Robust Multi-threading

Jeff Napper Lorenzo Alvisi

Department of Computer Sciences
The University of Texas at Austin
{jmn,lorenzo}@cs.utexas.edu

1. Introduction

Networked services are increasingly in demand as traditional services become web-enabled. These high-throughput services today use a pool of worker threads to service requests in parallel on multi-processor machines [15, 8]. As businesses streamline practices, these services are often critical (e.g., web services for an online retailer or just-in-time supply chaining), requiring methods to manage faults.

The relatively little attention devoted so far to improving the robustness of these applications has focused on fault tolerance techniques—such as rollback-recovery [2] and state machine replication [12]—that can be added transparently to existing applications. Unfortunately, conjugating transparency with practicality has proven especially difficult in these applications. The proposed recovery-based techniques have not moved beyond the design stage, while implementations based on state machine replication require strong assumptions about the interface used to access shared data that are often violated by real programs. Indeed, it is precisely the lack of a suitable consistent, widespread interface in use for accessing shared data and resources that complicates transparent approaches. Without a clean, concise interface to support transparency (e.g., network send/receive operations for message passing applications), an implementation may be forced to use a lower level interface not well-suited to fault-tolerance (e.g., thread scheduling in [12]). Existing interfaces to shared data provide synchronization without addressing failures. We instead propose an interface that is inherently well-suited to the implementation of robust applications because it incorporates fault-tolerance as a first class citizen: multi-object transactional operations.

Using a transactional interface to shared data, a thread may fail safely at any time during execution without inhibiting the rest of the application. To ensure an application can still make progress under these conditions, we provide lock-free access to shared data. Failed threads may then be treated independently, either be restarted, recovered using rollback-recovery techniques, or simply *ignored*, depending on the application's needs.

For example, when requests are independent, clients could simply retry failed requests. Recovery of a failed thread would typically be unnecessary, although thread failures may trigger the creation of new threads to maintain throughput. Note that to ensure highly available services, the system should recover resources from failed and stalled threads to prevent resource leaks.

Admittedly, it is not enough that correct threads have unfettered access to shared data and available resources—under some failure models we need to also prevent failed threads from corrupting both shared data and another thread's private data (e.g., by corrupted pointer dereferencing or Byzantine failures). This remains an intriguing area of future research. For example, it may be possible to design systems where threads can be failed *intentionally*—to recover runaway resources, to rejuvenate the system, or to terminate threads that violate security constraints—while other threads continue to service requests.

In the rest of this document, we focus on the implementation of lock-free data structures. In Section 3 we present our approach, but first we put our work in the larger research context in the next section.

2. Background

Wait-free synchronization [5] offers an elegant approach to handling threads that fail or are delayed. It advocates building wait-free implementations of concurrent objects, i.e. implementatations that guarantee that any thread can complete any operation in a finite number of steps, irrespective of the execution speed of other threads. Unfortunately, eliminating starvation has proved in general difficult to implement efficiently [5, 3], thereby motivating the introduction of weaker progress guarantees. *Lock-free* (or nonblocking) concurrent objects [5] guarantee progress to at least one thread, allowing starvation but not livelock; *obstruction-free* concurrent objects [6] guarantee progress only in the absence of contention, thus allowing both starvation and livelock.

While many have devised specific wait-free and lock-free implementations for individual data structures, a general and efficient approach applicable to any data structure is clearly more desirable. Herlihy [5] has shown that compare-and-swap (CAS), an operation that atomically and conditionally modifies a memory location, is *universal*—sufficient to provide wait-free implementations of arbitrary data structures. The CAS primitive and the similar LL/SC are the only widely available universal primitives. Researchers have extended CAS to provide an efficient multi-word CAS (MWCAS) [4]. MWCAS greatly simplifies the task of implementing wait-free and lock-free data structures compared to simple CAS, but maintains the same interface. Transactional memory has also been proposed as a different interface to provide nonblocking access [13]. Unfortunately, implementations often i) require static transactions (i.e., memory locations are predetermined) [13], ii) cannot return early indication of failure, or iii) require a different API for read-only transactions [7].

3. Lock-Free Transactional Objects

Neither MWCAS nor transactional memory provide the best interface for developers. Though MWCAS is a useful primitive, developers using MWCAS must address the low-level details of concurrent updates to the memory representations of data structures and keep track of expected values for conditional update. The API provided by transactional memory is typically easier than MWCAS for the developer, but the approach yields a host of new problems and still does not avoid the fundamental problem of MWCAS—programmers must focus on memory layouts of objects rather than operations thereupon.

We propose a new approach—lock-free transactional *object* operations. Rather than requiring attention to objects' memory representations, we provide a transactional framework that ensures that lock-free multi-object operations occur in a serializable order. Our framework restricts neither the representations of the objects nor the operations that the objects support. For example, our framework supports both copying an entire object or saving only modified portions of the object during an operation, allowing developers to optimize their implementation according to the size of the object.

