Stable Adaptive Work-Stealing for Concurrent Multi-core Runtime Systems

Yangjie Cao*, Hongyang Sun†, Depei Qian* and Weiguo Wu*
*School of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an, China
Email: {caoyj@stu., depeiq@, wgwu@}xjtu.edu.cn
†School of Computer Engineering, Nanyang Technological University, Singapore
Email: sunh0007@ntu.edu.sg

***Abstract*—The proliferation of multi-core architectures has led to explosive development of parallel applications using programming models, such as OpenMP, TBB, and Cilk, etc. With increasing number of cores, however, it becomes harder to efficiently schedule parallel applications on these resources since current multi-core runtime systems still lack efficient mechanisms to support collaborative scheduling of these applications. In this paper, we study feedback-driven adaptive scheduling based on work stealing, which provides an efficient solution for concurrently executing a set of applications on multi-core systems. To dynamically estimate the number of cores desired by each application, a stable feedback algorithm, called A-Deque, is proposed using the length of active deques, which more precisely captures the parallelism variation of the applications. Furthermore, a prototype system is built by extending the Cilk runtime system, and the experimental results show that feedback-driven scheduling algorithms have more advantages for scheduling parallel applications with dynamic changing parallelism, and better overall performances are achieved with more accurate and stable feedback mechanism. Compared with existing algorithms, A-Deque improves the performances by up to 19.13% and 28.96% with respect to average response time and processor utilization respectively.**

***Keywords*-Multi-core architectures; Multi-core runtime systems; Feedback-driven adaptive scheduling**

I. Introduction

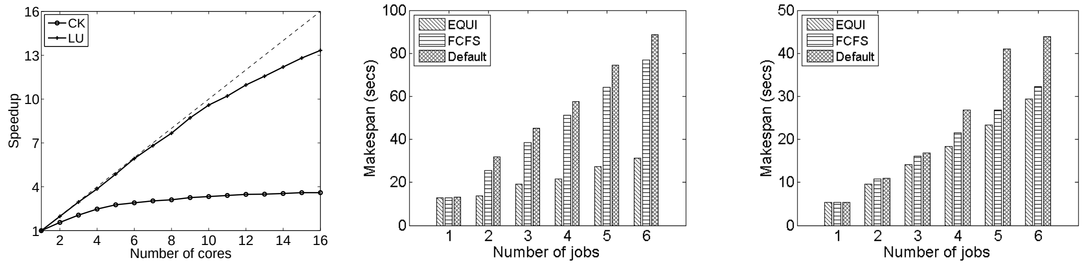
Recent developments in microprocessor design show a clear trend towards multi-core and many-core architectures. This radical shift in processor design results from diminishing returns of increasing processor frequencies, ILP (Instruction Level Parallelism) and deeper pipelines [1]. In the near future, it will be very common to have a multi-core processor with dozens or hundreds of cores on the chip. In the multi-core era, however, exploiting all the advantages offered by these processors will not be easy, and one of the great challenges is how to efficiently utilize the available computing power.

To exploit the hardware resources of modern processors, various programming models for multi-core systems have been developed, such as OpenMP [2], TBB [3], Cilk [4] which is recently extended to Intel Cilk Plus [5], etc. Compared with other parallel programming models, such as MPI and POSIX threads, these models, supported by their flexible runtime systems, provide good programmability, portability, and ability to manage dynamic parallelism for multi-core systems. When using these programming models in practice, however, there are still many issues that should be addressed to efficiently utilize the increasing number of cores. Firstly, many multi-core runtime systems will have poor scalability. Currently, it is a typical requirement in many multi-core runtime systems to explicitly or implicitly (via function calls) specify the number of cores to use for the execution of an application. As more cores are becoming available, many applications will start to experience diminishing returns with increased processor allocation. Without knowing the execution characteristic of the application on a particular hardware platform, manually specifying the number of cores to use will be a challenging task for the user. Secondly, competitions for processor resources are unavoidable in current multi-core runtime systems. Nowadays, it is very common for multiple users or applications to share a high-performance computing platform. However, many multi-core runtime systems have been traditionally designed to use a fixed number of processors (usually the maximal available processors) for each individual parallel application. Since different applications can have very different execution characteristics with respect to speedups or processor utilizations as shown in Fig. 1(a), executing multiple applications simultaneously in the traditional manner can easily lead to unnecessary resource competition, thus reduces the overall system throughput.

As a result, using these solutions by themselves, the performance does not scale well with increasing number of cores, particularly in the presence of concurrent parallel applications. To demonstrate this with an example, Fig. 1(b) and Fig. 1(c) give the results of running multiple copies of two Cilk applications on an Intel Xeon server[1]. As shown in these figures, we can see that the overall running time (makespan) of both applications under the default scheduler, which runs each copy of the Cilk application on all available cores, becomes much worse than that of the FCFS (First-Come First-Serve) scheduler when increasing the number of concurrently running copies.

