BWS: Balanced Work Stealing for Time-Sharing Multicores

Xiaoning Ding
Intel Labs, Pittsburgh, PA
xiaoning.ding@intel.com

Kaibo Wang
The Ohio State University
wangka@cse.ohio-state.edu

Phillip B. Gibbons
Intel Labs, Pittsburgh, PA
phillip.b.gibbons@intel.com

Xiaodong Zhang
The Ohio State University
zhang@cse.ohio-state.edu

Abstract

Running multithreaded programs in multicore systems has become a common practice for many application domains. Work stealing is a widely-adopted and effective approach for managing and scheduling the concurrent tasks of such programs. Existing work-stealing schedulers, however, are not effective when multiple applications time-share a single multicore—their management of steal-attempting threads often causes unbalanced system effects that hurt both workload throughput and fairness.

In this paper, we present BWS (Balanced Work Stealing), a work-stealing scheduler for time-sharing multicore systems that leverages new, lightweight operating system support. BWS improves system throughput and fairness via two means. First, it monitors and controls the number of awake, steal-attempting threads for each application, so as to balance the costs (resources consumed in steal attempts) and benefits (available tasks get promptly stolen) of such threads. Second, a steal-attempting thread can yield its core directly to a peer thread with an unfinished task, so as to retain the core for that application and put it to better use. We have implemented a prototype of BWS based on Cilk++, a state-of-the-art work-stealing scheduler. Our performance evaluation with various sets of concurrent applications demonstrates the advantages of BWS over Cilk++, with average system throughput increased by 12.5% and average unfairness decreased from 124% to 20%.

Categories and Subject Descriptors D.4.1 [*Operating Systems*]: Process Management—Scheduling; D.3.4 [*Programming Languages*]: Processors—Run-time environments

Keywords work stealing; multicore; time sharing; fairness

1. Introduction

In the multicore era, an application relies on increasing its concurrency level to maximize its performance, which often requires the application to divide its work into small tasks. To efficiently distribute and execute these tasks on multicores, fine-grained task manipulation and scheduling must be adopted [Saha 2007]. A common practice is that the application spawns multiple worker threads (*workers* for brevity) and distributes the tasks dynamically among its workers with a user-level task scheduler.

Work stealing [Blumofe 1994, Burton 1981], as a standard way to distribute tasks among workers, has been widely adopted in both commercial and open-source software and libraries, including Cilk [Blumofe 1995, Frigo 1998] and Cilk++ [Leiserson 2010], Intel Threading Building Blocks (TBB) [Kukanov 2007], Microsoft Task Parallel Library (TPL) in the .NET framework [Leijen 2009], and the Java Fork/Join Framework [Poirier 2011]. In work stealing, workers execute tasks from their local task queue. Any newly spawned tasks are added to the local queue. When a worker runs out of tasks, it steals a task from another worker's queue and executes it. Work stealing has proven to be effective in reducing the complexity of parallel programming, especially for irregular and dynamic computations, and its benefits have been confirmed by several studies [e.g., Navarro 2009, Neill 2009, van Nieuwpoort 2001].

Existing work-stealing schedulers, however, are not effective in the increasingly common setting where multiple applications time-share a single multicore. As our results show, state-of-the-art work-stealing schedulers suffer from both system throughput and fairness problems in such settings. An underlying cause is that the (time-sharing) operating system has little knowledge on the current roles of the threads, such as whether a thread is (i) working on an unfinished task (a *busy worker*), (ii) attempting to steal tasks when available tasks are plentiful (a *useful thief*), or (iii) attempting to steal tasks when available tasks are scarce (a *wasteful thief*). As a result, wasteful thieves can consume resources that should have been used by busy workers or useful thieves. Existing work-stealing schedulers try to mitigate this problem by having wasteful thieves yield their cores spontaneously. However, such yielding often leads to significant unfairness, as a frequently yielding application tends to lose cores to other concurrent applications. Moreover, system throughput suffers as well because the yielded core may fail to go to a busy worker or may be switched back to the wasteful thief prematurely.

In this paper, we present BWS (Balanced Work Stealing), a work-stealing scheduler for time-sharing multicore systems that leverages new, lightweight operating system support. BWS improves both system throughput and fairness using a new approach that minimizes the number of wasteful thieves by putting such thieves into sleep and then waking them up only when they are likely to be useful thieves. (Useful thieves become busy workers as soon as they successfully steal a task.) Moreover, in BWS a wasteful thief can yield its core directly to a busy worker for the same application, so as to retain the core for that application and put it to better use.

We have implemented BWS in Cilk++ and the Linux kernel, and performed extensive experiments with concurrent running benchmarks. Our experiments show that, compared with the original Cilk++, BWS improves average system throughput by 12.5% and reduces average unfairness from 124% to 20%. The experiments also show another benefit of BWS, which is to reduce the performance variation. On average, BWS reduces the performance variation by more than 11% as measured by the coefficient of variation of execution times.

The rest of the paper is organized as follows. Section 2 presents further background on prior work-stealing schedulers and their limitations, and the limitations of current OS support for work stealing. Section 3 and Section 4 describe the design and implementation of BWS, respectively. Section 5 provides a comprehensive evaluation of BWS. Section 6 discusses related work. Finally, Section 7 concludes the paper.

2. Problem: Time-Sharing Work-Stealing Applications

In this section, we first discuss the basics of work stealing in more detail, and then look into the challenges of work stealing in multiprogrammed environments. Next, we describe ABP, a state-of-the-art work stealing scheduler, and discuss its limitations. Finally, we discuss the limitations of current OS support for efficient and fair work stealing.

2.1 Work Stealing Basics

A work-stealing software system provides application developers with a programming interface for specifying parallel tasks over a shared memory. The system handles the tasks as parallel procedure calls, and uses stack frames as the major data structure for bookkeeping task information. To minimize the overhead associated with handling fine-grained tasks, it uses lazy task creation techniques [Mohr 1991]. We will refer to an application developed and supported by a work-stealing software system as a *work-stealing application*. In practice, such applications span a wide spectrum, including document processing, business intelligence, games and game servers, CAD/CAE tools, media processing, and web search engines.

A *user-level task scheduler* manages tasks and distributes then dynamically among the worker threads (*workers*). It uses a queue to manage the tasks ready to execute for each worker, one queue per worker. During the execution, each worker dequeues the tasks from its queue and executes them. New tasks dynamically generated in the execution (e.g., by a *spawn* or *parallel for* in Cilk++) are enqueued into the worker's task queue. When a worker runs out of tasks, the worker (*thief*) selects another worker (referred to as a *victim*) and tries to steal some tasks from the victim's task queue. If there are available tasks in the victim's queue, the steal is successful, and the thief dequeues some tasks from the victim's queue and continues to process the tasks. Otherwise, the thief selects another victim. Recall from Section 1 that workers processing and generating tasks are *busy workers*, and that we informally distinguish between *useful thieves*, who help distribute tasks promptly whenever there is available parallelism, and *wasteful thieves*, who waste resources on unsuccessful steal attempts.

2.2 Issues of Work Stealing with Multiprogramming

Rapid increases in the number of cores and memory capacity of multicore systems provide a powerful multiprogramming environment for concurrently running multiple jobs and parallel applications. One multiprogramming management option is to time-slice all the cores so that each application is granted a dedicated use of the cores during its scheduling quanta. This approach suffers from low efficiency when some applications can only partially utilize the cores. Similarly, a static partitioning of the cores among the applications (i.e., space-sharing) suffers from the same inefficiencies. Approaches such as process control that dynamically partition the cores among the applications at process granularity [McCann 1993, Tucker 1989] improve upon the static case, but their OS-level mechanisms are slow to react to application phase changes. Moreover, such approaches impose a space-sharing paradigm into standard (time-sharing) operating systems. Thus, work stealing has been advocated as a powerful and effective approach to scheduling in multiprogrammed multicores [Blumofe 1998], where several parallel applications are executed concurrently.

