DWS: Demand-aware Work-Stealing in Multi-programmed Multi-core Architectures

Quan Chen
Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
chen-quan@sjtu.edu.cn

Long Zheng
[1]Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
[2]University of Aizu, Japan
longzheng@sjtu.edu.cn

Minyi Guo*
Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
guo-my@cs.sjtu.edu.cn

ABSTRACT

Traditional work-stealing schedulers perform poorly in multi-programmed multi-core architectures, because all the programs tend to use all the cores and thus incur serious core contention. To relieve this problem, this paper proposes a Demand-aware Work-Stealing (DWS) task scheduler, with which a work-stealing program uses cores according to its realtime demand on the cores. If multiple programs scheduled by DWS run in a multi-core architecture concurrently, the cores are first evenly allocated to the co-running programs. At runtime, if a program cannot fully utilize its cores, it releases some of its allocated cores. Otherwise, if a program demands more cores, it tries to use the free cores released by its co-running programs. Experimental results show that DWS can achieve up to 32.3% performance gain for co-running programs compared to traditional work-stealing schedulers with the ABP yielding mechanism.

Categories and Subject Descriptors

D.3.4 [**Programming Language**]: Processors

General Terms

Run-time Environments, Performance

Keywords

Work-stealing, Multi-programmed, Core allocation

1. INTRODUCTION

In the multi-core era, to fully utilize the cores, parallel programs that consist of many threads are increasingly popular. However, programmers need to assign tasks to threads manually in multi-threading, which is often burdensome for developing parallel programs. To relieve the burden, many parallel programming environments with dynamic load-balancing polices have been proposed for providing user-friendly programming interfaces. In these programming environments, such as MIT Cilk [9], Cilk++ [23], TBB [26], X10 [22] and OpenMP [5], a parallel program divides its work into fine-grained tasks. By scheduling the tasks to different worker threads (denoted by "workers" for short) dynamically, the workloads can be balanced in multi-core architectures automatically.

Work-stealing [8] is the best-known dynamic load balancing policy. In a multi-core architecture, existing work-stealing schedulers launch a worker for each core and provides an individual task pool for each worker. Most often, each worker pushes tasks to and pops tasks from its own task pool. Only when a worker's task pool is empty, it tries to steal tasks from other workers until it gets a task. Work-stealing works well if the multi-core architecture does not execute multiple programs concurrently.

However, in the multi-programmed environments where multiple programs co-run in a multi-core architecture, if a worker cannot steal a task successfully in a few steal attempts, the large number of unsuccessful steals it performs waste computational resources, which could be used by other workers to execute tasks. To reduce the resource wasting, existing work-stealing schedulers (e.g., MIT Cilk and TBB) implement the *ABP yielding mechanism* [4]. In ABP, if a worker fails to steal a task, it yields the core spontaneously so that other workers can use the core to execute the tasks.

The above ABP yielding mechanism suffers from two critical drawbacks: unfair resource allocation and serious cache contention. As for the first drawback, it is very possible that a worker that yields its core c cannot get c back even when its tasks are ready, because the other worker that is using c may not yield c back. Generally speaking, the more often a worker yields the core, the less computational resource the worker gets. The computational resources are not allocated to the programs according to their demands. As for the second drawback, if multiple data-intensive programs co-run in a multi-core architecture, their workers contend for the cache because workers of different programs can be scheduled to the same cores by the OS-level thread scheduler with the *time-sharing* scheme. The cache contention degrades the performance of all the co-running programs.

To overcome the two drawbacks, we propose a *Demand-aware Work-Stealing* (DWS) task scheduler. If multiple programs scheduled by DWS co-run on a multi-core architec-

*Minyi Guo is the correspondence author of this paper.

PMAM'14, February 15-19 2014, Orlando, FL, USA

http://dx.doi.org/10.1145/2560683.2560696.

ture, they take disjoint cores evenly in the beginning based on *space-sharing* scheme. **In DWS, a work-stealing program does not aggressively take all the cores but using cores according to its realtime demands**. If a program desires fewer cores, it releases some of its cores. Otherwise, if a program desires more cores, it tries to take the cores released by other programs. Because the programs do not use all the cores, the contention on the cores is partly relieved. Furthermore, DWS can relieve the cache contention because the co-running programs take disjoint cores.

To the best of our knowledge, DWS is the first work-stealing scheduler that can dynamically balance cores among the co-running programs adopting the space-sharing scheme without a centralized OS-level core allocator. The main contributions of this paper are as follows.

- We have modified the algorithm adopted by each worker in traditional work-stealing scheduler, with which a worker goes to sleep if it fails to steal a task for too many times. It enables a program to release the under-utilized cores, so that other programs can use the released cores more efficiently.
- We have proposed a *coordinator* in DWS to manage the workers. For any program, its coordinator collects runtime information and schedules its workers based on the collected information. It enables a program to grab and use the under-utilized cores released by other programs, so that the program can achieve better performance.
- We have implemented and evaluated DWS. The experimental results show that DWS can significantly improve the performance of co-running programs up to 32.3% compared to traditional work-stealing schedulers.

The rest of this paper is organized as follows. Section 2 describes the problem and explains the motivation of DWS. Section 3 presents design and the implementation of DWS. Section 4 gives the experimental results. Section 5 discusses related work. Section 6 draws conclusions and sheds light on future work.

2. MOTIVATION

In modern multi-core processors, a core provides its computation ability to different threads in units of time slices. Threads that reside on the same core share the core by taking its different time slices.

