Software transactional memory

From Wikipedia, the free encyclopedia

In computer science, **software transactional memory** (STM) is a concurrency control mechanism analogous to database transactions for controlling access to shared memory in concurrent computing. It functions as an alternative to lock-based synchronization, and is typically implemented in a lock-free way. A transaction in this context is a piece of code that executes a series of reads and writes to shared memory. These reads and writes logically occur at a single instant in time; intermediate states are not visible to other (successful) transactions. The idea of providing hardware support for transactions originated in 1993 by Maurice Herlihy and J. Eliot B. Moss. In 1995 Nir Shavit and Dan Touitou extended this idea to software-only transactional memory (STM). STM has recently been the focus of intense research and support for practical implementations is growing.

Contents

Performance

Unlike the locking techniques used in most modern multithreaded applications, STM is optimistic: every thread completes its modifications to shared memory without regard for what other threads might be doing, recording every read and write that it makes in a log. Instead of placing the onus on the writer to make sure it does not adversely affect other operations in progress, it is placed on the reader, who after completing an entire transaction verifies that other threads have not concurrently made changes to memory that it accessed in the past. This final operation, in which the changes of a transaction are validated and, if validation is successful, made permanent, is called a *commit*. A transaction may also *abort* at any time, causing all of its prior changes to be rolled back or undone. If a transaction cannot be committed due to conflicting changes, it is typically aborted and re-executed from the beginning until it succeeds.

The benefit of this optimistic approach is increased concurrency: no thread needs to wait for access to a resource, and different threads can safely and simultaneously modify disjoint parts of a data structure that would normally be protected under the same lock. Despite the overhead of retrying transactions that fail, in most realistic programs conflicts arise rarely enough that there is an immense performance gain over lock-based protocols on large numbers of processors.

However, in practice STM systems also suffer a performance hit relative to fine-grained lock-based systems on small numbers of processors (1 to 4 depending on the application). This is due primarily to the overhead associated with maintaining the log and the time spent committing transactions. However, even in this case performance is typically no worse than twice as slow; STM advocates believe this penalty is justified by the conceptual benefits of STM.

Theoretically (worst case behaviour) when there are n concurrent transactions running in the same time, there could be need of O(n) memory and processor time consumption. Actual needs depends on implementation details (one can make transaction fail early enough to avoid overhead), but there will be also cases (although rare) where lock based algorithms have better theoretical computing time than software transactional memory.

Conceptual advantages and disadvantages

In addition to their performance benefits, STM greatly simplifies conceptual understanding of multithreaded programs and helps make programs more maintainable by working in harmony with existing high-level abstractions such as objects and modules. Lock-based programming has a number of well-known problems that frequently arise in practice:

- They require thinking about overlapping operations and partial operations in distantly separated and seemingly unrelated sections of code, a task which is very difficult and error-prone for programmers.
- They require programmers to adopt a locking policy to prevent deadlock, livelock, and other failures to make progress. Such policies are often informally enforced and fallible, and when these issues arise they are insidiously difficult to reproduce and debug.
- They can lead to priority inversion, a phenomenon where a high-priority thread is forced to wait on a low-priority thread holding exclusive access to a resource that it needs.

In contrast, the concept of a memory transaction is much simpler, because each transaction can be viewed in isolation as a single-threaded computation. Deadlock and livelock are either prevented entirely or handled by an external transaction manager; the programmer need never worry about it. Priority inversion can still be an issue, but high-priority transactions can abort conflicting lower priority transactions that have not already committed.

On the other hand, the need to abort failed transactions also places limitations on the behavior of transactions: they cannot perform any operation that cannot be undone, including most I/O. Such limitations are typically overcome in practice by creating buffers that queue up the irreversible operations and perform them at a later time outside of any transaction.

Composable operations

In 2005, Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy described an STM system built on Concurrent Haskell that enables arbitrary atomic operations to be composed into larger atomic operations, a useful concept impossible with lock-based programming. To quote the authors:

> Perhaps the most fundamental objection [...] is that *lock-based programs do not compose*: correct fragments may fail when combined. For example, consider a hash table with thread-safe insert and delete operations. Now suppose that we want to delete one item A from table t1, and insert it into table t2; but the intermediate state (in which neither table contains the item) must not be visible to other threads. Unless the implementor of the hash table anticipates this need, there is simply no way to satisfy this requirement. [...] In short, operations that are individually correct (insert, delete) cannot be composed into larger correct operations.
> —Tim Harris et al, "Composable Memory Transactions", Section 2: Background, pg.2

With STM, this problem is simple to solve: simply wrapping two operations in a transaction makes the combined operation atomic. The only sticking point is that it's unclear to the caller, who is unaware of the implementation details of the component methods, when they should attempt to re-execute the transaction if it fails. In response, the authors proposed a **retry** command which uses the transaction log generated by the failed transaction to determine which memory cells it read, and automatically retries the transaction when one of these cells is modified, based on the logic that the transaction will not behave differently until at least one such value is changed.

The authors also proposed a mechanism for composition of *alternatives*, the **orElse** keyword. It runs one transaction and, if that transaction does a *retry*, runs a second one. If both retry, it tries them both again as soon as a relevant change is made. This facility, comparable to features such as the POSIX networking *select ()* call, allows the caller to wait on any one of a number of events simultaneously. It also simplifies programming interfaces, for example by providing a simple mechanism to convert between blocking and nonblocking operations.

