Thesis Proposal: Improving the Performance of Synchronization in Concurrent Haskell

Ryan Yates

5-5-2014
Introduction

Outline

- Thesis
  - Why Haskell?
- Preliminary work
  - Hybrid TM for GHC
  - Obstacles to Performance
- Remaining Implementation Work
- Opportunities for Improvement
- Timeline
Haskell has existing and ongoing work that explores how to express concurrent and parallel execution. This high-level expressiveness gives opportunities to gain performance. Our focus is the synchronization performance of Concurrent Haskell.
Why Haskell?

- Active research community.
- Existing uses of STM.
- Purity by default and expressive type system gives unique opportunities and challenges.
  - Controlling of effects avoids tricky problems.
  - Lazy evaluation makes reasoning about performance difficult.
Hybrid TM for GHC.

- Full Haskell transactions can be executed as hardware transactions.
- Global lock on commit phase of software fallback elided with HLE.
- Fully software fallback acquires global lock for validation and commit.
Three Forms of Transactions

Hardware:
- XBEGIN
- XEND

Software (HLE):
- start
- commit

Software:
- Lock acquire
- Lock release
- commit

Software (HLE) and Hardware have separate transactions with distinct commands.
Start transaction with `XBEGIN`.
Read values directly from `TVars`.
Write values directly to `TVars`.
Commit transaction with `XEND`.
Check the fallback lock!
Start transaction with empty TRec.
- Each entry stores TVar, initial value, and new value.

Reads
- Search the TRec.
- If not found, read from the TVar and store in TRec.

Writes
- Write to TRec.
Fallback Commit

- Acquire global lock.
- Check that initial reads in TRec match values in TVars.
- Perform writes to TVars.
- Release global lock.
HLE Commit

- Acquire global lock. ← Elide the global lock.
- Check that initial reads in TRec match values in TVars.
- Perform writes to TVars.
- Release global lock.
Respecting the Fallback Commit
Obstacles to performance

C Call Overhead

Hardware transactions are sensitive to memory operations. Inlining read and write operations rather than calling C functions greatly increased performance.

Indirection

Indirection increases the number of memory locations that need to be read.
Indirection

Node
- key
- value
- parent
- left
- right
- color

TVar
- value
- watch

Watch Queue
- thread
- next
- prev

Watch Queue
- thread
- next
- prev

TRec
- prev
- index
- tvar
- old
- new
- ...
Handling retry and orElse

**retry**  Discard the effects of the current transaction and attempt the transaction again.

**orElse a b**  If action a reaches retry, then discard the writes of a and perform the action b.

The original implementation blocks the transaction’s thread when it encounters a retry without an orElse. It then waits for a change from a TVar in the read set to trigger a wake up.
Handling retry with Hardware Transactions

To support the same implementation of retry in a hardware:

- Track the *read* set to know which TVars to *wait* on.
- Track the *write* set to know which TVars to *wake* up.

Alternatively we could wake up *all* blocked transactions on *every* commit.
Use constant space to approximate the read and write sets.

Communicate the write set *and* discard writes by using XABORT.

- Only 8 bits can be communicated!
- Read-only transactions can use XEND.
- If we had non-transactional writes, we could do more.
We cannot do a *partial* rollback of writes.
Transactions that reach `retry` before any writes can move immediately to the alternative branch.
Our idea: rewrite transactions to buffer writes until after the decision to invoke `retry`.
Opportunities for Improvement

- Purity by default.
  - More code is free to move.
  - More flexibility for improvements.
  - Less reliance on static analysis.

- Ongoing work explores high-level APIs for concurrent and parallel programming.
  - This work is built on simple primitives.
  - Has the same challenges as our preliminary work.

- New and refined APIs.
  - Leverage the expressiveness of the types system.
  - Use high-level properties to improve synchronization.
Timeline

- **Summer 2014**
  - Finish `retry` and `orElse` support in hybrid TM.
  - Start implementing benchmarks that exercise `retry` and `orElse`.

- **Fall 2014**
  - Finish benchmarks for `retry` and `orElse`.
  - Implement compiler passes for improving code with `orElse`.
  - Start work on efficient `TArray` support.
  - Look for collaboration opportunities with `LVar` researchers.

- **Spring 2015**
  - Implement a NOrec-based STM and hybrid TM for GHC.
  - Apply what we have learned to “Lightweight Concurrency” branch.
  - Explore improvements to IO manager.
  - Explore further compile-time distinctions between transactions.
Timeline

- **Summer 2015**
  - Implement improved random supply inside transactions.
  - Formalize the Haskell TM’s interaction with unsafe operations.

- **Fall 2015**
  - Make run-time system improvements specific to parallel and concurrent libraries.
  - Explore transactions in the context of distributed processes.
  - Begin writing thesis

- **Spring 2016**
  - Continue work on unfinished projects.
  - Explore new opportunities discovered in the work done so far.
  - Finish writing thesis and prepare defense.