In a multi-core system that executes multiple work-stealing programs concurrently, every program aggressively tries to take all the time slices. Serious contention for the time slices happens among workers of different programs. If the workers are not well-scheduled among the cores, the time slices are not allocated to the co-running programs according to their demands. In this case, the co-running programs are seriously slowed down. We say their performance is not balanced if the difference among their slowed-down times is too large.

Suppose m work-stealing programs co-run on a multi-core system with k cores. The k-core system executes $m \times k$ workers concurrently. The target problem of this paper can be expressed as *"how to schedule the $m \times k$ workers of the m co-running work-stealing programs on a k-core system so that they can get good and balanced performance?"*

There are two general schemes, *time-sharing* and *space-sharing*, for scheduling the $m \times k$ workers among the k cores. In time-sharing, threads can be scheduled to any core and therefore workers of different work-stealing programs can be scheduled to the same core. Most current operating systems (e.g., Linux) schedule threads via time-sharing scheme by default. In space-sharing, cores are divided into m core groups and each of the m programs runs on one group of cores. It is worth noting that workers of the same program share cores in its core group in time-sharing although the cores are allocated to the co-running work-stealing programs in space-sharing.

Fig. 1 shows the problem of scheduling m work-stealing programs (denoted by P_1, ..., P_m) on a k-core system. In the figure, w_{i1}, ..., w_{ik} are the k workers of P_i $(1 \leq i \leq m)$, the solid lines represent the scheduling with the time-sharing scheme and the dashed lines represent the scheduling with the space-sharing scheme. However, neither time-sharing nor space-sharing can effectively balance the performance of the co-running work-stealing programs.

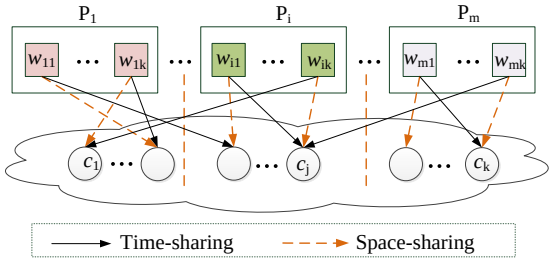

Figure 1: Schedule m work-stealing programs on a k-core system.

2.1 Drawbacks of time-sharing scheme

As shown in Fig. 1, it is inevitable that some workers are scheduled onto the same core since there are $m \times k$ workers overall but there are only k cores. To balance the performance of workers on the same core, it is important to avoid the situation where some workers take too many time slices while the other workers can only get a few time slices. Traditional work-stealing tries to avoid the situation by letting a free worker yield the core to other workers on the same core if the free worker fails to steal a task [1]. However, this solution leads to seriously unbalanced performance if the workers on the same core belong to different programs. We discuss two main drawbacks of using time-sharing to schedule co-running work-stealing programs as follows.

To show the first drawback, we assume P_i consists of small and short tasks but P_m consists of large and long tasks in Fig. 1. For w_{i1} and w_{mk} that are scheduled to the same core c_j in Fig. 1, once w_{i1} yields the core to w_{mk}, it can hardly get the core back since w_{mk} would take the core for a long time to process the large and long tasks. In this case, the execution of w_{i1} is seriously delayed until w_{mk} yields the core. In consequence, P_i gets worse performance compared with P_m since most computation resources (i.e., time slices) are occupied by workers of P_m. Generally speaking, the

[1]This yielding algorithm is named the "ABP algorithm" since it was proposed by Arora, Blumofe and Plaxton in [4].

more a worker yields the core, the more its execution is delayed.

To show the second drawback, we assume both P_i and P_m consist of many data-intensive tasks. In this case, both w_{i1} and w_{mk} access caches and DRAM frequently. If w_{i1} is running on the core c_j, it reads its data into the cache and the data of w_{mk} will be evacuated from the cache due to the conflict in the cache. Once w_{mk} takes the time slices of c_j, w_{mk} reads its data into the cache again and evacuates the data of w_{i1} for the same reason. Both w_{i1} and w_{mk} need to read their data into the cache many times due to the contention for the cache. The contention for the caches and the large number of cache misses degrades the performance of both w_{i1} and w_{mk}.

Due to the two main drawbacks, the time-sharing scheme is not a competitive candidate for balancing the performance of co-running work-stealing programs.

2.2 Limitations of space-sharing scheme

In the space-sharing scheme, as shown in Fig. 1, the co-running programs occupy different cores. Therefore workers on the same core belong to the same work-stealing program.

In a work-stealing program, tasks often have the same features (e.g., number of instruction, compute-intensive or data-intensive). Consequently, the workers of a work-stealing program have similar demands on the computation resources as well. Therefore, workers on the same core will get a similar number of time slices if the space-sharing scheme is adopted. The space-sharing scheme overcomes the first drawback of the time-sharing scheme that is caused by scheduling workers of different programs to the same core.

Equipartition [24] is one of the most popular policies that can be adopted in the space-sharing scheme when allocating cores to programs. Each program is allocated $\frac{k}{m}$ cores if m work-stealing programs co-run on a k-core system. However, with equipartition policy, the performance of co-running programs is not balanced since they often demand different amount of computation resources for achieving the same performance. In other words, the m co-running programs would be slowed down by different times if each of them runs on $\frac{k}{m}$ cores. Therefore, the space-sharing scheme with the simple but popular equipartition policy cannot balance the performance of co-running work-stealing programs.

