Brief Announcement: Selfishness in Transactional Memory

Raphael Eidenbenz
Computer Engineering and Networks Lab
ETH Zurich, Switzerland
eidenbenz@tik.ee.ethz.ch

Roger Wattenhofer
Computer Engineering and Networks Lab
ETH Zurich, Switzerland
wattenhofer@tik.ee.ethz.ch

ABSTRACT

In order to be efficient with selfish programmers, a multicore transactional memory (TM) system must be designed such that it is compatible with good programming incentives (GPI), i.e., writing efficient code for the overall system coincides with writing code that optimizes an individual program's performance. By implementing a selfish strategy, we show that under most contention managers (CM) proposed in the literature so far, TM systems are not GPI compatible, whereas a simple randomized CM is GPI compatible.

Categories and Subject Descriptors: D.4.1 [Process Management]: Concurrency, Scheduling, Threads; J.4 [Social and Behavioral Sciences] Economics

General Terms: Performance, Human Factors, Experimentation

1. INTRODUCTION

As multicore architecture is evermore pervasive in today's computers, efficient software is to consist of several concurrent execution threads. Writing concurrent code is a challenge faced by modern developers. The paradigm of Transactional Memory (TM)[4] has emerged as a promising approach to keep this challenge manageable. A TM system provides the possibility for programmers to wrap critical code that performs operations on shared memory into transactions. The system then guarantees an exclusive code execution such that no other code being currently processed interferes with the critical operations. To achieve this, TM systems employ a mechanism called contention manager (CM). If two processes want to access the same resource, a CM resolves the conflict, i.e., it decides which transaction may continue and which must abort. The aborted transaction will be restarted by the system until it is executed successfully. One might think that it is in the programmer's interest to choose the placement of atomic blocks as beneficial to the TM system as possible. Unfortunately, in current TM systems, it is not necessarily true that if a thread is well designed—meaning that it avoids unnecessary accesses to shared data—it will also be executed faster. On the contrary, most CMs proposed so far privilege threads that incorporate long transactions rather than short ones. This is not a severe problem if there is no competition for the shared resources among the threads. In large software projects, however, there are many developers, and conflicting threads often originate from different programmers. Given developers act selfish, it is hence realistic to assume competition in many cases. Developers are inclined to write program code that boosts their individual transactions, even if this is bad for the project as a whole. Consequently, a TM system that allows boosting an individual transaction's performance at the expense of the overall system performance suffers an avoidable loss of efficiency.

2. GOOD PROGRAMMING INCENTIVES

Avoiding unnecessary locks and partitioning transactions whenever possible is beneficial to a TM system since the former prevents conflicts and the latter gives more freedom to the CM. Partitioning here means that a transaction is divided into two transactions without overhead. We say a CM $\mathcal{M}$ *rewards partitioning* of transactions if in a system managed by $\mathcal{M}$, it is rational for a programmer to always partition a transaction whenever the program logic allows her to do so. Further, $\mathcal{M}$ *punishes unnecessary locking* if in a system managed by $\mathcal{M}$, it is rational for a programmer to never lock resources unnecessarily, i.e., she only locks a resource when required by the program logic. One can expect that, from a certain level of selfishness among developers, a CM which incentivizes these two crucial aspects of good programming, performs better than the best incentive incompatible CM. We are therefore interested in the question of which CMs are good programming incentive (GPI) compatible. A CM is *GPI compatible* if it rewards partitioning and punishes unnecessary locking. Most CMs proposed in [1, 2, 5, 6] incorporate a mechanism that accumulates some sort of priority for a transaction. In the event of a conflict, the transaction with higher priority wins against the one with lower priority. The priority is typically supposed to measure, in one way or another, the work already done by a transaction. The proposed CMs base priority on a transaction's time in the system, the number of conflicts won, the number of aborts or the number of resources accessed. The intuition behind this approach is that aborting old transactions discards more work already done and thus hurts the system efficiency more than discarding newer transactions. Although this approach seems legitimate, it incentivizes programmers to not partition transactions as the built-up priority would be lost. In some cases even locking resources unnecessarily is rational since this increases priority.

THEOREM 2.1. *Polite, Greedy, Karma, Timestamp and Polka are not GPI compatible. Randomized is GPI compatible.*

One example of a CM which is not priority-based is Randomized (cf. [5]). To resolve conflicts, Randomized simply flips a coin in order to decide which competing transaction to abort. The advantage of this simple approach is that it bases decisions neither on information about a transaction's history nor on predictions about the future. This leaves programmers little possibility to boost their competitiveness. Employing such a simple Randomized CM is not a good solution although it rewards good programming. A transaction's probability $\mathcal{P}$ of running until commit decreases exponentially with the number of conflicts, i.e., $\mathcal{P} \sim p^{|C|}$ where p is the

