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

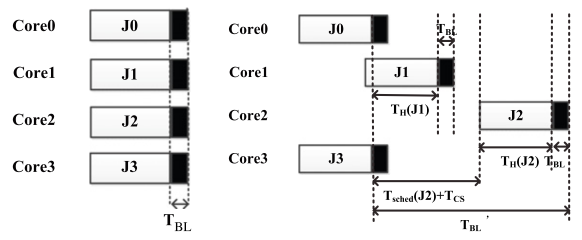
Abundant computing resources and memory capacity of multicore machines offer a powerful environment for simultaneously executing multiple multithreaded applications. Most mainstream operating systems, such as Linux, adopt independent time-sharing scheduling policy. With this policy, threads of the same 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.



(a) Dedicated environments. (b) Competitive environments.

Fig. 1. Barrier latency.

- $w(J)$: the number of share (weight) that J is assigned in the proportional fair share scheduler. We assume that each thread in the system has the same weight value.
- $Received(J, t1, t2)$: the CPU time obtained by J in the interval $[t1, t2)$.
- $Lag(J, t)$: the deviation of the CPU time of J for a period of time t.
- $T_E(J, t)$: the time taken by J to do useful work for a period of time t.
- $T_{WS}(J, t)$: the time taken by J to do useless spinning for a period of time t.
- T_{BL}: the barrier latency in the dedicated mode.
- T'_{BL}: the barrier latency in competitive environments.
- $Run(J)$: indicating whether J is preempted at the start of a barrier (1: not preempted; 0: preempted).
- $T_{sched}(J)$: the amount of time that a preempted, laggard thread J waits to be scheduled (not including the amount of time that scheduler switches to the context of J) from the start of a barrier.
- T_{CS}: the overhead of a context-switches operation.
- $T_H(J)$: if $Run(J) = 0$, indicating the time distance between when J is scheduled to run next and when it reaches a barrier; otherwise, indicating the time distance between the start of a barrier and when J reaches a barrier.
- $T_{cpt}(J)$: the time distance between the start of a barrier and that J reaches and completes a barrier in competitive environments.

An ideal fair scheduling strategy can guarantee that the CPU time consumed by a thread is strictly proportional to its weight. The deviation of the actual CPU time consumed by a thread in the interval $[0, t)$ is used to evaluate the fairness of the scheduling strategy. It is defined as follows:

$$Lag(J, t) = t \times P \times w(J) - Received(J, 0, t). \tag{1}$$

This paper focuses on the barrier synchronization. Therefore, we assume that wasteful works performed by a thread only include spinning operations. The relation of $Received(J, 0, t)$, $T_E(J)$, $T_{WS}(J)$ can be given by this equation:

$$Received(J, 0, t) = T_E(J) + T_{WS}(J). \tag{2}$$

According to Eqs. (1) (2), we can arrive at:

$$T_E(J) = t \times P \times w(J) - Lag(J, t) - T_{WS}(J). \tag{3}$$

Normally, an independent time-sharing thread scheduler, such as CFS in Linux, allows thread siblings to be scheduled to run on any cores. The scheduler can delay the execution of a laggard thread with preemption, thereby extending the barrier latency. We assume that a multithreaded application exhibits a good load balance, and thus thread siblings reach and complete a barrier simultaneously in the dedicated mode. As shown in Fig. 1, we can arrive at:

$$T_{cpt}(J) = \begin{cases} T_{sched}(J) + T_{CS} + T_H(J) + T_{BL}, & \text{if } Run(J) = 0 \\ T_H(J) + T_{BL}, & \text{if } Run(J) = 1 \end{cases} \tag{4}$$

$$T'_{BL} = \max_J \{T_{cpt}(J)\}. \tag{5}$$

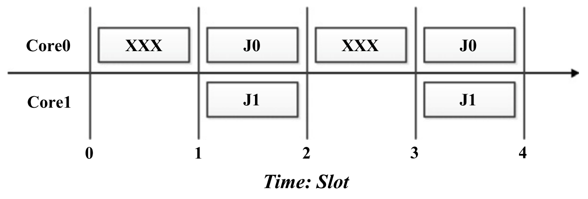


Fig. 2. CPU fragmentation in co-scheduling.

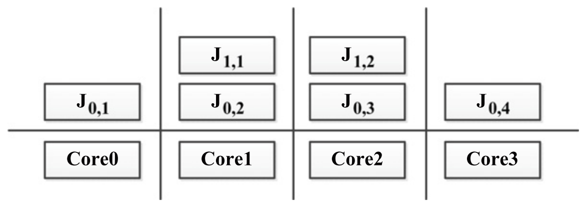


Fig. 3. A scenario of thread-to-core assignment with balance scheduling when P is not a factor of m.

From Eqs. (4) (5), we can observe that barrier latency can be extended significantly when preemptions of laggard threads happen. During the long barrier latency, a waiting thread J performs a significant amount of useless spinning before being blocked (increasing $T_{WS}(J)$) which lowers system throughput, and cannot be scheduled to make progress after being blocked. Moreover, schedulers like CFS in Linux do not supply enough CPU time for threads with long sleeping time when they become runnable (increasing $Lag(J, t)$), in order to prevent a long-sleeping threads from starving others. Therefore, according to Eq. (3), applications with barrier operations may suffer from unfair performance degradation in competitive environments.

Co-scheduling is an attractive approach to minimizing barrier latency (eliminating $T_{sched}(J)$ and T_{CS} in Eq. (4)). In fact, most realistic execution environments suffer from imbalanced load such as when P is not a factor of m. In these cases, the restricted requirement of synchronous progress can cause CPU fragmentation [2,8–10]. Fig. 2 shows an example for the CPU fragmentation problem. $J0$ and $J1$ cannot be scheduled until slot 1, although both become runnable at slot 0 because there is only one core idle at slot 0 (increasing $Lag(J0, t)$ and $Lag(J1, t)$).