However, early studies on work stealing in multiprogrammed environments demonstrated a tension between how aggressively thieves try to steal work (in order to quickly balance loads) and the wasted resources such steal attempts incur [Blumofe 1998]. In effect, work stealing is a double-edged sword. When a thief cannot obtain tasks quickly, the unsuccessful steals it performs waste computing resources, which could otherwise be used by other threads. More importantly, the execution of thieves may impede the execution of busy workers that would generate new tasks, causing livelock where no workers make useful progress. If unsuccessful steals are not well controlled, applications can easily be slowed down by 15%–350% [Blumofe 1998].

2.3 The ABP Algorithm and its Limitations

To handle the above problems, work-stealing schedulers (e.g., those in Cilk++ and Intel TBB) implement a yielding mechanism, based on the solution proposed by Arora, Blumofe, and Plaxton (referred to herein as **ABP**, and shown in Algorithm 1[1]) [Arora 1998, Blumofe 1998]. When a worker runs out of tasks, it yields its core spontaneously to give way to other threads (line 16). The worker is repeatedly switched back to make steal attempts, and if the attempts fail, the worker yields the core again. It repeats these operations until it successfully steals a task or the computation completes.

Algorithm 1 – ABP Work Stealing Algorithm

```
 1: $t$ : a task
 2: $w$ : current worker
 3: $v$ : a victim worker $w$ selects to steal from
 4:
 5: procedure RandomSteal(w)
 6:     Randomly select a worker $v$ as a victim
 7:     if $w$ can steal a task $t$ from $v$ then
 8:         enqueue $t$
 9:     end if
10: end procedure
11:
12: repeat
13:     if local task queue is not empty then
14:         dequeue a task $t$ and process $t$
15:     else
16:         yield()
17:         RandomSteal(w)
18:     end if
19: until work is done
```

2.3.1 Drawbacks of ABP

ABP, while the state-of-the-art, suffers from two critical drawbacks for time-sharing multicores: significant unfairness and degraded throughput. Workers in ABP use the *yield* system call to relinquish their cores. *When* they get switched back (i.e., rescheduled by the OS) to continue stealing is determined by OS scheduling policies and the workloads on the system, instead of the availability of tasks ready for steals. If thieves for an application cannot resume stealing when tasks become available (i.e., when they transition from wasteful thieves to useful thieves), the concurrency level of the application is limited. Significant unfairness arises when such limiting of concurrency is applied unevenly across co-running applications. As will be illustrated below and demonstrated in Section 5, such scenarios are common when ABP is used in multiple work-stealing applications co-running on time-sharing multicores. The unfairness also arises when work-stealing applications share the same set of cores with non-work-stealing applications (e.g., pthread applications). In general, the more frequently a work-stealing application yields its cores, the more its execution is delayed.

While *unfairness* arises whenever yielding thieves are not switched back for their work in time, *throughput* is degraded whenever wasteful thieves either (i) fail to yield their cores or (ii) yield their cores but are switched back prematurely. Depending on OS scheduling policies, there are cases in which *yield* calls return without actually relinquishing the cores, for example, when the caller is the only ready thread or the thread with the highest priority on the core. This makes the yielding mechanism ineffective. Because the OS is not aware of whether or not a thread is a wasteful thief, multiple wasteful thieves may be scheduled on the same core. Such thieves yield the core back and forth wasting resources without making progress. If there are suspended threads (perhaps from a different application) that are ready to do useful work, these threads are needlessly delayed, reducing overall system throughput.

2.3.2 Illustrative Examples

We will now provide a quantitative illustration of the above problems using a few representative experiments. We select four benchmarks, BFS, EP, CG, and MM, and run them on a 32-core machine. (Please refer to Section 5 for benchmark description and machine configuration.) In the experiments, we first run each benchmark alone to get its solo-run execution time. Then, we run two benchmarks concurrently to see how much their executions are slowed down due to the co-running. As they may have different execution times, we run each benchmark multiple times so that their executions are fully overlapped.

We first use our results for BFS and EP to illustrate the fairness problem. Compared to the solo-runs, BFS is slowed down significantly by ***377%***, while EP is slightly slowed down by ***5%***. We define unfairness as the difference between the two slowdowns, in this case 372%. Though BFS creates 32 workers and can achieve a speed up of 21 when it runs alone on the 32 cores, we find that it has only about 5 active workers when it co-runs with EP. This experiment clearly confirms that the actual concurrency levels of work-stealing applications can be seriously limited when they co-run with other applications, because their workers may yield cores prematurely and may not be rescheduled in time.

[1] Detailed implementations may vary in different work-stealing software systems. For example, Cilk++ implements the algorithm faithfully, while in Intel TBB a worker yields its core when the number of failed steals exceeds a threshold. Though the implementations may vary, the problems we identify (and address) are the same.

To illustrate the throughput problem, we use our results for CG and MM. After we run each of CG and MM alone to obtain its solo-run execution time, we run them concurrently in two different scenarios. In the first scenario, the number of workers in each application is 32, and in the second scenario, CG has 16 workers and MM has 32 workers. The first scenario represents the case in which the number of thieves in CG is not controlled and its workers may yield cores frequently. The second scenario represents the case in which the number of thieves of CG is under control, and yieldings are less frequent than those in the first scenario. This is because each worker has more work to do and spends a larger proportion of its time working than in the first scenario. We compare the execution times of CG and MM in these two scenarios against their solo-run execution times with 32 workers.

In the first scenario, due to the co-running, the execution time of CG is significantly increased by 144%, while the execution time of MM is increased by only 37%. In the second scenario, the execution time of CG is increased by 97%, and the execution time of MM is increased by 12%. In both scenarios, the execution of CG is delayed by much larger percentages than the execution of MM. This unfairness is caused because the workers in CG yield cores more frequently than the workers in MM. Compare to the first scenario, reducing the number of workers in CG in the second scenario improves the performance of *both* applications. This indicates that in the first scenario, the yielding mechanism is only partially successful in reducing the resource waste caused by steal attempts.

The experiments also show that co-running the two work-stealing applications together may increase the throughput compared to time-slicing all the cores (so that each application gets all the cores during its slice). This is indicated by the slowdowns of the benchmarks being less than 100% in the second scenario, and is consistent with previous studies [Iancu 2010]. We have also co-run CG and MM with a non-yielding work-stealing scheduler, and found that CG was slowed down by more than 600% and MM was slowed down by about 100%. This significant performance degradation confirms the observation in a previous study [Blumofe 1998]. It also demonstrates the need for a work-stealing scheduler that can make efficient use of computing resources on multicores.

2.4 OS Limitations and Insufficient Mechanisms to Support Work Stealing

As discussed above, the OS has little knowledge on the current roles of the threads, such as whether a thread is a busy worker, a useful thief or a wasteful thief. As a result, it may, for example, suspend a busy worker to schedule a wasteful thief. Moreover, it is unable to properly schedule thieves, likely either (i) delaying an application's execution by schedling too few (useful) thieves, or (ii) reducing system throughput by scheduling too many (wasteful) thieves.

Operating systems do provide several mechanisms that could address this lack of user-level information, through its existing API. One type of such mechanisms is policy-based. For example, an application can choose one of a few pre-cooked scheduling algorithms, or a thread can adjust its scheduling priority. However, these mechanisms depend on the scheduling policies hard-coded in the OS, which lack the flexibility to meet the customized scheduling requirements of work-stealing applications.

The other type of mechanisms increases the flexibility by enabling a thread to voluntarily suspend its execution. But, which thread the core will be granted to is still determined by OS scheduling policies. Operating systems usually provide three suspension mechanisms, but none of them can meet the requirements of work-stealing schedulers for fairness and throughput efficiency.

