On Designing NUMA-Aware Concurrency Control for Scalable Transactional Memory

Mohamed Mohamedin Roberto Palmieri Sebastiano Peluso Binoy Ravindran
ECE Department, Virginia Tech
{mohamedin, robertop, peluso, binoy}@vt.edu

Abstract

NUMA architectures posed the challenge of rethinking parallel applications due to the non-homogeneity introduced by their design, and their real benefits are limited to the characteristics of the particular workload. We name as *partitionable transactional workloads* such workloads that may be able to exploit the distributed nature of NUMA, such as transactional workloads where data and accesses can be easily partitioned among the so called NUMA zones. However, in case those workloads require the synchronization on shared data, we have to face the issue of exploiting the NUMA architecture also in the concurrency control for their transactions. Therefore in this paper we present a NUMA-aware concurrency control for transactional memory that we designed for promoting scalability in scenarios where both the transactional workload is prone to scale, and the characteristics of the underlying memory model are inherently non-uniform, such as NUMA architectures.

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

Keywords Transactional Memory, NUMA, Scalability

1. Overview

Transactional Memory (TM) [8] is a powerful programming abstraction for implementing concurrent applications. TM frees programmers from the complexity of managing multiple threads that access the same set of shared objects, and its programmability advantage is compelling with the advent of multi-core architectures, which have exacerbated the multithreaded programmability challenge by requiring greater exposition of concurrency in software for improved application performance.

To continue to push the performance boundary of today's multi-core architectures, hardware designers have continually increased the per-processor core count in multi-core architectures. This has had many consequences. In particular, scalability issues of single bus-based architectures have led to Non-Uniform Memory Access (NUMA) [12] designs, which are increasingly becoming the de-facto standard for emerging multi/many-core platforms (e.g., Intel QuickPath Interconnect, Opteron/HyperTransport [1, 16]).

In a NUMA design, one memory socket is physically attached to one processor socket (or to one die inside a socket), and it represents what is called a "NUMA zone". We say that a thread executing on a socket accesses a *local* NUMA zone when it accesses a memory location within the NUMA zone connected to that socket. Otherwise, we say that the thread accesses a *remote* NUMA zone. In the former case the latency is very small (e.g., 9 *nsec* in DDR3-2000 memory [10]), since there is no usage of the shared bus, while in the latter case the latency and the hardware contention is higher because the usage of the shared bus is required.

This distinction is of particular interest for a broad range of transactional workloads, which we call *partitionable transactional workloads*, wherein application data can be partitioned such that transactions mostly access data that are stored in the partitions where they are running, and therefore they are well-suited for exploiting the characteristics of NUMA. Examples of such workloads include most OLTP applications, e.g., TPC-C [15] benchmark, as well as applications that have driven the design of scalable in-memory distributed transactional systems, e.g., [13].

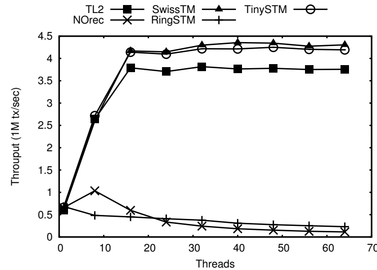
In this paper we focus on maximizing the performance of such workloads when they are executed on software TM (STM) that are deployed on NUMA architectures. The reason is simple: *today's STM implementations are not prone to scale on NUMA architectures even when the workloads are partitionable*. To support our claim we conducted an experimental study by using a typical commodity NUMA machine: a 64-core AMD Opteron 6376 server (2.3 GHz, and 128 GB of memory), which is structured in 4 sockets of 2 dies each, with a NUMA zone of 8 cores per die.

In the study, we assessed the performance of the most popular TM algorithms in the literature, including TL2 [4], SwissTM [6], TinySTM [7], RingSTM [14], and NOrec [3], by configuring TL2, SwissTM, and TinySTM to rely on a lock table that is partitioned across NUMA zones. We used a version of the Bank benchmark (i.e., a micro-benchmark that represents bank accounts and enables monetary transfers among the accounts), which meets our definition of partitionable workload: the bank accounts were partitioned across NUMA zones, and threads operated only on accounts stored in their local NUMA zone. Each transaction produces 10 transfer operations involving two bank accounts each.

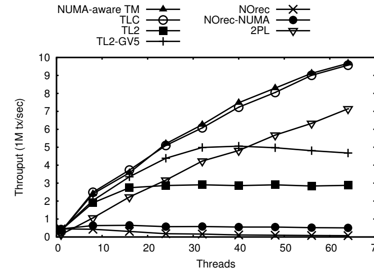
Figure 1(a) shows the throughput (committed transactions per second) under varying number of threads. We can notice that all of the algorithms stop scaling with more than 16 threads, where 16 is the maximum number of parallel threads allowed within a single socket, since the cost of updating their global meta-data, e.g., a shared logical clock, becomes significantly high in that case.