Balance scheduling assigns thread siblings onto different cores. Therefore, it can reduce barrier latency (reducing $T_{cpt}(J)$ in Eq. (4)) compared to independent time-sharing scheduling, when P is a factor of m. But in most realistic situations, P is not a factor of m. Fig. 3 shows a scenario: a 4-threaded (J_0) and 2-threaded (J_1) application co-run on a 4-core machine. Because each thread from J_0 occupies a different core respectively, all threads ($J_{1,1}$ and $J_{1,2}$) from J_1 are always assigned to overloaded cores (Cores 1 and 2). The application J_1 may be significantly slowed down due to high scheduling latency (reducing $Received(J_{1,1}, 0, t)$ and $Received(J_{1,2}, 0, t)$). Moreover, most operating systems use lazy algorithms to balance global loads in order to maintain warm hardware state, and thus CPU loads could be temporarily imbalanced between the invocations of the load balancer, even in the off-by-one imbalance [4] (dynamically guaranteeing that the number of threads on each core is within one for each other), which is closest to balanced load. Therefore, $T_{cpt}(J_{0,2})$ and $T_{cpt}(J_{0,3})$ tend to be a large value due to that $J_{0,2}$ and $J_{0,3}$ are running on overloaded cores while $J_{0,1}$ and $J_{0,4}$ are running on underloaded cores.

In summary, although co-scheduling is effective in reducing barrier latency, it introduces CPU fragmentation problem which increases $Lag(J, t)$ of each thread J in systems. $T_{cpt}(J)$ of each thread J may tend to a large value with both the independent time-sharing scheduling and balance scheduling, so as to extend barrier latency. During long barrier latency, spinning–waiting threads waste computing resources (increasing $T_{WS}(J)$), and probably cannot retain fair computing resources to their applications after being blocked (increasing $Lag(J, t)$). According to Eq. (3), applications with barrier operations may suffer from unfair performance degradation and lower the overall system throughput. Can we make good use of waiting threads to reduce $T_{cpt}(J)$ of each laggard sibling J so as to accelerate barrier operations, while not increasing $Lag(J, t)$? This paper presents *Tidon*.

3. Time Donating Barrier

In this section, we introduce the design of *Tidon* in detail, and provide an analysis in terms of system throughput and fairness with *Tidon*.

3.1. Basic ideas

Our motivation is to reduce barrier latency in competitive environments, aiming at lowering the chance that waiting threads waste computing resources and relinquish computing resources to other applications without changing the underlying scheduling policy in mainstream operating systems. Therefore, *Tidon* still adopts independent time-sharing policy to maintain fairness in providing the CPU time to threads while maximizing overall CPU utilization.

Algorithm 2 overviews *Tidon*. We adopt the concept of *Barrier Participation Status* (BAPS) to reflect whether a thread can make progress. The default BAPS value of each thread is FALSE, which indicates that the corresponding thread is doing useful work. With *Tidon*, the current thread T entering into the barrier first notifies the OS to update its BAPS value. Then T polls a memory location for checking whether all siblings have entered into the barrier, in order to respond to the completion of the barrier quickly and avoid unnecessary context-switches (lines 4–6). If some siblings have not reached the barrier, *Tidon* applies time donating policy (line 8), which will be described in the next subsection.

Algorithm 2 The Overview of *Tidon*

```
Input: The current thread $T$
Output: $T$ returns from the current barrier
 1: $T$ indicates its arrival by executing a critical section;
 2: update its BAPS;
 3: repeat
 4:     check the shared completion flag;
 5:     if all siblings have entered into the barrier then
 6:         return
 7:     else
 8:         Apply time donating policy;
 9:     end if
10: until $T$ returns
```

The interactions between the user-level synchronization and OS (lines 2 and 8) are through two new, lightweight system calls implemented by us.

3.2. Time donating policy

Algorithm 3 describes the time donating policy within *Tidon*. The core idea is to donate timeslices of waiting threads to their preempted, laggard siblings with the goal to accelerate the progress of laggard siblings. The procedure is: If the BAPS value of the current thread T is FALSE, it means all threads have entered into the barrier (refer to the description of the system call *update_barr_status* in the next section), and thus the current thread T returns to the user-level code (lines 1–3). Else, it means T cannot make forward progress. And then *Tidon* traverses the siblings of T and schedule a preempted, laggard sibling T_s to the core C, using the remaining timeslice offered by T (lines 4–9). The OS judges whether a thread is laggard through its BAPS value. In this way, preempted, laggard threads can make forward progress timely with the donated CPU time from waiting threads.

Algorithm 3 Time Donating Policy

```
Input: $T$: the current thread; $C$: the core that the thread $T$ is running on;
Output: Apply Time Donating Policy
 1: if The BAPS value of the thread $T$ is FALSE then
 2:     return
 3: end if
 4: for each sibling thread $T_s$ of $T$ do
 5:     if $T_s$ is preempted and its BAPS value is FALSE then
 6:         inform the scheduler to schedule $T_s$ to the core $C$, and
            allocate the remaining timeslice of $T$ to $T_s$
 7:         return
 8:     end if
 9: end for
10: run the thread next to $T$ according to the scheduling decision
    of OS scheduler
```

3.3. Analysis

This subsection provides an analysis in terms of system throughput and fairness with *Tidon*.

As analyzed in Section 2.3, reducing $Lag(J, t)$ and $T_{cpt}(J)$ of each thread J can alleviate unfair performance degradation of applications with barrier operations and loss of system throughput. With *Tidon*, waiting threads continue to obtain CPU time according to the underlying fair policy, and donate the CPU time to their siblings. Therefore, $Lag(J, t)$ is almost eliminated.