The first suspension mechanism, the *yield* system call, makes its caller thread relinquish its core without blocking. The OS scheduler puts back the caller into a ready thread queue, and makes a rescheduling decision to allow other threads to run. The caller will be rescheduled automatically later by the OS scheduler. The other two suspension mechanisms are sleeping based. A thread can be put into sleep by waiting for a condition denoted by a variable to become true. When the thread sleeps, the core is granted to other threads. When the condition is met, the thread is woken up and put back to the ready thread queue to be scheduled. A thread can also be put into sleep by waiting for a timer to expire.

The above mechanisms incur unfairness when a work-stealing application time-shares the same set of cores with other applications. No matter which of the three suspension mechanisms is used, if the workers relinquish their cores frequently, other applications can take advantage of this and get more chances to run. The problem is more serious if the *yield* mechanism is used, because *yield* indicates that the caller is currently not eager to run and the OS scheduler is usually reluctant to reschedule it very soon. As we have explained, the use of the *yield* system call in work-stealing schedulers also incurs performance concerns, because the *yield* callers may fail to yield their cores, or may be switched back prematurely due to other *yield* calls.

In summary, current operating systems do not provide the mechanisms that can meet the requirements of work-stealing schedulers. This makes work-stealing schedulers unable to effectively address the fairness and throughput issues when they adjust the concurrency levels of work-stealing applications. To solve this problem, we propose a new task scheduling algorithm and new OS support to establish a balanced work-stealing (BWS) system, which will be presented in the next section.

3. The Design of BWS

In this section we describe the design of BWS, including the work-stealing algorithm used in BWS and the new

OS support. Our design is motivated by the considerations and issues outlined in the previous section. Our goals are to maximize throughput and minimize unfairness. We define *unfairness* to be the difference between the larger and smaller slowdowns among co-running applications, where slowdown is relative to the application's performance in isolation.

3.1 BWS Overview

BWS abides by the following three principles in order to minimize the extent that thieves impede busy workers, the resource waste, and the unfairness.

- The *priority principle*. The timeslices that the OS scheduler allocates to a work-stealing application should be used for processing tasks first. Only extra timeslices that are not being used by workers for processing tasks are used for thieves to do work stealing. This is mainly to prevent the execution of thieves from impeding the execution of busy workers. It also helps to put computing resources to best use.
- The *balance principle*. To maintain a high throughput for a work-stealing application, the cost of having steal attempts must be paid in order to quickly distribute dynamically generated tasks. However, the cost must be controlled: too many thieves will not improve application throughput.
- The *efficiency principle*. Both throughput and fairness are improved by having each application running at high parallel efficiency. If an application phase can make highly productive use of additional cores, its concurrency level should be increased in order to acquire more time slices.

At a high level, BWS enforces the *priority principle* by making thieves yield their cores to busy workers. It achieves the *balance principle* by dynamically putting thieves into sleep or waking them up. It realizes the *efficiency principle* by waking up sleeping thieves whenever there are tasks waiting to be stolen, in order to increase the application's concurrency level.

In more detail, the work-stealing algorithm in BWS adjusts the number of sleeping thieves based on how easily thieves can find tasks to steal. When it is easy for thieves to find tasks, more thieves are woken up to maximize throughput. Otherwise, if a thief has performed a number of unsuccessful steals, it puts itself into sleep and relinquishes its core to save computing resources. Every time a thief fails to steal from a victim, it checks the status of the victim. If the victim is working on an unfinished task but is currently preempted, the thief yields its core directly to the victim. Otherwise, the thief tries to steal from another victim.

Our design relies on two operating system features not found in current operating systems, which we now introduce:

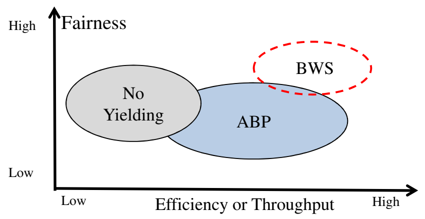


Figure 1. BWS improves both fairness and throughput, compared to current work-stealing implementations

- The OS discloses the running status of the workers to the work-stealing scheduler.
- The OS provides a new yielding facility, which enables a worker to yield its core directly to one of its peer workers.

These are discussed in further detail in Section 4. The first mechanism enables the work-stealing scheduler to find a thread that needs computing resources in a timely manner. The second mechanism enables the above step where a thief yields its core directly to the (busy but preempted) victim.

Note that BWS' use of sleep/wake-up to adjust the number of executing thieves has two key advantages over ABP's use of yield. First, it prevents thieves from being scheduled prematurely. While yielded threads can be switched back at any time, sleeping thieves cannot be switched back as long as they are asleep. Second, because the action of waking-up a thread indicates there are some tasks waiting to be processed by the thread, the thread can be rescheduled at the earliest time supported by the OS scheduler. Thus, awakened thieves can resume stealing promptly to increase the application's concurrency level. Yielded threads, in contrast, have low OS priority to be rescheduled.

Figure 1 depicts a qualitative comparison of BWS, ABP, and a work-stealing scheduler without any yielding mechanism. The X-axis represents how efficiently computing resources are used to execute work-stealing applications. The more efficient, the higher throughput the system will achieve. The Y-axis shows the fairness. With the "No Yielding" scheduler, unsuccessful steals waste computing resources and delay task processing. Thus, the efficiency and system throughput are low. ABP improves efficiency and throughput, but at the cost of fairness. Compared with ABP, BWS addresses the fairness issue, and further improves efficiency and throughput.

3.2 BWS Work Stealing Algorithm

Waking up sleeping thieves requires the involvement of non-sleeping workers to detect the availability of tasks and make wake-up system calls. Busy workers generate new tasks and have the information on whether tasks are available. Thus, an intuitive solution is to have busy workers wake up sleeping workers. However, this violates the work-first principle, because work overhead is increased.[2] To minimize the impact on work overhead, BWS lets thieves wake up sleeping workers.

Algorithm 2 shows the work stealing algorithm. As discussed above, management work is conducted by thieves to minimize the work overhead. Thus, we present only the algorithm for stealing. This algorithm is called whenever the worker's local task queue is empty.

When a thief steals a task successfully, it tries to wake up two sleeping workers (lines 11–15 in Algorithm 2). However, because a thief turns into a busy worker when it gets a task to process, waking up other workers will delay the execution of the task. To minimize the overhead, BWS offloads this management work to other thieves. For this purpose, BWS uses a counter (referred to as the *wake-up counter*) for each worker to bookkeep the number of wake-ups it desires to issue. The actual wake-up operations are to be carried out by other thieves. Thus, this worker starts to process the task immediately after it updates its wake-up counter. When a thief finds a preempted busy worker (lines 18–19), it yields the core to the busy worker.

A thief carries out management work only when it cannot get a task or yield its core to a busy worker (lines 21–40). Specifically, if the victim is a busy worker and the wake-up counter of the victim is greater than 0, it reduces the wake-up counter of the victim by 1 and increases the wake-up counter of itself by 1, implying that it will handle one wake-up operation for the worker. Then, it continues to steal. If the victim a thief tries to steal from is sleeping and the thief has pending wake up operations to process, it wakes up the sleeping worker.

To avoid the high cost associated with global synchronization impacting scalability, workers in BWS function in a peer-to-peer manner. Each thief relies on the information accumulated in its two local counters to determine whether it is a useful thief or a wasteful thief. One is the wake-up counter. The other is the *steal counter*. When a worker becomes a thief, it starts to count how many unsuccessful steals it has made with the steal counter. It is considered to be a useful thief as long as the counter value is below a pre-set threshold. When the counter value exceeds the threshold, it considers whether to sleep. It first checks the wake-up counter. If the wake-up counter is 0, it is considered to be a wasteful thief and is put into sleep. Otherwise, it reduces the wake-up counter by 1 and resets the steal counter. This is to cancel out a wake-up operation, because carrying out both of the wake-up and sleep operations increases the number of unnecessary context switches and incurs unnecessary overhead.