In addition, the space-sharing scheme cannot overcome the second drawback of the time-sharing scheme since the workers of the same program contend for the caches and DRAM as well.

To balance the performance of co-running programs, we should allocate different numbers of cores to different programs according to their demands on the cores. If the performance of a work-stealing program is not scalable enough to utilize $\frac{k}{m}$ cores, we can allocate the program fewer cores. Otherwise, if the performance of a program is scalable enough to utilize more than $\frac{k}{m}$ cores, it is better to allocate the program more cores for better performance according to its demand.

However, it is hard to decide how many cores should be allocated to each program. Especially, the demand of a work-stealing program on the cores changes dynamically since the number of the queued tasks in task pools changes. As far as we know, for any program, the relationship between its performance and the number of allocated cores cannot be obtained without profiling it many times and the relationship changes with the changing of the program's working set.

Even worse, since operating systems have no knowledge of programs' demands on the cores, it is not realistic for current operating systems to find an appropriate core allocation for co-running programs at runtime.

To improve and balance the performance of co-running programs, we propose *Demand-aware Work-Stealing* (DWS) that is not a centralized OS-level job scheduler for multiple programs but a work-stealing scheduler for a single program.

With DWS, a work-stealing program can find out its own realtime demands on the cores according to the number of queued tasks. If all the co-running programs adopt DWS, they find out their realtime demands on the cores and cooperate with each other to dynamically adjust the cores among them at runtime.

3. DEMAND-AWARE WORK-STEALING

In this section, we first present the design of DWS. Then, we introduce the worker algorithm and the coordinator algorithm in DWS. Lastly, we describe the implementation of DWS.

3.1 Design of DWS

Without loss of generality, in Fig. 2, we have built a runtime architecture for a k-core system that executes m work-stealing programs (p_1, ..., p_m) concurrently. All the programs use DWS as the task scheduler to schedule their tasks.

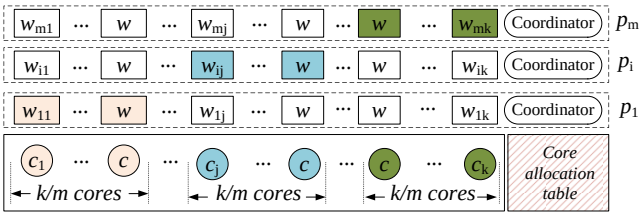

Figure 2: Runtime architecture of a k-core system that executes m work-stealing programs scheduled by DWS concurrently.

In a k-core system, DWS launches k *workers* and a *coordinator* for a work-stealing program. DWS affiliates each of its workers with an individual hardware core. For ease of description, we use w_{i1}, ..., w_{ik} to represent the k workers of program p_i. For p_i, its worker w_{ij} is affiliated with core c_j. Because every program has a worker on each core, there are m workers on each hardware core. For example, the m workers w_{1j}, ..., w_{mj} of the m work-stealing programs are affiliated with c_j.

As shown in the figure, to ensure the basic fairness in allocating cores to programs, each program is allocated $\frac{k}{m}$ adjacent cores evenly in the beginning based on space-sharing scheme. To implement the space-sharing allocation, the workers are put into sleep accordingly. Take core c_j that is allocated to p_i for example. On c_j, only the worker w_{ij} of program p_i is active while all the other $m-1$ workers on c_j are in sleep model. Because a hardware core does not execute multiple active workers concurrently and only the active workers execute tasks, the core does not execute tasks of different programs concurrently and thus relieve the

interference among different programs.

Although the initial even allocation ensures the fairness, it is not adaptive and cannot utilize the cores efficiently because the co-running programs often desire different numbers of cores during their execution. It is very possible that the cores allocated to some programs are idle while the cores allocated to other programs are overloaded. Therefore, with DWS, each work-stealing program launches a coordinator, with which the co-running programs can cooperate with each other to balance the cores according to their realtime demands.

To make the cooperation effective, all the work-stealing programs periodically check their realtime demands on the cores based on the following observation.

For any work-stealing program scheduled by DWS, if an active worker on core c cannot successfully steal new tasks in a few steal attempts when it completed its current task, the worker cannot efficiently utilize the core because the computational resources are wasted on useless steals. In this situation, the program releases core c by putting the active worker to sleep and its performance would not be damaged seriously because the active worker on c does not contribute to the progress of the program.

On the other hand, if there are a great amount of queued tasks, the program tries to acquire some free cores by waking up its sleeping workers on the free cores released by other programs. This situation happens frequently in work-stealing programs because tasks are dynamically generated at runtime. It is very possible that a large number of tasks are spawned and increases the number of queued tasks suddenly.

To support the above dynamic core releasing and core obtaining, we use a *core allocation table*, to record the allocation of cores to programs, and map the table into the global shared memory. Table 1 shows the core allocation table for the k-core system in Fig. 2.

Table 1: The structure of the core allocation table for a k-core system.

Cores	c_1	c_2	...	c_j	...	c_k
Programs	p_1	p_1	...	p_i	...	p_m

In summary, by adjusting hardware cores among the co-running work-stealing programs dynamically according to their demands, the performance of the co-running programs is improved and balanced. To support the above design, we propose the worker algorithm and the coordinator algorithm in DWS as follows.