In order to analyze the impact of *Tidon* on $T_{cpt}(J)$, we define $T'_{sched}(J)$, $T'_H(J)$ and $T'_{cpt}(J)$ with *Tidon* corresponding to $T_{sched}(J)$, $T_H(J)$ and $T_{cpt}(J)$ defined in Section 2, respectively. Then, we can arrive at:

$$T'_{cpt}(J) = \begin{cases} T'_{sched}(J) + T_{CS} + T'_H(J) + T_{BL}, & \text{if } Run(J) = 0 \\ T'_H(J) + T_{BL}, & \text{if } Run(J) = 1. \end{cases} \tag{6}$$

For each laggard thread J that gets extra CPU time donated from waiting threads to make forward progress, it is clear that $T'_{sched}(J)$ and $T'_H(J)$ will be less than $T_{sched}(J)$ and $T_H(J)$, respectively. Therefore, $T'_{cpt}(J)$ is less than $T_{cpt}(J)$. The more threads reach and complete the barrier turning into waiting threads, the more CPU time can be donated to accelerate the progress of remaining laggard threads, so as to improve the efficiency of barrier in competitive environments.

4. The implementation of *Tidon*

We have implemented *Tidon* based on GNU OpenMP runtime library (libgomp) with the support from the OS. We choose libgomp and Linux because of their broad acceptance and the availability of their opensource code.

Tidon relies on two new system calls: *update_barr_status* and *time_donating_to*. These two system calls require only slight changes to the Linux kernel (about 100 lines of code in Linux kernel 3.6.1). The system call *update_barr_status* is used to update the BAPS values. If all siblings of the current thread have entered into the barrier, the BAPS value of each thread from the application (which the current thread belongs to) is reset to FALSE. Otherwise, the BAPS value of the current thread is set as TRUE. As Linux does not correlate threads from the same application, we introduce our implementation that guarantees threads from the same application to find each other quickly before the system call *time_donating_to*.

4.1. How to find siblings from the same application

Generally, threads are created by calling *do_fork* in Linux. The OS judges whether the new task is a thread through the parameter *clone_flags*. *Tidon* adds two member variables: *main_thr* and *thread_list* to the data structure *task_struct* associated with each task in the Linux kernel. If a newly created task p is a thread, it will be added into the member variable *thread_list* of the current thread T, and the variable *main_thr* of p will point to T. Otherwise, *Tidon* treats p as the main thread. The variable *thread_list* of the main thread stores all of its cloned threads. Therefore, each thread can access the list of siblings in $O(1)$ complexity.

4.2. How to implement time donation based on CFS

The system call *time_donating_to* is used to donate the remaining timeslice of the caller to a designated thread. Its implementation is based on the default scheduler in Linux, that is, the Completely Fair Scheduler (CFS) [11]. CFS is designed to try its best to maximize overall CPU utilization while maintaining balance (fairness) in providing the CPU time to threads. To determine the balance, CFS uses a single time-based red–black tree for each core to track all the runnable threads indexed by their *virtual runtime* which indicates the amount of time the regarding thread has run on the core. The smaller a thread's virtual runtime is, the higher its need for the core. Threads with lower *virtual runtime* are stored toward the left side of the tree, and threads with the higher *virtual runtime* are stored toward the right side of the tree. The scheduler generally picks the left-most node of the tree to schedule next to maintain fairness in every scheduling event. The preempted thread updates its *virtual runtime* and is then inserted back into the tree if it is still runnable.

CFS dynamically calculates a timeslice for each runnable thread in every scheduling event. Therefore, it is more difficult to determine the remaining length of the timeslice for a thread in CFS than static-timeslice-based schedulers [12,13]. Inspired by the implementation of *yielding* mechanism in the previous work [9], we exchange the *virtual runtime* values of a thread A and thread B in order to allocate the remaining timeslice of A to B. However, this mechanism can reduce the CPU time obtained by B compared to the original scheduling path in CFS. Therefore, we further optimize the mechanism by exchanging the *virtual runtime* values (after being updated) of A and B again when B uses up the timeslice. At the same time, we should also change the position of the thread A in its red–black tree.

With the support of two new system calls mentioned above, *Tidon* modifies codes concerned with the implementation of barriers in the GNU OpenMP.

5. Performance evaluation

With the implementation of *Tidon*, we carry out our experiments on a machine consisting of two eight-core 2.6 GHz Intel Xeon E5-2670 chips with hyper-threading disabled. We use Redhat Enterprise Linux 6.2 with the kernel version 3.6.1 and OpenMP 3.0. In this section, we first introduce characteristics of the selected benchmarks and our experimental methodology, then present the experimental results.

5.1. Benchmarks

Table 1 describes the benchmarks we select. The problem size of benchmarks from NPB 3.3 is configured as *Class A*, and the input set of benchmarks from PARSEC 2.1 is *simmedium* data set. Fig. 4 shows the barrier behavior of the OpenMP implementations for parallel benchmarks in our testbed. Based on the average inter-barrier interval, we divide parallel benchmarks into three classes: fine-grained, medium-grained and coarse-grained. We can see that both the fine-grained and medium-grained applications are barrier-intensive.

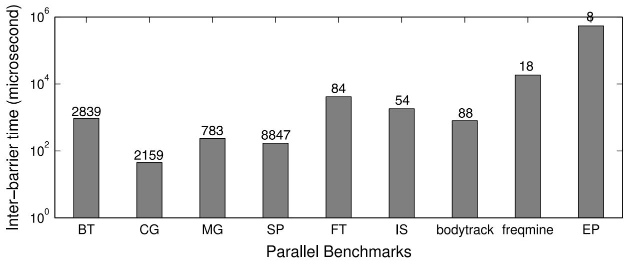


Fig. 4. The height of the bars indicates the average time between two barriers, while the labels beyond bars show the number of executed barriers.

Table 1
Benchmarks.

