J. Parallel Distrib. Comput. 89 (2016) 13–24

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Optimizing memory transactions for large-scale programs

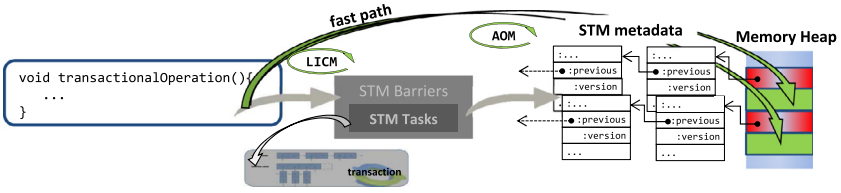
Fernando Miguel Carvalho [a,b,*], João Cachopo [a]

[a] INESC-ID/Instituto Superior Técnico, University of Lisbon, Portugal
[b] ADEETC, Instituto Superior de Engenharia de Lisboa, Polytechnic Institute of Lisbon, Portugal

HIGHLIGHTS

- A new technique of adaptive object metadata (AOM) that eliminates the extra STM metadata.
- AOM with LICM (lightweight identification of captured memory) provide a fast path for non-contended objects.
- Results that show performance with an STM that rivals a fine-grained lock in a large-scale benchmark.
- Integrated in Deuce STM full support for in-place metadata that is required by LICM and AOM.
- Innovative adaptation of Deuce STM: maintains original API, and enhances any existing STM.

GRAPHICAL ABSTRACT



ARTICLE INFO

Article history:
Received 8 November 2013
Received in revised form
20 February 2015
Accepted 3 December 2015
Available online 14 December 2015

Keywords:
Software Transactional Memory
Runtime optimizations
Concurrent programming

ABSTRACT

Even though Software Transactional Memory (STM) is one of the most promising approaches to simplify concurrent programming, current STM implementations incur significant overheads that render them impractical for many real-sized programs. The key insight of this work is that we do not need to use the same costly barriers for all the memory managed by a real-sized application, if only a small fraction of the memory is under contention—lightweight barriers may be used in this case. In this work, we propose a new solution based on an approach of *adaptive object metadata* (AOM) to promote the use of a fast path to access objects that are not under contention. We show that this approach is able to make the performance of an STM competitive with the best fine-grained lock-based approaches in some of the more challenging benchmarks.

1. Introduction

The idea of providing hardware support for transactions was firstly introduced by Knight [18] to check the correctness of parallel executions of Lisp programs. Later, Herlihy and Moss [16] extended the concept to the notion of Transactional Memory (TM) and then Shavit and Touitou [26] proposed a software realization of the same idea, called Software Transactional Memory (STM), however, with a different interface from the original TM proposal.

One of the key selling points of TM is that it simplifies the development of concurrent programs, because programmers do not have to enforce isolation through some concurrency control mechanism, but instead, they just have to say which groups of operations should be executed *atomically*. To that end, Harris and Fraser [13]

* Corresponding author at: INESC-ID/Instituto Superior Técnico, University of Lisbon, Portugal.
E-mail addresses: mcarvalho@cc.isel.ipl.pt (F.M. Carvalho), joao.cachopo@ist.utl.pt (J. Cachopo).

http://dx.doi.org/10.1016/j.jpdc.2015.12.001
0743-7315/

proposed the use of an `atomic` construct to provide a style of *conditional critical region* [17] in the Java programming language. Alternatively, and following the same approach, the Deuce STM [20] provides a simple API based on an `@Atomic` annotation to mark methods that must have a transactional behavior.

Unfortunately, one of the consequences of making an STM completely *transparent* to the programmer is that it may add large overheads to the program. A major source of overheads in STM-based programs is the use of *STM barriers* whenever a memory location is accessed within a transaction. These barriers are responsible for keeping track of what is done inside a transaction (at a minimum, STM barriers need to keep track of what is read and written), so that the STM runtime is able to ensure the intended transactional semantics.

This problem was pointed out by some authors (e.g., [6,22]), who raised questions about the practical applicability of STMs to large-scale programs. In fact, whereas STMs have shown promising results when applied to micro-benchmarks, they typically perform worse than coarse-grained lock-based approaches in larger benchmarks with a large number of shared objects.

Yet, we claim that it is possible to reduce substantially the STM-induced overheads for a large-scale program if we assume that the amount of memory under contention – that is, memory being concurrently accessed both for read and for write – is only a small fraction of the total amount of memory accessed by that program.

The key idea is that for non-contended memory (that is, memory not being concurrently accessed both for read and for write) we can avoid the full-blown STM barriers, which should be used only for accessing (the relatively rare) memory under contention. Instead, for the frequent non-contended memory accesses we use lightweight barriers that access directly the target memory, thereby significantly reducing the overheads imposed by the STM. Our approach combines two orthogonal techniques – *lightweight identification of captured memory* (LICM) and *adaptive object metadata* (AOM) – to reduce the overheads of accessing objects that are not under contention.