Aiming at addressing this problem in the current multi-core runtime systems, adaptive scheduling algorithms are studied in this paper. Compared to scheduling all applications in a time-sharing manner as described above, adaptive scheduling based on space-sharing seems to provide a more efficient solution for simultaneously executing a set of applications. Since the parallelism of most applications often changes over time, adaptive scheduling takes advantage of the application malleability by dynamically allocating a variable number of processors to each job during runtime, thus achieves better utilization of the available computing resources. Fig. 1(b) and Fig. 1(c) also show the results of running the same set of applications as described previously, but with the simple space-sharing scheduler EQUI (Equi-Partitioning) [6] that at any time divides the total number of processors evenly among all running jobs. The results demonstrate that EQUI has much better performance in terms of makespan, especially when the applications have sublinear speedups. While this simple example shows the benefit of adaptive scheduling, in the rest of this paper we will study more effective mechanisms than

[1]The detailed information related to this experiment can be found in Section V.



(a) Speedup (b) CK (c) LU

Figure 1. The speedup and performance comparison of different scheduling strategies using two Cilk applications.

EQUI that can better capture and explore the parallelism variations of the jobs.

Although some existing work have studied adaptive scheduling, most results are based on theoretical analysis and simulation approaches [7], [8], [9], [10], [11]. Unlike these results, in this paper we study the benefits of adaptive scheduling based on solid experiments conducted on practical systems and actual workloads. The adaptive runtime system we build is based on the well-known work-stealing strategy, which has been shown to have provably-good performances in terms of both theory and practice [4], [12], [13]. The main contributions of the paper are the following:

- An adaptive runtime system is implemented based on the work-stealing load balancing strategy. The runtime system has the ability to dynamically change the number of cores allocated to each job so that it can effectively exploit the runtime characteristics of jobs and eliminate the need of manually specifying the number of cores required by most existing multi-core runtime systems.
- To dynamically estimate the number of cores desired by each job, a stable feedback algorithm, called A-Deque, is proposed using the utilization of active workers and the length of active deques. Compared to existing algorithms, A-Deque tends to capture more precisely the parallelism of the jobs, and more importantly it solves the desire instability problem of existing algorithms.
- A prototype system is built by heavily modifying the original Cilk runtime system, and the experimental results show that feedback-driven scheduling algorithms have more advantages for scheduling parallel applications with dynamic changing parallelism, and better overall performance will be achieved with more accurate and stable feedback mechanism.

The rest of this paper is organized as follows: Section 2 briefly introduces adaptive scheduling based on work stealing. Section 3 describes how to obtain stable parallelism feedback using the length of active deques. Section 4 gives the detailed implementation of adaptive scheduling framework ACilk. Our experimental results are presented in Section 5 and Section 6 concludes the paper.

II. Adaptive Scheduling Based on Work Stealing

In order to present our adaptive runtime system and feedback-driven algorithm, it is necessary to review the basic concepts in adaptive scheduling and work stealing. In this section, we will first describe work stealing and adaptive scheduling separately, and then we combine the two and discuss challenges in adaptive work stealing.

A. *Work Stealing*

Work stealing [13] is a popular thread-level scheduling mechanism to schedule parallel computations with dynamic parallelism. Because of the good performance and ease of implementation, it has been successfully applied to runtime systems in Cilk [4], Cilk Plus [5], TBB [3] and OpenMP [14].

In traditional work stealing, an application is given a fixed set of processors throughout execution. Each processor (or worker) maintains a double-ended queue, called *deque*, which contains the ready threads of the job. A processor treats its own deque as a stack and treats the deque of another processor as a queue. At any time, each processor works as follows: (1) when the thread it is currently running spawns a new thread, the processor pushes the parent onto the bottom of its deque and starts working on the child thread; (2) when the running thread completes or blocks, the processor checks its own deque. If the deque is not empty, it pops the thread from the bottom of the deque and starts working on it. In case the deque is empty, the processor becomes a "thief" and starts work stealing. In this process, the thief randomly chooses another processor, called "victim", and removes the thread from the top of victim's deque if it is not empty. If the victim's deque is empty, the thief restarts the stealing process by randomly choosing another victim until its finds a thread to work on. Clearly, when an application first starts to run, all of its allocated processors have empty deques except one processor that works on the root thread of the job.

Work stealing has been shown to have provably-efficient performances in terms of both time and space bounds [12], [13]. Moreover, unlike centralized schedulers based on work sharing such as the Greedy scheduler [15], [16], a work stealing scheduler operates in a decentralized manner without knowing all the available threads of a job at any time. Therefore, due to ease of implementation, it has also been shown to be an effective thread scheduling mechanism in practice.

B. *Adaptive Scheduling*

Adaptive scheduling provides an efficient solution to better utilize the available processor resources for simultaneously executing a set of applications, thus has gained popularity recently [7], [8], [9], [10], [11], [17], [18], [19], [20]. Since the parallelism of most applications often changes over time, adaptive scheduling takes advantage of the application malleability and gives a variable processor allocations to the jobs.

One common approach used in adaptive scheduling is the two-level scheduling framework [7]. In this framework, the executions of the jobs are divided into scheduling quantum, and the processors are reallocated based on the interaction between the job-level thread scheduler and the global-level resource allocator or processor controller. Specifically, at the beginning of each scheduling quantum q, a thread scheduler for each job calculates its processor desire $d(q)$, that is, how many processors the job needs, in this quantum. The processor controller at the global level then based on the processor desires of all jobs and its scheduling policy decides a processor allocation $a(q)$ for the job in quantum q. This process, called *request-allocation protocol* [9], will repeat after each scheduling quantum until the completion of the job.

