Parallel Computing 50 (2015) 25–52

Contents lists available at ScienceDirect

Parallel Computing

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

Self-tuning Intel Restricted Transactional Memory

Nuno Diegues [a,b,*], Paolo Romano [a,b]

[a] INESC-ID, Rua Alves Redol 9, Lisbon, Portugal
[b] Instituto Superior Técnico, Universidade de Lisboa, Av. Rovisco Pais 1, Portugal

ARTICLE INFO

Article history:
Received 9 April 2015
Revised 2 October 2015
Accepted 3 October 2015
Available online 9 October 2015

Keywords:
Hardware Transactional Memory
Performance
Self-tuning

ABSTRACT

The Transactional Memory (TM) paradigm aims at simplifying the development of concurrent applications by means of the familiar abstraction of atomic transaction. After a decade of intense research, hardware implementations of TM have recently entered the domain of mainstream computing thanks to Intel's decision to integrate TM support, codenamed RTM (Reduced Transactional Memory), in their last generation of processors.

In this work we shed light on a relevant issue with great impact on the performance of Intel's RTM: the correct tuning of the logic that regulates how to cope with failed hardware transactions. We show that the optimal tuning of this policy is strongly workload dependent, and that the relative difference in performance among the various possible configurations can be remarkable (up to 10 × slow-downs).

We address this issue by introducing a simple and effective approach that aims to identify the optimal RTM configuration at run-time via lightweight reinforcement learning techniques. The proposed technique requires no off-line sampling of the application, and can be applied to optimize both the cases in which a single global lock or a software TM implementation is used as fall-back synchronization mechanism.

We propose and evaluate different designs for the proposed self-tuning mechanisms, which we integrated with GCC in order to achieve full transparency for the programmers. Our experimental study, based on standard TM benchmarks, demonstrates average gains of 60% over any static approach while remaining within 5% from the performance of manually identified optimal configurations.

1. Introduction

The ubiquity of multi-core processors in mainstream architectures has motivated the need to identify programming paradigms capable of simplifying the development of concurrent applications. In this scope, Transactional Memory (TM) [1] is a promising approach because it exposes a simple interface: it only requires programmers to identify which code blocks should run atomically, and not how atomicity should be achieved.

This contrasts with traditional lock-based synchronization schemes, where, in order to achieve good scalability, programmers need to design complex lock acquisition schemes. These are often prone to deadlocks/livelocks [2], are hard to reason about and debug [3–5], and, even worse, hinder software composability [6].

* Corresponding author at: Instituto Superior Técnico, Universidade de Lisboa, Av. Rovisco Pais 1, Portugal. Tel.: +351968427763.
E-mail address: nmld@tecnico.ulisboa.pt (N. Diegues).

http://dx.doi.org/10.1016/j.parco.2015.10.001
0167-8191/

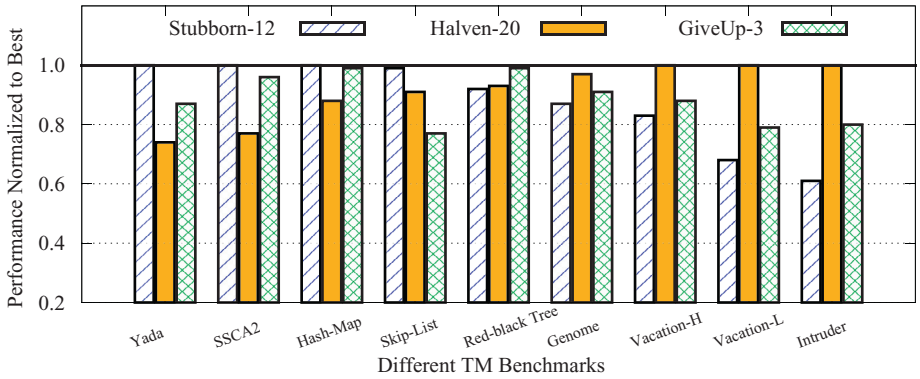

Fig. 1. Relative performance of three example RTM configurations with respect to the Best Static configuration in each benchmark with 8 threads (we show 9 different benchmarks). The configurations used differ in the number of retries allowed for hardware transactions and in how they deal with different types of hardware aborts. This experimental data highlights that no configuration performs consistently better than the others, and that all static configurations can be far from the optimal performance at least for certain workloads.

Conversely, TM exposes a simple and familiar abstraction of atomic transactions, which, on one hand, shelters programmers from the complexity of locks [7,8], while allowing on the other hand for efficient implementations. These were shown to achieve performance similar, and sometimes even superior, to complex, ad hoc designed fine-grained lock synchronization schemes [9–12]. As such, TM promises to conflate ease of usage, efficiency and scalability.

Recently, the maturing of TM research has reached an important milestone with the release of the first mainstream commercial processors providing hardware support for TM. In particular, Intel has augmented the $\times$ 86 instruction set with Transactional Synchronization Extensions under the name Restricted Transactional Memory (RTM). This commodity Hardware Transactional Memory (HTM) implementation is available in the 4th generation core processor, which is widely adopted and deployed, ranging from tablets to server machines.

1.1. Problem

One important characteristic of the Intel HTM is its best-effort nature: due to inherent hardware/architectural limitations, RTM gives no guarantees as to whether a hardware transaction can commit successfully, even in absence of concurrency and data conflicts. This is easily understandable as the implementation heavily relies on the usage of processors' caches, which have limited space. Indeed, although solutions providing stronger progress guarantees for HTM have been proposed in literature, the alterations required to existing CPU architectures are currently perceived as overly invasive and risky [13].[1] For this reason, the best-effort nature of HTM is a sweet spot in the design choices, being shared by every HTM implementation currently proposed by industrial CPU manufacturers, including Intel [14], AMD [15], IBM [16,17] and Oracle [18], and it appears unlikely that alternative designs will be pursued in the near term future.

As such, a programmer using hardware transactions in RTM must decide what should be done upon the abort of that hardware transaction: under which circumstances should an aborted transaction be re-executed using RTM, or when should it resort to an alternative fall-back software-based synchronization scheme?

In this paper we show that there is no definite answer to this question: there is no one-size fits all solution that yields the best performance across all possible workloads. To better illustrate this important statement, we provide experimental evidence summarized in Fig. 1. This figure shows the performance of three example configurations, using Intel HTM, across some popular TM benchmarks. A detailed description of these configurations and benchmarks will be provided, respectively, in Section 3 and 8. These results show the relative performance of each configuration, with respect to the optimal one that we found for each benchmark. The outcome of this plot supports our claim: no configuration performs consistently better than all the others, and they all perform excellently in some benchmarks and poorly in others.

This important fact means that the programmer is left with the responsibility of finding out the best choices for his application—a problem that is cumbersome and time consuming to tackle via off-line profiling, given that there are many available configurations. Even worse, in fact, there may not exist a single optimal solution to be found statically, when in the presence of dynamic workloads. These facts have also been recently acknowledged by Intel researchers, highlighting the importance of developing adaptive techniques to simplify the tuning of RTM [19].

[1] IBM System Z processors represent a notable exception: they guarantee transactions to commit if they abide certain constraints in terms of footprint and instructions, provided that they do not conflict. However, we note that those requirements are quite strict (at most 32 instructions, accessing up to 256 bytes of memory).