2008
YEAR

PRESENTATION

The University of Auckland | New Zealand

Computer Science 703

Advance Computer Architecture

2008 Semester 1

Lecture Notes for 6May08

Herlihy/Moss: Transactional Memory

James Goodman

Test

- Tuesday, 13May, in-class
- Coverage: through STM
- Open book, notes
  - Calculators allowed (but not needed)
  - No communication devices

Lecture Time Change!

- After next week, no lectures at 3pm on most Tuesdays/Thursdays due to conflict.
- There will be additional lectures on *Mondays,* including the 19th & 26th of May and the 2nd of June.

Background

Precursor: T. F. Knight, "An architecture for mostly functional languages," In *Proc. ACM Lisp and Functional Programming Conference*, pp. 105–112 Aug. 1986.)

"Knight describes a hardware system to parallelize a single thread program speculatively, and to execute it on a multiprocessor system. A compiler divides a program into a series of code blocks called transactions. For doing the division, the compiler assumes that these transactions do not have memory dependencies. These blocks then execute optimistically on the processors.

"The hardware enforces correct execution and uses caches to detect when a memory dependence violation between threads occurs [27]."

--Larus/Rajwar

"Transactional Memory"

M. Herlihy and J.E.B. Moss, Transactional Memory: Architectural Support for Lock-Free Data Structures, *Proc. International Symposium on Computer Architecture (ISCA-93),* ACM Press, 1993, pp. 289-300. This paper (Herlihy/Moss) coined the term "Transactional Memory"

> "Our transactions satisfy the same formal serializability and atomicity properties as database-style transactions …, but they are intended to be used very differently. Unlike database transactions, our transactions are short-lived activities that access a relatively small number of memory locations in primary memory. The ideal size and duration of transactions are implementation-dependent, but, roughly speaking, a transaction should be able to run to completion within a single scheduling quantum, and the number of locations accessed should not exceed an architecturally specified limit."

Lock-free data structures do not require mutual exclusion. They avoid common problems:

1. priority inversion
2. convoying
3. deadlock

Justification of HTM

"Experimental evidence suggests that in the absence of inversion, convoying, or deadlock, software implementations of lockfree data structures often do not perform as well as their locking-based counterparts.

"A *transaction* is a finite sequence of machine instructions, executed by a single process, satisfying the following properties:

> "Serializability: Transactions appear to execute serially, meaning that the steps of one transaction never appear to be interleaved with the steps of another. Committed transactions are never observed by different processors to execute in different orders.
>
> "Atomicity: Each transaction makes a sequence of tentative changes to shared memory. When the transaction completes, it either commits, making its changes visible to other processes (effectively) instantaneously, or it aborts, causing its changes to be discarded."

No mention of isolation!

New Instructions

"We assume here that a process executes only one transaction at a time. Although the model could be extended to permit overlapping or logically nested transactions, we have seen no examples where they are needed."

Memory Instructions

- *Load-transactional* (LT) reads the value of a shared memory location into a private register.
- *Load-transactional-exclusive* (LTX) reads the value of a shared memory location into a private register, "hinting" that the location is likely to be updated.
- *Store-transactional* (ST) tentatively writes a value from a private register to a shared memory location. This new value does not become visible to other processors until the transaction successfully commits (see below).

Defines **read set** as locations read by LT, **write set** as locations read by LTX, **data set** as union.

Manipulation instructions:

- *Commit*: make changes permanent (visible). Indicates success/failure
- *Abort*: discards all updates to the write set.
- *Validate*: TRUE indicates transaction has not (yet) aborted.

No *Begin Transaction* operation

Hardware detecting conflict may cause spontaneous abort, but no immediate indication to software.

Transactional & non-Transactional Operations

Two caveats:

(From TR): For brevity, we have chosen not to specify how transactional and non-transactional operations interact when applied concurrently to the same location. We expect that such a conflict is almost always an error. One reasonable choice is to abort a transaction when a non-transactional operation tries to revoke its ownership. Another choice is to signal some kind of error condition.

"We also leave unspecified the precise circumstances that will cause a transaction to abort. In particular, implementations are free to abort transactions in response to certain interrupts (such as page faults, quantum expiration, etc.), context switches, or to avoid or resolve serialization conflicts."