3.2 Worker algorithm in DWS

As mentioned before, in DWS, to utilize all the cores efficiently, a worker goes to sleep and releases its core if the worker failed to steal a task for too many times. To support this strategy, we have modified the worker algorithm in traditional work-stealing schedulers so that each worker records how many times in succession it fails to steal tasks from other workers in a variable "*failed_steals*". If a worker fails to steal a task from a victim worker, it increases its *failed_steals* by one. Otherwise, if a worker successfully obtains a task from its own task pool or steals a task from another worker, it resets its *failed_steals* to 0.

A worker decides whether to go to sleep according to its *failed_steals*. if its *failed_steals* is larger than a given threshold (denoted by T_SLEEP), it goes to sleep because it cannot utilize its core efficiently.

If a worker decides to release its core, the correspondence item in the core allocation table is set as 0, which represents that the core is free and can be taken by other programs.

Algorithm 1 shows the detailed work-stealing algorithm adopted by workers in DWS. A worker first tries to obtain a task from its own task pool if it is free. When the worker's task pool is empty, the worker becomes a thief and tries to steal a task from a randomly chosen victim worker. Once the worker obtains or steals a task t successfully, it executes the task t (lines 20-22).

Algorithm 1: Work-stealing algorithm in DWS

```
   Input: w: current worker
 1 int failed_steals = 0;          // num of failed steals
 2 while work is not done do
 3     if w is free then
 4         if its task pool is not empty then
 5             w obtains a task t from its own task pool ;
 6             failed_steals = 0 ;
 7         else
 8             w randomly selects v as victim worker ;
 9             if v has a non-empty task pool then
10                 w steals t from v ;
11                 failed_steals = 0 ;
12             else
13                 failed_steals ++ ;
14                 if failed_steals > T_SLEEP then
15                     w goes to sleep ;
16                     w waits to be woken up ;
17                 end if
18             end if
19         end if
20         if t then
21             w executes t ;
22         end if
23     end if
24 end while
```

3.3 Coordinator in DWS

The coordinator of a work-stealing program manages its workers and its cores in two steps. In the first step, it checks the number of queued tasks and decides whether to wake up sleeping workers. In the second step, it calculates how many sleeping workers should be woken up and obtains free cores released by other programs for the woken-up workers. Take p_i in Fig. 2 for example, we present the two steps as follows.

For program p_i, its coordinator periodically checks the number of its active workers and the number of its queued tasks in all its task pools. If each worker only needs to process a few tasks on average, the coordinator will not wake up sleeping workers due to the small number of tasks. If each worker needs to process a great many tasks on average, on the other hand, the coordinator tends to wake up several sleeping workers.

However, waking up the proper number of sleeping cores is challenging. If too many sleeping workers are woken up, these workers are put into sleep again in a short time once

they finish the queued tasks. The extra overheads of waking the workers up and putting them to sleep again degrade p_i's performance. On the other hand, if there are a great amount of queued tasks but only a few sleeping workers are woken up(e.g., one worker), the active workers cannot maximally speed up the execution.

Therefore, we require that the coordinator satisfies three constraints. The first constraint is that the more queued tasks in the task pools, the more sleeping workers should be woken up. The second constraint is that a program can take its allocated cores back if the coordinator tends to wake up more sleeping workers than the free cores. The third constraint is that a program cannot take the cores that are not released by other programs. This constraint ensures that the performance of any program would not be seriously degraded.

After careful study, we model the three constraints using the following four parameters: the number of queued tasks, the number of active workers, the number of overall free cores and the number of its cores that are using by other programs.

In the model, we suppose that the work-stealing program p_i has N_b queued tasks and N_a active workers. We further assume that the k-core system has N_f free cores and there are N_r cores out of p_i's k/m cores are using by other programs.

Therefore, to achieve the best performance, the more queued tasks exist in the task pools, the more sleeping workers should be woken up. After careful study, obeying the above guideline, the coordinator tends to wake up N_w sleeping workers, and N_w can be calculated in Eq. 1.

$$N_w = \frac{N_b}{N_a} \tag{1}$$

After N_w is calculated, the coordinator of p_i decides to wake up which sleeping workers. More precisely, based on N_w, N_f and N_r, the coordinator of p_i decides to wake up how many workers and decides to wake up which sleeping workers as follows.

However, in reality, it is very possible that the k-core system does not have enough free cores for N_w workers. Therefore, before the coordinator of p_i starts to wake up workers, the coordinator checks whether there are N_w free cores or not in the whole system by checking the core allocation table. We use N_f to represent the number of free cores in the system. In addition, we use N_r to represent the number of its cores that are used by other programs.

- If $N_w \leq N_f$, p_i randomly selects N_w free cores and wakes up p_i's sleeping workers on the N_w selected free cores.

- If $N_f \leq N_w \leq N_f + N_r$, the coordinator of p_i tends to wakes up more sleeping workers than the free cores. Fortunately, if p_i can gets its cores that are used by other programs back, the overall cores are enough for the N_w workers. Therefore, in this case, p_i first uses all the N_f free cores and then gets $N_w - N_f$ its cores that are used by other programs back by waking up the N_w workers on the correspondence cores.

- If $N_w > N_f + N_r$, even p_i can take all its cores that are used by other programs back, the overall cores are still not enough for N_w workers. In this case, the coordinator of p_i would not wake up the N_w sleeping workers. p_i only takes all the N_f free cores and takes its N_r cores that are used by other programs back. Therefore, the coordinator wakes up $N_f + N_r$ sleeping workers on the correspondence cores.