Proposed language support

The conceptual simplicity of STMs enable them to be exposed to the programmer using relatively simple language syntax. Tim Harris and Keir Fraser's "Language Support for Lightweight Transactions" proposed the idea of using the classical *conditional critical region* (CCR) to represent transactions. In its simplest form, this is just an "atomic block", a block of code which logically occurs at a single instant:

```
// Insert a node into a doubly-linked list atomically
atomic {
    newNode->prev = node;
    newNode->next = node->next;
    node->next->prev = newNode;
    node->next = newNode;
}
```

When the end of the block is reached, the transaction is committed if possible, or else aborted and retried. CCRs also permit a *guard condition*, which enables a transaction to wait until it has work to do:

```
atomic (queueSize > 0) {
    remove item from queue and use it
}
```

If the condition is not satisfied, the transaction manager will wait until another transaction has made a *commit* that affects the condition before retrying. This loose coupling between producers and consumers enhances modularity compared to explicit signaling between threads. "Composable Memory Transactions" took this a step farther with its **retry** command (discussed above), which can, at any time, abort the transaction and wait until *some value* previously read by the transaction is modified before retrying. For example:

```
atomic {
    if (queueSize > 0) {
        remove item from queue and use it
    } else {
        retry
    }
}
```

This ability to retry dynamically late in the transaction simplifies the programming model and opens up new possibilities.

One issue is how exceptions behave when they propagate outside of transactions. In "Composable Memory Transactions", the authors decided that this should abort the transaction, since exceptions normally indicate unexpected errors in Concurrent Haskell, but that the exception could retain information allocated by and read during the transaction for diagnostic purposes. They stress that other design decisions may be reasonable in other settings.

Implementation issues

One problem with implementing software transactional memory is that it's possible for a transaction to read inconsistent state (that is, to read a mixture of old and new values written by another transaction). Such a transaction is doomed to abort if it ever tries to commit, but it's possible for inconsistent state to cause a transaction to trigger a fatal exceptional condition such as a segmentation fault or even enter an endless loop, as in the following contrived example from Figure 4 of "Language Support for Lightweight Transactions":

Transaction A	Transaction B
`atomic { if (x != y) while (true) { } }`	`atomic { x++; y++; }`

Provided *x=y* initially, neither transaction above alters this invariant, but it's possible transaction A will read *x* before transaction B updates it but read *y* after transaction B updates it, causing it to enter an infinite loop. The usual strategy for dealing with this is to intercept any fatal exceptions and periodically check if each transaction is valid; if not, it can be aborted immediately, since it's doomed to fail in any case.

Implementations

A number of STM implementations (on varying scales of quality and stability) have been released, many under liberal licenses. These include:

- The TL2 lock-based STM from the Scalable Synchronization research group at Sun Microsystems Laboratories, as featured in the DISC 2006 article "Transactional Locking II".
- The STM library, as featured in "Composable Memory Transactions", is part of the standard Glasgow Haskell Compiler distribution.
- SXM, an implementation of transactions for C# by Microsoft Research. Documentation, Download page.
- Several implementations by Tim Harris&Keir Fraser, based on ideas from his papers "Language Support for Lightweight Transactions", "Practical Lock Freedom", and an upcoming unpublished work.
- The Lightweight Transaction Library (LibLTX), a C implementation by Robert Ennals focusing on efficiency and based on his papers "Software Transactional Memory Should Not Be Obstruction-Free" and "Cache Sensitive Software Transactional Memory".
- LibCMT, an open-source implementation in C by Duilio Protti based on "Composable Memory Transactions". The implementation also includes a C# binding.
- RSTM The Rochester STM written by a team of researchers lead by Michael Scott. The code was written by Mike Spear, Virendra Marathe, and Chris Heriot, with contributions from Athul Acharya, David Eisenstat, Bill Scherer, Arrvindh Shriraman, and Vinod Sivasankaran.
- SCAT reseach group's implementation of AtomJava.
- JVSTM implements the concept of Versioned Boxes proposed by João Cachopo and António Rito Silva, members of the Software Engineering Group - INESC-ID
- TARIFA is a prototype that brings the "atomic" keyword to C/C++ by instrumenting the assembler output of the compiler.
- STM for Perl 6 has been implemented in Pugs via the Glasgow Haskell Compiler's STM library.
- DSTM2 Sun Lab's Dynamic Software Transactional Memory Library
- Durus is a simple, yet mature, complete and fast, STM implementation for Python, allowing both STM inside a single process and STM in a server/multiple clients architecture. In addition to its built-in storage format, there are others available such as one based on Berkeley DB available here.

References

- Nir Shavit and Dan Touitou. Software Transactional Memory. *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pp.204–213. August 1995. The paper originating STM.
- Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software Transactional Memory for Dynamic-Sized Data Structures. *Proceedings of the Twenty-Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, 92–101. July 2003.
- Tim Harris and Keir Fraser. Language Support for Lightweight Transactions. *Object-Oriented Programming, Systems, Languages, and Applications*, pp.388–402. October 2003.
- Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable Memory

Transactions. *ACM Conference on Principles and Practice of Parallel Programming 2005* (PPoPP'05). 2005.
- Robert Ennals. Software Transactional Memory Should Not Be Obstruction-Free (dead link). Archived: Software Transactional Memory Should Not Be Obstruction-Free.
- Michael L. Scott et al. Lowering the Overhead of Nonblocking Software Transactional Memory gives a good introduction not only to the RSTM but also about existing STM approaches.

External links

- Cambridge lock-free group
- Software transactional memory Description; Derrick Coetzee
- Transactional Memory Bibliography