One important aspect of two-level adaptive scheduling is how to calculate processor desires from the thread scheduler. Since the future parallelism of the job is usually unknown, the desire calculation is usually based on the execution history of the job in the previous quantum, such as measurements about the job's processor utilizations or average parallelism [7], [10]. Another aspect is for the processor controller to decide the processor allocation of each job. In this paper, we use the well-known dynamic equi-partitioning (DEQ) policy [21], which we will describe in detail in Section IV.

C. Adaptive Work Stealing

Compared with conventional thread schedulers that use only a fixed set of processors at any time, adaptive scheduling has the additional challenge of dealing with variable processor allocations at different times. When the thread scheduler uses distributed work stealing, this task becomes even more challenging, since the scheduler does not possess global information on the deques of the processors.

To handle processor changes, therefore, we adopt the concept of *mugging* [17]. In particular, when the processor allocation decreases from quantum q to $q+1$, the job loses $a(q)-a(q+1)$ processors, who may have non-empty deques. These deques are not attached to any processor at this time, therefore becomes muggable. When any processor of the job runs out of work during quantum $q+1$, instead of immediately stealing work from another processor, it will first look for muggable deques. If there are indeed deques waiting to be mugged, it will claim any such deque as its own and starts working on its bottom-most thread. Otherwise, if there is no muggable deque, it will start stealing as normal. On the other hand, when the processor allocation increases from quantum q to $q+1$, the job gains $a(q+1)-a(q)$ additional processors with empty deque. Again, each of these processors will first look for a muggable deque, which may be available from previous quantum, before stealing work as described before.

Moreover, besides dealing with processor changes, another very important challenge in adaptive work stealing is how to calculate processor desires for a job in each scheduling quantum. In the next section, we will design a novel desire calculation strategy that directly utilizes the lengths of the active deques, which solves the desire instability problem of an existing scheduler.

III. Stable Desire Calculation Using Length of Active Deques

In this section, we propose an novel desire calculation algorithm, called A-Deque, based on the utilization of the active workers and the length of active deques. We show that the processor desires calculated by A-Deque well reflects the parallelism of the job, and more importantly, it solves the desire instability problem of an existing scheduler.

A. A Novel Algorithm: A-Deque

A-Deque works based on both processor utilization and length of active deques in the quantum. Intuitively, the status of the processors in terms of whether they are busy or idle indicates the utilization of the allocated resources to a job, thus it should be used to determine the number of processors in the next quantum. Moreover, the total length of the active deques of the job at any time gives the number of ready threads that can be stolen when the job is provided with enough processors to execute, thus it indicates the unexploited parallelism of the job. A-Deque explores both of these indicators and computes the processor desire for the job in the following way.

Suppose a quantum q starts at time t_q and lasts L units of time. Recall that $a(q)$ denotes the processor allocation for the job in quantum q. Since an allocated processor is either working, mugging, or stealing at any time $t \in [t_q, t_q+L]$, let $X_j(t)$ denote the status of the jth processor at time t, where $1 \leq j \leq a(q)$. Specifically, if processor j is either working or mugging at t, we have $X_j(t) = 1$. Otherwise, if processor j is stealing at t, we have $X_j(t) = 0$. As mugging is a result of reduced processor allocation, the time spent on mugging is considered as not wasted [8].

Let $e(t)$ denote the number of active deques of the job at time $t \in [t_q, t_q+L]$, including the ones that are not attached to any processor thus are waiting to be mugged. Hence, we have $e(t) \geq a(q)$. For the jth active deque, let $Q_j(t)$ denote its length, or the number of ready threads on the top of the deque waiting to be stolen at time t. The processor desire for the job in next quantum $q+1$ is then calculated based on both $X_j(t)$ and $Q_j(t)$ as follows:

$$d(q+1) = \frac{1}{L}\int_{t_q}^{t_q+L}\left(\sum_{j=1}^{a(q)} X_j(t) + \beta \sum_{j=1}^{e(t)} Q_j(t)\right) dt, \quad (1)$$

where $\beta \geq 1$ is the exploration parameter that controls how aggressive the scheduler exploits the job's parallelism.

For instance, suppose a processor j is busy working at time t and has one more ready thread on its deque, that is, $X_j(t) = Q_j(t) = 1$. From this deque's perspective, an extra processor would be able to steal its ready thread, thus explores the available parallelism of the job. Setting $\beta = 1$ will satisfy this requirement. However, since the ready thread is in higher level of the job's structure, it is more likely to spawn more threads in the future. Thus, having a larger value for β, such as setting $\beta = 2$, will further explore the unexposed parallelism of the job. To explore the entire parallelism of the job and to smooth out the value of processor desire, this calculation is taken from all processors and deques, and is averaged over the entire quantum as shown in Eq (1). In case that no more thread is spawned for the extra processors, the processor desire will then be quickly reduced to the number of busy processors in the following quantum.

B. An Existing Algorithm: A-Steal

We now describe an existing adaptive work stealing algorithm, called A-Steal [17], which calculates the processor desire for a job in each quantum based on only the utilization of the job's allocated processors in the previous quantum. The calculation uses a simple multiplicative-increase multiplicative-decrease strategy first introduced in [7].

