only in an IO context. The Nil constructor still returns a pure Tree value. In the Node definition the **mutable** keyword indicates that the left and right subtree fields can be mutated, as we will see later. This mutation will be safe because it will be guaranteed to occur in an IO context.

Pattern matching in a **case** expression allows the programmer to bind variable names to fields of each constructor along with the code to evaluate if the scrutinized value matches that constructor. For example, we can write a function to search the tree for the presence of a key:

```
1  contains :: Key → Tree → IO Bool
2  contains searchKey tree
3    = case tree of
4        Nil → return False
5        Node key leftVar rightVar → ...
```

The **case** expression on Line 3 scrutinizes the value bound to the tree parameter. This value can either be a Nil value, leading to the code on Line 4, or a Node value, leading to Line 5. The variables introduced by the pattern match normally hold the values from the fields of the object being scrutinized. With mutable constructor fields, however, we cannot directly access the value as it may change over time. Instead the left and right fields will be bound to *references* of type Ref Tree. The underlying value can be accessed by primitive operations readRef and writeRef, which, when executed in IO, perform the access. The key variable is an immutable field and is bound to its value. We can now give the code for the Node case alternative:

```
5        Node key leftVar rightVar →
6          case compare searchKey key of
7            EQ → return true
8            LT → do
9              leftTree ← readRef leftVar
10             contains searchKey leftTree
11           GT → do
12             rightTree ← readRef rightVar
13             contains searchKey rightTree
```

The compare function gives an ordering value for the two keys. For the less than (LT) and greater than (GT) case alternatives, we use Haskell's **do**-notation to bind actions to access the object. On Line 9, for example, leftTree is a new variable bound to the value found in the leftVar field of tree. The key, leftVar, and rightVar variables are scoped to only the code to the right of the right arrow (→). In the previous TStruct work, field accesses did not have this safety and invalid dereferences could easily be expressed.

4.2 TM Extensions

In our extended version of mutable fields, we include support for the STM monadic context. Pattern matching an STM mutable field binds a TRef transactional reference; dereferencing is provided by readTRef and writeTRef operations. Since an immutable field pattern match will bind the *value* of the field, we do not need the unsafe APIs for non-transactional accesses that TStruct required for good performance.

We also added support for a single mutable array field per constructor by allowing the keyword **mutableArray** on the last parameter type. This extension supports such structures as the hashed array mapped trie (HAMT) (to be discussed in Section 5.1.2). When constructing a mutable array field, the size of the array and the initializing value are given as arguments for the field. Pattern matching binds to a TRefArray which can be accessed by operations that take an additional index argument. We leave safe nontrivial initialization of these arrays for future work.

4.3 Code Generation

The layout of each constructor's heap object is fixed at code generation time. With our extensions there are up to five parts to each heap object: header, extended header, pointer fields, non-pointer fields, and mutable array fields. The header is a reference to a shared *info table* that describes the layout of the heap object for GC. The extended header holds STM metadata; it is compatible with the TStruct layout. Pointer fields come next, to reduce the amount of memory traversed by GC. If the constructor has a mutable array field, its size is in the last non-pointer field. If the mutable array fields are pointers, then they are also traversed by GC.

GHC generates code for case expressions to evaluate the scrutinized expression and then jump to the correct code alternative based on which constructor matches. Each alternative begins with code to read the values of the various bound variables. The offsets for the reads come from the heap object layout. References are not bound to values but instead are GHC-internal pointers. Just before code generation these are expanded to two fields, an address and an offset. The address is bound to the pointer-tagged version of the main object, already a live value as it is used to choose the case alternative. The offset is bound to a constant that is adjusted to account for the specific tag bit pattern for the constructor. This constant avoids the need to mask the tag at run time, allowing the address of the field to be computed with a simple add. Addresses internal to a heap object cannot be stored on the heap, as GC requires all addresses it traverses to point to the header of a heap object.

4.4 Run-time Support

Two changes to TStruct layout were required for mutable fields. First, fields with mutable arrays have pointers at indexes beyond the non-pointer part of the heap object. When a field access is recorded in the transactional record it is now treated as a pointer access if its index is before or after the non-pointers. Second, all TStructs have a single info table, but for constructors with mutable fields we generate a new info table for each unique layout.

