Improving Haskell STM Performance

Ryan Yates and Michael L. Scott

Department of Computer Science, University of Rochester, Rochester, NY, USA

Introduction

As computer hardware approaches physical speed limits, we are faced with the need to write more programs that execute in parallel where a significant amount of the work of the program is done on independent processing resources at the same time. Such programs are, however, hard to express with our present software technology.

Transactional Memory [1] (TM) aims to make it easier to express programs that can run in parallel by allowing programmers to group program actions into transactions that appear from the outside to happen all at once. This idea is a natural fit for the Haskell programming language. While Haskell’s transactional memory support [2] is popular, it is not as fast as we would like!

Our research addresses this problem in three ways:

1. Improving the internal representation of transactional data (TStruct).
2. Using hardware transactional memory (HTM) to boost performance.
3. Extending the language so the programmer can express efficient transactional data directly (future work).

Much of the poor performance from Haskell’s TM support stems from a bloated representation of transactional data. Values that are changed in transactions are isolated into objects called TVar’s. While this provides a simple interface for programming, it comes at the cost of performance. When multiple values need to be accessed in a transaction, distinct TVar’s may have to be loaded from distant memory locations. With our TStruct primitive operations, several values can be stored adjacent to each other, leading to faster access times. In addition, less space is required to store primitive values such as integers. The diagram below illustrates the benefits of using TStruct to implement a popular data structure called the Hashed Array Mapped Trie (HAMT).

Threads

Improving the internal representation of transactional data with TStruct improves performance, as can be seen in the plot above. As more and more parallel threads performing transactions (along the x-axis) perform operations on a central HAMT data structure, more work gets done (y-axis). This shows that TStruct is more effective at working in parallel than the original TM implementation, though not as effective as a very difficult to make specialized version HAMT called CTrie [3]. This CTrie is also limited to atomic operations on a single structure, not a composition of structures as in transactions.

Without TStruct

Multiple objects are used to represent a single node

Unnecessary indirection

With TStruct

Avoids unnecessary indirection

Compact representation

Hardware Transactional Memory

Processors from Intel and IBM now support Hardware Transactional Memory (HTM). Their support is limited, however, and work must be done to ensure that transactions that are too large have a software fallback. We have augmented Haskell’s TM support to attempt transactions using hardware and fall back to the software implementation. Critical to the success of HTM is getting the transactions as small and quick as possible. Our TStruct work helps with this. Some of our ongoing work aims at using static analysis to further improve HTM by recognizing key properties of some transactions and avoiding unneeded bookkeeping up front.

References


Acknowledgements

This work was funded in part by the National Science Foundation under grants CCR-0963759, CCF-116055, CCF-1337224, and CCF-1422649, and by support from the IBM Canada Centres for Advanced Studies.