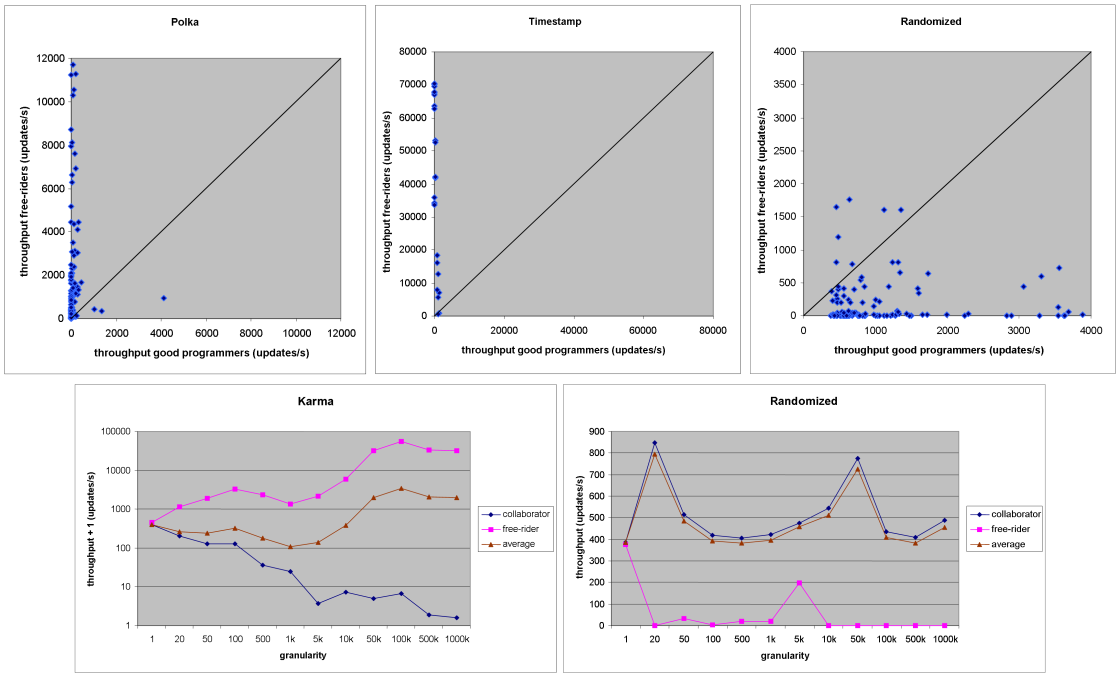

Figure 1: (top) Plot of all cases simulated under a Polka, a Timestamp and a Randomized CM. If a point is above the diagonal line this indicates that in the corresponding test run, a free-rider had a larger throuhput than a good programmer. (bottom) Average throughput in the red-black tree benchmark with 15 collaborators and one free-rider. For Karma, we used a logarithmic scale.

probability of winning an individual conflict and C the set of conflicts. However, there is great potential for further developement of CMs based on randomization.

3. SIMULATIONS

We implemented free-riders in DSTM2 [3], a software transactional memory system in Java, and let them compete with the threads originally provided by the authors of the included benchmark. In particular, we added threads that use coarse transaction granularities, i.e., instead of just updating one resource, a free-rider updates several resources per transaction at once. We tested and compared the performance of free-riding threads with collaborative threads in two benchmarks, one with a shared ordered list and one with a shared red-black tree. In both, there is a total number of 16 threads which start using a shared data structure for 10 seconds, before they are all stopped. All operations are update operations, i.e., a thread either adds or removes an element. We ran various configurations of the scenario in both benchmarks managed by the Polite, Karma, Polka, Timestamp or the Randomized contention manager. The variable parameters were the number of free-riders (0, 1, 8, 16) among the 16 threads and their transaction granularity γ. The benchmarks were executed on a machine with 16 cores, namely 4 Quad-Core Opteron 8350 processors running at a speed of 2 GHz. To get accurate results every benchmark was run five times with the same configuration and averaged. The results show that a free-rider can outperform and sometimes almost entirely deprive the collaborative threads of access to the shared resources if the TM system is managed by the Polite, Karma, Polka or the Timestamp CM. With Randomized on the other hand, the collaborative threads are much better off than the “free-riders”(cf. Figure 1). In all of our tests, if the system was managed by Polite, the free-riders were always better off. Under Karma, they were better off in 92% of all cases and if they used granularities γ of at least 20 operations per transaction, they always performed better. With Polka, the free-rider success rate was 70% over all runs and 100% for $\gamma \in \{20, 50, 100\}$. Of all tests run with Timestamp, free-riding paid off in 92% of the cases and in 100% if the granularity γ was at least 20. Under Randomized, free-riders had a larger throughput in only 7% of all cases.

4. CONCLUSION

While Transactional Memory constitutes an inalienable convenience to programmers in concurrent environments, it does not automatically defuse the danger that selfish programmers might exploit a multicore system to their own but not to the general good. A TM system thus has to be designed strategy-proof such that programmers have an incentive to write code that maximizes the system performance. Priority-based CMs are prone to be corrupted. CMs not based on priority seem to feature incentive compatibility more naturally.

5. REFERENCES

[1] H. Attiya, L. Epstein, H. Shachnai, and T. Tamir. Transactional contention management as a non-clairvoyant scheduling problem. In *PODC ’06: Proc. 25th ACM symposium on Principles of Distributed Computing*, 308–315, 2006.

[2] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *PODC ’05: Proc. 24th ACM symp. on Principles of Dist. Computing*, 258–264, 2005.

[3] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. *SIGPLAN Not.*, 41(10):253–262, 2006.

[4] M. Herlihy and J. E. B. Moss. Transactional memory:architectural support for lock-free data structures. *SIGARCH Comp. Arch. News*, 21(2):289–300, 1993.

[5] W. N. Scherer III and M. L. Scott. Contention Management in Dynamic Software Transactional Memory. In *PODC Workshop on Concurrency and Synchronization in Java Programs (CSJP), St. John’s, NL, Canada*, July 2004.

[6] W. N. Scherer III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC ’05: Proceedings of the 24th annual ACM symposium on Principles of Distributed Computing*, pages 240–248, 2005.