Alert-on-Update: A Communication Aid for Shared Memory Multiprocessors *

Michael F. Spear Arrvindh Shriraman Hemayet Hossain
Sandhya Dwarkadas Michael L. Scott
Computer Science Dept., University of Rochester
{spear,ashriram,hossain,sandhya,scott}@cs.rochester.edu

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

General Terms Design, Experimentation, Performance

Keywords transactional memory, communication, coherence, events

1. Introduction

A traditional advantage of shared-memory multiprocessors is their ability to support very fast *implicit* communication: if thread A modifies location D, thread B will see the change as soon as it tries to read D; no explicit *receive* is required. There are times, however, when B needs to know of A's action *immediately*. Event-based programs and condition synchronization are obvious examples, but there are many others. Consider a program in which B reads V from D, computes a new value V' (a potentially time-consuming task), and uses *compare-and-swap* to install V' in D only if no other thread has completed an intervening update. If some thread A *has* completed an update, then all of B's work subsequent to that update will be wasted.

More significantly, suppose we generalize the notion of atomic update to implement *software transactional memory* (STM) [4]. Now A may not only force B to back out (abort) and retry, it may also allow B to read mutually inconsistent values from different locations. If B does not learn of A's action immediately, these inconsistent values (which should not logically be visible together) may cause B to perform erroneous operations that cannot be undone [5]. STM systems typically avoid such errors by performing *incremental validation* prior to every potentially dangerous operation—in effect, they poll for conflicts. Since validation effort is proportional to the number of objects read so far, the *total* cost is quadratic in the number of objects read, and may cripple performance [5].

Interprocessor interrupts are the standard alternative to polling in shared-memory multiprocessors, but they are typically triggered by the operating system and have prohibitive latencies. This cost is truly unfortunate, since most of the infrastructure necessary to inform remote threads of a change to location L is already present in the cache coherence protocol. What is missing is a way to reflect write notices up to user-level code.

We propose a simple technique to selectively expose cache events to user programs. Using our technique, threads register an *alert handler* and then selectively mark lines of interest as *alert-on-update* (AOU). When a cache controller detects a remote write, a replacement, or (optionally) a local write of a marked line, it notifies the local processor, effecting a spontaneous subroutine call to the current thread's alert handler. AOU can be used to implement fast mutexes, fast rollback in nonblocking algorithms, hazard pointers [2], debugger watchpoints, active messages [6], and various forms of code security. This paper focuses on STM.

2. Implementing AOU

Alert-on-update can be implemented on top of any cache coherence protocol: coherence requires, by definition, that a controller be notified when the data cached in a local line is written elsewhere in the machine. The controller also knows of conflict and capacity evictions. AOU simply alerts the processor of these events when they occur on marked lines. The alert includes the address of the affected line and the nature of the event. Our simulator employs an invalidation-based MESI snooping protocol.

An application enables AOU by using an instruction to register a handler. It then indicates its interest in individual lines using `aload` (alert-load) instructions. One variety of `aload` requests alerts on remote writes only; another requests alerts on any write, including local ones. Our STM experiments use the first variety only. Both varieties also generate alerts on capacity and conflict misses. Hardware provides `set_handler`, `aload`, `clear_handler`, and `arelease` instructions; special registers to hold the handler address, the PC at the time an alert is received, and a description of the alert; an interrupt vector table entry for alerts received while in kernel mode; and two bits per cache line to indicate interest in local and/or remote writes. For simplicity, we implement the alert bits in a non-shared level of the memory hierarchy (the L1 cache). If the L1 were shared we would need separate bits for each local core or thread context.

Perhaps the most obvious way to deliver a notification is with an interrupt, which the OS can transform into a user-level signal. The overhead of signal handling, however, makes this option unattractive. We propose, therefore, that interrupts be used only when the processor is already running in kernel mode. If the processor is running in user mode, an alert takes the form of a spontaneous, hardware-effected subroutine call.

We make no assumptions about the calling conventions of AOU-based code. Typically, our alert handlers consist of a subroutine bracketed by code reminiscent of a signal trampoline. Additional alerts are masked until the handler re-enables them. A status register allows the handler to determine whether masked alerts led to lost information. Capacity/conflict alerts, likewise, in-

* This work was supported in part by NSF grants CCR-0204344, CNS-0411127, CNS-0509270, and CNS-0615139; an IBM Faculty Partnership Award; financial and equipment support from Sun Microsystems Laboratories; and financial support from Intel and Microsoft.

PPoPP 2007, San Jose, CA

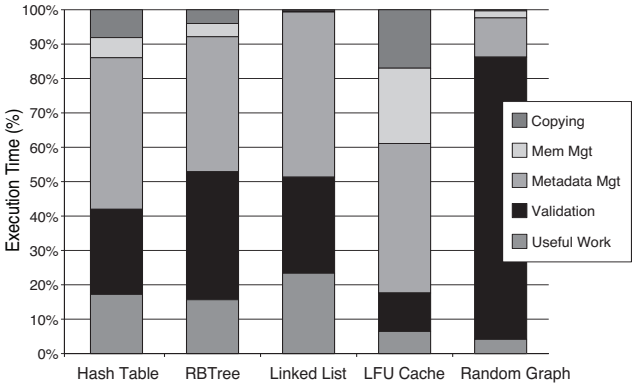

Figure 1. Single-thread overheads in RSTM; Validation and metadata management dominate even in the absence of contention.