Supported by the above model, a work-stealing program scheduled by DWS can use cores according to its realtime demands.

3.4 Implementation of DWS

We have implemented DWS by modifying MIT Cilk, which is one of the earliest parallel programming environments that implement work-stealing [17]. All the programs developed for Cilk can run in DWS without any modifications. Users can improve the performance of work-stealing in multi-programmed environment by simply updating Cilk to DWS without modifying the programs at all.

To support the *core allocation table*, the first-launched work-stealing program creates a new file and maps the file into the shared memory using "mmap()" which is part of the Posix standard [20] and is available in most operating systems (e.g., Linux). Once the shared memory is occupied, all the following programs can easily access the *core allocation table* using "mmap()".

We have also modified the worker algorithm in MIT Cilk for DWS according to the algorithm in Section 3.2. In our current implementation, we set the threshold T_SLEEP=k on a k-core system according to our experiment (to be discussed later in Section 4.3). Furthermore, every program spawns a helper thread that implements the algorithm in Section 3.3 as its coordinator besides its workers, when it is launched in DWS.

For a program scheduled by DWS, its coordinator goes to sleep every T milliseconds. If T is too small, the overhead of the coordinator is heavy and degrades the overall performance. On the other hand, if T is too large, the coordinator cannot schedule the workers in a timely manner. The overall performance of the program is therefore degraded as well. Observing from empirical results, we suggest setting $T = 10$ so that the overhead of the coordinator is negligible.

4. EVALUATION

We use a server that has two Intel Quad-core Xeon(R) E5620 processors as the multi-core system to evaluate the performance of DWS. In the processor, Intel *Hyper-Threading Technology* (HTT) that delivers two processing threads per physical core is adopted. Therefore, the hardware experimental platform can be viewed as a multi-core computer with 16 cores. In addition, the computer has 32GB DRAM and runs Linux 2.6.32-38. Accordingly, each work-stealing program launches 16 workers on the experimental platform by default.

Same to [16], we evaluate the effectiveness of DWS by executing two benchmarks in Table 2 concurrently. While most of the benchmarks are from the examples of MIT Cilk, we developed some extra benchmarks by adapting a few OpenMP programs. We call the co-running benchmarks in a test as a benchmark mix and use a two-tuple (i, j) to represent the test in which p-i and p-j co-run on the experiment multi-core platform. We make sure the execution of the programs

Table 2: Benchmarks used in the experiments

ID	Name	Description
p-1	FFT	Fast Fourier Transform
p-2	PNN	Polynomial Neural Network
p-3	Cholesky	Cholesky decomposition
p-4	LU	LU decomposition
p-5	GE	Gaussian Elimination algorithm
p-6	Heat	Five-point heat distribution
p-7	SOR	2D Successive Over-Relaxation
p-8	Mergesort	Merge sort on 4E6 numbers

in each test is totally overlapped by executing them for multiple times.

Fig. 3 illustrates the way of calculating the execution time of the co-running programs p-i and p-j. In Fig. 3, p-i is run a times and p-j is run b times.

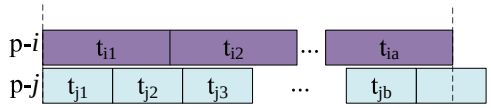

Figure 3: Measure the execution time of co-running programs.

The execution time of p_i and p_j (denoted by T_i and T_j) are calculated in Eq. 2.

$$T_i = \frac{\sum_{r=1}^{a} t_{ir}}{a}, T_j = \frac{\sum_{r=1}^{b} t_{jr}}{b} \quad (2)$$

In the following experiment, we compared the performance of DWS with the performance of two other popular strategies for scheduling co-running programs: Time-sharing + ABP yielding mechanism (denoted by ABP for short) and Space-sharing + Equi-partitioning policy (denoted by EP for short). In ABP, the operating system schedules the co-running programs using the default thread scheduler in OS with time-sharing and MIT Cilk that implements the ABP yielding policy is used to schedule each program. In EP, the 16 cores are evenly and statically allocated to the co-running programs.

4.1 Performance of DWS

Fig. 4 only shows the performance of eight benchmark mixes as other benchmark mixes perform similarly. In the figure, DWS-1 and DWS-2 represent the execution time of the first benchmark and the second benchmark in the correspondence benchmark mix with DWS respectively. The other bars represent the execution time of the benchmarks with the correspondence task schedulers.

For each benchmark, we first run it alone on the experimental platform for ten times to get its average non-interference execution time as its baseline execution time. When we run a benchmark alone, it uses all the 16 cores.

From Fig. 4 we can find that DWS significantly improves the performance of co-running programs, with the execution time reduction up to 32.3% compared to ABP and with the execution time reduction up to 37.1% compared to EP.

The good performance of DWS comes from the demand-aware core allocation. Although the co-running benchmarks

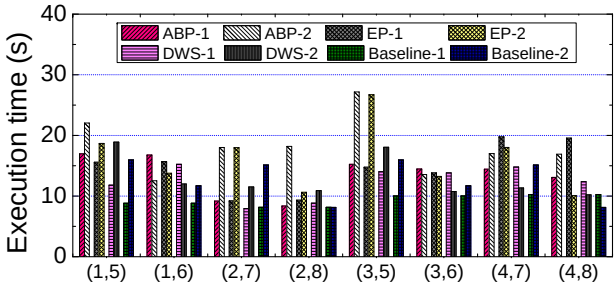

Figure 4: Performance of benchmark mixes in ABP, EP and DWS.

take cores evenly in the beginning, they release some of their allocated cores or try to get more cores according to their realtime demands on the cores at runtime. Therefore, the cores are adjusted among the co-running programs dynamically. In addition, by putting workers that repeatedly fail to obtain tasks to sleep, computational resources will not be wasted by these useless workers and can be used by other programs to execute the tasks. The sleeping workers are woken up later if the number of queued tasks increases dramatically. The improved valid resource utilization also partly improves the performance of DWS.