Algorithm 2 – BWS: Balanced Work Stealing Algorithm

```
 1: Local Variables:
 2: $t$ : a task
 3: $n$ : value of the steal counter of a worker
 4: $c$ : value of the wake-up counter of a worker
 5: $w$ : current worker
 6: $v$ : a victim worker $w$ selects to steal from
 7:
 8: procedure RANDOMSTEAL
 9:     repeat
10:         Randomly select a worker $v$ as a victim
11:         if $w$ can steal a task $t$ from $v$ then
12:             enqueue $t$
13:             $w.c \leftarrow w.c + 2$
14:             reset $w.n$
15:             return
16:         else
17:             if $v$ has an unfinished task then
18:                 if $v$ has been preempted then
19:                     yield core to $v$
20:                 else
21:                     if $v.c > 0$ then
22:                         $v.c \leftarrow v.c - 1$
23:                         $w.c \leftarrow w.c + 1$
24:                     end if
25:                 end if
26:             else
27:                 if $v$ is sleeping and $w.c \neq 0$ then
28:                     wake up $v$
29:                     $w.c \leftarrow w.c - 1$
30:                 end if
31:             end if
32:             $w.n \leftarrow w.n + 1$
33:         end if
34:         if $w.n > SleepThreshold$ then
35:             if $w.c = 0$ then Sleep
36:             else
37:                 $w.c \leftarrow w.c - 1$
38:             end if
39:             reset $w.n$
40:         end if
41:     until work is done
42: end procedure
```

4. The Implementation of BWS

BWS relies on two new OS supporting mechanisms, which require only slight changes to the Linux kernel (about 100 lines of code in two existing files concerned with thread scheduling in Linux kernel 2.6.36). First, BWS needs the

[2] Work overhead measures the overhead spent on task scheduling. It is defined as the ratio between the execution time of a work-stealing application running with a single thread (with the overhead to support task scheduling) and the execution time of its sequential version (without any scheduling overhead) [Frigo 1998]. As a key principle for designing work-stealing schedulers, the work-first principle requires that the scheduling overhead associated with each task (and finally work overhead) be minimized [Frigo 1998]. For example, techniques like lazy task creation [Mohr 1991] have been used to reduce the overhead of task generation to a level similar to that of function calls. The violation of work-first principle causes performance degradation, because tasks in a work-stealing application are usually very small [Kumar 2007, Sanchez 2010].

OS to disclose whether a worker is currently running on a core. On current operating systems, an application can only get brief running status of its threads, e.g., whether a thread is sleeping or whether it is terminated. However, whether a thread is currently taking a core and running is not directly disclosed by the OS. To provide the support, we add a new system call, which returns to the calling thread the running status of a peer thread (identified by its thread id) in the same application.

Second, we implement a new yielding support to allow a thread (*yielder*) to yield its core to a designated second thread (*yieldee*) in the same application. When the yielder makes a *yield_to* call (with a thread id argument), it is suspended. The rest of its timeslice is passed to the designated yieldee. Then the yieldee is scheduled immediately on the core that the yielder was running on, using the remaining time-slice offered by the yielder.

With the OS support, we implement a prototype of BWS based on Intel Cilk++ SDK preview (build 8503). The implementation must handle and prevent a special case, in which all the thieves are sleeping, because then parallelism changes in the application would go undetected. To avoid this case in the implementation, BWS uses a thief as a "watchdog", which executes a similar algorithm to Algorithm 2. One difference between the watchdog worker and other thieves is that the watchdog worker does not go to sleep. The other difference is that, when the watchdog worker steals a task, it wakes up sleeping workers itself (instead of relying on other thieves). Before the watchdog worker begins to process the task, it appoints one of the workers it wakes up as the new watchdog worker. Then, it becomes a normal worker.

5. Experiments

With the prototype implementation, we tested the performance of BWS. In this section, we first introduce our experimental setup, then present the experimental results.

5.1 Experimental Setup

We carried out our experiments on a workstation with four 2.26GHz Intel Xeon 7560 processors. Each processor has 8 cores. The memory size is 64GiB. The operating system is 64-bit Ubuntu Linux 10.04LTS. The kernel version is 2.6.36. The kernel parameter sched_compat_yield is set to 1.[3] The *SleepThreshold* is set to 64.

We selected the following benchmarks and measured their execution times in varying scenarios. All the selected benchmarks are computation intensive to minimize the performance impact of I/O operations.

group 1	BFS, EP, MM, RayTracer, LOOP
group 2	CG, CLIP, MI

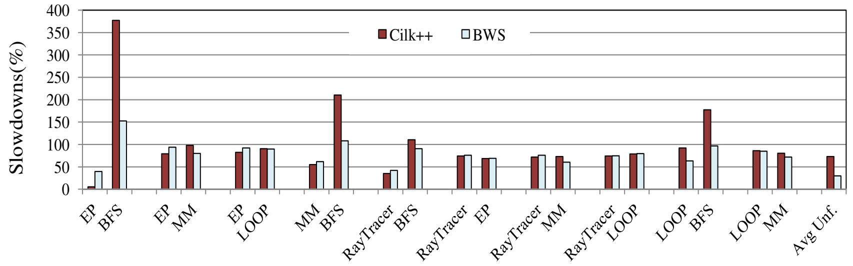
Table 1. Benchmarks are divided into two groups based on scalability: good (group 1) and fair (group 2).

- BFS traverses a graph with 10 million nodes using a breadth-first algorithm.
- CG and EP benchmarks were adapted from their OpenMP implementations in the NPB NAS benchmark suite [Jin 1999]. CG finds an estimation of the smallest eigenvalue of a large sparse matrix with a conjugate gradient method. EP computes a large number of gaussian pseudo random numbers with a scheme well suited for parallel computing. The problem size for both benchmarks is class B.
- CLIP executes a parallelized computational geometry algorithm to process a large number of polygon pairs. The algorithm implements a common operation in spatial database systems, which is to compute the areas of the intersection and the union of two polygons [Wang 2011].
- MI implements the matrix inversion algorithm. The size of the matrix is 500×500.
- MM is an example application in the Cilk++ package that multiplies two matrices. The size of each matrix is 1600×1600.
- RayTracer is a 3D renderer that renders an 800×600 image using ray tracing techniques. It casts 4 rays of light from each pixel to generate a vivid 3D world.
- LOOP is a micro-benchmark. It creates 32 threads, each of which performs a busy loop until the main thread notifies them to finish.

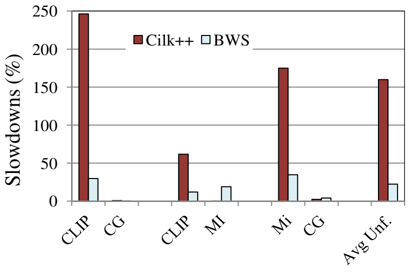
The above benchmarks except LOOP were developed with Cilk++. LOOP was developed with pthreads, and is used as a representative of non-work-stealing multi-threaded programs. The computation carried out in LOOP is very simple, in order to minimize the interference from other factors that may affect the performance of co-running applications, such as contention for cache space and memory bandwidth.

Based on the scalability of the benchmarks, we divide them into two groups. In the first group, BFS, EP, MM, RayTracer, and LOOP show good scalability, with speed-ups exceeding 20 when 32 cores are used. In the second group, CG, CLIP, and MI are not as scalable as those in the first group. Although allocating more cores to each of the benchmarks can improve the performance marginally, the speed-ups are below 16 with 32 cores.

