

# Contention Management in Dynamic Software Transactional Memory – Errata\*

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

April 2005

Our paper [2] at the 2004 ACM PODC workshop on Concurrency and Synchronization in Java Programs (CSJP'04) presented preliminary comparisons of several out-of-band contention managers for the Dynamic Software Transactional Memory (DSTM) system of Herlihy et al. [1]. In the course of subsequent work (joint with Virendra J. Marathe) we have identified several problems with the implementation used to collect our results. Corrected, these change the results of our experiments enough to merit dissemination of revised graphs and conclusions.

## 1 Implementation Problems

**Missing contention management calls.** Several “hooks” for contention management were missing in our code, including the notification that an object had been opened for write or visible read access.

**Inefficient visible reads.** In the case of visible reads, a reader was added to the beginning of the reader list every time it opened an object, causing it to be listed multiple times if the same object was opened more than once. This was not a correctness problem, but led to unnecessary overhead for both the reader and any subsequent writer.

**Incomplete early release.** The early release operation needs to remove an object from two lists: one containing all opened objects, the other containing objects eligible for early release. Our code removed the object from the latter but not the former, resulting in unnecessary validation overhead.

**Coarse-Grained Time Measurement.** The implementation of `Thread.sleep()` in our local Java library accepts a nanosecond argument, but incurs a minimum wait of several milliseconds. The default wait of 100 ns, used in several policies, was implicitly greatly inflated.

To implement fine-grain waits we replaced calls to `Thread.sleep()` with busy-wait loops that call `System.nanoTime` (which does have high accuracy and low overhead in our Java implementation). Subsequent trial-and-error experimentation suggested a default wait of 1000 ns in the Eruption, Karma, KillBlocked, Kindergarten, QueueOnBlock, and Timestamp managers. Additional experimentation with the Polite manager led to a new set of parameters for exponential backoff: a minimum of  $2^4$  ns, rising to a maximum of  $2^{26}$  ns over a maximum of 22 cycles of delay. Finally, we found that the previous strategy of losing 50% of accumulated priority upon each abort in the Eruption manager actually reduced throughput for all benchmarks. The new version of the policy retains priority at aborts.

## 2 Revised Results

We are still experimenting with implementations of early release, and are not yet confident of our ability to separate implementation artifacts from the impact of fewer conflicts. We have therefore left `IntSetRelease` out of the results reported here. In its place we report results for an `IntSetUpgrade` benchmark in which we open all list nodes for read-only access, then upgrade to write access for only those nodes that actually need to be modified.

---

\*This work was supported in part by NSF grants EIA-0080124, CCR-9988361, and CCR-0204344, by DARPA/AFRL contract number F29601-00-K-0182, and by financial and equipment grants from Sun Microsystems Laboratories.

As before, our results were obtained on a 16-processor SunFire 6800 with 1.2Ghz UltraSPARC III processors. Our test environment, however, is upgraded to Sun's Java 1.5 HotSpot VM. (We had previously used a beta version.) We ran each benchmark with each of the contention management policies described in the original paper (modified as described above) for 10 seconds. We completed three passes of this test regime for both visible and invisible read implementations, varying the level of concurrency from 1 to 128 threads.

Figures 1–5 show averaged results for the counter and LFUCache benchmarks, the read-black tree-based integer set benchmark, and the two linked list-based integer set benchmarks. Each graph is shown both in total and zoomed in on the first 16 threads (where multiprogramming does not occur).

### 3 Conclusions

As in the original paper, we find that different contention management policies work better for different benchmark applications, and that no single manager provides all-around best results. However, the group of managers that yield top performance narrows considerably: Polite does well for many benchmarks, but Karma, Eruption, and Kindergarten yield good performance in each case where Polite does less well. We therefore recommend choosing one of these managers for any realistic application of DSTM, though the exact conditions under which any one manager outperforms the others remain unclear.

Choosing between visible and invisible reads, however, is no longer as difficult a proposition as we had previously reported. While visible reads yield far better throughput than invisible reads with the IntSet benchmarks, invisible reads only outperform visible reads with very high levels of contention in the RB-Tree benchmark. As a result, we recommend visible reads as a general default strategy.

### Acknowledgments

We are indebted to Sun's Scalable Synchronization Research Group for providing the SunFire machine used in the experiments as well as an experimental version of DSTM that supports both visible and invisible reads. We are further grateful to Virendra Marathe for his discovery of the coarse granularity of `Thread.sleep()`.

### References

- [1] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing*, pages 92–101, Boston, MA, July 2003.
- [2] W. N. Scherer III and M. L. Scott. Contention Management in Dynamic Software Transactional Memory. In *Proceedings of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, NL, Canada, July, 2004.

