Introduction to STM Systems

Over the last few decades, much of the gain in software performance can be attributed to increases in CPU clock frequencies. However around 2004, 50 years of exponential improvement in the performance of sequential computers ended [1]. Industry's response to these changes was to introduce single-chip, parallel computers, variously known as "chip multiprocessors," "multicore," or "manycore" computers.

In order to get a continued speedup on these processors, applications need to be able to harness the parallelism of the underlying hardware. This is commonly achieved using multi-threading. Therefore, there is an increasing need for programmers to produce applications in which multiple processes execute in parallel and coordinate to achieve some shared task.

Yet writing correct and scalable multi-threaded programs is far from trivial. While it is well known that shared resources must be protected from concurrent accesses to avoid data corruption, guarding individual resources is often not sufficient. Sets of semantically related actions may need to execute atomically to avoid semantic inconsistencies. For instance, if a thread *t1* wishes to access data-items *x* and *y* atomically, then between the accesses of *t1*, no other thread (say *t2*) should access these data-items. Thus the following execution should not be allowed:

t1.read(x) t2.write(x) t2.write(y) t1.read(y)

Traditionally locks have been used to synchronizing threads. Threads wishing to the access multiple shared data-items atomically, obtain locks for all the shared data-items, performs the access and then releases the locks. This approach is called as two phase locking (2PL).

Issues with Locking: Lock-based parallel programming, however, is widely acknowledged as a difficult and error-prone task. Several classes of bugs that can result from mistakes in this area cannot be found by static analysis, and can sometimes be difficult to find even with the most rigorous testing, as they are only triggered under precise timing conditions while executing tasks in parallel. All of this increases the cost of developing high performance parallel software [2].

More importantly, systems built using locks are difficult to compose without knowing about their internals. Correct fragments may fail when combined [3]. Composition of software is a very useful property which is used to build large software systems using simpler software systems. Using composition existing pieces of correctly executing software can be trivially combined to form a larger piece of software that executes correctly. It is the basis of modular programming.

Alternative to locking – STM Systems

Due to these difficulties with locks, alternative approaches were looked into. Software transactional memory (STM) is an approach which has garnered significant interest as an elegant alternative for developing parallel programs. Software transactions are units of execution in memory which enable concurrent threads to execute seamlessly. Software transactions address many of the shortcomings of lock based systems. This idea originated from transactions in databases.

Unlike the lock-based programming, STM approach is optimistic: a thread completes modifications to shared memory without regard for what other threads might be doing, recording every read and write that it is performing in a log. The STM system then looks into the log and validates if the actions of the thread can be allowed to become permanent (called *committed*) or not (*aborted* then). The benefit of this 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.

Another advantages of STMs is that it provides a very promising approach for composing software components [3]. STMs achieve composition through nesting of transactions. A transaction is called

nested if it invokes another transaction as a part of its execution.

Programming support for STMs

In order to execute code accessing shared data-items as transactions, the user designates piece of code in an atomic block. For instance, the following code will be executed as a transaction by the STM system.

```
atomic {
   if (x != null)
     x.foo();
   y = true;
}
```

The STM system's responsibility is to execute these transactions as if they were atomic — as if the entire body of the transaction were executed at a single moment of time. As discussed above, a transaction executes to completion then it is committed and its effects are visible to other transactions. Otherwise it is aborted and none of its effects are visible to other transactions.

Thus it can be seen that not much change is required to the existing code. Sections of the code which access shared objects have to be designated as transactions. The STM system ensures that they execute atomically.

Conclusion

Software transactional memory (STM) is an approach which has garnered significant interest as an elegant alternative for developing parallel programs. Software transactions are units of execution in memory which enable concurrent threads to execute seamlessly. Software transactions can be employed to address many of the shortcomings of lock based systems.

References

[1] K. Olukotun and L. Hammond, "The Future of Microprocessors," ACM Queue, Vol. 3(7), pp. 26–29, 2005.

[2] James R. Larus and Ravi Rajwar, "Transactional Memory", Morgan & Claypool, 2006.

[3] Tim Harris, Maurice Herlihy, Simon Marlow, and Simon Peyton-Jones. Composable memory transactions. In Proceedings of PPOPP, Jun 2005.