Exploiting Off-the-Shelf Virtual Memory Mechanisms to Boost Software Transactional Memory

Amin Mohtasham[*], Paulo Ferreira[†], and João Barreto[‡]

Instituto Superior Técnico - Universidade Técnica de Lisboa/INESC-ID

Abstract

Transactional memory (TM) is known as a promising successor for traditional lock-based mutual exclusion methods. Amongst different approaches to design TMs, *software transactional memory (STM)* has gained a reputation for being flexible and portable. However, as a result of extra instrumentation for loading and storing data in shared memory, many STMs exhibit considerable sequential overheads, which incur in poor performance in low concurrency workloads. We claim that the hardware and kernel-level mechanisms that already support virtual memory on commodity computers can also play an unexpected role: to fulfill the core concurrency control needs of an STM. By taking advantage of such powerful and fast hardware mechanisms that the traditional STM design neglects, this approach enables us to devise novel STMs with unprecedentedly low sequential overheads. In this work we propose one such method. Preliminary evaluation results show that our method is promising and can alleviate STMs sequential overhead.

1 Introduction

With the advent of *Chip Multi Processors*, parallel programming has become increasingly important. The vast amount of research in the field has yielded new programming paradigms that help non-expert programmers exploit the available hardware parallelism.

Traditional synchronization methods are based on explicit locks and suffer from their trade-off between complexity and performance [11]. On the other hand, *Transactional Memory (TM)* has built up a promising reputation due to its ease of use and scalability [10]. In contrast with explicit lock-based methods, which require the programmer to be an expert in designing an efficient parallel application, TMs offer a simple programming interface for the non-expert programmers. This helps them to take the advantage of available hardware parallelism power without concerning about complex synchronization issues.

One can distinguish two different approaches to implement a transactional memory system: *hardware transactionl memory (HTM)* and *software transactional memory (STM)* [8]. HTM suffers from high implementation and verification costs, as well as limitations to run diverse sets of transactional workloads [3]. Although there are next generation processors with built-in TM support, such as IBM BlueGene/Q [7], IBM zEC12 [13] and Intel Haswell processors [12], it is still highly uncertain whether and when processors with built-in TM support will become the norm in all categories of parallel computing devices.

STM, on the other hand, does not rely on sophisticated hardware and supports flexible transactional programming. However state-of-the-art STMs exhibit considerably higher sequential overhead than HTM and lock-based mutual exclusion methods. This stems from their inherent extra overhead in loading and storing shared memory objects inside a transaction (e.g. maintaining extensive read/write sets and validating them at commit time) [3]. In fact, a recent research on STM shows increasing concerns with the sequential overhead of STM algorithms [3, 4, 14, 9]. For instance, Spear *et. al* [14] propose *TML*, a transactional framework with a very low sequential overhead. TML is assumed to be applied with read-dominated workloads where, to the best of our knowledge, outperforms all off-the-shelf STMs. Of course, STM relies on hardware primitives that are widely supported by commodity CMPs, from simple load/store instructions to atomic instructions such as *compare-and-swap* or *load-linked/store-conditional* [5, 2, 6].

In this paper, we advocate that STM algorithms ought to take advantage of another component that, despite widely available in today's machines, is neglected by mainstream STM: the page-level protection and translation mechanisms that the memory management unit (MMU) offers for virtual memory support. We claim that one can significantly reduce the sequential overhead of STM by taking advantage of the page-level support of the MMU. More precisely, we propose a novel approach that, by relying on the page-level mechanisms offered by the MMU, can boost most mainstream STM algorithms. The key insight of our approach is that a careful use of page-level protection mechanisms can substantially reduce the number of read accesses to shared locations that actually require running the expensive STM read barriers. Instead, STM access barriers are only run in write accesses and read accesses that are suspected to be conflict-prone.

As a proof-of-concept of our approach, we have designed, implemented and evaluated PGSTM, a simple STM which relies on available virtual memory management system. We describe how PGSTM takes advantage of MMU to achieve reductions in sequential overhead for read/write accesses.

The idea of having memory management hardware lock pages in order to synchronize concurrent tasks is not new. Abadi *et al.* [1], use that approach to add strong atomicity to an STM offering weak atomicity. They split the process virtual memory space into a *transactional* and a *non-transactional* region and use hardware page protection tools to achieve their goal.

2 A proof-of-concept: PGTML

At each moment, each active transaction in PGSTM can be either *read-only* or *read-write*. Initially, all transactions start as read-only transactions. As soon as a transaction tries to write to a memory address, it acquires a global *write-lock*, turns into a read-write transaction and remains until it commits.

PGSTM relies on a fundamental assumption: that, at each moment, there can only exist one read-write transaction and the rest are doomed to fail. If a transaction is currently in read-write mode (thus all other transactions are currently read-only), it has full access to all shared data. In contrast, read-only transactions can only read data that has not been written by the currently active read-write transaction.