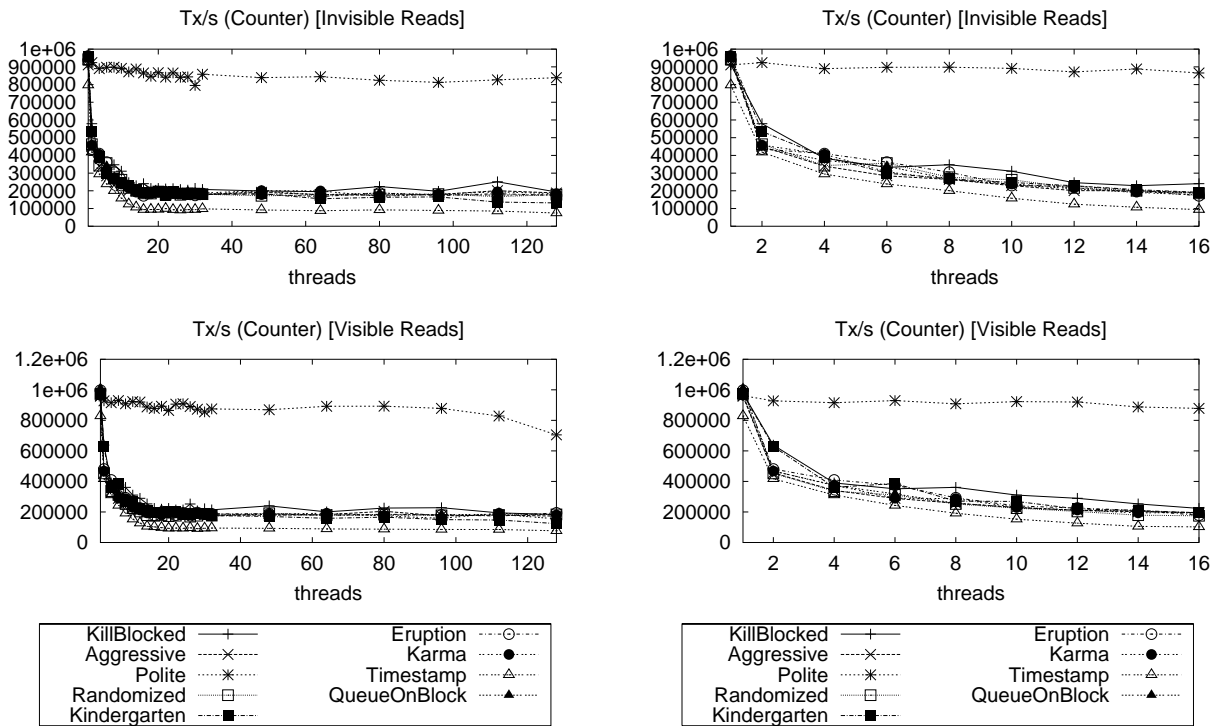


Figure 1: Counter benchmark results

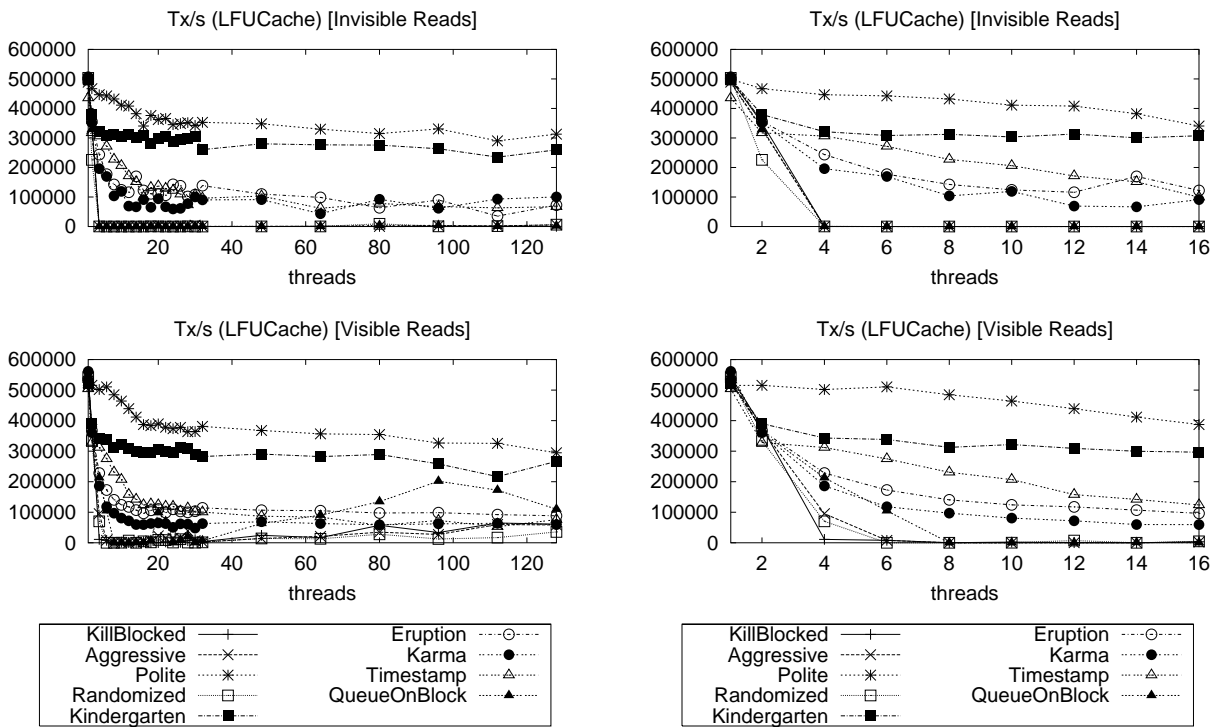


Figure 2: LFUCache benchmark results

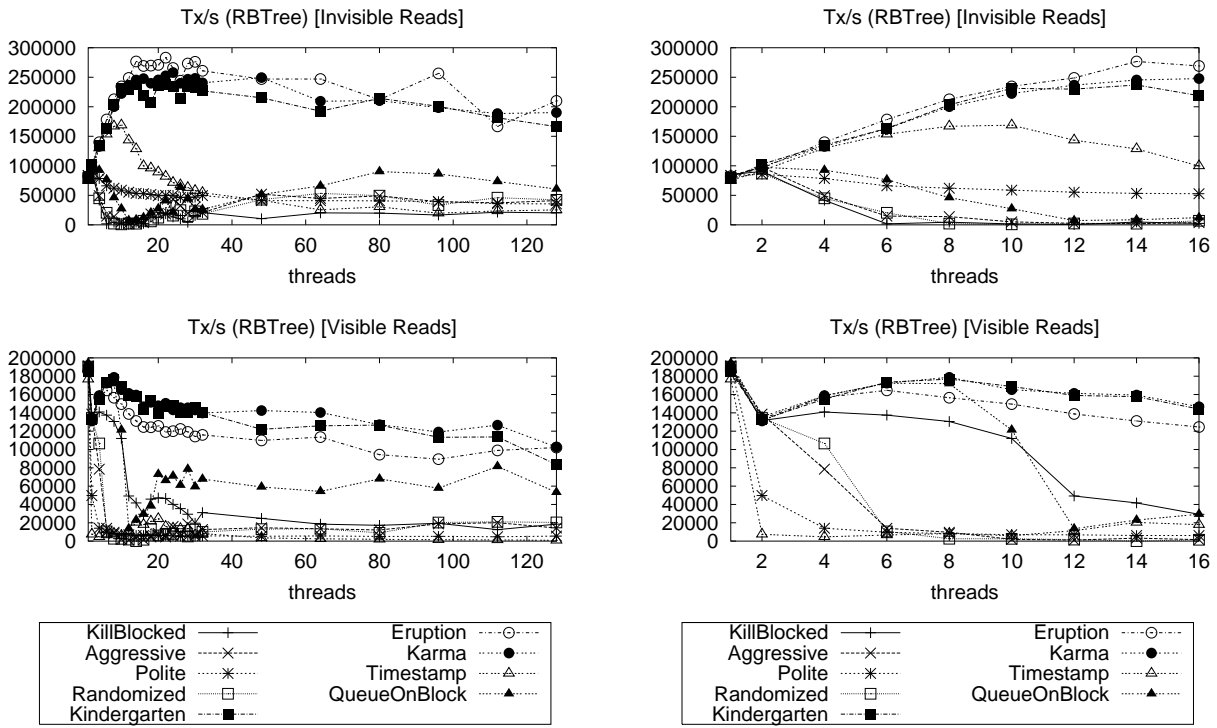