Compared with EP, the better performance of DWS origins from the dynamic adjustment of cores among the co-running programs. For any benchmark mix, in EP, because a program can only use its allocated cores, it is very possible that the cores allocated to some programs are idle while the cores allocated to other programs are overloaded in EP. The cores are not utilized efficiently in EP. In DWS, on the other hand, the idle cores will be released by their owners and other programs that desire more cores can utilize these released cores to accelerate their execution. Therefore, DWS works better than EP.

Careful readers may find that the performance of p-7 in (2, 7) is even slightly better than its baseline performance. The good performance of p-7 comes from the improved data locality and reduced contention for caches that are side effects of DWS. Because the workers of p-7 are scheduled to the same CPU in DWS, the data locality is improved. In addition, Because DWS puts some workers to sleep and the active workers are enough for the tasks, the contention for caches and memory bandwidth is partly relieved. This result verifies the assertion that the space-sharing scheme in DWS can relieve cache contention compared to the time-sharing scheme.

4.2 Effectiveness of the coordinator

As presented before, with the coordinators of the co-running programs, the high-demand programs can use the cores released by low-demand programs. To evaluate the effectiveness of the coordinator, we compared the performance of DWS with DWS-NC, in which the cores are not balanced among the co-running programs. Different from ABP and EP, in DWS-NC, the workers are still put to sleep and woken up in the same way as in DWS. However, DWS-NC does not ensure that a core only executes a single active worker.

Fig. 5 shows the performance of benchmark mixes in DWS-NC and DWS. From the figure, we can find that DWS-NC performs worse than DWS. The coordinator in DWS is very

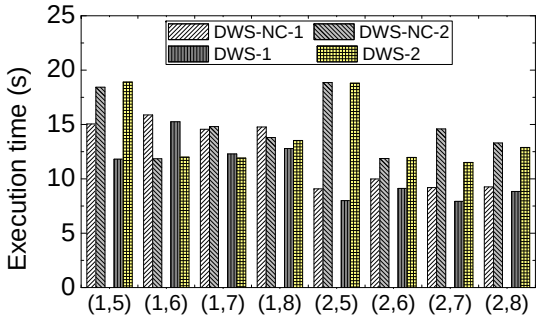

Figure 5: Performance of eight benchmark mixes in DWS-NC and DWS.

helpful for improving the performance of the programs that often desire a large number of cores, because their coordinators can help them to get more cores.

4.3 Impact of T_SLEEP

This experiment discusses the impact of the threshold T_SLEEP that is introduced in Section 3.2 on the performance of DWS. For benchmark mix (1, 8), Fig. 6 shows the performance of p-1 (FFT) and p-8 (Mergesort) in DWS with different T_SLEEP values. Other benchmark mixes show similar results.

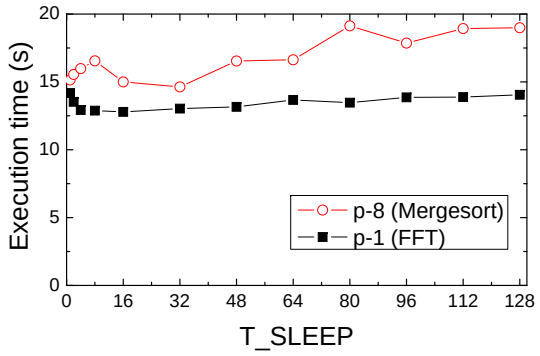

Figure 6: Performance of p-1 and p-8 in (1, 8) in DWS with T_SLEEP of different values.

From the figure we can find that p-1 and p-8 achieve the best performance when T_SLEEP is 16 or 32 on the 16-core experimental platform. If T_SLEEP is too small (e.g., =1), the workers go to sleep easily once they fail to steal tasks. In this case, the coordinator needs to wake them up shortly for queued tasks because most workers are sleeping due to the small T_SLEEP. The extra overheads caused by the frequent wake-up operation degrade the overall performance. If T_SLEEP is too large (e.g., =128), on the other hand, a worker will not go to sleep even if there are only several queued tasks. In this case, the computational resources are wasted by these workers on useless steals. According to this experiment, we suggest choosing T_SLEEP = k or $2k$ on a k-core system.

4.4 Discussion

It is worth noting that DWS does not degrade the performance of a single work-stealing program on multi-core system, although it is proposed to improve and balance the performance of multiple co-running work-stealing programs. Compared with traditional work-stealing, the workers in DWS also utilize all the cores while the only overhead in DWS is incurred by the coordinator. Our experiment shows that the overhead is negligible.

The general ideas in DWS can be applied to other parallel programming environments. For example, DWS can be easily adapted to work-sharing since the difference between work-stealing and work-sharing is the strategy of balancing tasks among workers and that does not affect the applicability of DWS. More generally, for multiple co-running parallel programs that are developed with any dynamic load-balancing model, if each of the co-running programs can put some its workers (or threads) to sleep and schedule workers accordingly as proposed in DWS, then their performance can be improved and balanced in multi-core systems.