GHC's garbage collection is generational and when objects in older generations are mutated to point to values in a younger generation they need to be treated as roots in minor (nursery-only) collections. GHC uses a *mutated list* to track these objects and the GC write barrier adds mutated heap objects to the list. As an optimization, every mutable object has two possible info tables—clean and dirty. When mutated, if the header is clean it is switched to point to dirty and the object is added to the mutated list. Subsequent mutations do not add the object to the list again.

5 Performance Evaluation

Our results are from a 2-socket, 36-core, 72-thread Intel Xeon E5-2699 v3 running Fedora 24. We use a branch of the 8.0.2 version of GHC that supports TStruct and fine control over GHC's thread pinning mechanism. We fill cores of one socket (1–18), then the second socket (19–36), then hyperthreads (37–72). We compare the performance of three TM systems:

Hybrid– Attempt an entire transaction in HTM (Sec. 3.1.2), then fall back to elision of the coarse-grain commit lock (Sec. 3.1.1), then fall back to the coarse-grain STM.

Fine HTM– Attempt the commit phase of fine-grain STM in a hardware transaction (Sec. 3.1.3), then fall back to the existing fine-grain GHC implementation.

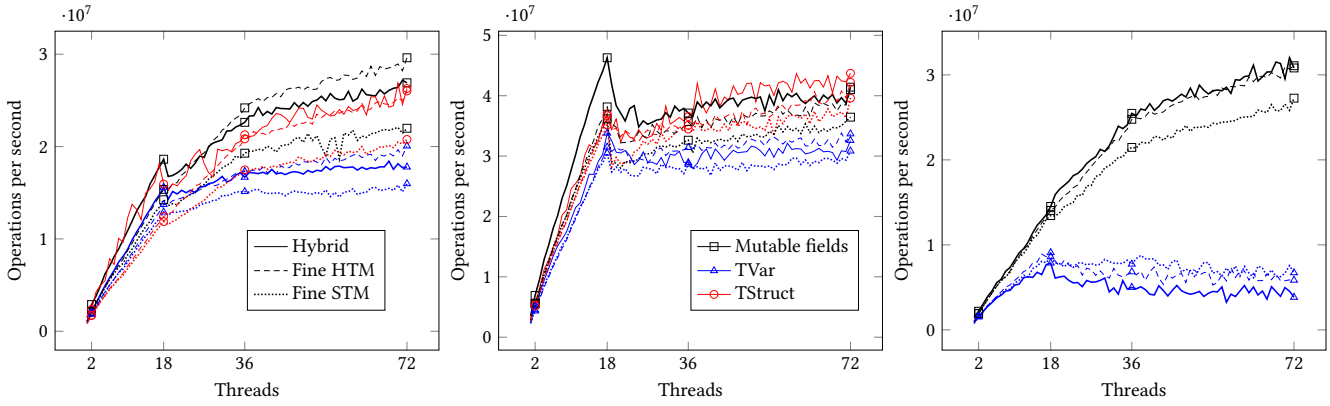
Fine STM– GHC's existing fine-grain code.

5.1 Data structures

We have results from three data structures implementing a concurrent set. We initialize each structure with 50,000 entries from a key space of 100,000. Worker threads then perform random insert, remove, and lookup operations in a 90:5:5 ratio. Because inserts and removes are balanced, the structure is expected remain half full, meaning that half of the inserts and removes will not change the structure and will thus be read-only transactions.

5.1.1 Red–Black Tree

For red–black trees, the TStruct work reduced both the size of nodes and the number of indirections relative to the equivalent TVar version. Mutable fields offer the same benefits, but better code generation: we can express trees as a two-way sum type, with a Node constructor (with data and mutable color and pointer fields) and a Nil constructor. With the TStruct implementation, the empty tree had to be represented as a top-level value rather than a constructor, and pointers to nodes could not take advantage of GHC's pointer tagging.



(a) Red–black tree. **(b)** HAMT. **(c)** Treap.

Figure 1. Throughput of TVar, TStruct, and mutable field structures with coarse-grain hybrid, fine-grain software with hardware commit, and fine-grain software runtimes. Note the differing y-axis scales. The line style key in (a) indicates the TM variant; the shape and color key in (b) indicates the data structure. Both keys combine to describe the lines in all three plots.

