Future Generation Computer Systems 54 (2016) 469–477

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

Time Donating Barrier for efficient task scheduling in competitive multicore systems

Song Wu*, Yaqiong Peng, Hai Jin

Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China

HIGHLIGHTS

- We present *Tidon*, a time donating barrier synchronization mechanism.
- *Tidon* leverages waiting threads to accelerate barrier in competitive environments.
- *Tidon* alleviates the performance degradation of barrier-intensive applications.
- *Tidon* maintains good fairness among co-running applications.
- We implement a prototype of *Tidon* and show its efficiency by experiments.

ARTICLE INFO

Article history:
Received 9 May 2014
Received in revised form
12 December 2014
Accepted 14 April 2015
Available online 3 June 2015

Keywords:
Cloud
Synchronization
Competitive
Scheduling

ABSTRACT

Nowadays, co-locating multithreaded applications on a multicore system has increasingly become a common case in cloud data centers, where multiple threads generally compete for computing resources. These competitive environments may suffer problems of system throughput and fairness caused by barrier operations in multithreaded applications. This is because most implementations of the barrier synchronization are based on the spin-then-block mechanism in which spinning–waiting threads probably waste computing resources and relinquish cores to other co-running applications after they are blocked. This paper attempts to find a new and intuitive way to improve the efficiency of barrier in competitive environments, and answer the question: Can we leverage the timeslices of waiting threads to accelerate barrier operations?

Targeting this question, we propose a novel barrier synchronization mechanism named *Tidon* (Time Donating Barrier). The basic idea of *Tidon* is to donate the timeslices of waiting threads to their preempted, laggard siblings in order to accelerate barrier operations, different from traditional static spinning and blocking. We implement *Tidon* based on the GNU OpenMP runtime library (libgomp) and Linux kernel with new, lightweight system calls. Our evaluation with various sets of co-running applications demonstrates that the advantages of *Tidon* include (1) alleviating the performance degradation of barrier-intensive applications (e.g. improving the performance by up to a factor of 17.9 and 2.3 compared to the default barrier implementation of OpenMP in Completely Fair Scheduler and Balance Scheduling, respectively) while not hurting or even improving the performance of non-barrier-intensive applications, and (2) maintaining good fairness among co-running applications (e.g. improving the fairness by up to a factor of 19.8 and 1.7 compared to the default barrier implementation of OpenMP in Completely Fair Scheduler and Balance Scheduling, respectively).

1. Introduction

Nowadays, cloud data centers generally consist of multicore machines. As the computing resources and memory capacity of multicore machines are abundant, the cloud providers tend to co-locate multiple multithreaded applications on a multicore system, in order to maximize resource efficiency. Lots of multithreaded applications are implemented in the Bulk-Synchronous single-program, multiple-data (SPMD) programming model that has a pattern of computation phases and communication with barrier synchronization [1–4]. Therefore, the performance of multithreaded applications highly depends on barrier operations, which

* Corresponding author.
E-mail address: wusong@hust.edu.cn (S. Wu).

http://dx.doi.org/10.1016/j.future.2015.04.005
0167-739X/© 2015 Elsevier B.V. All rights reserved.

generally use the state-of-the-art spin-then-block mechanism such as Linux futex and the barrier synchronization implemented by GNU OpenMP [5,6].

Unfortunately, schedulers of most mainstream operating systems are unaware of synchronization operations within multithreaded applications in order to maximize overall CPU utilization, and thus the barrier latency could be significantly extended due to preemptions of laggard threads (this paper calls threads that have not reached the barrier as laggard threads) in competitive environments. During the long barrier latency, spinning–waiting threads probably waste computing resources and relinquish cores to other co-running applications after they are blocked. This may significantly aggravate both the system throughput and fairness.

To improve the efficiency of synchronization in competitive environments, co-scheduling [7,3] is a representative approach which allows thread siblings (this paper calls threads from the same application as siblings for each other) to be synchronously scheduled and de-scheduled. Despite its effectiveness in minimizing barrier latency, co-scheduling can cause CPU fragmentation in most realistic situations, leading to deployment impediment [2,8,9]. Balance scheduling is a probabilistic co-scheduling, which dynamically assigns thread siblings to different cores and can perform similarly or better than co-scheduling for the performance of applications without the drawbacks of co-scheduling [8]. However, we show that when the system load is imbalanced, the progress of threads on overloaded cores may be much behind the progress of threads on underloaded cores, and thus the barrier latency may be still long.