Recall that $X_j(t)$ denotes the status of the jth processor at time t, where $1 \leq j \leq a(q)$. The usage of the allocated processors in quantum q is then given by $w(q) = \int_{t_q}^{t_q+L} \sum_{j=1}^{a(q)} X_j(t)dt$. Since maximum possible usage of the quantum is $a(q)L$, the utilization of the processors is $u(q) = w(q)/(a(q)L)$. The quantum is said to be "efficient" if the utilization satisfies $u(q) \geq \delta$, where δ is a threshold that is usually set in the range of 80% to 95%. Otherwise, the quantum is said to be "inefficient". In addition, the quantum is said to be "satisfied" if we have $a(q) \geq d(q)$. Otherwise, the quantum is "deprived". The processor desire for the job in next quantum $q+1$ is calculated depending on whether quantum q is efficient or inefficient and whether it is satisfied or deprived as follows:

$$d(q+1) = \begin{cases} d(q) \cdot \rho & \text{if } q \text{ is efficient and satisfied,} \\ d(q)/\rho & \text{if } q \text{ is inefficient,} \\ d(q) & \text{if } q \text{ is efficient and deprived,} \end{cases}$$

where ρ is a responsiveness parameter that can be set in the range of 1 to 3. In both A-Deque and A-Steal, the processor desire for the first quantum is fixed to be 1, since the job usually starts with a single thread.

Note that A-Steal also actively explores the potential parallelism of the job by increasing its processor desire by a multiplicative factor ρ each time. Since such calculation is blind to the actual parallelism of the job, it can result in desire instability as we will show in the next subsection. A-Deque, on the other hand, performs such exploration with more precision and stability, as it directly makes use of the information about the length of active deques, which is a strong indicator on job's actual parallelism.

C. Desire Stability of A-Deque and A-Steal

It was shown in [19], [10] that another adaptive scheduler based on centralized work sharing, called A-Greedy [7], exhibits desire instability problem, even when the parallelism of the job is constant. Since both A-Steal and A-Greedy use multiplicative-increase multiplicative-decrease strategy to calculate processor desires, such instability problem can also be observed in A-Steal. In this section, we use a simple data-parallel program to show the desire instability of A-Steal, and to compare it with A-Deque. The result can clearly be applied to other data-parallel applications with constant parallelism over a period of time.

Suppose that we have a data-parallel application written in Cilk [4] as shown in Fig. 2, where N children threads are spawned by the parent thread at almost the same time[2], and each child contains a large amount of work to be done in the $Work()$ function. The graph at the right of Fig. 2 shows the DAG (Directed Acyclic Graph) that represents the structure of the program. The parallelism of this application is therefore constant at N for a long period of time.

To schedule this application with A-Deque or A-Steal, we make the following two assumptions. First, we assume that the desires of the job can be satisfied by the global-level

[2]The N threads are spawned with a small delay after each iteration of the *for* loop. Compared to the large amount of time to complete the function $Work()$, however, such delay can be negligible.

```
for (i = 0; i < N; i++){
   spawn Work(i);
}
sync;
```

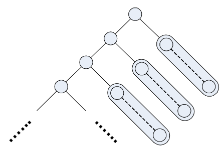


Figure 2. A simple data-parallel section of a program written in Cilk and its DAG representation.

processor controller as much as possible. This corresponds to light to medium workloads, in which two-level adaptive schedulers tend to work better compared to non-adaptive schemes [20], [10]. Second, we assume that the ready threads of a deque can be stolen as quickly as possible by steal attempts from other processors. Since the victims are chosen uniformly at random, this is usually true for a reasonable number of processors and when the quantum is set to be sufficiently long. Given these two assumptions, the processor desire and hence the processor allocation of the job can be shown to exhibit unstable behavior as shown in Fig. 3(a), where the responsiveness parameter of A-Steal is set to be $\rho = 2$, the utilization threshold is set to $\delta = 0.8$, and the parallelism of the job is at $N = 10$.

Although Fig. 2 only gives a simple example, it is not hard to see that such desire instability problem of A-Steal will remain in many other data-parallel programs like this. Varying parameters ρ and δ can alleviate the problem for a specific parallelism value. However, it will inevitably affect the responsiveness of the desires or the utilization of the processors for other sections of the job with different parallelism.

Fig. 3(b), on the other hand, shows the processor desires calculated by A-Deque for the same application when its exploration parameter is set to be $\beta = 2$. Compared to A-Steal, which catches up with the job's parallelism in about $\log_\rho N$ steps, but never converges to N, A-Deque converges to the target parallelism in about N/β steps, and exhibits no desire oscillation afterwards. For comparable parameter values in ρ and β, A-Steal tends to have better convergence for large N initially, but its desire instability will delay the execution of the job and cause waste of the processors for the majority of time steps in the steady state. A-Deque, on the other hand, is more conservative in estimating the processor desires, but guarantees stability, no steady-state error and as shown in Fig. 3(b) a small amount of transient overshoot[3]. These properties not only ensure more efficient job execution and resource utilization, but also help to reduce scheduling overheads in practice caused by context switching and cache reloading when adjusting processor allocations for a job [19], [10].

IV. Implementations

To implement the feedback-driven scheduling algorithms, we build an adaptive scheduling framework called ACilk (Adaptive Cilk), as shown in Fig. 4, which is an extension to the Cilk runtime system [4]. Cilk is a language for multithreaded parallel programming based on ANSI C

[3]The desire overshoot is because of the parent thread that continues after the *for* loop, but immediately blocks when executing the *sync* statement. Since A-Deque does not have advanced information about the program structure, it requests for more processors to explore the potential parallelism from the parent thread. The extra processor is immediately released in the next quantum when the parent blocks and no longer spawns more threads.

