J. Parallel Distrib. Comput. 74 (2014) 2845–2859

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

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

Adaptive thread mapping strategies for transactional memory applications

Márcio Castro [a,*], Luís Fabrício W. Góes [b], Jean-François Méhaut [c]

[a] Federal University of Rio Grande do Sul, Institute of Informatics, Av. Bento Gonçalves, 9500 - Campus do Vale - 91501-970 - Porto Alegre, Brazil
[b] University PUC-Minas, Computer Science Department, Avenida Dom José Gaspar, 500 - 30535-610 - Belo Horizonte, Brazil
[c] University of Grenoble, CEA-DRT - LIG Laboratory, ZIRST 51 Avenue Jean Kuntzmann - 38330 - Montbonnot, France

HIGHLIGHTS

- We propose adaptive thread mapping strategies based on single metrics.
- We propose a new strategy based on association rule learning.
- We implement all the proposed adaptive strategies in a TM system.
- We achieved performance improvements of up to 64.4% on a set of synthetic applications.
- We achieved performance improvements of up to 16.5% on the STAMP benchmark suite.

ARTICLE INFO

Article history:
Received 15 June 2013
Received in revised form
25 April 2014
Accepted 28 May 2014
Available online 9 June 2014

Keywords:
Transactional memory
Thread mapping
Adaptivity
Multicore

ABSTRACT

Transactional Memory (TM) is a programmer friendly alternative to traditional lock-based concurrency. Although it intends to simplify concurrent programming, the performance of the applications still relies on how frequent they synchronize and the way they access shared data. These aspects must be taken into consideration if one intends to exploit the full potential of modern multicore platforms. Since these platforms feature complex memory hierarchies composed of different levels of cache, applications may suffer from memory latencies and bandwidth problems if threads are not properly placed on cores. An interesting approach to efficiently exploit the memory hierarchy is called thread mapping. However, a single fixed thread mapping cannot deliver the best performance when dealing with a large range of transactional workloads, TM systems and platforms. In this article, we propose and implement in a TM system a set of adaptive thread mapping strategies for TM applications to tackle this problem. They range from simple strategies that do not require any prior knowledge to strategies based on Machine Learning techniques. Taking the Linux default strategy as baseline, we achieved performance improvements of up to 64.4% on a set of synthetic applications and an overall performance improvement of up to 16.5% on the standard STAMP benchmark suite.

1. Introduction

There was a 30-year period in which the advances in semiconductor technology and computer architectures improved the performance of a single processor at a high annual rate of 40% to 50% [21]. However, issues such as dissipating heat from increasingly densely packed transistors began to limit the rate at which processor frequencies could be increased. This was one of the main reasons why most of semiconductor manufacturers are now investing in multicore processors [2].

Consequently, applications must now evolve to efficiently exploit the potential of multicore platforms. Sequential applications thus need to be split into pieces (*e.g.*, tasks) that can be executed in parallel by threads, each one running on a processor/core [17]. The side effect is that the application data, which were accessed by a single thread on a sequential application, is now shared among several concurrent threads. Thus, it is necessary to use synchronization mechanisms to coordinate concurrent accesses to these shared data.

Traditional synchronization mechanisms such as *locks*, *mutexes* and *semaphores* have been proven to be more error-prone [13],

* Corresponding author.
E-mail addresses: mbcastro@gmail.com, mbcastro@inf.ufrgs.br (M. Castro), lfwgoes@pucminas.br (L.F.W. Góes), jean-francois.mehaut@imag.fr (J.-F. Méhaut).

http://dx.doi.org/10.1016/j.jpdc.2014.05.008
0743-7315/

largely due to well-known problems such as deadlocks and livelocks [25], and are difficult to manage in large scale systems. Due to those issues, researchers have been looking for alternative mechanisms. One of such mechanisms that has been subject of intense research in the last years is Transactional Memory (TM) [16,10]. The TM programming model allows programmers to write parallel portions of the code as transactions, which are guaranteed to execute atomically and in isolation regardless of eventual data races. At runtime, transactions are executed speculatively and the TM runtime system continuously keeps track of concurrent accesses and detects conflicts. Conflicts are then solved by re-executing conflicting transactions. This model removes from the programmer the burden of correct synchronization of threads and provides a straightforward way of extracting parallelism from applications.

Although the TM programming model simplifies concurrent programming, the performance of TM applications on multicores still relies on how frequent they synchronize, the amount of contention (conflicts between transactions) and the way transactions access shared data on memory (memory access pattern) [5]. In order to alleviate the cost of accessing the main memory, multicore processors usually feature complex memory hierarchies composed of different levels of cache (private and shared). As a drawback, this can potentially increase memory access latency and degrade bandwidth usage if threads are not properly placed on cores.