We can also easily adapt DWS into other parallel architectures (e.g., asymmetric multi-core architectures that have cores with different frequencies). During the execution of co-running programs, we can identify if a program is data-intensive or compute-intensive based on the number of instructions and the number of memory accesses which can be obtained easily through hardware performance counter or PAPI [21]. When the co-running programs start, data-intensive programs take slow cores since they do not need too much computation resources and the compute-intensive programs take the fast cores. After that, the techniques proposed in DWS can be applied to further improve and balance the performance of the co-running programs.

Not surprisingly, DWS has one limitation. If a multi-core system only executes a single work-stealing program at a time, DWS cannot improve the performance of the work-stealing program. It is easy to identify the situation by checking the number of co-running programs. If a work-stealing program is the only program on a multi-core system, traditional random work-stealing is used instead to schedule the program. Furthermore, if the program is memory-bound, our previous CAB scheduler [14] and CATS scheduler [12, 13] can be adopted to improve its performance by reducing the cache misses. If the program is CPU-bound, our previous WATS scheduler [10, 11] can be adopted to improve its performance on asymmetric multi-core architectures by balancing tasks to asymmetric cores according to the tasks' workloads dynamically. Therefore, the above limitation will not affect the applicability of DWS.

5. RELATED WORK

Work-stealing is increasingly popular for automatic load balancing inside parallel applications. There has been a lot of research work on its adaptation and improvement [32, 25, 18]. However, if multiple work-stealing programs co-run on the same multi-core system, they suffer from poor and unbalanced performance. Improving and balancing performance of co-running work-stealing programs has become a popular research issue [7, 27]. *Time-sharing* and *space-sharing* are the two most popular schemes for balancing computation resources among co-running programs.

Traditional work-stealing adopts the ABP algorithm [4] to balance time slices among workers on the same core. However, it does not work well if operating systems adopt the time-sharing scheme to schedule threads as discussed in Section 2.1. To solve the problem, BWS [16] has been proposed. In BWS, if a worker of program p fails to steal a task, it

yields the core to another worker of p that is executing a task. With this strategy, the time slices of cores are balanced among different programs. As discussed in Section 2, if configured appropriately, the space-sharing scheme works better than the time-sharing scheme since the interference among the co-running programs is avoided [24]. DWS balances and improves the performance of the co-running work-stealing programs adopting the space-sharing scheme while BWS is based on the time-sharing scheme.

Based on the space-sharing scheme, many studies have been done to find the appropriate core allocation method (either static or dynamic) [31, 28, 6]. In these studies, a centralized job scheduler allocates cores to the co-running programs and each program uses a task scheduler to balance the workloads among its allocated cores [4].

In [15], SelfAnalyzer is proposed to analyze the performance of co-running applications. According to the feedback of SelfAnalyzer, PDPA (*Performance-Driven Processor Allocation*) is proposed to distribute processors to different applications. In [24], the job scheduler allocates processors from one program to another based on their realtime parallelism. In [1], A-GREEDY in which the task scheduler of each program provides continual parallelism feedback to the job scheduler in the form of requests for cores is proposed. With the feedback from all the co-running programs, the job scheduler adjusts the allocation of cores to programs periodically. In [3], A-STEAL is proposed on the basis of A-GREEDY. For each of the co-running program, A-STEAL uses work-stealing to schedule the tasks on its allocated cores while A-GREEDY uses a centralized algorithm to schedule tasks. In [1] and [3], the authors did not evaluate A-GREEDY or A-STEAL through experiment. In [30], ABG is proposed to relieve the unstable feedback that exists in A-GREEDY and A-STEAL. Compared with A-GREEDY, ABG ensures both good and stable performance of the co-running programs. Besides ABG, more studies [2, 19, 29] have been done to improve the performance of A-GREEDY and A-STEAL.

However, the centralized job scheduler becomes a bottleneck with more co-running programs since all the programs need to interact with the job scheduler. Different from these studies, with DWS, the co-running programs cooperate with each other to adjust cores among them dynamically in a distributed manner without interference of any centralized job scheduler. To the best of our knowledge, DWS is the first work-stealing scheduler that balances cores among the co-running programs adopting the space-sharing scheme without a centralized job scheduler. In addition, in existing studies based on the space-sharing scheme, workers are not put to sleep and woken up as needed to improve performance as we do in DWS.

6. CONCLUSIONS

Although traditional work-stealing policy works efficiently if multi-core architectures only process a single program at a time, it suffers from poor performance if multiple work-stealing programs co-run on the same multi-core architectures. To solve this problem, we have designed, implemented and evaluated the Demand-aware Work-Stealing (DWS) task scheduler. In DWS, a work-stealing program does not aggressively take all the cores but only take cores according to its realtime demands on the cores. If a worker fails too many times in succession to steal a task, the worker goes to sleep for saving computational resources. The experimental results show that the techniques adopted in DWS are effective and DWS can achieve up to 32.3% performance gain for co-running work-stealing programs, compared to the traditional work-stealing task schedulers with ABP yielding mechanism.

One promising future research direction is to extend the techniques proposed in DWS to more hardware architectures (e.g., asymmetric multi-core architectures) and more dynamic load-balancing models (e.g., work-sharing). Another future research avenue is to optimize work-stealing for multi-core systems with heterogenous accelerators (such as GPGPU).