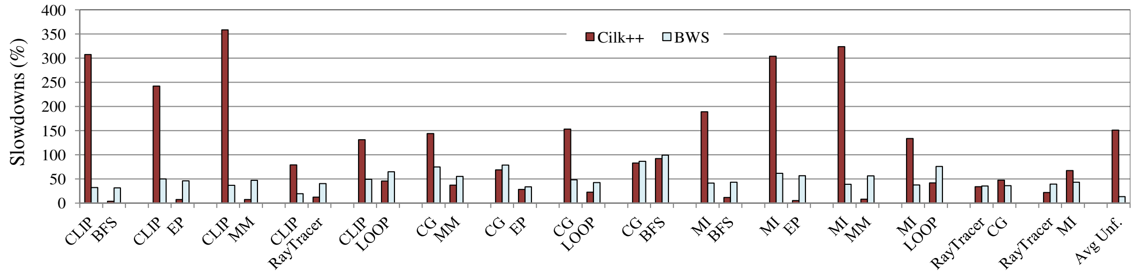
[3] With the default Linux kernel configuration, whether a thread calling *yield* can actually yield the core depends on the unfinished time quantums of this thread and other threads co-running with it on the core. Work-stealing applications usually cannot perform well with this configuration, and their performance is similar to that without yielding operations. In the experiments, we changed the configuration to improve the throughput of work-stealing applications with the original Cilk++ and to make the comparison fair.



(a) Co-runs of benchmarks in group 1



(b) Co-runs of benchmarks in group 2



(c) Co-runs of benchmarks from different groups

Figure 2. Slowdowns of benchmarks with the original Cilk++ and the Cilk++ with BWS scheduler. Subfigure (a) shows the slowdowns of the benchmarks from group 1 when they co-run, Subfigure (b) shows the slowdowns of the benchmarks from group 2 when they co-run, and Subfigure (c) shows the slowdowns for the co-runs of benchmarks from different groups.

In the experiments, we first run each benchmark alone to get its solo-run execution time, averaged over 5 runs. Then, we run two benchmarks concurrently. As benchmarks have different execution times, we run each benchmark multiple times (between 8 and 100 times, depending on the benchmark's solo-run time) so that their executions are fully overlapped. We call the concurrent executions of two benchmarks a *co-run*. For each benchmark in a co-run, we average its execution times, compare the average execution time (T_c) against its solo-run execution time (T_s), and calculate the slowdown, which is $(T_c - T_s)/T_s$. For each possible combination of two benchmarks, we first co-run the two benchmarks with the original Cilk++ (with ABP). Then we co-run them with the Cilk++ with the BWS scheduler, and compare their slowdowns with the two different systems. We set the number of threads in each benchmark to be 32. As the benchmarks in each combination have the same number of threads, in the ideal case, they show similar slowdowns.

To measure unfairness and performance, we define the unfairness metric and system throughput metric as follows. The unfairness metric is the difference between the larger slowdown and the smaller slowdown between the co-running benchmarks. We use *Weighted-Speedup* to measure system throughputs, which is the sum of the speedups of the benchmarks (i.e. $\Sigma(T_b/T_c)$), where T_b is the execution time of a benchmark in the baseline case [Mutlu 2008, Snavely 2000]. As an example, suppose two benchmarks have the same solo-run execution time, and when they co-run, each of them is slowed down by 100%. In our setting $T_b = T_s$, so the weighted-speedup throughput of the co-run is $1/2+1/2 = 1$. A throughput of 1 implies that the co-run's throughput is the same as if the benchmarks were run consecutively.

5.2 Performance in Terms of Fairness, Throughput, and Execution Time Variation

In this subsection, we show that the BWS scheduler can effectively achieve both goals of reducing unfairness and improving system throughput. We will also show that BWS can reduce the performance variation of work-stealing applications, as a by-product.

Figure 2 compares the slowdowns of the benchmarks for both the original Cilk++ (with ABP) and the Cilk++ with BWS scheduler. In each subfigure, the last two bars show the average unfairness. As shown in the figure, with the original Cilk++, when two benchmarks co-run, one benchmark may be slowed down by a much larger degree than the other benchmark. Usually, the one that is less scalable shows a larger slowdown. For example, when CLIP and MM co-run, CLIP is slowed down significantly by 358%, while MM is slowed down by only 7%. When MI and EP co-run, MI is slowed down dramatically by 304%, and EP is only slowed down by 5%. The reason is that the workers in the less scalable benchmarks are more likely to run out of tasks and yield their cores. BFS is slowed down by large percents when it co-runs with other benchmarks in group 1. This is because the granularity of the tasks in BFS is very small. Workers in BFS perform steals frequently, and relinquish cores frequently on unsuccessful steals. On average, the unfairness is 73% for benchmark pairs from group 1 (in Figure 2(a)), and 160% for benchmark pairs from group 2 (in Figure 2(b)), and 151% for the remaining benchmark pairs which mix the benchmarks from both groups (in Figure 2(c)).

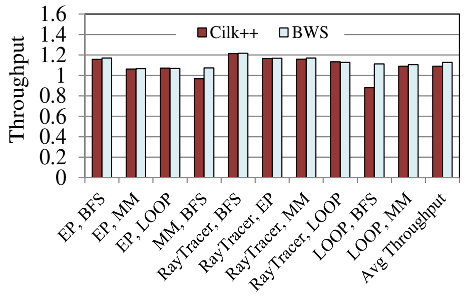
Our BWS scheduler can avoid the execution of a work-stealing application being excessively delayed. Thus, for each pair of co-running benchmarks, the difference between their slowdowns can be significantly reduced. For example, when CLIP and MM co-run, their slowdowns are 36% and 47%, respectively. When MI and EP co-run, the slowdowns are 61% and 56%, respectively. On average, the BWS scheduler can significantly reduce the unfairness to 30% for benchmark pairs from group 1, to 22% for benchmarks from group 2, and to 13% for the remaining benchmark pairs.

Our BWS scheduler not only improves fairness, but also increases system throughput. Figure 3 shows the throughputs of the co-runs for both the original Cilk++ and the Cilk++ with BWS scheduler. For fair comparison, when we calculate the throughput of a co-run for the two different systems, we use the same baseline, in which each benchmark runs alone with the original Cilk++ for the same amount of time.[4] For the co-runs of the benchmarks from group 1, BWS improves throughput by 4% on average over the original Cilk++. These benchmarks have good scalability, and their workers spend only a small amount of time on stealing with the original Cilk++. Thus, there is only a limited opportunity for BWS to show improvements. In contrast, the benchmarks in group 2 have only fair scalability, and their workers spend more time on stealing than those in group 1. Thus, for the co-runs of these benchmarks, BWS improves throughput by much larger degrees (33% on average). For the co-runs of benchmarks from different groups, BWS improves throughput moderately: the average throughput is 11% higher than that with the original Cilk++.

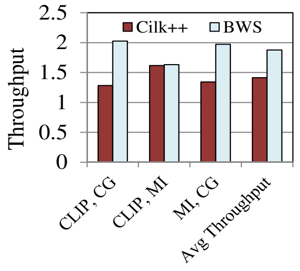
To understand how BWS improves system throughput and fairness, we have instrumented the OS scheduler to monitor the context switches during the co-runs. We select the co-runs of benchmarks from different groups. For each execution of a benchmark, we collect the number of context switches, as well as the numbers of the following two types of context switches: 1) a *to-thief context switch* that grants the core to a thief, and 2) an *external context switch* that grants the core to a thread from another application.

Compared to the original Cilk++, BWS incurs far fewer context switches. For example, BWS incurs 98% fewer context switches for the co-run of CLIP and MM, and 99% fewer context switches for the co-run of MI and EP. For most co-runs, the numbers of context switches are reduced by more than 70%.