The many drawbacks of transactional memory disappear: our framework i) allows dynamic transactions (where operations are not predetermined) ii) returns early failure indication, and iii) does not use a special API for read-only transactions. Finally, our algorithm is disjoint-access parallel [9] for multi-object updates, relies only on CAS as a hardware primitive, and supports an implementation compatible with current nonblocking garbage collectors [10].

Our system models a pool of cooperating threads that operate on shared data. Shared data is represented as a set of objects that support read or write operations. Threads may run simultaneously (as on a multi-processor) and there are no bounds on relative speeds between threads. A thread may fail by halting or be arbitrarily delayed (e.g., by a page fault). Threads execute a transaction on a set of objects. We consider only a single transaction per thread and do not address nested transactions. Our transactions follow the *restricted model* [14] in which a transaction has a beginning, performs one or more read operations, and may attempt write operations on objects that it has read previously. A thread then either attempts to *commit* or *abort* the transaction so that all operations are either visible or not, respectively. Until a transaction commits or aborts, it is *pending*. Operations from different transactions may be interleaved during an execution provided that the execution is 1-*copy serializable* (1-SR) [1]—equivalent to an execution in which transactions execute sequentially on a single copy of the objects.

For our applications of interest (e.g., web services), read-only transactions can significantly outnumber update transactions—hence, we take special care in allowing read-only transactions to proceed concurrently with writes. Borrowing from multiversion databases, we keep several versions of an object so that reads can proceed in parallel as new versions are written. Our multiversion approach also allows a thread to keep all writes to its particular version local, reducing cache contention on multi-processor machines.

We keep versions in a lock-free queue [11], but a thread performing a transaction keeps all reads and writes locally. When a thread attempts to commit the transaction it (i) adds write operations to the object version queues, (ii) checks for violations of 1-SR, and iii) commits the transaction if consistent. Read-only transactions do not need a special API. They simply do not enqueue any updates and thus also cannot interfere with write operations. The lock-free queues guarantee a thread can access particular versions and can always enqueue a new operation. A thread can help commit any transaction, including transactions that it did not initiate, provided that a write of the transaction has been enqueued. A thread attempting to commit a transaction only helps another transaction if both transactions write to the same object and the other transaction's operation is enqueued earlier.

Transactions are allowed to commit if the execution is 1-copy serializable using a multiversion serialization graph (MVSG) representing transaction dependencies: Bernstein and Goodman show that the execution is 1-SR iff the MVSG is acyclic. Keeping track of the multiversion serialization graph dynamically as transactions execute enables many of the desirable features of our framework, including dynamic transactions (where operations do not need to be predeclared), and early failure indication. By checking for 1-SR consistency at every operation, at commit a thread need only check that write operations were enqueued in the desired order (otherwise conflicting transactions may be forced to abort to ensure 1-SR). Finally, our algorithm cleanly garbage collects old versions, though the specifics of our approach are beyond the scope of this paper.

4. Conclusions

We believe that robust multi-threaded applications are becoming increasingly important as networked services face increased demand and require multi-threading to maintain throughput on multi-processor machines. Our approach to improve robustness is to create an environment that can tolerate thread failures, be they accidental or intentional, without affecting the safety properties of the application and with minimal disruption to its liveness properties. This paper outlines the first step in this direction, lock-free object transactions.

References

[1] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, 1983.

[2] O. P. Damani, A. Tarafdar, and V. K. Garg. Optimistic recovery in multi-threaded distributed systems. In *Proc. IEEE SRDS*, pages 234–243, October 1999.

[3] M. Greenwald. *Non-Blocking Synchronization and System Design*. PhD thesis, Standford University, 1999.

[4] T. Harris, K. Fraser, and I. Pratt. A practical multi-word compare-and-swap operation. In *Proc. IEEE DISC*, pages 265–279, 2002.

[5] M. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, 1991.

[6] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. IEEE ICDCS*, pages 522–529, 2003.

[7] M. Herlihy, V. Luchangco, M. Moir, and W. N. S. III. Software transactional memory for dynamic-sized data structures. In *Proc. ACM PODC*, pages 92–101, July 2003.

[8] J. Hu, I. Pyarali, and D. C. Schmidt. The object-oriented design and performance of JAWS: A high-performance web server optimized for high-speed networks. *PDCP*, 3(1), March 2000.

[9] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proc. ACM PODC*, pages 151–160. ACM Press, 1994.

[10] M. M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proc. ACM PODC*, pages 21–30, July 2002.

[11] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. ACM PODC*, pages 267–275, 1996.

[12] J. Napper, L. Alvisi, and H. Vin. A fault-tolerant java virtual machine. In *Proc. IEEE DSN*, pages 425–434, June 2003.

[13] N. Shavit and D. Touitou. Software transactional memory. In *Proc. ACM PODC*, pages 204–213, 1995.

[14] R. E. Stearns, P. M. Lewis, and D. J. Rosenkrantz. Concurrency control for database systems. In *Proc. IEEE FOCS*, pages 19–32, 1976.

[15] The Apache Software Foundation. Apache performance tuning, 2004.