7. ACKNOWLEDGMENTS

This work was partially supported by Shanghai Excellent Academic Leaders Plan (No. 11XD1402900), Program for Changjiang Scholars and Innovative Research Team in University (IRT1158, PCSIRT) China, NSFC (Grant No. 61272099, 61261160502) and Scientific Innovation Act of STCSM (No.13511504200). This work was also partially supported by JSPS Research Fellowships for Young Scientists Program.

8. REFERENCES

[1] K. Agrawal, Y. He, W. Hsu, and C. Leiserson. Adaptive scheduling with parallelism feedback. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 100–109. ACM, 2006.

[2] K. Agrawal, Y. He, and C. E. Leiserson. An empirical evaluation of work stealing with parallelism feedback. In *the 26th IEEE International Conference on Distributed Computing Systems*, pages 19–19. IEEE, 2006.

[3] K. Agrawal, C. E. Leiserson, Y. He, and W. J. Hsu. Adaptive work-stealing with parallelism feedback. *ACM Transactions on Computer Systems*, 26(3):7, 2008.

[4] N. Arora, R. Blumofe, and C. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 119–129. ACM, 1998.

[5] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of openmp tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.

[6] M. Bhadauria and S. McKee. An approach to resource-aware co-scheduling for CMPs. In *International Conference on Supercomputing*, pages 189–199. ACM, 2010.

[7] R. Blumofe and D. Papadopoulos. The performance of work stealing in multiprogrammed environments. *Performance evaluation review*, 26:266–267, 1998.

[8] R. D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, MIT, Sept. 1995.

[9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *The Journal of Parallel and Distributed Computing*, 37(1):55–69, Aug. 1996.

[10] Q. Chen, Y. Chen, Z. Huang, and M. Guo. WATS: Workload-Aware Task Scheduling in Asymmetric Multi-core Architectures. In *IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2012.

[11] Q. Chen and M. Guo. Adaptive workload aware task scheduling for single-ISA multi-core architectures. *ACM Transactions on Architecture and Code Optimization (to appear)*, 2013.

[12] Q. Chen, M. Guo, and Z. Huang. CATS: Cache aware task-stealing based on online profiling in multi-socket multi-core architectures. In *International Conference on Supercomputing*. IEEE, 2012.

[13] Q. Chen, M. Guo, and Z. Huang. Adaptive cache aware bi-tier work-stealing in multi-socket multi-core architectures. *IEEE Transactions on Parallel and Distributed System*, 24(12):2334–2343, 2013.

[14] Q. Chen, Z. Huang, M. Guo, and J. Zhou. CAB: Cache aware bi-tier task-stealing in multi-socket multi-core architecture. In *International Conference on Parallel Processing*, pages 722–732. IEEE, 2011.

[15] J. Corbalán, X. Martorell, and J. Labarta. Performance-driven processor allocation. In *Proceedings of the 4th conference on Symposium on Operating System Design and Implementation-Volume 4*, pages 5–5. USENIX Association, 2000.

[16] X. Ding, K. Wang, P. B. Gibbons, and X. Zhang. BWS: balanced work stealing for time-sharing multicores. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 365–378. ACM, 2012.

[17] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. *ACM Sigplan Notices*, 33(5):212–223, 1998.

[18] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. SLAW: a scalable locality-aware adaptive work-stealing scheduler. In *IEEE International Symposium on Parallel and Distributed Processing*, 2010.

[19] Y. He, W. Hsu, and C. Leiserson. Provably efficient online nonclairvoyant adaptive scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 19(9):1263–1279, 2008.

[20] I. . IEEE Std 1003.1-2001 (Open Group Technical Standard. *Standard for Information Technology–Portable Operating System Interface (POSIX)*. 2001.

[21] M. Johnson, H. McCraw, S. Moore, P. Mucci, J. Nelson, D. Terpstra, V. Weaver, and T. Mohan. Papi-v: Performance monitoring for virtual machines. 2012.

[22] J. Lee and J. Palsberg. Featherweight X10: a core calculus for async-finish parallelism. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 25–36. ACM, 2010.

[23] C. E. Leiserson. The cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[24] C. McCann, R. Vaswani, and J. Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, 1993.

[25] M. M. Michael, M. T. Vechev, and V. A. Saraswat. Idempotent work stealing. In *Proceedings of ACM symposium on Principles and practice of parallel programming*, pages 45–54. ACM, 2009.

[26] J. Reinders. *Intel threading building blocks*. O'Reilly, 2007.

[27] S. Sen. Dynamic processor allocation for adaptively parallel work-stealing jobs. Master's thesis, MIT, Aug. 2004.

[28] P. Sobalvarro and W. Weihl. Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. In *Job Scheduling Strategies for Parallel Processing*, pages 106–126. Springer, 1995.

[29] H. Sun, Y. Cao, and W.-J. Hsu. Efficient adaptive scheduling of multiprocessors with stable parallelism feedback. *IEEE Transactions on Parallel and Distributed Systems*, 22(4):594–607, 2011.

[30] H. Sun and W.-J. Hsu. Adaptive B-Greedy (ABG): A simple yet efficient scheduling algorithm. In *IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8. IEEE, 2008.

[31] A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *ACM SIGOPS Operating Systems Review*, volume 23, pages 159–166. ACM, 1989.

[32] L. Wang, H. Cui, Y. Duan, F. Lu, X. Feng, and P.-C. Yew. An adaptive task creation strategy for work-stealing scheduling. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 266–277. ACM, 2010.