5.1.2 Hashed Array Mapped Trie

A *hashed array mapped trie* (HAMT) [1] is a set implementation that hashes the key of each value and then uses successive bit slices of the hash to control multi-way branching at each level of a search tree. To avoid wasted space for null pointers, internal nodes employ a compact representation that combines a presence bitmap with a dense array of pointers. Again building on the TStruct work, which in turn employs code from the stm–containers library [19], we use mutable fields for the compact arrays of both internal and leaf nodes. As in our red–black tree, HAMT benefits from better code generation. It offers three constructors, for empty trees, internal array nodes, and leaf array nodes.

5.1.3 Treap

A *treap* [14] is a binary tree that simultaneously maintains left-to-right ordering for keys and priority-queue (heap) ordering for randomly assigned "priorities." The randomization serves to maintain expected log-time operations without the complexity of typical rebalancing schemes. Code for the treap is much simpler than that of the red–black tree or HAMT. We experimented with several versions to assess their impact on on performance: one in which recursive calls return a triple of values, one that uses continuation passing, and one that passes mutable locations by reference. Working with more than one of these would have been extremely difficult without the type safety provided by mutable fields.

Treaps allow us to observe an important factor in the performance of hybrid transactions. When a key is inserted or removed from a treap, the code performs writes up the spine of the data structure. As they reach the top, these writes are likely to be superfluous, writing the value that is already in the field. The fallback path tracks values read and written in its TRec fields, allowing it to refrain from rewriting existing values, but all-hardware transactions cannot effect this optimization without redundant reads. As a consequence, all-hardware transactions that write to the treap are likely to conflict-abort regardless of the number of attempts.

We initially used a policy in which the maximum number of pre-fallback retries, for both all-hardware transactions and the fallback commit, was proportional to the number of active threads. The high conflict rate in treap suggested that we should lower the number of attempts for all-hardware transactions. Performance suffered, however, if we also lowered the number of attempts for the fallback commit: for that we needed to retain the proportional number of retries. For red–black trees, uniformly proportional retries had yielded the best results.

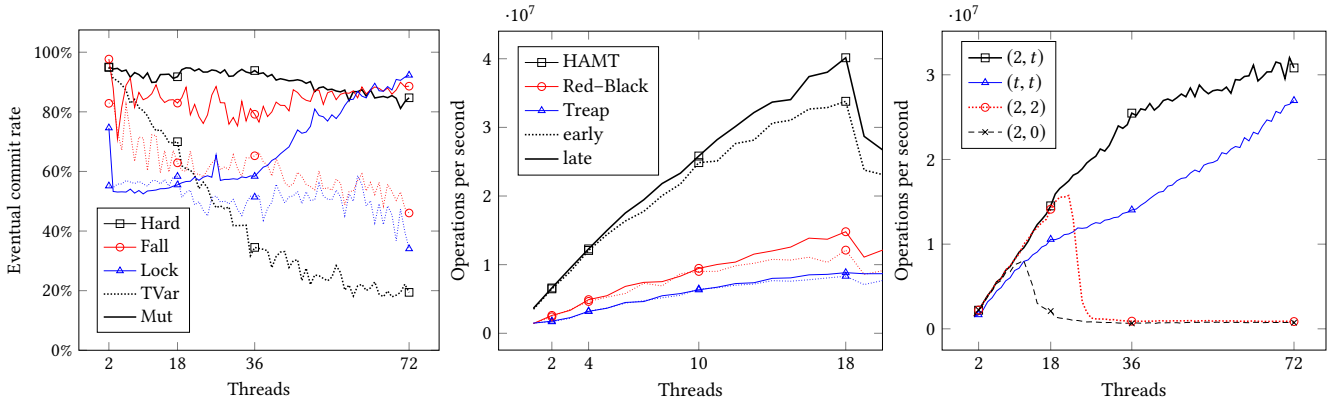
Treaps were not included in the TStruct work, and we were unable to evaluate their performance without the benefit of mutable fields. We do, however, compare with a TVar version.

5.1.4 Other Data Structures

The TStruct work included results for skip lists and cuckoo hash tables. While the code for these is unsafe, and uses software transactions only, the results suggest that these structures, too, should scale well in the hybrid system. Unfortunately, they require complex non-transactional initialization, a feature beyond the scope of our initial implementation of mutable fields.

5.2 Results

Figure 1 presents performance results for the red–black tree, HAMT, and treap. Our coarse-grain hybrid system does well in all cases, though Fine STM outperforms it slightly at scale on the red-black tree and ties it on treap. On HAMT, performance peaks with only one socket and no active hyperthreads. At this point, the hybrid system with mutable fields



(a) Treap eventual commit rate. **(b)** Early vs lazy subscription. **(c)** Treap retry policies.