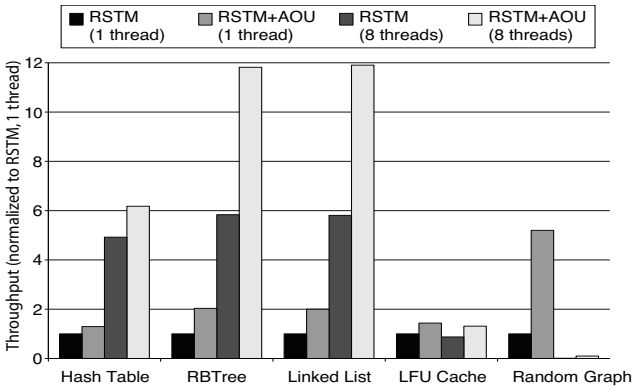

Figure 2. Speedup of RSTM using AOU. All results are normalized to RSTM, 1 thread.

dicate that the local cache has lost the ability to precisely track a line. Moreover, exclusive loads and prefetches, failed *compare-and-swap*s, silent stores, and false sharing within lines may all lead to "spurious" alerts. For these reasons user software must treat alerts conservatively: any change to a marked line will result in an alert, but not all alerts provide precise information or correspond to interesting changes. We expect the overhead of spurious alerts to be significantly less than that of fruitless polling.

We assume that all `aloade`d lines belong to the active thread. Thread and process schedulers must execute `clear_handler` (which also `arelease`s all `aloade`d cache lines) on context switches, and must permit each thread or process to perform a conservative polling operation and to re-`aload` lines at the beginning of each quantum. It is *not* necessary to execute `clear_handler` when returning from the kernel to the most recently executing user thread, and thus simple system calls and many common interrupts will not force conservative polling. If an event occurs on an `aloade`d line while the processor is executing in the kernel, the alert will be delivered as an interrupt. The interrupt handler simply remembers the contents of the special registers, and if control eventually returns to user space without a context switch, the kernel effects a deferred call to the user-level handler. Moreover kernel routines themselves can use alert-on-update, so long as the interrupt handler is able to distinguish between user and kernel lines, and so long as all the kernel lines are `arelease`d before returning to user space.

3. AOU for STM

Figure 1 shows a single-thread execution-time breakdown for five benchmarks running under our RSTM system [1]. Incremental validation accounts for an average of 37% (up to 82%) of run time, with additional time devoted to metadata management (bookkeeping), some of which serves to track the objects for which validation may be needed. AOU can eliminate much of this overhead.

Our AOU-optimized version of RSTM maintains an estimate, `alimit`, of the number of object headers (metadata structures) that can be kept in the cache in AOU mode. Subject to this limit, our system `aload`s headers at *open* time. Additional headers are opened in the usual way, and added to a transaction's *read list* of objects that are validated incrementally on future *open*s. For up to `alimit` objects, bookkeeping and validation costs are eliminated.

On any alert or context switch our handler marks the current transaction as aborted. For capacity/conflict eviction, the handler reduces `alimit` for future transactions. (We increase `alimit` occasionally at the beginning of transactions, to exploit any reduction in cache pressure.) The handler then returns, allowing an aborted transaction to continue to the next validation point (this avoids deferring alerts during memory management and other library calls). On every *open* and at commit time we check to see if we have been aborted. We also incrementally validate any objects that are on our read list because we *open*ed more than `alimit` of them. As long as all headers fit in the cache, validation becomes a constant-time operation, and there is no overhead for read list management.

Though we do not currently do so, it is possible to `aload` transaction descriptors, causing a transaction to receive an alert immediately when it is aborted by a competitor. Such *immediate aborts* could avoid significant useless work prior to the next *open* or *commit*. In related work, we have used `aload` to create an STM that allows transactions to modify objects in place without losing nonblocking semantics. Most other nonblocking systems use an extra level of indirection to *replace* modified objects.

Figure 2 shows normalized throughput at 1 and 8 threads for several microbenchmarks, comparing RSTM with and without `AOU`. We used the GEMS/Simics simulation infrastructure from the Multifacet group at the University of Wisconsin–Madison to simulate a 16-way CMP. All experiments were run with eager acquire and the Polka contention manager [3]. We observe speedup of as much as $5\times$, and see that AOU-enabled RSTM scales as well, if not better than, unmodified RSTM.[1]

References

[1] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the Overhead of Software Transactional Memory. In *ACM SIGPLAN Workshop on Transactional Computing*, Ottawa, ON, Canada, June 2006.

[2] M. M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. on Parallel and Distributed Systems*, 15(8), Aug. 2004.

[3] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proc. of the 24th ACM Symp. on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[4] N. Shavit and D. Touitou. Software Transactional Memory. *Distributed Computing*, 10(2):99-116, Feb. 1997.

[5] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict Detection and Validation Strategies for Software Transactional Memory. In *Proc. of the 20th Intl. Symp. on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[6] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Intl. Symp. on Computer Architecture*, pages 256-266, Gold Coast, Australia, May 1992.

[1] RandomGraph livelocks on multiple threads due to our use of eager acquire. We have not yet evaluated RSTM+AOU with lazy acquire.