LICM allows the automatic identification of transaction-local objects at runtime, for which no STM barriers are needed, but its use does not eliminate all of the excessive overheads caused by an STM in many large-scale benchmarks. Yet, it complements well the second technique that we propose in this paper. AOM is an optimization technique for multi-versioning STMs, based on the JVSTM [10] general design, that is adaptive because the structure of the metadata used for each transactional object changes over time, depending on how the object is accessed.

By combining AOM with LICM, we are able to outperform the coarse-grained (and compete with the best fine-grained) lock-based approaches in some of the more challenging benchmarks, while retaining ease of programming.

In Section 2, we present some motivation for this work and present an overview of our approach. Then, in Section 3 we introduce the key aspects of the JVSTM that are relevant to understand our proposal. In Section 4, we describe in detail the AOM design and discuss the correctness of its operations. In Section 5, we present an experimental evaluation for a variety of benchmarks. In Section 6, we discuss related work on efficient support for STMs. Finally, in Section 7 we conclude and discuss some future work.

2. Motivation and solution overview

Despite the promising results of STMs for micro-benchmarks, when we apply STMs to larger benchmarks, such as STMBench7 [12] or Vacation [2], they typically perform worse than the single-threaded sequential version of the same benchmark. This effect was observed by Fernandes and Cachopo [10], who show that using Deuce with either TL2 STM [8] or LSA STM [25] in STMBench7 achieves a throughput up to 100 times lower than using a coarse-grained lock. Interestingly, however, they also show that by manually instrumenting STMBench7 with the JVSTM (rather than with Deuce), it was possible to get better performance than with the medium-grained locks, which suggests that, after all, it is possible to get good performance from STMs in large-scale applications.

The large gap in performance observed in that work may be attributed to the difference in the STMs used, to the amount of barriers that are introduced into the benchmark by each approach (Deuce vs manual),[1] or, most probably, to a combination of both.

To help us pinpoint the causes for the differences observed by Fernandes and Cachopo [10], we ran STMBench7 with three synchronization approaches: (1) using Deuce to instrument all of the code and executing it using 3 different STMs: TL2, LSA, and JVSTM[2]; (2) using the coarse-grained and medium-grained lock-based synchronization of STMBench7; and (3) using an STMBench7 that was manually instrumented to use the JVSTM.

In Fig. 1, we show the throughput of the benchmark for each of the synchronization approaches, when the number of threads varies from 1 to 48 (in Section 5 we describe the details of the experimental platform). The results show that for this workload the JVSTM performs better than the other STMs (even when all the instrumentation is made by Deuce), but the more telling aspect is the huge gap in performance between using the JVSTM with Deuce or manually: Whereas when using the JVSTM with Deuce the throughput never gets above the sequential non-instrumented execution, in the manual case we get a speedup of 3 times, outperforming even the medium-grained lock-based approach.

Given that the JVSTM used in both cases is exactly the same, this difference must result from the over-instrumentation made by Deuce. This over-instrumentation has two consequences: (1) it adds more STM barriers to the execution of the benchmark; and (2) it adds to each object extra metadata, which needs to be traversed when accessing those objects. Both affect performance negatively.

Our goal is to suppress STM barriers and avoid the metadata indirections in situations where they are not really necessary—that is, when accessing objects that are not under contention. By integrating our solution in Deuce we expect to be able to achieve results similar to those obtained with the manual use of the JVSTM, albeit without the intervention of the programmer (thereby, making the STM easier to use).

The first part of our solution is based on the use of LICM, which we introduced in [5] and further enhanced in [4]. LICM provides a new lightweight runtime technique to elide STM Barriers when accessing *captured memory*—that is, memory allocated inside a transaction that cannot escape (i.e., is captured by) its allocating transaction, as defined by Dragojevic et al. [9]. Captured memory corresponds to newly allocated objects that did not exist before the beginning of their allocating transaction and that, therefore, are held within the transaction until its successful commit. In our LICM work, we were concerned with how to perform the *runtime capture analysis* efficiently. In our solution, we label new objects with a *fingerprint* that uniquely identifies their creating transaction, and then check if the accessing transaction corresponds to that fingerprint, in which case we avoid the barriers. This check is simply an identity comparison between the fingerprint of the accessing transaction and the fingerprint of the accessed object, and, thus, a very lightweight operation. The idea of the fingerprint

[1] When using Deuce, all classes are automatically instrumented, whereas when manually using the JVSTM only the classes of the effectively shared objects are instrumented.

[2] For this purpose we integrated in Deuce the lock-free version of the JVSTM, which was not available yet.