Brief Announcement: Transactional Memory Retry Mechanisms

Michael F. Spear, Andrew Sveikauskas, and Michael L. Scott
Department of Computer Science, University of Rochester*
{spear, asveikau, scott}@cs.rochester.edu

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

General Terms: algorithms, design, performance

Keywords: transactional memory, condition synchronization, Bloom filters

1. INTRODUCTION

Software TM systems typically support condition synchronization through a `retry` mechanism [2]. Using `retry`, a transaction explicitly self aborts and deschedules itself when it detects that a precondition for its operation does not hold. The runtime then tracks the set of locations read by the `retry`er, and refrains from rescheduling it until at least one location in the set has been modified by another transaction.

When a transaction T calls `retry` after reading locations $\{l_1 \dots l_r\}$, the standard implementation modifies the metadata of each location l_i to indicate that any transaction that subsequently writes l_i must wake T. After marking all such locations, T re-checks its validity (to avoid a timing window) and yields the processor. Though elegant and simple, this implementation has several potential drawbacks. First, explicitly marking each location l_i requires exclusive access to l_i's metadata in the cache, in a manner analogous to "visible reader" conflict detection. Previous work suggests that the invalidation of lines in concurrent readers may have a substantial performance cost [3]. Second, when garbage collection is unavailable, tracking locations for retrying transactions appears to prevent the reclamation of *any* shared data whenever a transaction is in the `retry` state. Third, for hardware or "best-effort" TM [1], software-based `retry` does not easily virtualize: registration as a "visible reader" appears to require that a thread re-execute its entire transaction in software *before* it can yield the processor.

We have developed a retry mechanism based on Bloom filters that is orthogonal to the TM implementation. Our retry avoids the pitfalls outlined above, but serializes writer transactions after their commit point when there are retrying transactions. Implementation details and evaluation of our mechanism are available in a technical report [4].

*This work was supported in part by NSF grants CNS-0411127, CNS-0615139, CCF-0702505, and CSR-0720796; and by financial support from Intel and Microsoft.

Copyright is held by the author/owner(s).
PODC'08, August 18–21, 2008, Toronto, Ontario, Canada.
ACM 978-1-59593-989-0/08/08.

2. RETRY WITH BLOOM FILTERS

We maintain a global set of retrying transactions, each represented by a Bloom filter and a handle object (e.g., a semaphore) for wakeup. To retry, transaction T constructs a Bloom filter representing the locations it has read and adds this filter, together with its handle, to the global set. T then re-validates its read set. If validation fails, T removes its filter from the global set and restarts; otherwise, it yields the CPU and awaits notification that it can resume. When T wakes, it removes its filter from the global set and restarts. Transactions in a hardware or hybrid TM can use this mechanism easily, provided that the code for `retry` is able to obtain the current transaction's read set.

When committing, a writer W must wake any `retry`er whose read set intersects its write set. It does so by inspecting the Bloom filters in the global set. When W has k writes, it can incur up to $O(k)$ overhead per `retry`er. This overhead can be avoided (at the expense of spurious wakeups) if W creates a filter of its writes and compares it to each posted read set via constant-time intersection.

Our current results [4] (for software TM) are mixed: both Bloom-filter and (optimized) visible-reader retry outperform naive sleep, but neither is consistently better than the other. We are hopeful that further experience with transactional workloads will clarify the tradeoff, or identify application characteristics that favor a particular implementation. Our current implementations and microbenchmarks are available as a patch to RSTM [5].

3. REFERENCES

[1] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid Transactional Memory. *12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 2006.

[2] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable Memory Transactions. *10th ACM SIGPLAN 2006 Symp. on Principles and Practice of Parallel Programming*, pages 48–60, Chicago, IL, June 2005.

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

[4] M. F. Spear, A. Sveikauskas, and M. L. Scott. Transactional Memory Retry Mechanisms. TR 935, Dept. of Computer Science, Univ. of Rochester, June 2008.

[5] Univ. of Rochester. Rochester Software Transactional Memory. *http://www.cs.rochester.edu/research/synchronization/rstm/.*