Classes	Benchmarks	Benchmark suites
Fine-grained	BT	NPB 3.3 (OpenMP version)
	CG	
	MG	
	SP	
Medium-grained	FT	
	IS	
	Bodytrack	PARSEC 2.1
Coarse-grained	Freqmine	
	EP	NPB 3.3 (OpenMP version)

Table 2
Co-runners.

Co-runners	Load configurations
CG IS EP	1, 2, 4, 8, 16 threads

5.2. Experimental methodology

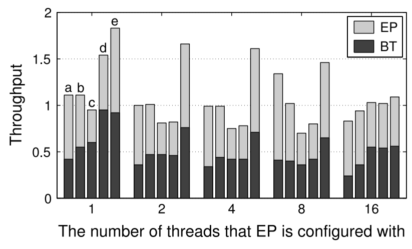
The OpenMP runtime system provides a tunable barrier implementation controlled by OMP_WAIT_POLICY environment variable. With the default setting, threads suffering a barrier poll for a period of time before sleeping. When OMP_WAIT_POLICY is set as passive, threads sleep immediately. We compare *Tidon* with the following configurations:

- CFS-De: OMP_WAIT_POLICY = default with CFS scheduler in Linux.
- CFS-Pa: OMP_WAIT_POLICY = passive with CFS scheduler in Linux.
- BS-De: OMP_WAIT_POLICY = default with balance scheduling.
- BS-Pa: OMP_WAIT_POLICY = passive with balance scheduling.

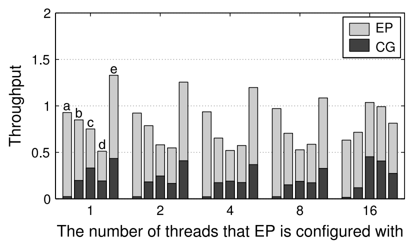
We conduct several experiments to compare *Tidon* with the above configurations, in order to answer the question: What is the performance comparison in terms of throughput and fairness?

Recent research has shown that the determining behavioral factor when parallel applications share a system is the granularity of synchronization operations, and the relatively fine-grained applications are likely to suffer from more performance degradation [14].

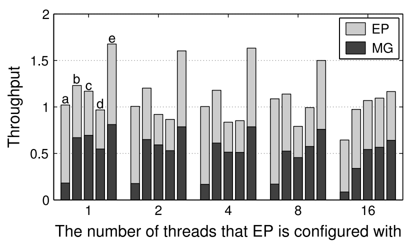
Therefore, we first co-run each parallel benchmark (main workload) with the most coarse-grained application EP (co-runner) in order to observe the performance degradation for each class of parallel applications in the most worse case. In order to coverage different types of co-running, we also conduct experiments with other co-runners, which are shown in Table 2. Among these co-runners, CG, IS, and EP represent fine-grained, medium-grained



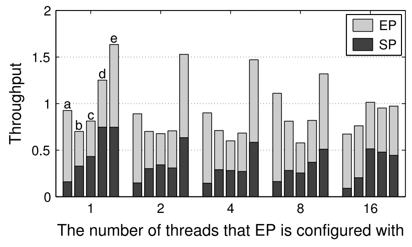
(a) BT.



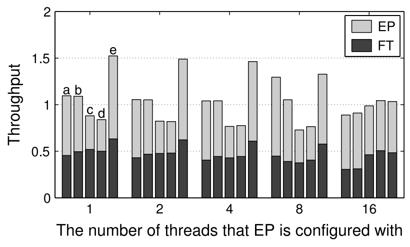
(b) CG.



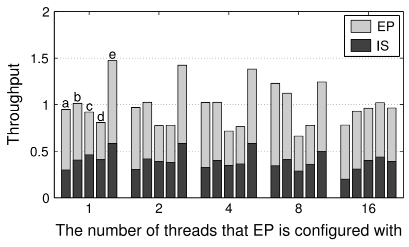
(c) MG.



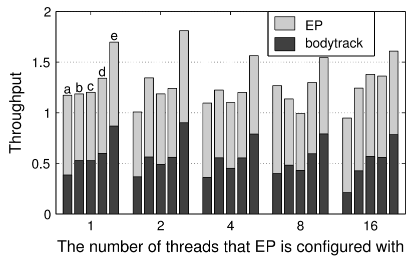
(d) SP.



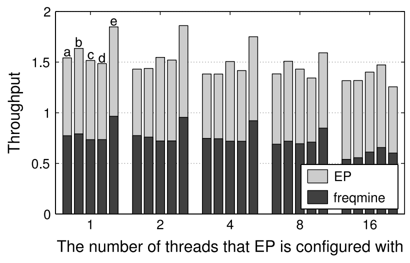
(e) FT.



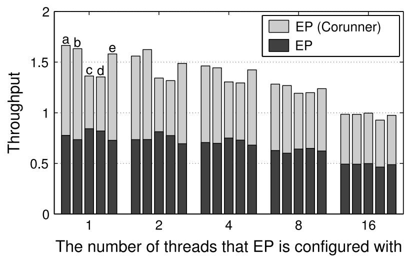
(f) IS.



(g) Bodytrack.



(h) Freqmine.



(i) EP.

Fig. 5. Throughput of co-running each parallel workload and EP with a breakdown of their speedups (a: CFS-De, b: CFS-Pa, c: BS-De, d: BS-Pa, e: *Tidon*).

and coarse-grained workload, respectively. In each co-running, the number of threads that the main workload configured with is set as 16, equal to the number of cores. We run each workload repeatedly in order to fully overlap their executions, and compute their speedups relative to solorun of each workload with the default barrier implementation of OpenMP.

To investigate system throughput and unfairness with different approaches, we use the weighted speedup [15] to measure system throughput, which is the sum of the speedups of corresponding benchmarks. The unfairness metric is defined as the ratio between the largest and smallest speedup among the co-running benchmarks [9,16].