Figure 2. (a) Eventual commit rate for treap, separated into all-hardware transactions (Hard), fallback hardware commit (Fall), and taking the fallback lock (Lock). Dotted lines are for the TVar version and solid lines are for the mutable fields version. (b) Early v. lazy subscription, with modified runtime that always acquires the lock during fallback. (c) Hybrid hardware transaction retry policies on treap. Key entries indicate all-hardware attempts followed by fallback hardware commit attempts (t is the number of threads).

outperforms Fine STM with TVars and TStructs by 52% and 28%, respectively. The TStruct version of HAMT outperforms mutable fields on hyperthreads, where we see an improved commit rate for fallback hardware commits. We attribute this to a small difference in layout with TStruct objects giving slightly better locality.

Treap shows the most drastic benefit from mutable fields. Note that without them, both Hybrid and Fine HTM perform worse than Fine STM. Our alternative versions of treap, discussed in Section 5.1.3, had similar performance; we show results for the tuple-returning version, which was generally the fastest. The indirections inherent in the TVar implementation greatly increase the read set of all-hardware transactions. Combining this with the inflated number of writes leads to a drastic decline in commit rate for all-hardware transactions. Figure 2a shows the rate at which treap transactions eventually commit, for each type of commit, on both the mutable field and TVar versions. To calculate the eventual commit rate for all-hardware transactions, we divide the number of all-hardware commits by the number of attempted transactions. Similarly, for fallback hardware commits, we divide the number of successful hardware commits by the number of transactions that fall back. Finally, for lock commits, we divide the number of successful validations by the number of lock acquisitions. This accounting lumps together hardware retry attempts, which should be significantly faster on successive attempts due to cache warming.

Figure 2b compares the performance of early and lazy lock subscription in our course-grain hybrid system. Under normal conditions there is no significant performance difference, as the fallback hardware transaction is very successful in avoiding lock acquisitions. To exhibit a difference in performance, we show results that skip the fallback hardware transaction and acquire the lock on fallback. This captures the behavior that would arise if, for example, hardware associativity conflicts interfered with fallback HTM. In this context, early subscription gives up to an 18% improvement in throughput.

Figure 2c looks at the retry policies mentioned in Section 5.1.3. Two attempts for each full hardware transaction and proportional attempts for each Fine HTM transaction gives the best performance on treap.

6 Conclusions and Future Work

Our work demonstrates transactional memory can provide not only attractive semantics but also good performance—and, in particular, that it can make effective use of hardware TM on current Intel processors. We explored several alternative means of leveraging HTM. In general, our results suggest that the best overall performance will be obtained with a three-level hybrid system that first attempts to execute a transaction entirely in hardware, then falls back to a software transaction body with an HTM commit phase, and finally resorts to a global commit-phase lock.

For scalable data structures without heavy reliance on Haskell's retry mechanism, we benefit significantly from using a compact Bloom-filter representation of read and write sets for transactional blocking and wakeup. We also can benefit from *lazy subscription* to the global commit lock within hardware transactions—an optimization that is fundamentally unsafe in languages like C, but can (with some effort) be made to be safe in Haskell.

Finally, we implemented the recent proposal for *mutable constructor fields* in Haskell (a major engineering effort), and

showed that they could be used to collapse indirection in transactional data structures, thereby achieving the benefits promised by earlier work on TStructs—namely, a major reduction in the memory footprint of transactions and a commensurate reduction in HTM abort rates.

There are several improvements to mutable fields that we leave to future work. We hope to offer type-safe initialization for mutable array fields along the lines of Haskell's ST monad [9] or the newly proposed and implemented linear types [2]. We might also hope to lift the restriction to a single mutable array at the end of a given structure.

Haskell's advanced type system is well suited to carrying additional information about transactions. Future work could use such information to generate better code. If, for instance, we know that a transaction cannot execute retry, we need not track its read set. Similarly, read-only transactions need not create, initialize, or postprocess a write set for waking up blocked peers.

The mutable fields feature also offers opportunities for further code improvements—e.g., to find and eliminate redundancies in the use of base-plus-offset addressing.

Acknowledgments

This work was supported in part by the National Science Foundation under grants CCR-0963759, CCF-1116055, CCF-1337224, and CCF-1422649, and by the IBM Canada Centres for Advanced Studies. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Phil Bagwell. 2001. *Ideal Hash Trees.* Technical Report. School of Computer and Communication Sciences, EPFL. http://lampwww.epfl.ch/papers/idealhashtrees.pdf.