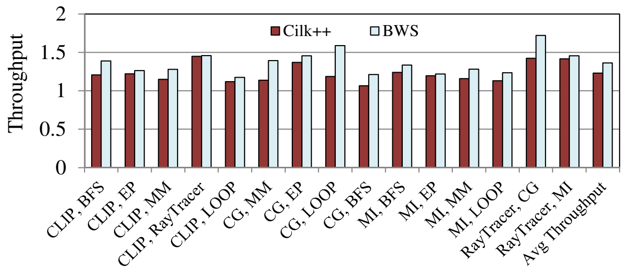
[4] Each benchmark has similar solo-run execution times with the two different systems (the difference is less than 5%), except CG and CLIP. With BWS, the solo-run execution time of CG is decreased by 25%, while the solo-run execution time of CLIP is increased by 17%. On average, the benchmark solo-runs are slowed down by 0.4% with BWS, compared with their solo-runs with the original Cilk++.



(a) Co-runs of benchmarks in group 1



(b) Co-runs of benchmarks in group 2



(c) Co-runs of benchmarks from different groups

Figure 3. Throughputs of different co-runs with the original Cilk++ and the Cilk++ with BWS scheduler. Subfigure (a) shows the throughputs of the co-runs with benchmarks from group 1, Subfigure (b) shows the throughputs of the co-runs with benchmarks from group 2, and Subfigure (c) shows the throughputs of the co-runs with benchmarks from different groups.

With the original Cilk++, a significant amount of context switches (93%) are to-thief switches. For example, when CLIP co-runs with MM, 98% of the context switches involving CLIP are to-thief switches, and 99% of the context switches involving MM are to-thief switches. The high percentages of to-thief context switches are caused by scheduling multiple thieves on the same core so the thieves yield the core back and forth quickly. BWS greatly reduces the to-thief context switches. With BWS, only 32% of context switches in the co-runs are to-thief context switches. The reduction of context switches and to-thief context switches shows that BWS improves system throughput by preventing thieves from being scheduled prematurely and by having thieves preferentially yield cores to threads that are making progress.

To show how BWS improves fairness, we compare the numbers of external context switches incurred by the original Cilk++ and by BWS. For BFS and the benchmarks in group 2, BWS significantly reduces the number of external context switches by 45%–96%. For example, when MI co-runs with EP, the number of external context switches in each execution of MI is reduced by 80% with BWS. This shows that with BWS, applications in group 2 become less likely to give up cores to their co-running applications, and thus excessive execution delay is avoided. We did not see such an obvious trend for the benchmarks in group 1 (except BFS). While the number of external context switches are reduced for some benchmarks (e.g., MM), the number is increased for others (e.g., EP).

Another important finding is that with the original Cilk++, the benchmarks usually show large performance variations,[5] which are caused by frequent core yieldings across applications. Taking CG as an example, when it co-runs with MM, its execution time varies in a wide range from 35 seconds to 65 seconds. With BWS, its execution time varies in a much smaller range from 23 seconds to 29 seconds.

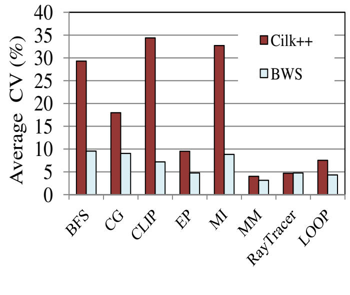
To monitor the performance variation, for each benchmark in each co-run, we calculate a Coefficient of Variation (CV) of its execution times. For each benchmark, we average its CV values across all the co-runs that include the benchmark, and show the average CV value in Figure 4(a). We also calculate an average CV value and show it in Figure 4(b) for each of the following types of executions.

- Type 1: Executions of a benchmark in group 2 with another benchmark in group 2
- Type 2: Executions of a benchmark in group 2 with a benchmark in group 1
- Type 3: Executions of a benchmark in group 1 with a benchmark in group 2
- Type 4: Executions of a benchmark in group 1 with a benchmark in group 1

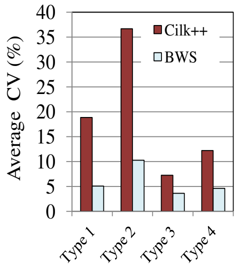
As shown in Figure 4(a), with the original Cilk++, benchmarks in group 2 and BFS show larger performance variations than other benchmarks, because their workers yield cores more frequently in the co-runs. Compared to their co-runners, the performance of benchmarks EP, MM, RayTracer, and LOOP changes only slightly across different co-runs.

As shown in Figure 4(b), type 2 executions show the largest performance variations among the four types of executions. This is because the benchmarks in group 2 may be slowed down by the largest percentages when they co-run with benchmarks in group 1. Type 3 executions show the smallest performance variations, because the performance of the benchmarks in group 1 is minimally impacted by benchmarks in group 2.

[5] Note that the impact of these variations is minimized in our co-run experiments because each benchmark is run many times in a tight loop.



(a)



(b)

Figure 4. Performance variation of the benchmarks

BWS reduces external context switches by letting workers relinquish their cores to their peer workers in the same application. Thus, it reduces the performance variation. For all the benchmarks, it reduces the average CV values to less than 10%. For type 2 executions, it can reduce the average CV value from 37% to 10%.

5.3 Experiments with Alternative Approaches

In Cilk++, a worker yields its core whenever it runs out of local tasks or its steal attempt fails. The execution of a work-stealing application is delayed if its workers relinquish the cores prematurely by this yielding mechanism. Thus, an intuitive proposal to reduce the execution delay is to lower the chance that workers yield their cores prematurely. For example, in Intel TBB, a worker does not yield its core until the number of failed steals exceeds a threshold, which is two times the number of workers. When the threshold is reached, the worker yields its core on each unsuccessful steal. For convenience, we call this method *delay-yielding*. Another method is that a worker yields its core once every time it has conducted a number (e.g., 64 in the experiments below) of failed steals. We call this method *selective-yielding*.

We have implemented these methods into Cilk++, and repeated all the co-runs for each of the methods. The threshold is 64 (i.e. 2×32) for the delay-yielding method. Figure 5 compares the average unfairness and average throughput of the two methods against that of the original Cilk++ and BWS. Compared to the original Cilk++, the two methods can slightly reduce unfairness. However, they undesirably lower system throughput. For example, the selective-yielding method can reduce the average unfairness from 124% to 108%, which is still serious. But it lowers the average throughput by about 8% from 1.2 to 1.1. In contrast, BWS both minimizes unfairness and improves throughput.

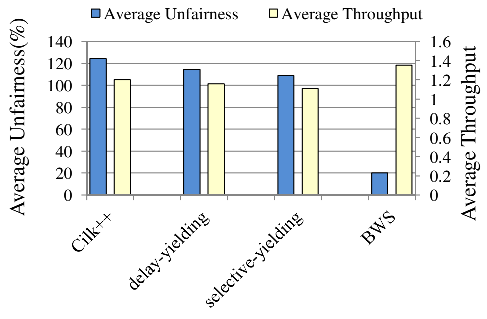


Figure 5. Average unfairness and average throughput for the original Cilk++, two alternative approaches, and BWS

The delay-yielding method or selective-yielding method can only modestly reduce unfairness. There are two reasons. First, the methods still use *yield* calls to relinquish cores. Second, in a work-stealing application, new tasks are dynamically generated by workers processing tasks. Though the methods enable a worker to continue stealing after unsuccessful steal attempts, the worker still may not be able to steal a task before it relinquishes its core if existing tasks are all being processed and new tasks have not been generated. In a multi-programmed environment, workers processing tasks may be preempted. This makes it more difficult for a worker to find a task to steal in a limited number of attempts. In contrast, BWS keeps useful thieves running and only puts wasteful thieves into sleep.

5.4 Parameter Sensitivity

In BWS, *SleepThreshold* is an important parameter. A large threshold value can help reducing unfairness by maintaining the concurrency level of a work-stealing application. However, a large threshold value may increase the resource waste and impact system throughput, because thieves are kept awake for a longer time with a larger threshold value. With the experiments in this subsection, we measure the impact of the threshold value on both fairness and throughput.