An appealing approach to efficiently exploit the memory hierarchy and alleviate these drawbacks is called *thread mapping* [18], which places threads on specific cores according to a predetermined strategy. However, the efficiency obtained from a thread mapping strategy relies upon matching the behavior of the application with the underlying system and platform characteristics. This issue becomes much more complex in TM due to two reasons: (i) the TM model uses speculation, hence TM applications present an irregular behavior (data dependencies between threads are only known at runtime); and (ii) each TM system implements its own mechanisms to detect and solve conflicts and thus the same TM application can behave differently when the underlying TM system is changed [7,4].

Due to the aforementioned issues, a single fixed thread mapping (which does not adapt itself to the current workload) cannot deliver the best performance in all cases. For instance, in some workloads it would be better to place threads on cores as close as possible in the cache hierarchy to increase cache sharing; while for others it would be better to distribute threads among different processors to reduce memory contention. Because of that, *adaptivity* becomes a key feature to increase performance for a wide range of different workload characteristics and platforms. Adaptivity has been studied in different contexts as a means of: performing dynamic load balancing on MPI [20]; generating and selecting a specific multithreaded version for a given loop at runtime on OpenMP [9]; and automatically selecting a TM algorithm adapted to the workload [28].

As opposed to those previously cited adaptive approaches, in this article we exploit adaptivity in the context of *thread mapping*. Since we are particularly interested in TM, we propose different adaptive thread mapping strategies that consider information from the TM application, TM system and multicore platform.

These strategies can be split into two categories: (i) strategies that do not require any prior knowledge; and (ii) strategies that require prior knowledge based on Machine Learning (ML) techniques. *Castro* et al. [7,6] proposed and evaluated a strategy that used ML to perform static and dynamic thread mapping on TM applications, showing promising results. However, the proposed approach was evaluated against simple non-adaptive thread mapping strategies that do not consider any information from the TM application, TM system and platform. In this article, we perform a more thorough evaluation of these previous works, comparing them to other new adaptive approaches.

Overall, the contributions of this article are:

- We propose two adaptive thread mapping strategies that do not require any prior knowledge from TM applications;
- We propose a new strategy based on association rule learning;
- We extend the work presented in [6] by comparing its performance results to those obtained with those new adaptive strategies;
- We implement all the proposed adaptive strategies in a TM system, so TM applications can benefit from adaptive thread mapping without any source code modification.

The rest of this paper is organized as follows. Section 2 presents the background and motivation for this research. Section 3 describes the proposed adaptive thread mapping strategies. Section 4 discusses the implementation details on a state-of-the-art TM system. Section 5 outlines our experimental methodology while Section 6 presents results. Finally, Section 7 discusses related work and Section 8 concludes the paper and points out future work.

2. Background and motivation

We first present the basic concepts of Transactional Memory in Section 2.1. Then, we discuss thread mapping and motivate this research in Section 2.2.

2.1. Transactional memory (TM)

Transactional Memory is an alternative synchronization solution to classic mechanisms such as locks and mutexes [25,16]. It makes it easier to write parallel programs by providing the programmer with a higher-level abstraction for synchronization, while leaving the implementation of the mechanism that provides this abstraction to the underlying system. Moreover, it provides an efficient model for extracting parallelism from applications [21].

Transactions are portions of code that are executed atomically and in isolation. Concurrent transactions *commit* successfully if their accesses to shared data do not conflict with each other. When one or more concurrent transactions conflict, only one transaction will *commit* whereas the others will *abort* and none of their actions will become visible to other threads [21]. Conflicts can be detected during the execution of transactions when the TM system uses an *eager conflict detection policy* whereas they are detected at commit-time when the system uses a *lazy conflict detection policy*. However, some TM systems also allow lazy transactions to detect conflicts before committing: this may be the case in which one conflicting transaction commits while the other is still running. In this case, the TM system may abort the running transaction due to the conflict with the committing transaction.

The TM system is in charge of re-executing aborted transactions. The choice among the conflicting transactions is done according to the *conflict resolution policies* implemented in the runtime system. Two common alternatives are to squash the transaction that discovers the conflict immediately (*suicide* strategy) or to wait for a time interval before restarting the conflicting transaction (*backoff* strategy).

Transactional Memory can be software-only (STM), hardware-only (HTM) or hybrid (HyTM) [16]. In this article we focus on STM, since it offers flexibility in implementing different mechanisms to detect and resolve conflicts and it does not require any specific hardware. STM allows us to carry out experiments on actual multicore platforms without relying on simulations.