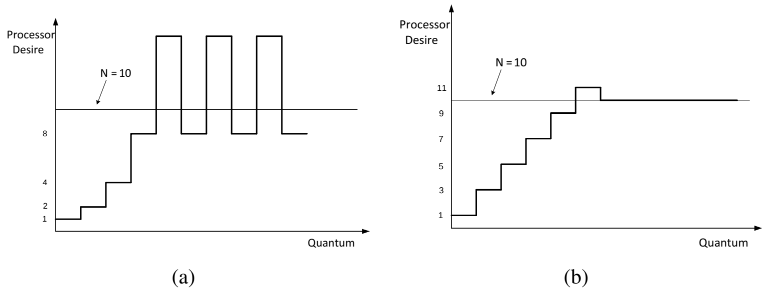


Figure 3. Processor desires calculated by (a) A-Steal and (b) A-Deque, when the parallelism of the job is constant at $N = 10$.

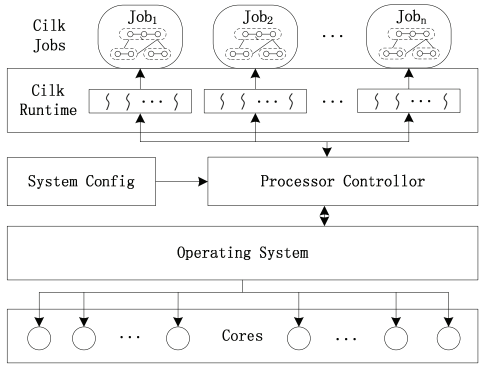


Figure 4. The adaptive scheduling framework of ACilk.

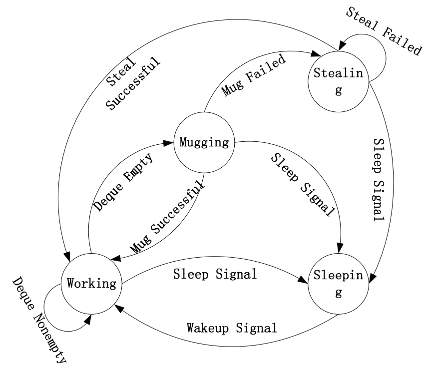


Figure 5. State diagram of a worker's execution in adaptive work stealing.

and it employs the work-stealing scheduler in its runtime system. Since the original Cilk does not support dynamically readjusting the jobs' processor allocations at runtime, based on POSIX threads libraries on Linux, ACilk modifies the Cilk runtime system to provide the ability to collect and feedback the processor desires and to handle dynamic processor allocations of jobs. To coordinate the reallocation of processors among jobs, a global processor controller is also implemented in ACilk, which takes the specified system configure information as its input, such as quantum length, feedback algorithms, and available cores, etc. Obviously, with all the information about the desire of each job in the runtime system, the global processor controller is able to provide fair and efficient processor allocations. The processor controller is implemented as a daemon process on Linux, which incurs little system overhead.

A. Processor Reallocation

To support dynamic processor reallocations for the jobs during runtime, ACilk introduces four different states, namely working, stealing, mugging, and sleeping to each processor or a worker in Cilk runtime system. ACilk ensures that the number of active workers used by a job always matches the number of physical processors assigned to it by controlling the state of the workers. The detailed process is described as follows. At the initialization stage, ACilk creates as many workers as the total number of physical processors for each job. After getting its first processor allocation (which is usually 1), ACilk puts the extra workers into the sleeping state. After each scheduling quantum, whenever the allocation of the job increases, some workers of the job are waken up. When the allotment decreases, the corresponding number of workers are put into sleeping state. Note that unlike the original work-stealing mechanism, whenever a worker runs out work, that is, its local deque becomes empty, it first enters the mugging state to look for muggable deques instead of immediately stealing work from another worker. Fig. 5 shows the state diagram of a worker's execution in adaptive work stealing.

To take advantage of the parallelism feedbacks, DEQ algorithm is implemented in the processor controller to dynamically allocate all the available processors among jobs. DEQ [21] is a variant of EQUI (Equi-partitioning) [6] that divides the total number of processors equally among all active jobs. Compared with EQUI, DEQ never allocates more processors to a job than the job's processor desire, hence it is better known for its efficiency and fairness in processor allocation [18]. Let $\mathcal{J}(q)$ denote the set of active jobs when a new quantum q begins. Based on the processor desires of all jobs collected by ACilk runtime, DEQ allocates the processors as shown in Algorithm 1, where $a_i(q)$ and $d_i(q)$ denote the processor allocation and the processor desire of job J_i in quantum q respectively, and P denotes the total number of available processors in the system.

Algorithm 1 $DEQ(\mathcal{J}(q), P)$

```
if 𝒥(q) = ∅ then
    return
S = {𝒥_i ∈ 𝒥(q) : d_i(q) ≤ P/|𝒥(q)|}
if S = ∅ then
    for each J_i ∈ 𝒥(q) do
        a_i(q) = P/|𝒥(q)|
    return
else
    for each J_i ∈ S do
        a_i(q) = d_i(q)
    DEQ(𝒥(q) − S, P − Σ_{J_i∈S} a_i(q))
```

B. Sampling Methods for A-Deque and A-Steal

As described in Section III, the desire calculation algorithms in A-Deque and A-Steal require utilization information of the allocated processors in a quantum, and A-Deque also needs the length of its active deques at any time during a quantum. Gathering these information can be very expensive in practice, which will incur a large amount of overhead in the implementation. In this subsection, we will present a more efficient implementation of the algorithms based on sampling methods that approximate the required statistics.