This paper, instead of working on underlying scheduling policies, proposes a new and intuitive way to reduce barrier latency by making good use of waiting threads. We present a novel barrier synchronization mechanism named *Tidon* (**Ti**me **don**ating barrier) on the top of time-sharing scheduling policy in mainstream operating systems. The basic idea of *Tidon* is to donate the timeslices of waiting threads to their preempted, laggard siblings. In this way, waiting threads can directly contribute to the completion of barriers.

In summary, this paper makes the following contributions:

- We analyze barrier latency in competitive multicore environments, and its impact on system throughput and fairness with different scheduling policies.
- We propose a barrier mechanism named *Tidon*, which donates the timeslices of waiting threads to their preempted, laggard siblings in order to accelerate barrier operations, so as to reduce the execution time of multithreaded applications in competitive multicore environments.
- We implement *Tidon* based on OpenMP and Linux kernel; the modifications to OpenMP and Linux kernel are lightweight. Evaluation with various sets of co-running applications shows that compared to other alternative policies, *Tidon* can (1) alleviate the performance degradation of barrier-intensive applications while not hurting or even improving the performance of non-barrier-intensive applications, and (2) maintain good fairness among co-running applications.

The rest of the paper is organized as follows. The next section presents further background on our definitive problem and a theoretical analysis. Sections 3 and 4 describe the design and implementation of *Tidon*, respectively. Section 5 provides performance evaluation. Section 6 overviews the related work, and Section 7 concludes the paper.

2. Background and problem analysis

In this section, we first discuss the basics of the barrier synchronization in more detail, and then introduce scheduling policies in competitive environments. Finally, we look into the challenges of the barrier synchronization in competitive environments.

2.1. Barrier basics

A barrier is a synchronization mechanism that ensures no threads can advance beyond a particular point in a computation until all threads have reached that point. Barriers are widely used to synchronize threads in multithreaded applications that exploit fork-join and SPMD parallelism. Barriers can also be used to separate sections of parallel code by parallelizing compilers.

Algorithm 1 The Spin-then-Block Barrier Algorithm

```
Input: The current thread $T$
Output: $T$ returns from the current barrier or is blocked
1: $T$ indicates its arrival by executing a critical section;
2: $pollcount = 0$;
3: repeat
4:    check the shared completion flag;
5:    if all siblings have entered into the barrier then
6:       return
7:    else
8:       $pollcount$++;
9:    end if
10: until $pollcount = Threshold\_Times$
11: $T$ is blocked;
```

Algorithm 1 shows the common barrier algorithm used by most implementations of barrier operations such as pthread and GNU OpenMP et al. The algorithm employs a central counter, and each thread increases the counter when it arrives at the barrier. Each thread first spins on a single, shared completion flag in order to respond to the low-latency barrier quickly and avoid unnecessary context-switches. When the spinning times reach the predefined threshold, the thread is blocked.

2.2. Scheduling policies in competitive environments

Abundant computing resources and memory capacity of multicore machines offer a powerful environment for simultaneously executing multiple multithreaded applications. Most mainstream operating systems, such as Linux, adopt independent time-sharing scheduling policy. With this policy, threads of the same multithreaded application are asynchronously scheduled to cores in competitive environments, in order to maximize overall CPU utilization while maintaining fairness in providing the CPU time to threads. Another scheduling policy under competitive environments is to simultaneously schedule threads of each running application to cores (co-scheduling [7]). It looks like that the multicore system is dedicated to each application during the scheduling quanta of the corresponding application. However, this approach suffers from CPU fragmentation and execution delay, leading to deployment impediment [2,8,9]. As an alternative solution to CPU fragmentation problem, balance scheduling (probabilistic co-scheduling) simply balances thread siblings on different cores instead of forcing the thread siblings to be scheduled simultaneously [8]. As a result, balance scheduling performs similarly or better than co-scheduling for the performance of applications in competitive environments [8].

In the following subsection, we will theoretically analyze barrier latency in competitive environments, and its impact on system throughput and fairness with the above three scheduling policies, respectively.

2.3. Problem analysis

For the convenience of our analysis, we first define some variables as follows:

- P: the total number of cores in the system.
- m: the total number of threads in the system.
- J: A thread.