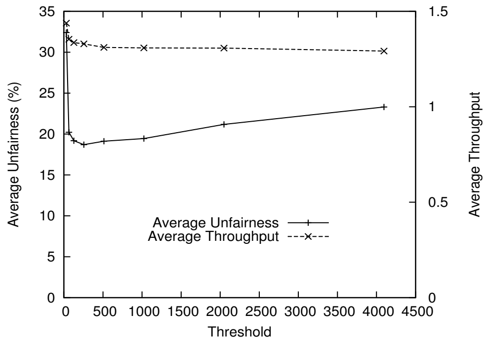


Figure 6. Average unfairness and average throughput achieved by BWS, when varying *SleepThreshold* from 32 to 4096

In the experiments, we vary the value of *SleepThreshold* from 32 to 4096, and execute all the co-runs on the Cilk++ with BWS scheduler. In Figure 6, we show the average unfairness and average throughput of these co-runs. When the threshold is less than 256, increasing the value can effectively reduce unfairness, especially when the threshold is small. For example, increasing the threshold from 32 to 64, the average unfairness value can be reduced from 32% to 20%. But the improved fairness comes at the cost of reducing the throughput from 1.44 to 1.35.

Increasing the threshold from 256 to 4096 both reduces the average throughput from 1.33 to 1.29 and increases the average unfairness from 19% to 23%. The reason that the fairness no longer improves when we increase the threshold beyond 256 is as follows. In some co-runs of benchmarks with different degrees of scalability, the benchmarks with better scalability are slowed down by larger percentages than their co-runners, and the difference between the slowdowns increases with the threshold.

Please note that the average unfairness and average throughput with the original Cilk++ are 124% and 1.20, respectively. Thus, Cilk++ with BWS always performs better than the original Cilk++, no matter which value from 32 to 4096 is chosen for *SleepThreshold*. However, values between 32 and 256 are clearly preferable to larger values.

5.5 Validation of Optimization Techniques

To get good performance, BWS uses a few techniques. One important technique is that the thieves in an application carry out most management work, e.g., inspecting the local data structures of their peer workers to collect information, making scheduling decisions, and waking up sleeping workers. This technique minimizes work overhead. The other critical technique is to keep the solution decentralized to avoid global coordination. Instead of maintaining global information, such as the number of available tasks or the number of busy workers, BWS executes the workers in an application in a peer-to-peer way. The workers accumulate partial information with simple data structures, e.g., wake-up counters and steal counters, and make scheduling decisions based on the information. This makes the solution scalable.

We have designed experiments to validate these optimization techniques. In the first experiment, we run the work-stealing benchmarks with a downgraded BWS, in which busy workers are in charge of waking up sleeping workers. With the downgraded BWS, when a busy worker generates a new task, it wakes up two sleeping workers if there are any, and a thief goes to sleep when its number of unsuccessful steals exceeds a threshold. The work overhead impacts not only the performance of work-stealing appliations when they run concurrently, but also their solo-run performance. Thus, we collect the solo-run execution times of the benchmarks to factor out any interference from co-running. We run each benchmark with 32 workers on 32 cores. Compared to BWS, with the downgraded BWS, the benchmarks are slowed down by 7% on average due to the increased work overhead. Among the benchmarks, MI is slowed down by the largest percentage (35%).

In the second experiment, we show the extent to which collecting global information in a work-stealing application can degrade the performance. We implement and use another work-stealing scheduler, named WSGI (work-stealing with global information), which collects the number of busy workers, the number of non-sleeping thieves, and the number of sleeping thieves. By dynamically putting thieves into sleep or waking up sleeping thieves, WSGI tries to keep a fixed ratio between the number of busy workers and the number of non-sleeping thieves. We set the ratio to 2 in the experiment to guarantee that available tasks can be distributed quickly to thieves [Agrawal 2008]. As we do in the previous experiment, we collect the solo-run execution times of the work-stealing benchmarks on 32 cores, with 32 workers in each benchmark. We compare the execution times against those with BWS. The comparison shows that the benchmarks are 43% slower with WSGI than they are with BWS. Among the benchmarks, CLIP shows the largest slow-down (177%). The performance degradation is due to the contention on the locks protecting the global information. This experiment clear demonstrates the importance of keeping the BWS design decentralized.

6. Related Work

6.1 Work Stealing in Multiprogrammed Environments

To efficiently execute work-stealing applications in multiprogrammed environments, two solutions have been proposed: ABP (as discussed in this paper) and A-Steal. A-Steal [Agrawal 2008] assumes there is a space-sharing OS job scheduler that allocates *disjoint sets of cores to different applications*. Based on whether the cores are efficiently used in one epoch, a work-stealing application requests that the OS job scheduler adjust the number of cores allocated to it for the next epoch. Based on the number of cores the OS grants it, the application in turn adjusts the number of workers it will use in the next epoch, in order to have exactly one worker per allocated core. Similar solutions were also proposed earlier to schedule parallel applications with user-level task scheduling on multiprogrammed multiprocessors [Agrawal 2006, Tucker 1989].

While the ABP solution has the fairness and throughput problems detailed in this paper, A-Steal can hardly be adopted on conventional multicore systems, where operating systems are usually designed for time-sharing the cores among applications. Space-sharing is typically used on batch processing systems. For both high performance and high efficiency, a space-sharing system requires that each application predicts the number of cores it needs and then readily adjusts to the number of cores allocated [McCann 1993, Tucker 1989]. Most applications must be redesigned on conventional systems to meet these requirements. At the same time, accurately predicting the demand for cores can be very challenging, considering various execution dynamics that can affect the demand such as paging, contention, irregularity in computation, etc.

Moreover, A-Steal's epoch-based adjustments work well only when the application's execution phases are much longer than the epoch length, so that the statistics collected in an epoch can be used to predict the number of cores preferred by the application in the next epoch. To amortize overheads, typical epoch lengths in A-Steal are tens to hundreds of milliseconds. However, our experiments show that parallel phases in some work-stealing applications can be as short as hundreds of microseconds. Such applications require much shorter epoch lengths, which may significantly increase the overhead to collect global information on execution efficiency, and thus limit application scalability. BWS does not suffer from these shortcomings.

6.2 System Support for User Level Thread Scheduling

Previous studies have proposed OS support for user level thread scheduling [Anderson 1992, Black 1990, Marsh 1991, Polychronopoulos 1998]. For example, scheduler activations [Anderson 1992] enable user-level thread schedulers to communicate with the OS scheduler via upcalls and downcalls. This provides a user-level thread scheduler with more control over the cores allocated to the application. For example, when a user-level thread is blocked, the user-level thread scheduler can be notified by the OS to perform rescheduling to prevent the kernel thread executing the user-level thread from being blocked. Usually, intensive modifications to the OS must be made to provide the support, and they have not been adopted by mainstream operating systems.[6] We believe that the OS support proposed for BWS, in contrast, are sufficiently lightweight to be adopted into Linux and other mainstream operating systems.

Though the tasks in work-stealing applications are called "threads" in some early articles, they are different from the user-level threads targeted in the above proposals. The granularities of the tasks in work-stealing applications can be very small, e.g., hundreds of CPU cycles. To minimize work overhead, lazy task creation and other similar techniques have been used to make them more like local procedure calls, rather than entities ready to be scheduled like user-level threads [Mohr 1991]. For example, the major data structure for a task is a stack frame. There are neither private stacks, contexts, priorities, nor preemption in scheduling the tasks. At the same time, the fine granularities also dictate that the management costs and scheduling overheads be kept very low. BWS is designed with the full consideration of these features of work-stealing applications, including its OS support, without requiring intensive modifications to the OS kernel.

[6] The idea of scheduler activations was once adopted by NetBSD and FreeBSD.