1) Approximating Processor Utilization: Firstly, to approximate the processor utilization in a quantum, we adopt the technique used in [22], which takes the ratio between the total number of purely unsuccessful steal attempts and the total number of all steal attempts. Specifically, for each job in quantum q, let $total\_steal_j$ denote the total number of steal attempts by the jth allocated processor, where $1 \leq j \leq a(q)$. Among all steal attempts, let $purely\_unsucc\_steal_j$ denote the total number of purely unsuccessful steal attempts. A steal attempt is called purely unsuccessful if the victim itself is attempting to steal work from other processors. The processor utilization $u(q)$ of the job in quantum q is then approximated by

$$u(q) = 1 - \frac{\sum_{j=1}^{a(q)} purely\_unsucc\_steal_j}{\sum_{j=1}^{a(q)} total\_steal_j},$$

and therefore we have $\int_{t_q}^{t_q+L} \sum_{j=1}^{a(q)} X_j(t)dt = u(q)a(q)L$.

Since a processor at any time is either working, mugging or stealing, the above ratio between the total number of purely unsuccessful steal attempts and the total number of all steal attempts gives a reasonable approximation for the inefficiency, that is $1-u(q)$, of the processors in quantum q. Apparently, the approximation will be more accurate with more steal attempts. Furthermore, since the work-stealing scheduler of Cilk runtime already has built-in counters to measure the steal attempts, collecting these information would incur very little extra overhead.

2) Approximating Active Deques Length: To approximate the length of active deques in a quantum, we again use the technique for approximating processor utilization, but combine it with the length of the deques sampled at the end of the quantum for better accuracy.

We introduce a new counter in Cilk runtime to accumulate the length of the victims' deques at every steal attempt or successful mugging for each processor j, and denote the accumulated length at the end of the quantum q by $length_j$. The approximated length of all active deques is then given by

$$Q(q) = e(t_q + L) \frac{\sum_{j=1}^{a(q)} length_j}{\sum_{j=1}^{a(q)} total\_steal_j},$$

where $e(t_q + L)$ denotes the number of active deques when quantum q ends at time $t_q + L$. Intuitively, the ratio between total accumulated deque length and total steal attempts indicates the average length of a single deque in the quantum. Again, we expect better accuracy when there are more steal attempts. In addition, we also use the lengths of the deques sampled at the end of the quantum as another approximation, and it is given by $Q'(q) = \sum_{j=1}^{e(t_q+L)} Q_j(t_q + L)$, where $Q_j(t_{q+L})$ denotes the length of the jth deque at time $t_q + L$ when quantum q ends.

The final approximation of the active deques length then takes a linear combination of the two approximations and is given by $\frac{1}{L} \int_{t_q}^{t_q+L} \sum_{j=1}^{e(t)} Q_j(t)dt = \alpha Q(q) + (1-\alpha)Q'(q)$. Intuitively, the first approximation is more accurate when there are more samples in steal attempts, thus should have higher weight. In our implementation, we set α to be the ratio between the total number of steal attempts in the quantum and the maximum possible steal attempts. Hence, the second approximation is always used in the calculation, and when there is no steal attempt in the quantum, the first approximation is simply ignored.

V. Experiments

As pointed out in previous sections, the WS (Work Stealing) algorithm implemented by original Cilk runtime system does not support dynamically readjusting the jobs' processor allocations at runtime. Therefore, manually specifying a fixed number of processors may easily lead to degraded performance when concurrently running multiple Cilk jobs, as shown in Fig. 1. In our experiments, we improve WS by applying the well-known algorithm EQUI [6] to equally share the total number of processors among all running jobs in the system. We name the new algorithm WS-EQUI and use it as a reference to evaluate the performances of different feedback-driven scheduling algorithms in the following experiments.

The experiments are all carried out on a server equipped with two Intel Xeon (E5620) quad-core processors, each with 2.4GHz clock speed and hyper-threading enabled, and the server has 8GB RAM. The operating system is Red Hat Enterprise Linux 5.2 (Linux kernel 2.6.18), and the compiler is GCC 4.1.2, with the compiling option "-g -O2". Six benchmarks are selected from the official released Cilk-5.4.6 for the experiments. The brief description and input sets of these benchmarks are listed in Table I.

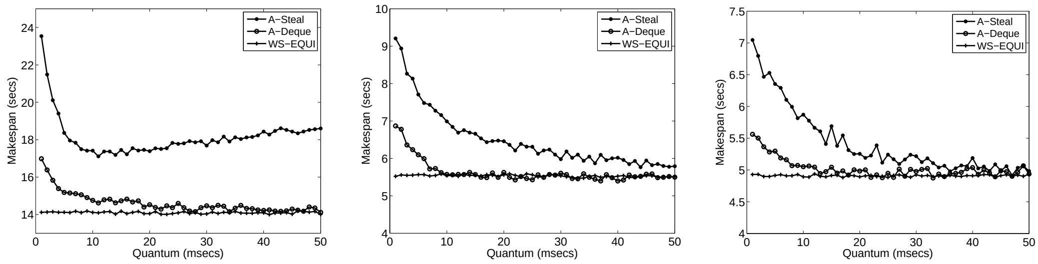
Table I
THE DESCRIPTION AND INPUT SETS OF THE BENCHMARKS