[2] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: Practical Linearity in a Higher-order Polymorphic Language. *Proc. of the ACM on Programming Languages* 2, POPL (Jan. 2018), 5:1–5:29.

[3] Luke Dalessandro, François Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. 2011. 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)*. Newport Beach, CA, 39–52.

[4] Dave Dice, Timothy L Harris, Alex Kogan, Yossi Lev, and Mark Moir. 2014. Hardware Extensions to Make Lazy Subscription Safe. *arXiv preprint arXiv:1407.6968* (2014).

[5] Keir Fraser. 2004. *Practical Lock-Freedom.* Ph.D. Dissertation. Univ. of Cambridge Computer Laboratory.

[6] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. 2005. Composable Memory Transactions. In *Proc. of the 10th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*. Chicago, IL, 48–60.

[7] Intel. 2012. *Intel Architecture Instruction Set Extensions Programming Reference.* Intel Corporation. Publication #319433-012.

[8] Intel. 2016. *Intel 64 and IA-32 Architectures Optimization Reference Manual.* Intel Coroporation. Publication #248966-033.

[9] John Launchbury and Simon Peyton Jones. 1994. Lazy Functional State Threads. In *Proc. of the ACM SIGPLAN 1994 Conf. on Programming Language Design and Implementation (PLDI)*. Orlando, FL, 24–35.

[10] Yossi Lev, Mark Moir, and Dan Nussbaum. 2007. PhTM: Phased Transactional Memory. In *2nd ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*. Portland, OR.

[11] Virendra J. Marathe, Michael F. Spear, and Michael L. Scott. 2008. Scalable Techniques for Transparent Privatization in Software Transactional Memory. In *Proc. of the Intl. Conf. on Parallel Processing (ICPP)*. Portland, OR, 67–74.

[12] Simon Marlow. 2016. Mutable Constructor Fields. https://github.com/simonmar/ghc-proposals/blob/mutable-fields/proposals/0000-mutable-fields.rst

[13] Simon Marlow, Alexey Rodriguez Yakushev, and Simon Peyton Jones. 2007. Faster Laziness Using Dynamic Pointer Tagging. In *Proc. of the 12th ACM SIGPLAN Intl. Conf. on Functional Programming (ICFP)*. Freiburg, Germany, 277–288.

[14] Conrado Martínez and Salvador Roura. 1998. Randomized Binary Search Trees. *J. ACM* 45, 2 (March 1998), 288–323.

[15] Alexander Matveev and Nir Shavit. 2013. Reduced Hardware Transactions: A New Approach to Hybrid Transactional Memory. In *Proc. of the 25th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*. Montréal, PQ, Canada.

[16] Torvald Riegel, Patrick Marlier, Martin Nowack, Pascal Felber, and Christof Fetzer. 2011. Optimizing Hybrid Transactional Memory: The Importance of Nonspeculative Operations. In *Proc. of the 23rd ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*. San Jose, CA, 53–64.

[17] Michael F. Spear, Michael Silverman, Luke Dalessandro, Maged M. Michael, and Michael L. Scott. 2008. Implementing and Exploiting Inevitability in Software Transactional Memory. In *Proc. of the Intl. Conf. on Parallel Processing (ICPP)*. Portland, OR, 59–66.

[18] The GHC Team. 2015. *The Glorious Glasgow Haskell Compilation System User's Guide.* https://downloads.haskell.org/~ghc/8.0.2/docs/html/users_guide/

[19] Nikita Volkov. 2016. STM-Containers: Containers for STM. https://hackage.haskell.org/package/stm-containers

[20] Ryan Yates and Michael L. Scott. 2017. Improving STM Performance with Transactional Structs. In *Proc. of the 10th ACM SIGPLAN Intl. Symp. on Haskell (Haskell 2017)*. Oxford, UK, 186–196.

A Artifact Appendix

A.1 Abstract

The artifact contains both source and binary distributions of two versions of our changes to the GHC compiler as well as the benchmarks and supporting code used in our evaluation. Building and running according to the instructions below will result in plots matching Figures 1 and 2 from the PPoPP'2019 paper *Leveraging Hardware TM in Haskell*.

