PODC '05 poster abstract

Randomization in STM Contention Management *

William N. Scherer III and Michael L. Scott
Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
{scherer, scott}@cs.rochester.edu

Abstract

The obstruction-free Dynamic Software Transactional Memory (DSTM) system of Herlihy et al. allows only one writing transaction at a time to access an object. Should a second require an object currently in use, a *contention manager* must determine which may proceed and which must wait or abort.

In this case study, we consider the impact of randomization when applied to our "Karma" contention manager. Previous work has shown that Karma tends to be a good choice of managers for many applications. We analyze randomized Karma variants, using experimental results from a 16-processor SunFire machine and a variety of benchmarks. We conclude that randomizing either abortion decisions or gain can be highly effective in breaking up patterns of livelock, but that randomized backoff yields no inherent positive benefit.

1 Introduction

Although early software transactional memory systems (STMs) were primarily academic curiosities, more modern STMs [1, 2, 3] have reduced runtime overheads sufficiently to outperform coarse-grained locks (with at least moderate contention). Dynamic software transactional memory (DSTM) [3] is a practical STM system novel in its support for dynamically allocated objects and transactions, and for its use of modular contention managers to separate issues of progress and correctness in data structures.

At its heart, contention management in DSTM is the question: how do we mediate transactions' conflicting needs to access a block of memory? In previous work [4], we have shown that the choice of contention management policies dramatically affects overall system throughput and that the Karma manager frequently gives top performance.

In the present work, we explore the impact of randomization in contention manager design. We study Karma as a top contention manager with many facets that can be randomized.

*This work was supported in part by NSF grants numbers EIA-0080124, CCR-0204344, and CNS-0411127, and by financial and equipment grants from Sun Microsystems Laboratories.

2 Contention Management

The contention management interface for the DSTM [4] includes notification methods for various events that transpire during the processing of transactions, plus two request methods that ask the manager to make a decision. Notifications include events such as beginning a transaction, successfully/unsuccessfully committing a transaction, attempting to open a block, and successfully opening a block. The request methods ask a contention manager to decide whether a transaction should (re)start and whether enemy transactions should be aborted.

Many researchers have found randomization to be a powerful technique for breaking up repetitive patterns of pathological behaviors that hinder performance. We evaluate this potential by randomizing facets of the Karma manager.

2.1 The basic Karma scheme

The Karma contention manager [4] tracks the cumulative number of blocks opened by a transaction as its *priority*. It increments this priority with each block opened, and resets it to zero when a transaction commits. It does not reset priorities if the transaction was aborted; this gives a boost to a transaction's next attempt to complete. Karma manages conflict by aborting an enemy transaction when the number of times a transaction has attempted to open a block exceeds the difference in priorities between the enemy and itself. Between attempts, it backs off for a

fixed period of time. Intuitively, Karma prefers not to abort a transaction that will take a large amount of effort to redo, but tries to maintain some ability for short transactions to eventually gain enough priority to finish even when competing with longer ones.

2.2 Randomized Backoff

The original Karma scheme backs off for a fixed period of time T between attempts to acquire an object. Randomized, we instead sleep for a uniform random amount of time between 0 and $2T$.

2.3 Randomized Abortion

In response to a `shouldAbort` query, the basic Karma manager returns `true` when the difference Δ between the current and enemy transactions' accumulated priorities is less than the number of times it has attempted to open a block. We randomize this abortion decision with a sigmoid function that returns `true` with probability biased to the higher-priority transaction: $(1 + e^{-\frac{1}{2}\Delta})^{-1}$.

2.4 Randomized Gain

The basic Karma manager gains one point of priority with each object that it successfully opens. Randomized, we instead gain as priority an integer randomly selected from the uniform interval $0..200$.

3 Methodology

All results were obtained on a SunFire 6800, a cache-coherent multiprocessor with 16 1.2Ghz UltraSPARC III processors. We tested in Sun's Java 1.5 HotSpot JVM.

We present experimental results for six benchmarks. IntSet, IntSetUpgrade, and RBTree are implementations of a set of integers; LFUCache simulates web caching [4]. Stack supports push and pop transactions. ArrayCounter transactions either increment each shared counter 0..255 in an array or decrement them in the opposite order; it is a "torture test" that exacerbates any tendency towards livelock.

We implemented all eight combinations of randomizing three facets of the Karma manager. We crossed each variant and benchmark, running for a total of 10 seconds. We display throughput results for eight threads: previous experiments suggest that eight threads is enough for inter-thread contention to affect scalability in the benchmarks, yet few enough that limited scalability of the benchmarks themselves does not skew the results. Figure 1 displays throughput results for the various benchmarks.

4 Analysis

In every benchmark, some combination of randomization improves throughput. In the ArrayCounter, IntSetUpgrade, and IntSet benchmarks, randomizing just abortion decisions yields the best performance. Randomizing both abortion and backoff gives very poor performance in ArrayCounter and RBTree; yet, it improves performance for LFUCache and Stack. Randomizing gain improves performance both alone, and in every combination with other types of randomization, for LFUCache and RBTree.

4.1 Interpretation of results

Randomizing abortion is particularly helpful for the ArrayCounter, IntSet, and IntSetUpgrade benchmarks. Why is this the case? One possible explanation is that randomizing abortion decisions is very powerful for breaking up semi-deterministic livelock patterns. Such livelocking patterns are particularly visible in ArrayCounter: an increment and a decrement that start at roughly the same time are very likely to have similar or identical priorities when they meet; they are thus prone to mutual abortion.