5.3. Performance comparison in terms of throughput and fairness

Fig. 5 shows the throughput of co-running parallel workloads and EP with different number of threads, which uses the weighted speedup as its metric. The weighted speedup indicates the throughput of co-running benchmarks relative to that of the situation when the co-running benchmarks are run consecutively. We can also observe the unfairness between each main workload and EP in Fig. 5 (the difference between their speedups). As mentioned in Section 2.3, when the number of cores is not a factor of the total number of threads, it intrinsically incurs load imbalance. Therefore, the system load is imbalanced when EP is configured with 1, 2, 4, and 8 threads. We will discuss results of both balanced and imbalanced system loads in turn.

5.3.1. Imbalanced cases

When the system load is imbalanced, the first thing to note is that CFS-De can generally achieve good throughput compared to CFS-Pa, BS-De and BS-Pa, but barrier-intensive applications (BT, CG, MG, SP, FT, IS and bodytrack) suffer from extremely unfair performance degradation. For example, when CG and EP (configured with two threads) co-run, compared to other methods (except *Tidon*), the throughput of CFS-De is improved by over 17.3% but the performance of CG is reduced significantly by over 86.2%. The performance unfairness between CG and EP is 39.5. The reason is that spinning–waiting threads delay the execution of useful siblings which extend the barrier latency significantly and has a negative impact on the performance of barrier-intensive applications. Compared to CFS-De, waiting threads with CFS-Pa yield computing resources immediately, and thus alleviate the performance degradation of barrier-intensive applications to some extent.

The second thing to note is that although both BS-De and BS-Pa significantly resolves the performance degradation of barrier-intensive applications compared to CFS-De, CFS-De outperforms them in terms of throughput nearly in all cases. As analyzed in Section 2.3, all threads from EP are assigned to overloaded cores with balance scheduling, while all threads from both the main workload and EP have the same probability of being assigned to the underloaded cores with CFS. Therefore, EP is significantly slowed down due to high scheduling latency on the overloaded cores, which outweighs the relative performance improvement of the main workload, so as to lower the system throughput.

The most important thing to note is that *Tidon* outperforms other approaches in terms of throughput in all cases except co-running two EP, by alleviating the performance degradation of barrier-intensive applications while not hurting or even improving the performance of EP. For example, when BT and EP (configured with two threads) co-run, the performance of BT is improved by

59%–110%, and the performance of EP is improved by 40%–163%, so as to improve the throughput by 64%–102%, compared to another four approaches. This is because *Tidon* enhances the benefits of CFS in system throughput by accelerating the completion of barrier synchronization with time donation, so as to save valuable computing resources and attain a fair amount of CPU time for both the barrier-intensive and non-barrier-intensive applications. For the co-running of two EP, the overhead of extra context-switches probably outweighs the benefits from time donation, since there are few barrier operations in EP. Nevertheless, the performance gap is only within 9% compared to another four approaches.

5.3.2. Balanced cases

In balanced cases, balance scheduling nearly outperforms other approaches including *Tidon* in terms of the system throughput. The reason is that as the progress of each thread is almost the same with the probabilistic co-scheduling feature of balance scheduling when the system load is balanced, barrier operations can be completed quickly. *Tidon* can match the throughput of balance scheduling in most cases (achieving 78.5%–96% and 81.7%–99.3% throughput of BS-De and BS-Pa, respectively), or even outperforms it in some cases: BT, MG and bodytrack (6.2%–16.7% and 7.5%–18% throughput improvement over BS-De and BS-Pa, respectively). Moreover, we can observe that both the balance scheduling and *Tidon* can provide a good fairness guarantee between each main workload and EP in balanced cases.

5.4. Experiments with different classes of co-runners

Fig. 6 compares the average throughput and unfairness of co-running each parallel benchmark with different classes of co-runners of *Tidon* against another four approaches. The results can be summarized as the following three types:

- **Type 1** *Tidon* outperforms other approaches in terms of both the average system throughput and fairness by 10%–97% and 1%–510%, respectively: EP (1 and 2 threads), IS (1, 4, 8 and 16 threads) and CG (1, 2 and 4 threads)
- **Type 2** *Tidon* outperforms other approaches in terms of the average system throughput by 1%–74%, and only slightly reduces the average fairness compared to BS-De and (or) BS-Pa by 2.3%–6.3%: EP (4, 8 and 16 threads) and IS (2 threads)
- **Type 3** *Tidon* outperforms other approaches in terms of the average system fairness by 4%–459%, and only slightly reduces the system throughput compared to CFS-De or BS-De by 2%–3%: CG (8 and 16 threads).

We can observe that CFS tends to achieve poor system fairness. Balance scheduling tends to achieve poor system throughput when the system load is imbalanced. *Tidon* enhances the benefits of CFS in system throughput by accelerating the progress of applications with barrier operations. Therefore, most cases of co-running various benchmarks with *Tidon* belong to type 1. In type 2 or 3, the improvement of system throughput (or fairness) outweighs the slight reduction in system fairness (or throughput). In summary, *Tidon* can make a better trade-off between system throughput and fairness.

6. Related work

Our work is related to the research in the synchronization and scheduling. We briefly discuss the most related work in turn.