In the Exokernel design, the OS provides applications with a *yield_to* mechanism [Engler 1995], which allows each application to schedule its kernel threads at user level. BWS could exploit this feature.

6.3 Other Related Work

Besides the ABP and A-Steal algorithms, there are other work-stealing algorithms proposed for various purposes such as improved data locality [Acar 2000] and extensions to distributed memory clusters [Blumofe 1997, Dinan 2009]. These studies did not address multiprogramming issues. Hardware support and OS support have been proposed to reduce the overheads of running work-stealing applications [Kumar 2007, Lee 2010, Sanchez 2010], e.g., task stealing overheads; these are orthogonal to BWS.

7. Conclusion

This paper introduced BWS (balanced work stealing), a novel and practical solution for efficiently running work-stealing applications on time-sharing multicores. BWS addresses the fairness and system throughput problems of existing work-stealing approaches via two means. First, it reduces the costs associated with wasteful thieves by putting such thieves into sleep and then waking them up only when they are likely to be useful thieves. Second, it minimizes unfairness by enabling thieves to yield their cores directly to busy workers for the same application, thereby retaining the cores for that application and putting them to better use. BWS reduces scheduling overheads through a decentralized design in which thieves perform scheduling bookkeeping and wake up sleeping workers when appropriate. The design relies on two new system calls added to the OS: one that returns the running status of a peer thread and another that yields the caller's core directly to a peer thread. The required changes to the OS are minimal (100 lines of code in two thread scheduling files in Linux).

Our evaluation with a prototype implementation in Cilk++ and the Linux kernel shows that compared to the original Cilk++, BWS improves average system throughput by 12.5% and reduces average unfairness from 124% to 20%, while also reducing application running time variance. Currently, we are making efforts to merge BWS into production work-stealing libraries and systems.

As part of future work, we intend to extend BWS to improve the fairness and throughput of work-stealing applications in virtualized environments, where physical CPU cores are usually over-committed with multiple virtual cores. Work-stealing applications face similar issues in these environments as they do in multi-programmed environments. Virtual cores running thief threads may impede the execution of other virtual cores doing useful work. However, due to the semantic gap challenge and the simple interface between hypervisor and virtual machines, addressing these issues seems more challenging in virtualized environments than in traditional multiprogrammed environments.

Acknowledgments

We thank Frans Kaashoek for his helpful suggestions as the shepherd for this paper, and the anonymous reviewers for their constructive comments. This research was supported by the National Science Foundation under grants CNS-0834393, CCF-0913150, and CNS-1019343 to the Computing Research Association for the CIFellows Project. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the Computing Research Association. This work was done under the umbrella of the Intel Science and Technology Center for Cloud Computing.

References

[Acar 2000] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *ACM SPAA '00*, pages 1–12, 2000.

[Agrawal 2006] Kunal Agrawal, Yuxiong He, Wen-Jing Hsu, and Charles E. Leiserson. Adaptive scheduling with parallelism feedback. In *ACM PPoPP '06*, pages 100–109, 2006.

[Agrawal 2008] Kunal Agrawal, Charles E. Leiserson, Yuxiong He, and Wen-Jing Hsu. Adaptive work-stealing with parallelism feedback. *ACM Trans. Comput. Syst.*, 26(3), September 2008.

[Anderson 1992] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst.*, 10(1):53–79, February 1992.

[Arora 1998] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *ACM SPAA '98*, pages 119–129, 1998.

[Black 1990] David L. Black. Scheduling support for concurrency and parallelism in the mach operating system. *IEEE Computer*, 23(5):35–43, May 1990.

[Blumofe 1995] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *ACM PPOPP '95*, pages 207–216, 1995.

[Blumofe 1994] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *IEEE FOCS '94*, pages 356–368, 1994.

[Blumofe 1997] Robert D. Blumofe and Philip A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *USENIX ATC '97*, pages 133–147, 1997.

[Blumofe 1998] Robert D. Blumofe and Dionisios Papadopoulos. The performance of work stealing in multiprogrammed environments. Technical Report TR-98-13, Department of Computer Science, University of Texas at Austin, 1998.

[Burton 1981] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *ACM FPCA '81*, pages 187–194, 1981.

[Dinan 2009] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *ACM SC '09*, pages 53:1–53:11, 2009.

[Engler 1995] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *ACM SOSP '95*, pages 251–266, 1995.

[Frigo 1998] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *ACM PLDI '98*, pages 212–223, 1998.

[Iancu 2010] Costin Iancu, Steven Hofmeyr, Filip Blagojevic, and Yili Zheng. Oversubscription on multicore processors. In *IEEE IPDPS '10*, pages 1–11, 2010.

[Jin 1999] Haoqiang Jin, Michael Frumkin, and Jerry Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report NAS-99-011, NASA, 1999.

[Kukanov 2007] Alexey Kukanov. The foundations for scalable multi-core software in Intel threading building blocks. *Intel Technology Journal*, 11(4):309–322, November 2007.

[Kumar 2007] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In *ACM ISCA '07*, pages 162–173, 2007.

[Lee 2010] I-Ting Angelina Lee, Silas Boyd-Wickizer, Zhiyi Huang, and Charles E. Leiserson. Using memory mapping to support cactus stacks in work-stealing runtime systems. In *ACM PACT '10*, pages 411–420, 2010.

[Leijen 2009] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *ACM OOPSLA '09*, pages 227–242, 2009.

[Leiserson 2010] Charles Leiserson. The Cilk++ concurrency platform. *J. Supercomput.*, 51(3):244–257, March 2010.

[Marsh 1991] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. In *ACM SOSP '91*, pages 110–121, 1991.

[McCann 1993] Cathy McCann, Raj Vaswani, and John Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 11 (2):146–178, May 1993.

[Mohr 1991] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Trans. Parallel Distrib. Syst.*, 2(3):264–280, July 1991.

[Mutlu 2008] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ACM ISCA '08*, pages 63–74, 2008.

[Navarro 2009] Angeles Navarro, Rafael Asenjo, Siham Tabik, and Călin Caşcaval. Load balancing using work-stealing for pipeline parallelism in emerging applications. In *ACM ICS '09*, pages 517–518, 2009.

[Neill 2009] Daniel Neill and Adam Wierman. On the benefits of work stealing in shared-memory multiprocessors. http://www.cs.cmu.edu/∼acw/15740/paper.pdf, 2009.

[Poirier 2011] Yolande Poirier. Java and parallelism computing: An interview with Java developer and researcher Dr. Gilda Garreton. http://www.oracle.com/technetwork/articles/java/gildagarreton-416282.html, 2011.

[Polychronopoulos 1998] Eleftherios D. Polychronopoulos, Xavier Martorell, Dimitrios S. Nikolopoulos, Jesus Labarta, Theodore S. Papatheodorou, and Nacho Navarro. Kernel-level scheduling for the nano-threads programming model. In *ACM ICS '98*, pages 337–344, 1998.

[Saha 2007] Bratin Saha, Ali-Reza Adl-Tabatabai, Anwar M. Ghuloum, et al. Enabling scalability and performance in a large scale CMP environment. In *ACM EuroSys '07*, pages 73–86, 2007.

[Sanchez 2010] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. Flexible architectural support for fine-grain scheduling. In *ACM ASPLOS '10*, pages 311–322, 2010.

[Snavely 2000] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *ACM ASPLOS '00*, pages 234–244, 2000.

[Tucker 1989] Andrew Tucker and Anoop Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *ACM SOSP '89*, pages 159–166, 1989.

[van Nieuwpoort 2001] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *ACM PPoPP '01*, pages 34–43, 2001.

[Wang 2011] Fusheng Wang, Jun Kong, Lee Cooper, et al. A data model and database for high-resolution pathology analytical image informatics. *J. Pathol. Inform.*, 2:32, July 2011.