

Session 3 presentations:

David Wood:

talk:

- coffee is good
- Hardware TM (in some form) is inevitable
- Virtualize TM into
 - Rollback: log-based rather than buffer and cache based
 - Access Summary: read/write sets (for conflict detection) implemented in per-process registers
 - Access Checks: detects concurrent access conflicts in cache and summarizes it for the OS

Q/A:

- logs may be useful for other (non-TM) work. e.g. speculation, GC read/write barriers
- how easy is it to change conflict detection policy? Currently exploring the level of software control necessary

Colin Blundell:

talk:

- Let actual programs drive semantic consensus
- Divorce semantics from conflict detection to allow a wide range of HW implementations
- Start simple (with a global lock) and refine/extend as workloads dictate

Q/A:

- What about I/O in the presence of speculation? May need to annotate I/O code.

Rick Hudson:

talk:

- Projectors can be finicky
- Synchronization and atomic block semantics subtly differ (non-repeatable reads, intermediate dirty read)
- Concurrent programming SDEs may be TM's killer app

Q/A:

- What isolation problems would you like to see solved, and how? Java allows good isolation, C/C++ is much harder.

Strong atomicity is a good option if you can do it, otherwise abort into 'semantically comfortable' regions.

- There will always be more S/W threads than H/W threads, right? Not necessarily, performance is usually good for

blocking code up to #HW threads, non-blocking is good for #S/W threads > #H/W threads. Now we use asynch. pre-emption, but we may want to switch to a co-operative mode.

Siddhartha Chatterjee:

talk:

- What is TM good for?
- Window of opportunity rapidly shrinking for TM killer app.
- Websphere? Second Life?
- Value demonstrated as a performance/productivity trade-off
- Single thread performance is vital
- What else could HW TM primitives be good for?
- TM is a programming model problem. Must be thought of as an entire SW stack.

-Keep HW extensions minimal (to avoid design team pushback)

-Policy separation.

Q/A:

-What is the killer app? Maybe web-apps, maybe on-line gaming (servers).

-You said keep it out of the core? Out of the proc. core. Easier to integrate into the memory system.

Eliot Moss:

talk:

-Ordering control is attractive to people used to wait/signal style programming

-OSes are almost certainly going to need a nicer interface (to compose, transactionally, syscalls)

-Locality/affinity, scheduling, and load balancing are as important as bulk parallelism (demonstrated by Galois)

Q/A:

-What about the VM adoption model (software maturing into hardware)? Virtual memory is a better analogy.

-Are there examples of tricky abstract locking? Trickiness depends on who you are. Experts will be able to understand this,
and so may not need CC generation.

-What level of programmer is this for? Expert programmers and library implementors. Abstract locks may be the wrong abstraction,
perhaps the Galois model is more convenient.

-The DB community tried a failed to make good on relaxing serializability in the 80s.

-What about Transaction ordering? Stanford has been working on mixing ordered and unordered transactions. Galois group and

IBM have also done some recent work with ordering. Ordering itself is important for message queues and transaction communication.

Discussion:

-What are simple transactional semantics?

Strong atomicity is a different way of thinking about data (compared to normal PL).

Transactions are simpler than explicit lock operations,

but is still harder to reason about than a single-threaded system. People are used to reasoning about things in a purely serial fashion.

Semantics also shouldn't overly constrain the breadth of compiler optimizations possible.

-What about single reads and writes?

They can be tricky, however in managed languages most of the coding is done in terms of classes/objects. Encapsulation could prevent most of these nasty cases.

-Are non-transactional writes to transactional data errors?

Possibly, but non-transactional reads seem to be a useful and common construct. Labeling non-transactional reads may not be sufficient (Splash benchmark as an example). Non-labeled code may not maintain semantics (unlike const in C/C++).

-UPenn position is not weak vs strong atomicity. The global lock idea is to allow programmers to think about atomic sections in a serial fashion, as well as to allow

serializing in some uncommon cases. Programmer-level retry and abort can mess up single-lock semantics.

-What about equivalent constructs to Condition Variables? Some abstraction to allow transaction ordering?

Retry may not need strict abort semantics, it could offer an early partial commit and wait. Retry is often criticized because its overkill and often only one or two conditions need to be checked rather than the whole read set. Retry with conditions can confuse composability. In a fixed-size work queue, retry will automatically check that the queue is empty for you.

-Retry in the simple serial model could lead to deadlock. If the purpose of retry/commit is to avoid rollback, then what about exceptions (which perform a kind of rollback)? What does conditional critical code look like in practice? Most of the synchronized code was guarding a series of method calls, some were doing wait/notify, but a significant few were making these deep, complex method calls.

-What about non-synchronized reads? David Bacon defined it to be an exception, has anyone looked at that?

It's very attractive, there's a significant question about efficiently detecting this dynamically, and recognizing it statically. This seems to make sense as many of these may be unintentional. However, detecting a non-transactional read at runtime is currently no cheaper than just making it strongly atomic (but the semantics are different (SA -> abort, exception -> error)).

-Which is better or preferable?

The consensus was that the implementation would be similar, although strong atomicity could quash a lot of compiler optimizations, requiring exact exception semantics at potentially erroneous reads would be at least as, if not more, expensive (because excepting operations act as reordering barriers).

-It seems that in a lot of cases, people want a transaction to act like a big fat instruction. So why not act like that and allow a certain amount of re-ordering?

Initialization is a big deal, and we can't reorder around it (so that no external thread sees uninitialized data). However, as long as the compiler can maintain a happens-before graph, it can re-order instructions. But, this may require global analysis in order to not have to treat all transactions as barriers.

-Volatile is too intrusive, ideally I would prefer separate notions of atomicity and ordering. Is this a good idea?

Some felt that it would be a mistake to break straight-line ordering semantics, however it would be occasionally useful to allow atomic operations to be re-ordered. Although, such concepts really are intended only for expert programmers. Given that serial ordering is so natural it seems like that should be the default, with a separate notation available for allowing atomic ops to be re-ordered by the compiler.

-Strong atomicity conflicts with compiler operations. Often compilers would like to cache in local data, values read from the heap from within a loop. However, if this heap data can

be changed (for instance, an array reference could be changed), then the compiler may have to read such previously cacheable data on every iteration.

-What about granularity issues for packed data?

You can't mix open/closed transactional/non-transactional in the same unit of concurrency. And padding all objects to cache-lines is very space inefficient and may increase the work for GC and memory

allocation (unless it can be targeted a just a very few objects).

-What do people think about open-nesting in general?

The need for it is clear, the open nesting solution is a possible solution. There may be a better option. However, with relational data bases as an example, it seems that library data structures should be implemented using something like open nesting by programming experts (much as are the b-tress in a DB). The difficulty is how to integrate the memory models of a closed-nested system (for beginner/user programmers) with a library implemented in an open-nested fashion, and how to offer decent assurances against deadlock.

Lunch

Informal Discussion:

-Benchmarks are vital. But different STMs have different APIs, so it's difficult to create a cross-platform set of benchmarks. Perhaps it would be fine to have a collection of programs, even if they have to be rewritten for each STM system. Perhaps the collection of benchmarks should require participants to distribute their source changes if they publish the numbers.

-Ease of use is also important, but most people don't do parallel programming. Is it going to be easier to get current concurrent programmers to switch to TM, or to use TM to teach concurrent programming to people who have only written serial code?