6.1. Synchronization technique

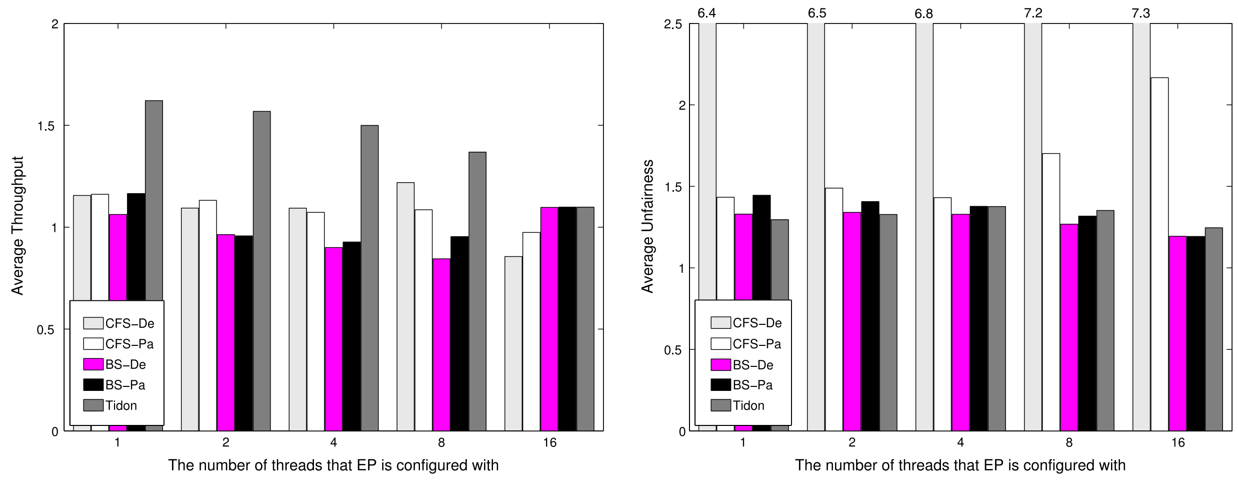
There are a number of works aiming at optimizing barrier algorithms by reducing lock contention [17,18] (line 1 as shown in Algorithm 1). Unlike this, the present paper aims at accelerating the progress of threads toward the barrier in competitive environments. In addition, the barrier performance can be also improved by replacing the lock-based part with wait-free synchronization [19], which is orthogonal to our work. Martínez proposes a speculative barrier that makes speculative threads execute past active barrier operations instead of waiting [20]. We think it is an effective approach for barrier synchronization in competitive environments, but threads must roll back to the synchronizing point when access conflicts are detected. The work between the synchronizing point and rollback point performed by a speculative thread wastes computing resources, which is similar to the problem of the barrier based on spin-then-block. In addition, the speculative barrier needs new hardware support.

6.2. Scheduling for parallel applications

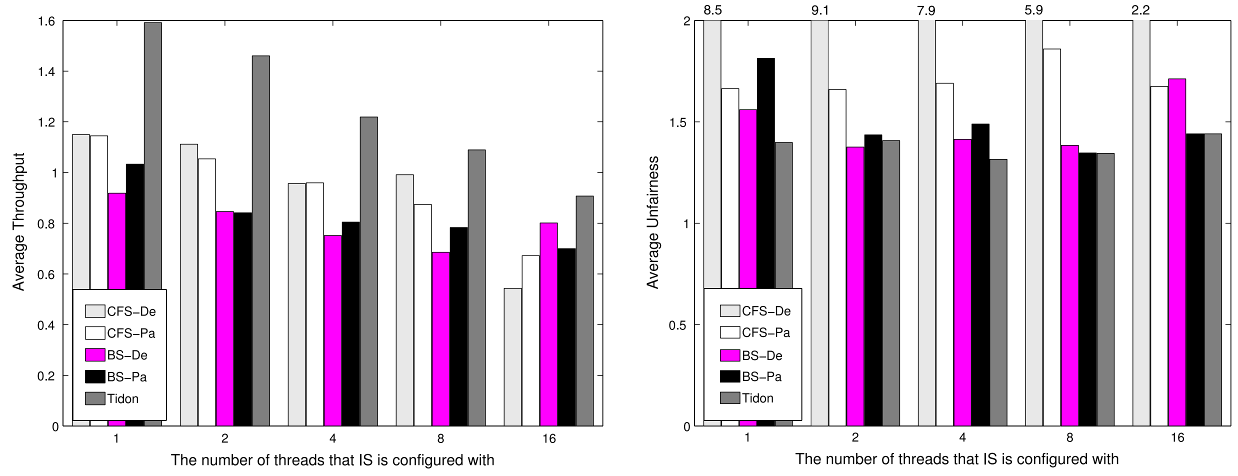
The shift from single-core to multicore has shifted the focus of research from executing one non-parallel application on a machine to (1) simultaneously executing non-parallel applications [16,21], (2) running one parallel application [22,23], (3) multiprogrammed workloads that include parallel applications [24,4, 25–27]. The third case has been prevailing execution environments in today's cloud data centers. Meanwhile, performance of parallel applications is limited by a variety of bottlenecks, e.g. critical sections, barriers and slow pipeline stages [28]. Previous research has shown that the determining behavioral factor is the granularity of synchronization operations in competitive environments, and fine-grained applications are more likely to see performance deterioration [14].

In order to improve the efficiency of the synchronization in competitive environments, early study presents gang scheduling (co-scheduling) [7]. It is a promising idea to ameliorate the synchronization for the parallel application when the overall workload is load-balanced and makes full use of the machine. However, these conditions are rarely met and most realistic workloads consequently suffer from both internal and external fragmentation, leaving resources and processors idle [2,8,9]. As an alternative solution to CPU fragmentation problem, demand-based co-scheduling is a common variant of co-scheduling [2,29,10], which only initiates co-scheduling for synchronizing threads. However, the progress of siblings toward a barrier may have been much different at the point of initiating co-scheduling, especially when the system load is imbalanced. Our evaluation of *Tidon* provides encouraging evidence that it may alleviate some of the need for co-scheduling.

