2nd Wkshp. on the Theory of Transactional Memory (WTTM), Cambridge, MA, Sep. 2010. In conjunction with DISC '10.

On the Orthogonality of Speculation and Atomicity

Michael L. Scott and Luke Dalessandro

Computer Science Department, University of Rochester
{scott, luked}@cs.rochester.edu

WTTM 2, September 2010

In a paper at DISC 2010 [1], we argue that atomic blocks should constitute the synchronization operations of a memory consistency model, and that traditional synchronization mechanisms (locks, condition variables, monitors, etc.) should be defined in terms of transactions, rather than the other way around. While speculation is likely to be used in any implementation of transactional memory, our model deliberately avoids mentioning this at the program level: atomic blocks are simply atomic; they do not abort or roll back in any way that affects the meaning of the program.

We provide a conventional retry mechanism [2] for condition synchronization, but its semantics are declarative, not operational: the dynamic behavior of a program containing retry is that of an execution (if there is one) in which retry statements are never encountered. In this sense, "if (!C) retry" might better be written "require(C)".

As a declarative mechanism, our retry does not specify the circumstances under which a speculative implementation should re-attempt a failed transaction. A typical implementation might, as suggested by Harris et al. [2], wait until something read by the transaction has changed. Alternatively, it might retry immediately, or after a small delay. Immediate retry is probably sufficiently important (for algorithms with explicit programmer-implemented speculation [5]) to merit its own source-level syntax (restart?), but again, the distinction is a matter of tuning, below the level of program semantics.

As originally presented, our model does *not* accommodate abort (abandon this transaction and do not retry) or orElse (try an alternative version of this transaction if preconditions are not met [2]). Both of these constructs require an explicit notion of speculation, as they "leak" the fact that a transaction has aborted. This note outlines how we might add such constructs as extensions.

We take the position that language-level speculation can and should be separated from atomicity. It is useful in its own right in sequential programs, and cleanly explains abort and orElse when used inside an atomic block.

Consider a spec construct, loosely modeled after the try-all of Shinnar et al. [4]:

```
spec {
    ...
    if (!C) fail
    ...
} else {...}
```

Operationally, the intent is that when fail is executed, all work since the beginning of the block is "undone," and execution continues in the else block. Most programmers have probably wished, at one time or another, for a try block that works like this.

To accommodate speculation in a memory model for sequential programs, we can extend the set of operations with begin_spec, end_spec, and fail. In any legal execution, these operations must occur in properly nested begin_spec... end_spec or begin_spec... fail pairs. We then define a tree-structured *visibility* order, $<_v$, that is a subset of program order, $<_p$. The intent of $<_v$ is to "hide" writes within a failed speculative region from reads that follow the region. For two operations a and c, we say $a <_v c$ iff (1) a is c's immediate predecessor in $<_p$ and c is not a fail operation; (2) c is a fail operation and a is the matching begin_spec operation, or (3) $\exists\ b$ such that $a <_v b <_v c$.

For data-race-free parallel programs, if spec blocks appear only inside atomic blocks, then no thread can ever see another thread's speculative writes, and the sequential model carries over cleanly. In keeping with our previous work, execution of such a program is *transactionally sequentially consistent* (TSC) if there exists a total order $<_g$ on all operations such that (1) $<_g$ is consistent with program order in all threads, (2) the operations of any given transaction are contiguous in $<_g$, and (3) $<_g$ induces a (tree-structured) visibility order $<_{gv}$ that explains the program's writes. (Because spec blocks occur only inside atomic blocks, a "side branch" of the tree will consist entirely of operations of a single thread.) We say that $a <_{gv} c$ iff (a) a is c's immediate predecessor in $<_g$ and c is not a fail operation; (b) c is a fail operation and a is the matching begin_spec operation; or (c) $\exists\ b$ such that

$a <_{gv} b <_{gv} c$. Each read is required to return the value written by the (unique) most recent previous write in $<_{gv}$.