Replace critical section with:

1. Use LT or LTX to read from a set of locations,
2. Use VALIDATE to check that the values read are consistent,
3. Use ST to modify a set of locations, and
4. Use COMMIT to make the changes permanent. If either the VALIDATE or the COMMIT fails, the process returns to Step (1).

"A more complex transaction, such as one that chains down a linked list (see Figure 3), would alternate LT and VALIDATE instructions. When contention is high, programmers are advised to apply adaptive backoff [3, 28] before retrying."

Observations

- No notion of "Begin Transaction"
- Can abort transaction explicitly, but didn't have concept of jumping/trapping on abort; had to do this explicitly. Presumably the technique is:
  1. Preload value into register from memory
  2. Validate()
  3. Use value in register

The Transactional Cache

"The idea is that the transactional cache holds all the tentative writes, without propagating them to other processors or to main memory unless the transaction commits. *If the transaction aborts, the lines holding tentative writes are dropped (invalidated); if the transaction commits, the lines may then be snooped by other processors, written back to memory upon replacement, etc.* We assume that since the transactional cache is small and fully associative it is practical to use parallel logic to handle abort or commit in a single cache cycle."

Detection Invalidation

"The VALIDATE instruction is motivated by considerations of software engineering. A set of values in memory is inconsistent if it could not have been produced by any serial execution of transactions. An *orphan* is a transaction that continues to execute after it has been aborted (i.e., after another committed transaction has updated its read set). It is impractical to guarantee that every orphan will observe a consistent read set. Although an orphan transaction will never commit, it may be difficult to ensure that an orphan, when confronted with unexpected input, does not store into out-of-range locations, divide by zero, or perform some other illegal action. All values read before a successful VALIDATE are guaranteed to be consistent. Of course, VALIDATE is not always needed, but it simplifies the writing of correct transactions and improves performance by eliminating the need for ad-hoc checks."

Deadlock

"The implementation described here aborts any transaction that tries to revoke access of a transactional entry from another active transaction. This strategy is attractive if one assumes (as we do) that timer (or other) interrupts will abort a stalled transaction after a fixed duration, so there is no danger of a transaction holding resources for too long."

From Tech Report from same authors:

"Deadlock (cyclic waiting) is impossible in this implementation because transactions never wait for one another. A high-priority transaction cannot be delayed indefinitely by a lower-priority transaction, because the latter will be aborted by a timer interrupt if it runs too long. Starvation, however, is still possible. We believe that the best way to avoid starvation is to advise programmers to adopt an adaptive backoff strategy: a transaction that repeatedly aborts should wait for some duration before retrying.

"We originally considered incorporating a backoff strategy in the cache coherence protocol itself. Our simulations, however, show that backoff schemes need to be tuned to perform well, and so hardware backoff seems overly inflexible. Anderson [Anderson, 1990] reports a similar experience in alleviating contention for spin locks: exponential backoff works well, but the parameters must be chosen carefully."

Practical Implications

1. "[F]or programs to be written in a uniform and portable manner, one needs to guarantee at the instruction set architecture level the minimum transaction size that the architecture supports. At present we do not have a good feel for what such a size might be, but it should probably be between 10 and 100. Since one might not want to put a fully associative cache of this size into every implementation of the architecture, schemes that use some hardware but handle larger transactions via software traps seem to be desirable. In fact, one can avoid hard limits on transaction size by offering the software overflow mechanism with all implementations." [TR]

2. "For programs to be portable, the instruction set architecture must guarantee a minimum transaction size, thus establishing a lower bound for the transactional cache size. An alternative approach is suggested by the LimitLESS directory-based cache coherence scheme of Chaiken, Kubiatowicz, and Agarwal [6]. This scheme uses a fast, fixed size hardware implementation for directories. If a directory overflows, the protocol traps into software, and the software emulates a larger directory. A similar approach might be used to respond to transactional cache overflow. Whenever the transactional cache becomes full, it traps into software and emulates a larger transactional cache. This approach has many of the same advantages as the original LimitLESS scheme: the common case is handled in hardware, and the exceptional case in software." [ISCA93]