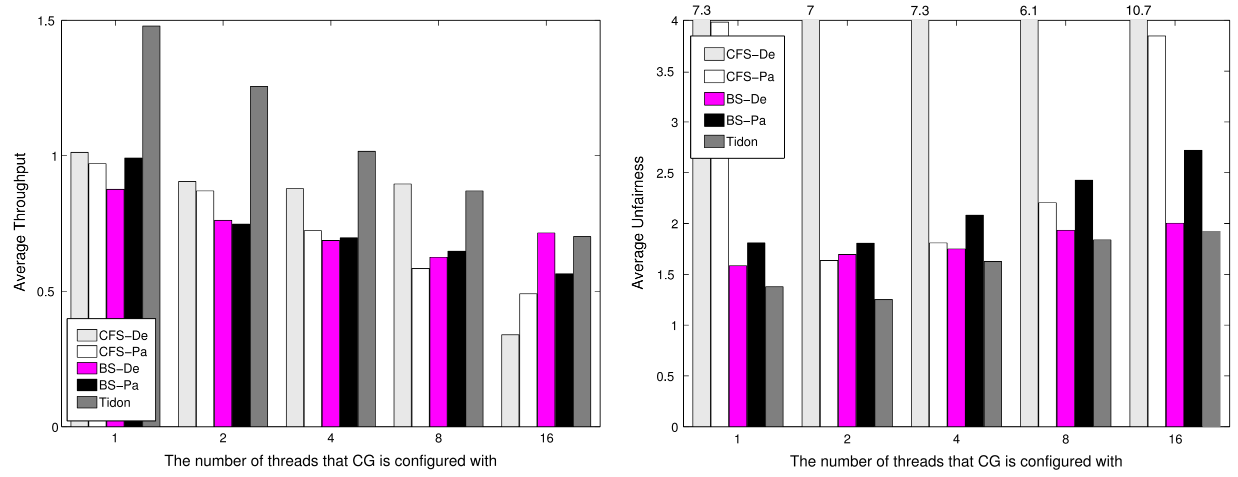
In addition, the work-stealing programming model with dynamic load balancing has been demonstrated to be a powerful and effective approach to achieving good performance of parallel applications in competitive environments [30]. In fact, work stealing is a double-edged sword because the unsuccessful steals of thieves probably waste computing resources [9]. Although Ding et al. proposes a solution named Balanced Work Stealing (BWS) to this problem [9], the overhead along with thread creation and d-e-que management of work-stealing can be still very high in some applications. Adaptive thread creation mechanism is proposed to alleviate these issues [31,32]. It is very effective to improve the efficiency of work-stealing in the dedicated mode. But in competitive environments, the execution of thieves may delay the execution of busy workers that would generate new threads, causing that no workers make useful progress. Our time donating policy can be considered to accelerate the generation of new threads by donating timeslices of thieves to busy workers, in order to enhance the multiprogrammed performance of work-stealing with adaptive thread creation.



(a) EP.



(b) IS.



(c) CG.

Fig. 6. Average throughput and unfairness in different system loads.

7. Conclusion

In this paper, we analyze barrier latency with different scheduling policies in competitive multicore environments, and its impact on system throughput and fairness. Then we present *Tidon*, which is a new and intuitive way to accelerate barrier operations by making good use of waiting threads. Our results indicate that *Tidon* can alleviate the performance degradation of barrier-intensive applications while not hurting or even improving the performance of non-barrier-intensive applications. Moreover, *Tidon* can maintain good fairness among co-running applications.

In the future, we plan to enhance the effectiveness of *Tidon* in reducing barrier latency. In the current version of *Tidon*, waiting threads simply accelerate the progress of preempted, laggard siblings without considering which ones are more urgent to be accelerated. The slower a thread's progress toward a barrier, the more urgent it is to be accelerated. A possible approach is to develop a prediction algorithm which predicts the progress of each thread, and preferentially to accelerate the most urgent thread. We also plan to explore the potential benefits of the time donating policy within *Tidon* in alleviating other bottlenecks of parallel applications in competitive environments. For example, the time donating policy within *Tidon* can be used to improve the efficiency of the general synchronization operations which may involve only a part of threads from an application, speculative barrier and work-stealing with adaptive thread creation in competitive environments as mentioned in Section 6. Moreover, *Tidon* can be easily ported to barrier implementations of other parallel runtime libraries such as pthread and Cilk + + by slight changes to codes concerned with the implementation of barriers.

Acknowledgments

The research was supported by National Science Foundation of China under Grant Nos. 61232008 and 61472151, National 863 Hi-Tech Research and Development Program under Grant No. 2014AA01A302, Research Fund for the Doctoral Program of MOE under Grant No. 20110142130005.