Relaxed memory models are analogous. Again assuming that spec blocks appear only inside atomic blocks, we define *strict serializability* (SS) as follows: $a <_{ss} c$ iff (a) $\exists$ transactions A, C: $a \in A$, $a <_v \mathsf{end}(A)$, $c \in C$, and $A <_t C$; (b) $\exists$ transaction A: $a \in A$, $a <_v \mathsf{end}(A)$, $c \notin A$, and $A <_v c$; (c) $\exists$ transaction C: $c \in C$, $a \notin C$, and $a <_v C$; or (d) $\exists\, b$: $a <_{ss} b <_{ss} c$. The reference to $\mathsf{end}(A)$ (A's end_txn operation) in clauses (a) and (b) makes failed speculative writes invisible to subsequent transactions, and keeps them invisible to accesses after the transaction in their thread.

Given this definition, a program execution is strictly serializable iff there exists a transaction order that, together with program and visibility order, induces a strict serial order $<_{ss}$ that explains the execution's reads. In this extended model, a read r is permitted to see a write w if they access the same location, $w <_v r \vee w <_{ss} r$, and there is no intervening write of the same location between w and r. (In some languages, r may also be permitted to see incomparable writes.) A transactional memory implementation (system) is strictly serializable if each of its realizable target executions is equivalent to (produces the same external effects as) some strictly serializable program execution. With appropriate definitions, a *transactional data-race-free* (TDRF) program can be shown to display TSC behavior on any SS system.

Having separated atomicity and speculation, we can model abort as follows:

```
atomic {
    spec {
        ...
        if (!C) fail      // abort
        ...
    } else { }
}
```

More interestingly, orElse, which supports disjunctive composition (atomically do *this* or, if its precondition isn't met, do *that*), can be modeled as

```
atomic {
    spec {
        ...
        if (!C1) fail
        ...               // this
    } else {
        ...
        if (!C2) fail
        ...               // that
    ...
    } else {retry}
}
```

Note that while transactions can fail spuriously at the implementation level in a typical TM system, spec blocks fail at the semantic level only when they reach a fail statement, and atomic blocks don't fail at all.

If we wish to make spec and atomic fully orthogonal, we must consider the behavior of atomic blocks or data races inside outermost (unnested) spec blocks. The motivation and desired semantics for such idioms are not immediately clear. If thread T sees speculative writes from a spec block in thread S that may subsequently fail, should T inherit S's speculative status? Alternatively, should S be allowed to "undo" its writes (via some sort of compensating action) while allowing T to continue? The first option would appear to induce significant implementation complexity; the second would appear to induce semantic complexity at least as severe as that of open nesting [3].

The bottom line: By separating speculation and atomicity, we (1) allow the former to be used without the latter in sequential contexts, e.g. for recovery from program-detected error conditions; (2) maintain the simplicity of "merely atomic" transactions; and (3) avoid conflating aborts due to implementation issues (hash table conflicts, hardware limitations, interrupts, etc.) with restarts due to failed preconditions. In parallel contexts, appropriate semantics for non-atomic speculation remain unclear.

Acknowledgments

The ideas in this note benefited from discussion with Mike Spear, Victor Luchangco, and the anonymous reviews of our DISC 2010 paper. Our work is supported in part by the National Science Foundation under grants CNS-0615139, CCF-0702505, and CSR-0720796.

References

[1] L. Dalessandro, M. L. Scott, and M. F. Spear. Transactions as the Foundation of a Memory Consistency Model. In *Proc. of the 24th Intl. Symp. on Distributed Computing*, Cambridge, MA, Sept. 2010.

[2] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable Memory Transactions. In *Proc. of the 10th ACM Symp. on Principles and Practice of Parallel Programming*, pages 48-60, Chicago, IL, June 2005.

[3] J. E. B. Moss and A. L. Hosking. Nested Transactional Memory: Model and Architecture Sketches. *Science of Computer Programming*, 63(2):186-201, Dec. 2006.

[4] A. Shinnar, D. Tarditi, M. Plesko, and B. Steensgaard. Integrating Support for Undo with Exception Handling. MSR-TR-2004-140, Microsoft Research, Dec. 2004.

[5] I. Watson, C. Kirkham, and M. Luján. A Study of a Transactional Parallel Routing Algorithm. In *Proc. of the 16th Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 388-400, Brasov, Romania, Sept. 2007.