Figure 3: RBTree benchmark results

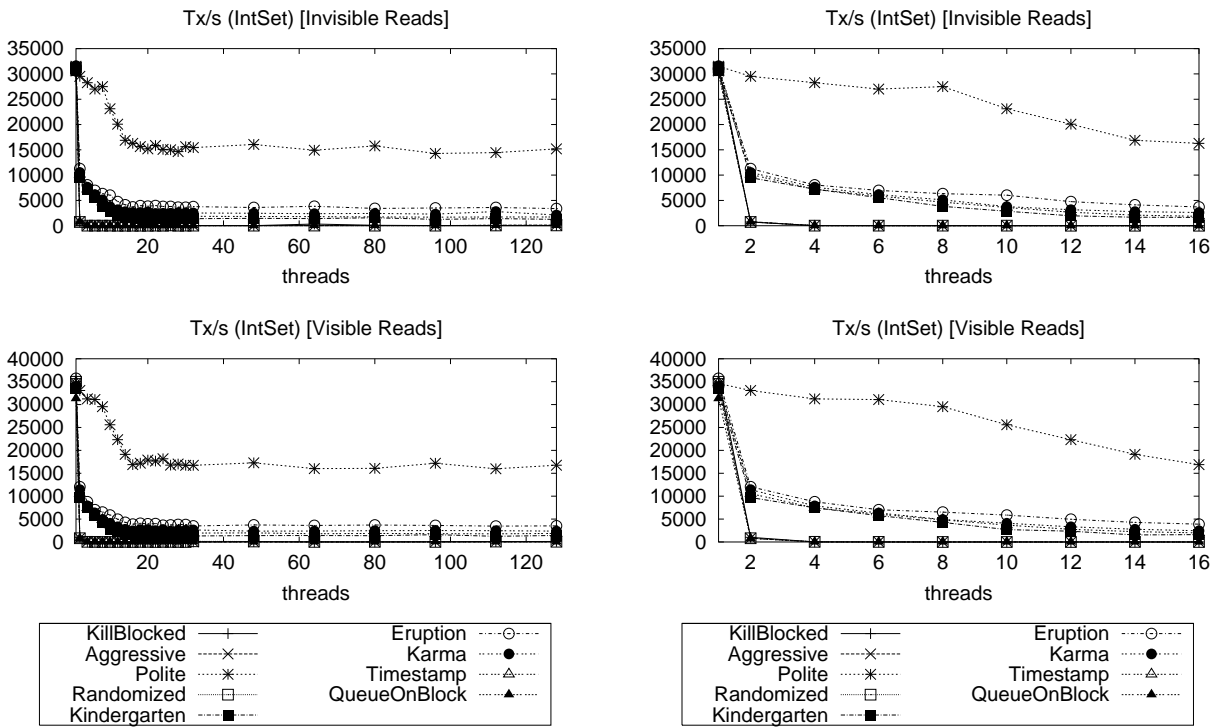


Figure 4: IntSet benchmark results

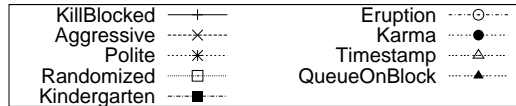
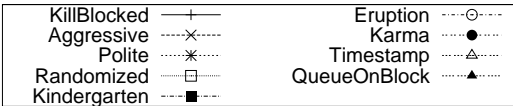