A.2 Artifact check-list (meta-information)

- **Program: TM throughput micro benchmarks are included.**
- **Compilation: GHC 8.0.2, gcc 7.3.0, g++ 7.3.0, and GNU make 4.1 are used to build our variants of GHC 8.0.2: hybrid, hybrid-early, and fine. These are used to build our benchmarks.**
- **Transformations: sed is used to switch hybrid source to hybrid-early.**
- **Binary: GHC 8.0.2 hybrid and fine.**
- **Run-time environment: Ubuntu 18.04.**
- **Hardware: Intel TSX.**
- **Run-time state: Sensitive to cache contention.**
- **Execution: Sole user execution for about 40 minutes for each figure on a large machine.**
- **Metrics: Logs record detailed per thread transaction statistics and per run throughput.**
- **Output: Plots in HTML5 files.**
- **Experiments: Scripts and manual steps.**
- **How much disk space required (approximately)?: 5GB**
- **How much time is needed to prepare workflow (approximately)?: 3 hours**
- **How much time is needed to complete experiments (approximately)?: 2 hours**
- **Publicly available?: Yes.**
- **Code/data licenses (if publicly available)?: BSD3, GPL, MIT, and GHC licenses.**

A.3 Description

A.3.1 How delivered

Our modifications to GHC are open source under the *The Glasgow Haskell Compiler License* and are hosted with binary and source distributions on Zenodo:

https://doi.org/10.5281/zenodo.1998472

After building the distribution will occupy about 5GB of disk space.

A.3.2 Hardware dependencies

Building and running our compiler and benchmarks requires a machine with Intel Transactional Synchronization Extensions (TSX). We have tested on a single socket, four core with two hyperthreads per core Intel i7-4770 and a dual socket, 18-core per socket, and two hyperthreads per core Intel Xeon E5-2699v3. In both these machines, TSX is not turned on. We followed instructions from Intel for enabling TSX for development. We recommend using a machine with TSX on by default.

A.3.3 Software dependencies

We have tested building and running on a clean install of Ubuntu 18.04. This required installation of `gcc`, `g++`, `make`, `libgmp-dev`, and `libncurses-dev`. It should work with any Linux distribution.

A.4 Installation

Detailed build instructions are included in the distribution. The source distributions of our variants of GHC are built with a standard `configure` script and `make`. Benchmarks are built with each compiler variant via a bash script build.sh. This needs the compilers to be put in a location the script can find by looking for each variant under an environment variable `GHC_COMPILERS`.

A.5 Experiment workflow

As described in the build instructions linked above, benchmarks can be run with the scripts included with the benchmarks: `fig-1.sh`, `fig-2a.sh`, and `fig-2bc.sh`.

A.6 Evaluation and expected result

Running the benchmark scripts results in plots in the `output` child directory. These plots should be consistent with the results from the matching figures in the paper. Some of the results are only apparent on machines with high thread counts. Relative order of code and compiler variations presented in the results should match.

We found results to be highly sensitive to thread affinity. We have provided some files (`topo-*`) that assign thread affinity by filling all the cores on a single socket before moving to the next and after all the discrete cores are assigned, filling hyperthreads in the same order. Individual machines may have a drastically different mapping from processor number to physical location.

A.7 Experiment customization

Additional parameters and configurations can be explored by following the pattern of the `fig-*.txt` files which specify a label, benchmark, compiler variant, benchmark code variant, full transaction attempt policy, and commit transaction attempt policy. For example, results on the coarse-grain STM (not shown in the paper) can be obtained by using the `hybrid` compiler and 0 for both the attempt policy columns. Additional benchmark command-line parameters can be set in the `fig-*.sh` scripts. The benchmark programs can be given a `--help` argument to see a list of configuration options.

The `plot.hs` and `plot-stats.hs` Haskell scripts are used to extract plots from the benchmark run logs. These scripts can be modified to plot additional information contained in the log files. The benchmarks output significant per-thread transaction event counts in a human readable format. In

addition GHC's runtime can output other statistics such as garbage collection.

A.8 Notes

We have benefited greatly from using Linux `perf` to get hardware transaction event counts and Intel's software development emulator to get detailed information about memory conflicts.

While we would like to support architectures with hardware transactions beyond Intel's TSX, GHC's support for other architectures is currently too limited.

A.9 Methodology

Submission, reviewing and badging methodology:

- https://cTuning.org/ae/submission-20180713.html
- https://cTuning.org/ae/reviewing-20180713.html
- https://www.acm.org/publications/policies/artifact-review-badging