The access separation between read-write and read-only transactions is done by splitting virtual memory into two regions, both pointing at same physical memory frames but with different access bits (Fig. 1). Whenever a read-write transaction intends to write to a memory address, it first

[*]amohtasham@gsd.inesc-id.pt

[†]paulo.ferreira@inesc-id.pt

[‡]joao.barreto@inesc-id.pt

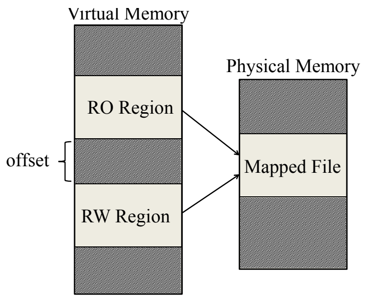

Figure 1: PGSTM's memory model

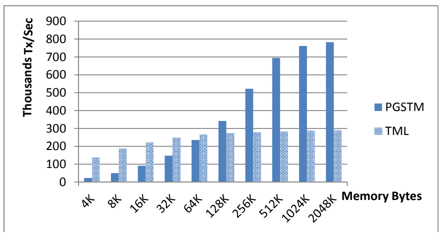

Figure 2: Throughput for different contention levels [64 threads, 10% write-load]

clears the read permission in the page table entry of the corresponding page in the read-only memory region. By doing that, it prohibits read-only transactions accessing that memory page. Then, the read-write transaction can safely write the value.

The advantage of this approach is that we eliminate any need for software-based validation before or after read accesses by read-only transactions. In fact, a read-only transaction can just perform the read directly. In case there is a conflict with the read-write transaction, the underlying memory management unit will automatically throw a trap, which will then cause the read-only transaction to abort and restart.

3 Evaluation

We have implemented PGTML and evaluated it in a host environment consisting of a quad AMD Opteron6272 with 64 cores total. We measure the system throughput in terms of *transactions/second*. We use a trivial *list look-up* benchmark. In this benchmark, each transaction tries to search an element in a list. R/W transactions update the found value to a new value. We define write-load as the probability that a transaction is read-write.

We compare our results with those of TML [14], a simple STM that, to the best of our knowledge, achieves the lowest sequential overhead amongst all STM algorithms proposed so far.

In Figure 2, we show how PGSTM behaves at different contention levels and 64 simultaneous threads. In high contention scenarios (i.e. small list sizes), PGSTM shows lower throughput, which is due to incurred overhead by page faults. However, as the list size increases, PGTSM starts to outperform TML with a high slope, while TML throughput remains steady. Hence, we can infer PGSTM is more fitted for medium and low contention scenarios.

4 Conclusion and Future Work

In this paper, we explored the feasibility of using hardware memory management unit with STMs to lower their sequential overhead and improve their throughput. Toward this end, we proposed PGSTM. PGTSM reduces the cost of read operations by taking advantage of commodity memory protection hardware. This algorithm is just a preliminary step, a mere proof that relying on MMU page-level mechanisms can boost the sequential overhead of STM.

In future, we intend to is to extend this work towards a generic approach that can generically boost the sequential performance of any mainstream STM solution.

References

[1] M. Abadi, T. Harris, and M. Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. *ACM Sigplan Notices*, 44(4):185–196, 2009.

[2] J. Barreto, A. Dragojevic, P. Ferreira, R. Filipe, and R. Guerraoui. Unifying thread-level speculation and transactional memory. In *Proceedings of the 13th International Middleware Conference*, pages 187–207. Springer-Verlag New York, Inc., 2012.

[3] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, 2008.

[4] L. Dalessandro, M. F. Spear, and M. L. Scott. Norec: streamlining stm by abolishing ownership records. In *ACM Sigplan Notices*, volume 45, pages 67–78. ACM, 2010.

[5] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *ACM Sigplan Notices*, volume 44, pages 155–165. ACM, 2009.

[6] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems (TOCS)*, 25(2):5, 2007.

[7] R. Haring and B. Team. The blue gene/q compute chip. In *Hot Chips*, volume 23, 2011.

[8] T. Harris, J. Larus, and R. Rajwar. Transactional memory. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, 2010.

[9] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *ACM SIGPLAN Notices*, volume 41, pages 14–25. ACM, 2006.

[10] M. Herlihy and J. E. B. Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.

[11] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *ACM SIGOPS Operating Systems Review*, volume 36, pages 5–17. ACM, 2002.

[12] J. Reinders. Transactional synchronization in haswell. *Intel Software Network*, 2012.

[13] C. Shum, F. Busaba, and C. Jacobi. Ibm zec12: The 3rd generation high frequency mainframe microprocessor. 2013.

[14] M. F. Spear, A. Shriraman, L. Dalessandro, and M. L. Scott. Transactional mutex locks. In *SIGPLAN Workshop on Transactional Computing*, 2009.