

TM Hardware

The hardware mechanisms needed for a bounded TM implementation (e.g., register checkpointing, read/write bits per cache line, mechanism to clear read/write bits on commit, hardware to abort a transaction if anyone tries to steal permission and transfer control to a different PC), are useful in other situations, such as for speculative optimizations and speculative execution of lock-based code. Also, schemes such as HybridTM could benefit from bounded TM hardware, and could serve as a stepping stone to an unbounded TM implementation.

Several people voiced the concerns that TM hardware should provide only mechanisms, needs to avoid over-commitment to particular policies, and must be able to coexist with existing software without transactions.

There was also a concern that if hardware vendors each come up with a different implementation of TM, then software won't work.

One comment made during the discussion was that computer hardware is definitely going to change (e.g., the number of cores doubling with every generation). Either the software stack can be rebuilt and change in parallel with hardware, or the changes in hardware will get ahead of the software.

Semantics

In general, people agree that TM needs good semantics. There seems to be a tension between having nice semantics and achieving good performance, however.

Some voiced the opinion that semantics seem more important than performance, and that there is a pressure (especially in software TM) to tradeoff semantics for performance.

One of the main semantic issues discussed was strong vs. weak atomicity / interactions between transactional and nontransactional code. Should it be possible for transactional and non-transactional code running in parallel which access the same data? If the answer is no, then should a particular error behavior be specified, and what should that behavior be?

One argument against having each read/write be a mini-transaction is that this decision might eliminate many compiler optimizations. Another question discussed was to what extent transaction boundaries should act as barriers which prevent reordering of instructions.

Another topic of discussion revolved around how TM interacts with data-level semantics. If there are ways for the programmer to specify data which is local/unshared/private etc., then TM might not need to track accesses to that data when checking for conflicts.

Other semantic issues mentioned include what kinds of synchronization are available in a TM system or how does one express concurrency.

Several people argued that the lack of a standard semantics / single parallel programming model for TM impedes the development of applications. It was suggested that perhaps TM needs some kind of abstract standard (e.g., like virtual memory) which provides a common abstraction, but whose implementation details can vary.

Using TM / Applications

There exist examples of complex pieces of code which would be simplified by transactions. One question that was posed was whether transactions might only be a tool for the expert programmer.

TM is likely not suited for all types of parallel applications. One example suggested was a program which spawns a thread for every item that comes into a webserver. Another class of examples might be programs where MPI is more suitable. There did not appear, however, to be consensus on exactly how far the transactional model should extend.

During the discussion, several potential applications for TM were mentioned. One question posed was whether TM can be used for real-time programs, such as real-time garbage collection. Is it possible to provide some precise performance model for a TM system?

One application mentioned in passing was game character interactions (e.g., in MMORPGS), where transferable resources equivalent to cash are exchanged.

It was suggested that having a commercial presence who can release an STM that people can use might aid in building a user community for TM.

Language / Compilers

One topic discussed was how language constructs might be used to specify private data. In recent work at Rochester, the TM system uses 4 variants of smart pointers.

Language constructs might be used for multiple purposes. For example, the construct might be used only as an aid / annotation to the compiler to help improve performance, but while keeping the same semantics (e.g., physical serializability). On the other hand, a language construct might also be used to mark places where the programmer intends to break physical serializability. It seems that the same mechanism / language construct might serve one or both purposes.

There was some discussion of X10, and in particular, the use of places. How difficult is it for programmers to write code and partition data into places? It may restrict the programming model, but on the other hand, it may force the programmer to reason about locality. The current implementation only supports transactions on one place, but there has been investigation into supporting atomic transactions over a bounded number of places. The current proposed implementation uses locks; how does one implement an optimistic scheme?

With regards to compilers for transactional memory, it was suggested that the primary concern should be to provide good semantics for TM, and that perhaps a good compiler can eliminate many performance concerns. Another comment was that TM requires a shift in some of the standard compiler optimizations; for example, instead of redundant operation elimination, TM needs idempotent operation elimination. TM also needs optimizations such as not logging things twice.

One question posed was whether TM should commit to the serializability of non-transactional memory operations or not. The questions of memory ordering prompt the question of what exactly we should count as a dependency. It was suggested that this question is important for compilers, since compilers can more closely implement whatever model we specify with dependency analysis.

Other Topics

Non-blocking vs. Blocking

One topic of discussion was how important non-blocking progress guarantees actually are.

In experiments where blocking code beats non-blocking code, is this result only because of the overhead of extra atomic operations in non-blocking code, or is it because blocking code is inherently less complex? Are CAS operations become cheaper or more expensive?

Some suggested that non-blocking codes usually win with enough processors; on the other hand, simply aborting a transaction before a context switch is not strictly non-blocking, but can improve performance.

Even with non-blocking guarantees in the system, it is still possible to introduce application-level blocking with constructs such as conditional critical regions or user-level retry.

Utility functions / cost

Has the idea of utility functions come up in other concurrent thread-schedule applications? One example of something that might benefit from such a scheme is a system which tries to properly balance the amount of locality / cache-affinity with good load balancing of tasks. This balancing is dependent on the algorithm, what else might be going on in the system, properties / constant factors in the hardware, etc. One challenge which hasn't not yet been addressed is how to devise smart, adaptive runtimes.