Hardware-Supported Optimistic Transactions

Jeremy H. Brown & Josie Ammer
(jhbrown@ai.mit.edu & josie@ai.mit.edu)

The Problem: We want to provide optimistic transactions as a programming language construct. However, transactions are expensive to implement; locking, logging, and maintaining the information needed to abort all incur significant overhead. Optimistic transactions, while potentially exploiting parallelism not detectable at compile-time, incur even greater overhead due to the bookkeeping necessary to detect dependency failures. Our solution is to decrease the overhead associated with these operations by implementing them in custom hardware.

Motivation: Transactions, and especially optimistic transactions, provide several powerful abilities generally unavailable in modern programming languages. First, in an environment with parallel operations, a transaction provides a simple means of defining an atomic operation; atomic operations aren't common in programming languages, however – the programmer who wants an atomic operation is typically left to implement it using locks and semaphores. Second, transactions provide the ability to abort, or completely undo, a partially completed operation; abort operations are entirely absent from popular programming languages. Third, optimistic transactions provide a means of exploiting parallelism hidden in sequential operations, even when compile-time analysis cannot prove the independence of the operations.

Previous Work: In [1], Knight proposes a scheme for optimistic parallel execution on a shared memory multiprocessor. Dependency checking is integrated into the cache coherence mechanism; all speculative writes must be kept in the processor-local caches, rather than written to main memory. When an optimistic computation discovers that one of its dependencies has been modified, it aborts and restarts. Knight's focus is on basic-block-level parallelism, so it is not a major problem that optimistic computations can write only a limited amount of data into the write-buffers before having to wait for older transactions to complete.

The Hydra [2] architecture is an implementation of cache-based optimistic parallel execution; four MIPS processors attempt to execute code sequences in parallel. Hydra relies on a compiler to mark potentially parallelism; currently, multiple loop iterations are attempted in parallel, and procedure call parameters are evaluated in parallel with the procedure itself using guesses as to the parameters' final values to enable forward progress. Unfortunately, [2] reports disappointing speedups on programs from the Spec95 benchmark suite.

The notion of using speculation-supporting hardware to perform explicit transactions is explored in neither [1] nor [2].

Approach: We are taking a two-pronged approach to the problem. First, we are designing a programming language in which optimism and transactions are explicit primitive constructs; this saves the compiler from having to perform elaborate analyses. Accompanying profiling tools will assist programmers in applying optimism successfully.

Second, we believe that optimistic transactions must be of coarse granularity for the benefits of the extracted parallelism to outweigh the bookkeeping overhead; in particular, transactions may need to write a great deal of data before completing. Thus, we are designing a memory system in which the main memory supports optimistic transactions by maintaining multiple "timestamped" versions of memory on a row basis, and by dynamically detecting dependency violations.

The main memory consists of pairs of DRAM banks: a "committed" bank, and a "speculative" bank. Each optimistic transaction is assigned a virtual time; memory accesses include the virtual timestamp of the transaction that issued them. Committed rows reside at their specified address in the committed DRAM bank. A tag on row A in the committed bank points to the least-speculative version of that row which resides at an arbitrary location in the speculative bank. A tag on the speculative row points to the next least speculative version of the row. Subsequent tags continue the ordered, linked-list. Tags are maintained in SRAM to allow fast linked-list traversal. Additional information in each tag specifies the virtual timestamp of its row, and tracks which words in the row have been written and read.

A read returns the most-speculative version of a row earlier than the read's timestamp; also, the accessed word in the speculative row is tagged as read. A write updates the word in the version of a row with the same timestamp as the write; if no such row exists, the write operation creates a new entry in the linked list which inherits the word-read bits of its predecessor in the list. If the write modifies a word marked as read, an exception is raised which is handled by software, which aborts all transactions later than the time of the write. This approach is conservative, aborting more transactions than sometimes necessary, but it has very small bookkeeping requirements.

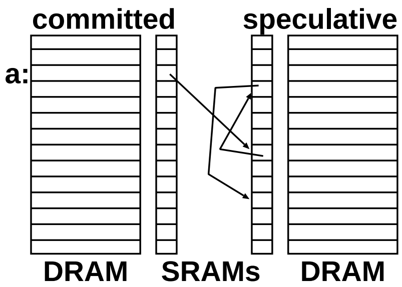
In the event that a write needs an additional row in the speculative bank, but no rows are available, an exception is raised; software aborts the most speculative transaction to free its rows in the speculative bank.

The overall advantage of this approach to transactional memory is that the overhead of speculation grows linearly with speculativeness (i.e. with length of linked-list traversal necessary to get to accessed data), so transactions more likely to commit (those with earlier virtual timestamps) are less-encumbered. Programs without transactions operate at full-speed into "committed" memory.

Difficulty: We are proposing to custom-design a complex memory system based on a mix of SRAM and DRAM, along with the processor and bus protocols to support time-stamped accesses. We expect the design to be quite difficult, and anticipate the actual construction of the entire system to be a major effort requiring significant investments of time and money.

Impact: Because the hardware, operating system, and programming language cooperate to provide efficient and accurate optimistic transactions, the programmer need not deal in the error-prone details of locks and semaphores to create transactional semantics. Furthermore, because of the novel memory architecture and circuitry, optimistic transactions will run with very little slowdown over normal code.

Our system will have a particularly dramatic impact on dynamic transaction-processing applications; most transactions will be independent and thus parallelizable using optimism, and those few which have interdependencies will nevertheless run correctly.



Future Work: Nearly the entirety of this project remains as future work. The next few steps are developing a software simulation of the transaction-aware memory system, bus, and processor, and developing programming idioms for using the transactional memory features. The layout, fabrication, and assembly of custom hardware remains quite far in the future, although accelerated simulations will become possible within the next year using a custom board being designed by other members of our group.

Research Support: This material is based upon work supported under a National Science Foundation Graduate Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. Support for this research was provided in part by the Defense Advanced Research Projects Agency under Rome Labs Contract Number F30602-98-1-0172.

References:

[1] T. Knight. An Architecture for Mostly Functional Languages. *Proceedings of the ACM Lisp and Functional Programming Conference*, pp. 105-112, August 1986.

[2] L .Hammond, M. Willey and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. *ACM SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*, Vol.33, No.11, pp. 58-69, November 1998.