In this paper we propose a NUMA-aware concurrency control for transactional memories, in order to obtain scalable performance in case of partitionable transactional workloads on NUMA machines. Our solution has two key design principles: *i)* threads running in different NUMA zones do not interfere with each other if the transactions they are executing do not conflict on common transactional objects; and *ii)* threads executing in the same zone are

[Copyright notice will appear here once 'preprint' option is removed.]



(a) Motivating results with no inter-NUMA zone conflicts.



(b) Results assessing the benefits of a NUMA-aware concurrency control with $<10\%$ of inter-NUMA zone conflicts.

Figure 1. Throughput of Bank benchmark configured to produce a partitionable workload.

allowed to share information any time to speed up their execution, as this cooperation does not significantly affect the performance of threads executing on other zones.

The core idea is to treat conflicts involving transactions that execute within the same zone differently from those that involve transactions on different zones. This allows the concurrency control to resolve some conflicts simply and efficiently within a single zone, e.g., by detecting a change of a shared per-zone logical clock, as updating shared variables locally at a zone is a fast operation. On the other hand, identifying inter-zone conflicts involves a different mechanism: each time a transaction accesses a remote NUMA zone, it leverages that zone's local view about the status of the memory to incrementally build the memory snapshot that it is allowed to read from. Thus, an inter-zone conflict is detected as soon as a transaction is required to update its readable snapshot. The conflict causes an abort whenever the snapshot cannot be updated without violating the correctness of the resulting execution.

2. Evaluation

We implemented our concurrency control in C++ language, and we integrated it into the RSTM framework[1]. We conducted a comprehensive evaluation by using the version of Bank benchmark and a 64-core machine as described in Section 1, and generating a percentage of conflict among threads running on different NUMA zones that is less than 10%.

As competitors, we considered two state-of-the-art STM algorithms that rely on global shared meta-data, i.e., TL2 and NOrec; two disjoint-access parallel [9] algorithms, i.e., TLC [2] and Strict 2-Phase Locking (2PL); and an optimized version of TL2 that was designed to alleviate the frequency of accesses to the shared global meta-data, i.e., TL2 GV5 [11]. We also developed a version of NOrec that was enhanced with our implementation of the NUMA-aware lock of [5], specifically, the C-BO-BO Lock.

Figure 1(b) shows the throughput under varying number of threads. We notice that our NUMA-aware TM provides the highest throughput and best scalability; its throughput almost linearly increases by increasing the number of threads. TLC's performance is very close to our solution since the level of inter-zone contention of this workload is very low, and TLC does not suffer from the typical high number of unnecessary aborts, which is common under slightly more contending workloads [2]. 2PL also scales well, but the overhead of acquiring locks on both read and written objects at encounter-time is evident, even with such low contention, and therefore has a much lower throughput than our solution and TLC. In contrast, TL2-GV5 stops scaling when the system size exceeds 32, as it uses a centralized single timestamp. TL2 has the same behavior starting from 16 threads. NOrec and NOrec-NUMA have very limited scalability up to 16 threads, due to NOrec's sequential commit phase, although it is clear that using a NUMA-aware lock enhances performance.

[1] `http://www.cs.rochester.edu/research/synchronization/rstm/`

Acknowledgments

This work is supported in part by the AFOSR under grant FA9550-14-1-0187.

References

[1] J. Antony, P. P. Janes, and A. P. Rendell. Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/Hypertransport. In *HiPC '06*, 2006.

[2] H. Avni and N. Shavit. Maintaining Consistent Transactional States Without a Global Clock. In *SIROCCO '08*, 2008.

[3] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *PPoPP '10*, 2010.

[4] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *DISC '06*, 2006.

[5] D. Dice, V. J. Marathe, and N. Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. In *PPoPP '12*, 2012.

[6] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching Transactional Memory. In *PLDI '09*, 2009.

[7] P. Felber, C. Fetzer, and T. Riegel. Dynamic Performance Tuning of Word-based Software Transactional Memory. In *PPoPP '08*, 2008.

[8] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010. ISBN 1608452352, 9781608452354.

[9] A. Israeli and L. Rappoport. Disjoint-access-parallel Implementations of Strong Shared Memory Primitives. In *PODC '94*, 1994.

[10] JEDEC. DDR3 SDRAM standard (revision F), 2012. `http://www.jedec.org/standards-documents/docs/jesd-79-3d`.

[11] Y. Lev, V. Luchangco, V. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a scalable software transactional memory. In *TRANSACT '09*, 2009.

[12] N. Manchanda and K. Anand. Non-Uniform Memory Access (NUMA). *New York University*, 2010.

[13] S. Peluso, P. Romano, and F. Quaglia. SCORe: A Scalable One-copy Serializable Partial Replication Protocol. In *Middleware '12*, 2012.

[14] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: Scalable Transactions with a Single Atomic Instruction. In *SPAA '08*, 2008.

[15] TPC Council. TPC-C Benchmark, Revision 5.11. Feb. 2010.

[16] D. Ziakas, A. Baum, R. A. Maddox, and R. J. Safranek. Intel® QuickPath Interconnect Architectural Features Supporting Scalable System Architectures. In *HOTI '10*, 2010.