Benchmark	Description	Input sets
CK	Rudimentary checkers	-b 10 -w 13
Fib	Fibonacci numbers	46
FFT	Fast Fourier transform	-n 2^{30}
LU	LU decomposition	-n 4096
Queens	The N queens problem	26
Strassen	Multiplies two randomly generated n x n matrices	-n 4096

To compare the performances of different scheduling algorithms, we use the following metrics: makespan, mean response time, and processor utilization. The makespan is defined as the completion time of the last completed job in the job set. The response time of a single job is the time elapsed from when the job arrives to when it completes, and the mean response time of the job set is used in our experiments. The utilization of the job's allocated processors is collected by counting the time of each processor during a quantum when the processor is doing useful work. Note that the time a processor spent on stealing is considered as wasted, because although the processor is also not idle during stealing, it is not contributing towards the work of the job.

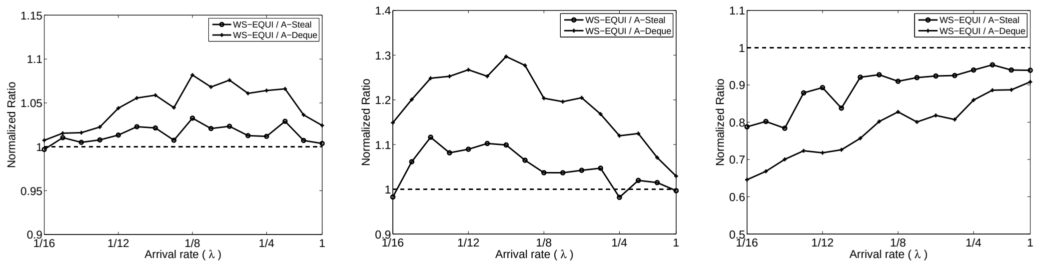
A. Scheduling Quantum and Overhead

In adaptive scheduling, the length of the scheduling quantum is an important system parameter, which may significantly affect the performance of a scheduling algorithm.



(a) CK (b) LU (c) Strassen

Figure 6. Impact of scheduling quantum and corresponding overhead on the performances of different scheduling algorithms.



(a) Makespan (b) Mean response time (c) Utilization

Figure 7. Performance Comparison of different algorithms with respect to makespan, mean response time and utilization.

Intuitively, smaller quantum length may lead to more efficiency for capturing changes in a job's parallelism, but inevitably incurs more scheduling overhead, including the cost of processor reallocation. In this subsection, we conduct a set of experiments to examine the impact of scheduling quantum and corresponding overhead on the performances of different scheduling algorithms. Specifically, only one job from the benchmark table I is used in each experiment. The quantum length is varied from 1ms to 50ms, the responsiveness parameter ρ and the utilization threshold δ of A-Steal is set to be 2 and 80% respectively, and the exploration parameter β of A-Deque is set to be 2. The experimental results for CK, LU and Strassen are shown in Fig. 6 while the other benchmarks have similar results and are not shown.

The experimental results demonstrate that, compared with A-Steal and A-Deque, the makespan of WS-EQUI is better and more stable with respect to varying scheduling quantum. The reason is that WS-EQUI is oblivious to the job's parallelism variations and always allocates all available processors to such single job, which leads to smaller running time with little scheduling overhead. The results also confirm that the original Cilk is more effective when only running one job in the system.

On the other hand, the makespans of A-Steal and A-Deque are impacted by varying scheduling quantum, especially that of A-Steal due to its unstable parallelism feedbacks. As can be seen from Fig. 6, the makespans of A-Steal and A-Deque increase rapidly when the quantum length is set to be 1ms, as the overhead incurred by the feedback-driven algorithms cannot be ignored in this case. With increasing quantum length, however, the makespans of A-Steal and A-Deque become smaller and gradually converge to that of WS-EQUI since the scheduling overhead is now better amortized over the entire scheduling quantum.

The experimental results indicate that the performance of feedback-driven algorithms are more sensitive to the scheduling quantum than WS-EQUI. Longer scheduling interval may lead to less scheduling overhead, but it may also result in slower response to the changes in parallelism of the job. Based on the experimental results, the length of the scheduling quantum is set to be 10ms in all following experiments, which seems to provide a good tradeoff between the responsiveness to the parallelism variations and the amount of scheduling overheads incurred.

B. Performance Comparison of Different algorithms

In this subsection, we evaluate and compare the performances of different scheduling algorithms using mixed workload, where jobs are released into the system according to the Poisson process and the inter-arrival time follows exponential distribution. The number of jobs is fixed to be 16 in this case, and the system load is proportional to the arrival rate λ of the jobs, which is varied from 1/16 to 1. Therefore, heavier system load corresponds to larger number of concurrently running jobs in the system. The scheduling quantum length is set to be 10ms and the parameters used in A-Steal and A-Deque are the same as in Section V-A.