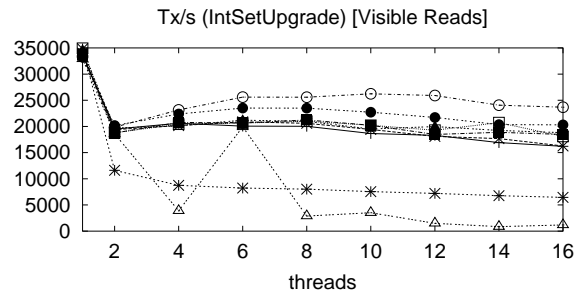
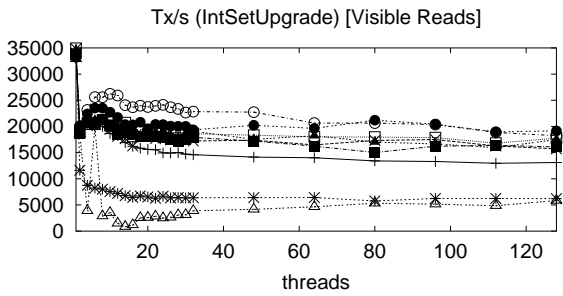
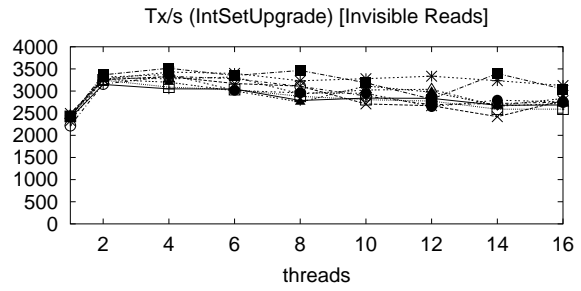
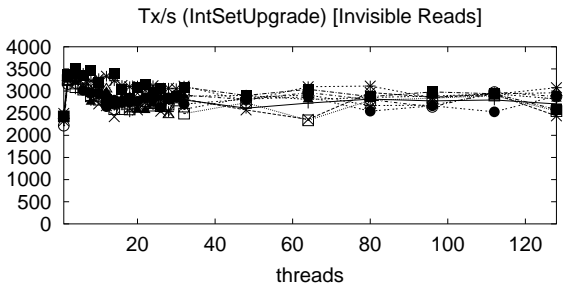


Figure 5: IntSetUpgrade benchmark results