References

[1] A.C. Dusseau, R.H. Arpaci, D.E. Culler, Effective distributed scheduling of parallel workloads, in: Proc. SIGMETRICS'96, 1996, pp. 25–36.
[2] E. Frachtenberg, D.G. Feitelson, F. Petrini, J. Fernandez, Adaptive parallel job scheduling with flexible coscheduling, IEEE Trans. Parallel Distrib. Syst. 16 (11) (2005) 1066–1077.
[3] T. Jones, Linux kernel co-scheduling for bulk synchronous parallel applications, in: Proc. ROSS'11, 2011, pp. 57–64.
[4] S. Hofmeyr, J.A. Colmenares, C. Iancu, J. Kubiatowicz, Juggle: proactive load balancing on multicore computers, in: Proc. HPDC'11, 2011, pp. 3–14.
[5] F.R. Johnson, R. Stoica, A. Ailamaki, T.C. Mowry, Decoupling contention management from scheduling, in: Proc. ASPLOS'10, 2010, pp. 117–128.
[6] K.K. Pusukuri, R. Gupta, L.N. Bhuyan, No more backstabbing... a faithful scheduling policy for multithreaded programs, in: Proc. PACT'11, 2011, pp. 12–21.
[7] D.G. Feitelson, L. Rudolph, Gang scheduling performance benefits for fine-grain synchronization, J. Parallel Distrib. Comput. 16 (4) (1992) 306–318.
[8] O. Sukwong, H.S. Kim, Is co-scheduling too expensive for SMP VMs?, in: Proc. EuroSys'11, 2011, pp. 257–272.
[9] X. Ding, K. Wang, P.B. Gibbons, X. Zhang, BWS: balanced work stealing for time-sharing multicores, in: Proc. EuroSys'12, 2012, pp. 365–378.
[10] H. Kim, S. Kim, J. Jeong, J. Lee, S. Maeng, Demand-based coordinated scheduling for SMP VMs, in: Proc. ASPLOS'13, 2013, pp. 369–380.
[11] Modular scheduler core and completely fair scheduler [CFS] http://lwn.net/Articles/230501/.
[12] D.R. Engler, M.F. Kaashoek, J. O'Toole Jr., Exokernel: An operating system architecture for application-level resource management, in: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, SOSP'95, ACM, New York, NY, USA, 1995, pp. 251–266.
[13] C.A. Waldspurger, W.E. Weihl, Stride scheduling: Deterministic proportional-share resource management, Tech. Rep., Cambridge, MA, USA, 1995.
[14] C. Iancu, S. Hofmeyr, F. Blagojevic, Y. Zheng, Oversubscription on multicore processors, in: Proc. IPDPS'10, 2010, pp. 1–11.
[15] A. Snavely, D.M. Tullsen, Symbiotic jobscheduling for a simultaneous multithreaded processor, in: Proc. ASPLOS-IX, 2000, pp. 234–244.
[16] D. Xu, C. Wu, P.-C. Yew, J. Li, Z. Wang, Providing fairness on shared-memory multiprocessors via process scheduling, in: Proc. SIGMETRICS'12, 2012, pp. 295–306.
[17] M.L. Scott, J.M. Mellor-Crummey, Fast, contention-free combining tree barriers for shared-memory multiprocessors, Int. J. Parallel Program. 22 (4) (1994) 449–481.
[18] L. Cheng, J. Carter, Fast barriers for scalable ccNUMA systems, in: Proc. ICPP'05, 2005, pp. 241–250.
[19] S. Al Bahra, Nonblocking algorithms and scalable multicore programming, Commun. ACM 56 (7) (2013) 50–61.
[20] J.F. Martínez, J. Torrellas, Speculative synchronization: applying thread-level speculation to explicitly parallel applications, SIGPLAN Not. 37 (10) (2002) 18–29.
[21] K. Tian, Y. Jiang, X. Shen, A study on optimally co-scheduling jobs of different lengths on chip multiprocessors, in: Proc. CF'09, 2009, pp. 41–50.
[22] E.Z. Zhang, Y. Jiang, X. Shen, Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs?, in: Proc. PPoPP'10, 2010, pp. 203–212.
[23] D. Tam, R. Azimi, M. Stumm, Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors, SIGOPS Oper. Syst. Rev. 41 (3) (2007) 47–58.
[24] M. Bhadauria, S.A. McKee, An approach to resource-aware co-scheduling for CMPs, in: Proc. ICS'10, 2010, pp. 189–199.
[25] L. Tang, J. Mars, N. Vachharajani, R. Hundt, M.L. Soffa, The impact of memory subsystem resource sharing on datacenter applications, in: Proc. ISCA'11, 2011, pp. 283–294.
[26] L.Y. Chen, D. Ansaloni, E. Smirni, A. Yokokawa, W. Binder, Achieving application-centric performance targets via consolidation on multicores: myth or reality?, in: Proc. HPDC'12, 2012, pp. 37–48.
[27] H. Sasaki, T. Tanimoto, K. Inoue, H. Nakamura, Scalability-based manycore partitioning, in: Proc. PACT'12, 2012, pp. 107–116.
[28] J.A. Joao, M.A. Suleman, O. Mutlu, Y.N. Patt, Bottleneck identification and scheduling in multithreaded applications, in: Proc. ASPLOS XVII, 2012, pp. 223–234.
[29] C. Weng, Q. Liu, L. Yu, M. Li, Dynamic adaptive scheduling for virtual machines, in: Proc. HPDC'11, 2011, pp. 239–250.
[30] R.D. Blumofe, D. Papadopoulos, The performance of work stealing in multi-programmed environments, in: ACM SIGMETRICS Performance Evaluation Review. Vol. 26, ACM, 1998, pp. 266–267.
[31] T. Hiraishi, M. Yasugi, S. Umatani, T. Yuasa, Backtracking-based load balancing, in: ACM Sigplan Notices. Vol. 44, ACM, 2009, pp. 55–64.
[32] L. Wang, H. Cui, Y. Duan, F. Lu, X. Feng, P.-C. Yew, An adaptive task creation strategy for work-stealing scheduling, in: Proc. CGO'10, 2010, pp. 266–277.

Song Wu is a Professor of computer science and engineering at Huazhong University of Science and Technology (HUST) in China. He received his Ph.D. from HUST in 2003. He is now the Director of Parallel and Distributed Computing Institute at HUST. He has also served as the Vice Director of Service Computing Technology and System Lab (SCTS) and Cluster and Grid Computing Lab (CGCL) of HUST. His current research interests include grid/cloud computing and virtualization technology.

Yaqiong Peng received her B.S. (2010) degree from Huazhong University of Science and Technology (HUST) in China. Currently she is a Ph.D. candidate of Computer Science and Technology at Huazhong University of Science and Technology (HUST). Her current research interests include operating systems and runtime environment.

Hai Jin is a Cheung Kung Scholars Chair Professor of Computer Science and Technology at Huazhong University of Science and Technology (HUST) in China. He is now Dean of the School of Computer Science and Technology at HUST. He received his Ph.D. in computer engineering from HUST in 1994. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. He had worked at The University of Hong Kong between 1998 and 2000, and as a visiting scholar at the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. He is the Chief Scientist of ChinaGrid, the largest grid computing project in China, and the Chief Scientist of National 973 Basic Research Program Project of Virtualization Technology of Computing System. His research interests include cluster computing and cloud computing, P2P computing, and computing system virtualization.