The combination of randomizing backoff and abortion produces great variance in how long a thread waits to abort an enemy transaction. In times when this wait period is shortened, a longer enemy transaction will have less of a chance to complete; transactions in RBTree and ArrayCounter are particularly long. In times when this wait period is lengthened, multiple shorter enemy transactions can complete, competing with one fewer enemy. Indeed, LFUCache and Stack transactions are very short; and with higher thread counts (not shown due to space limitations), this combination is less effective.

There is no obvious analogous deterministic pathology associated with tranaction priority levels. While backoff randomization helps in locking algorithms that have multiple contenders (which can get into simultaneous retry pathology), this problem does not arise in the 2-transaction case. Instead, one continues oblivious to the conflict and the other backs off. This is why randomizing backoff yields comparatively little direct benefit.

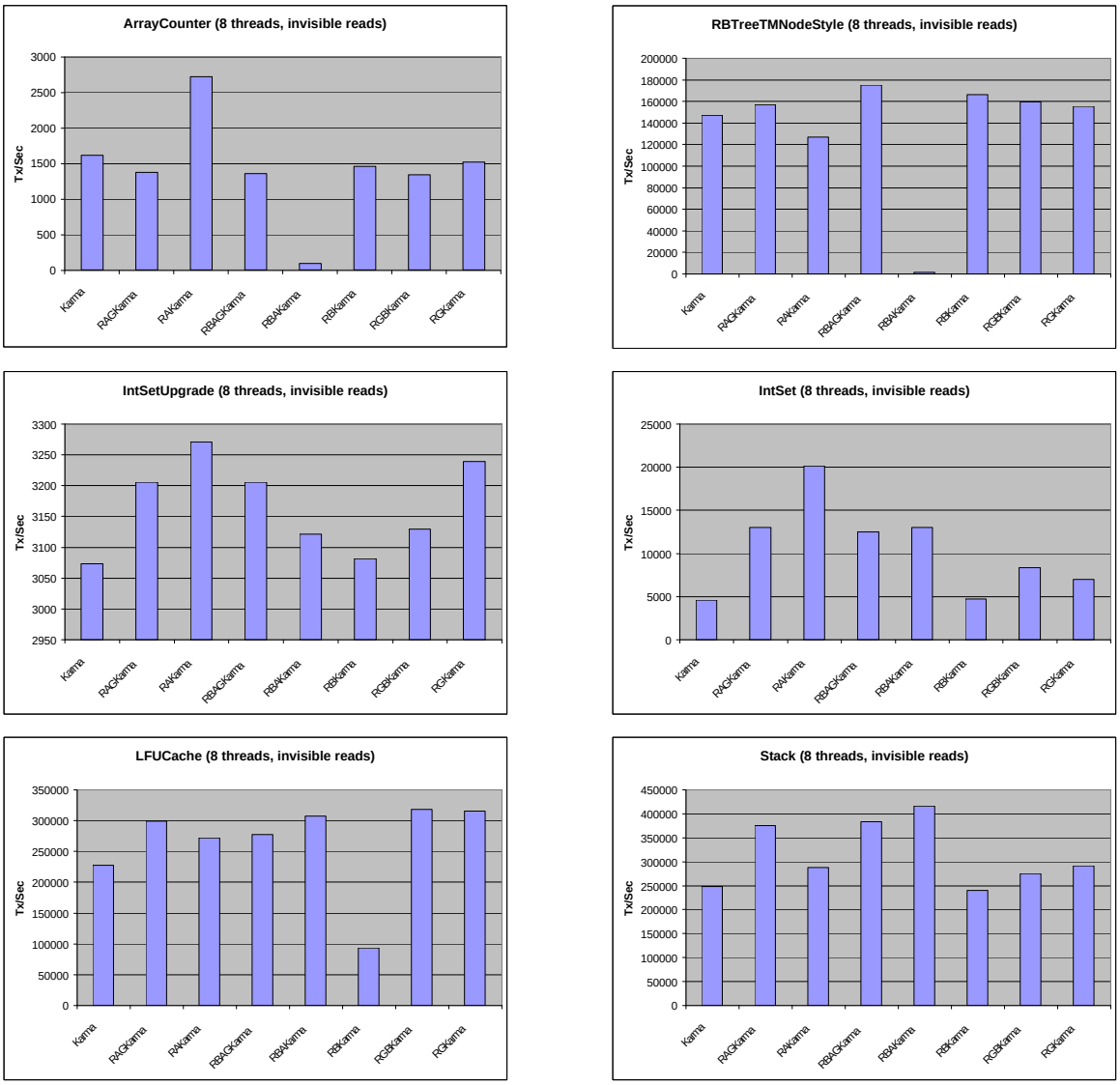

Figure 1: Throughput results for 8 threads and each combination of randomizing backoff (B), abortion (A) decisions, and/or gain (G) upon opening a block (ordered alphabetically)

4.2 Future work

As future work, we plan to analyse other randomized contention managers, more benchmarks, and systems with greater variability in transaction type.

Acknowledgments

We are grateful to Sun's Scalable Synchronization Research Group for donating the SunFire machine.

References

[1] K. Fraser. Practical Lock-Freedom. Ph. D. dissertation, UCAM-CL-TR-579, Computer Laboratory, University of Cambridge, Feb. 2004.

[2] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *OOPSLA 2003 Conf. Proc.*, Anaheim, CA, Oct. 2003.

[3] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proc. of the 22nd ACM Symp. on Principles of Distributed Computing*, pages 92–101, Boston, MA, July 2003.

[4] W. N. Scherer III and M. L. Scott. Contention Management in Dynamic Software Transactional Memory. In *Proc. of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, NL, Canada, July, 2004.