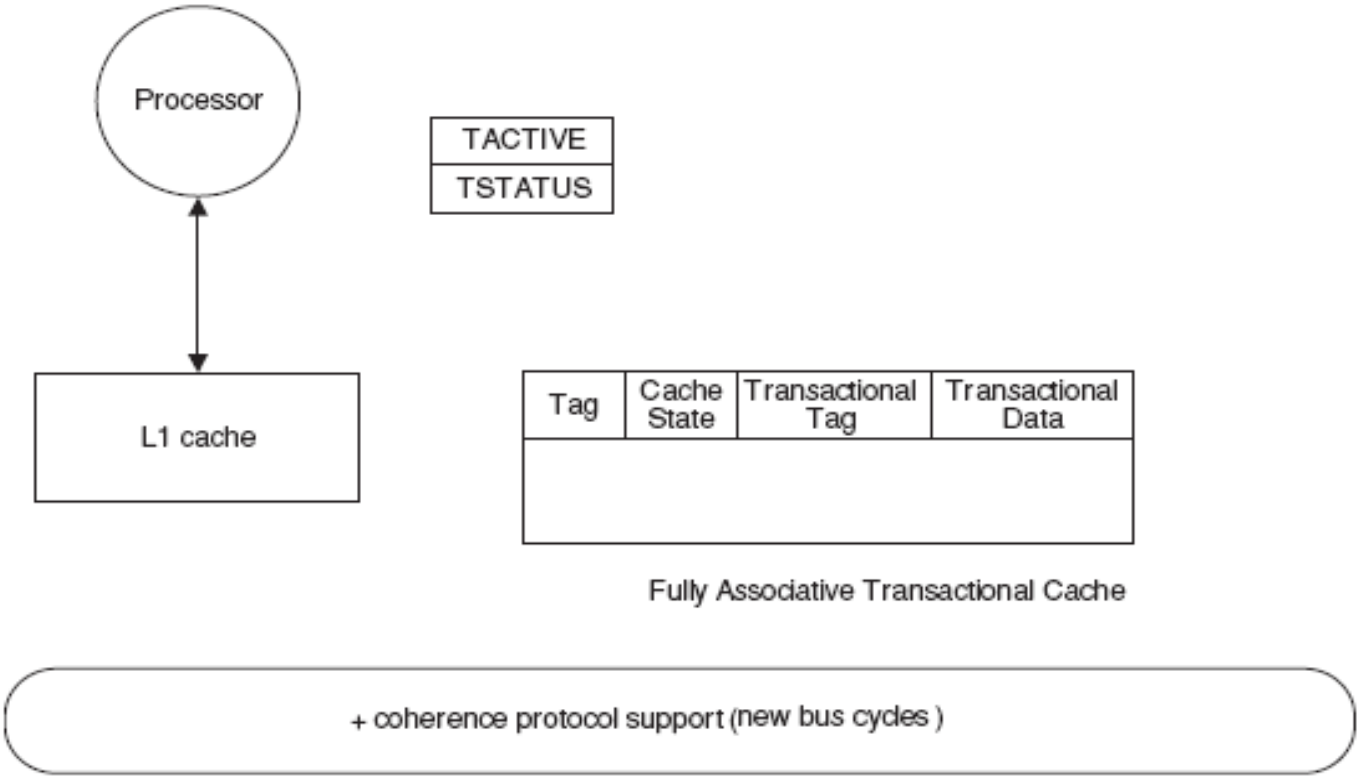

FIGURE 4.7: Herlihy and Moss Ttransactional Mmemory support

Implementation

- The *transactional cache* is a fully associative cache that holds all transactional writes without propagating their values to other processors or to main memory until the transaction commits. The transactional cache has additional tags with each line that add special meaning to the regular cache states. If tag is **empty**, the line has no data. If tag is **normal**, the line has committed data. An **xcommit** tag means the contents must be discarded on commit, and an **xabort** tag means the contents must be discarded on an abort.

- The cache coherence protocol is augmented by three new bus cycles. The **t_read** bus cycle is for a transactional read request that goes across the bus. This request can be refused (NACK) by a **busy** cycle. The **t_rfo** bus cycle is for a transactional read-for-exclusive request that goes across the bus. This can be refused (NACK) by a busy cycle. The **busy** bus cycle prevents too many transactions from aborting one another too often. This approach may starve some transactions but a queuing mechanism can address starvation. A **busy** response does not cause the transaction execution itself to abort immediately but records hardware state to allow the transaction to check for whether the transaction has aborted from the hardware's perspective. Until this check, the transaction may continue to execute without aborting."

  --Larus/Rajwar