The results, as shown in Fig. 7, indicate that the feedback-driven adaptive scheduling algorithms A-Steal and A-Deque generally achieve better performance than WS-EQUI with respect to all metrics. Only when the system has light load, A-Steal and A-Deque are slightly worse than WS-EQUI. The reason is that feedback-driven scheduling strategies take advantage of the parallelism feedback based on the information of execution history while WS-EQUI is oblivious to the job's parallelism and thus wastes many processor resources, especially when the system load is light. Furthermore, as shown in Fig. 7, the performances of A-Steal and A-Deque tend to gradually converge to that of WS-EQUI with heavy system load because in this case each job can only receive very few processors most of the time, and thus frequent processor reallocations have no obvious benefits.

The experimental results further demonstrate that A-Deque has better performance than both of the other algorithms for all system loads with respect to makespan, mean response

Table II
AVERAGE PERFORMANCE IMPROVEMENTS OF A-DEQUE OVER A-STEAL AND WS-EQUI.

	Makespan	Response time	Utilization
A-Deque / A-Steal	3.14%	13.57%	14.46%
A-Deque / WS-EQUI	4.61%	19.13%	28.96%

time, and utilization. As shown in Table II, the average makespan improvements of A-Deque over A-Steal and WS-EQUI are 3.14% and 4.61% respectively. While the makespan improvements seem small as it could be easily dominated by one large job in the job set with long executing time, the performances improvements of A-Deque with respect to mean response time and utilization are more significant. Specifically, the average performance improvements of A-Deque over A-Steal and WS-EQUI reach up to 19.13% and 28.96% with respect to mean response time and processor utilization respectively. These correspond to much improved performance for each individual user or application as well as better utilization of the system resources. In summary, these experimental results indicate that A-Deque directly benefits from its more accurate and stable parallelism feedbacks, as shown in Section III-C, and shows the best of its performance under light to medium system loads. In general, feedback-driven scheduling algorithms have been demonstrated to be beneficial for scheduling parallel applications with dynamic changing parallelism, and better overall performance will be achieved with more effective feedback mechanism.

VI. CONCLUSION

In this paper, we study feedback-driven adaptive scheduling based on work stealing load balancing strategy, which provides an efficient solution to better utilize the available processor resources and improve efficiency in concurrently executing parallel applications on multi-core platforms. The benefit of adaptive scheduling is that it not only eliminates the need of manually specifying the number of cores required by most existing multi-core runtime systems, but also enhances the overall system performance by exploiting the runtime characteristics of parallel applications. The experimental results demonstrate that feedback-driven adaptive scheduling algorithms achieve better performance with respect to makespan, mean response time and processor utilization, especially when more accurate and stable feedback mechanism is applied. For our future work, we plan to integrate our adaptive scheduling algorithm to the Linux kernel, which will provide more benefits to efficiently control and collaborate with multi-core runtime systems.

ACKNOWLEDGMENT

This work is supported in part by Natural Science Foundation of China under the grant No. 61073011 and China National Hi-tech Research and Development Program (863 Program) under the grants No. 2009AA01A135, 2009AA01A131.

REFERENCES

[1] M. Hill, M. Marty. Amdahl's law in the multicore era. Computer, 41(7):33–38, 2008.

[2] B. Chapman, and L. Huang. Enhancing OpenMP and its implementation for programming multicore systems. Parallel computing: architectures, algorithms, and applications. IOS Press Inc, Amsterdam, 2008.

[3] J. Reinders. Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. Sebastopol, CA: O'Reilly Media, 2007.

[4] M. Frigo and C. E. Leiserson and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, 1998.

[5] Intel Cilk Plus http://software.intel.com/en-us/articles/intel-cilk-plus/

[6] J. Edmonds. Scheduling in the dark. In *STOC*, 1999.

[7] K. Agrawal, Y. He, W.-J. Hsu, and C. E. Leiserson. Adaptive scheduling with parallelism feedback. In *PPoPP*, 2006.

[8] K. Agrawal, Y. He, and C. E. Leiserson. An empirical evaluation of work stealing with parallelism feedback. In *ICDCS*, 2006.

[9] Y. He, W.-J. Hsu, and C. E. Leiserson. Provably efficient online non-clairvoyant adaptive scheduling. In *IPDPS*, 2007.

[10] 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.

[11] Y. Cao, H. Sun, W.-J. Hsu and D. Qian. Malleable-Lab: A tool for evaluating adaptive online schedulers on malleable jobs. In *PDP*, 2010.

[12] R. D. Blumofe and C. E. Leiserson. Space-Efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, 1998.

[13] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[14] A. Duran, J. Corbalán, and E. Ayguadé. Evaluation of OpenMP task scheduling strategies, Proceedings of the 4th international conference on OpenMP in a new era of parallelism, 2008.

[15] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[16] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 1563–1581, 1966.

[17] K. Agrawal, Y. He, and C. E. Leiserson. Adaptive work stealing with parallelism feedback. In *PPoPP*, 2007.

[18] Y. He, W.-J. Hsu, and C. E. Leiserson. Provably efficient two-level adaptive scheduling. In *JSSPP*, 2006.

[19] H. Sun and W.-J. Hsu. Adaptive B-Greedy (ABG): A simple yet efficient scheduling algorithm. In *IPDPS*, 2008.

[20] H. Sun, Y. Cao, and W.-J. Hsu. Competitive two-level adaptive scheduling using resource augmentation. In *JSSPP*, 2009.

[21] 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.

[22] S. Sen. Dynamic processor allocation for adaptively parallel jobs. Master's thesis, Massachusetts Institute of technology, 2004.