Adaptive Work-Stealing with Parallelism Feedback

KUNAL AGRAWAL and CHARLES E. LEISERSON
Massachusetts Institute of Technology
and
YUXIONG HE and WEN JING HSU
Nanyang Technological University

Multiprocessor scheduling in a shared multiprogramming environment can be structured as two-level scheduling, where a kernel-level job scheduler allots processors to jobs and a user-level thread scheduler schedules the work of a job on its allotted processors. We present a randomized work-stealing thread scheduler for fork-join multithreaded jobs that provides continual parallelism feedback to the job scheduler in the form of requests for processors. Our A-STEAL algorithm is appropriate for large parallel servers where many jobs share a common multiprocessor resource and in which the number of processors available to a particular job may vary during the job's execution. Assuming that the job scheduler never allots a job more processors than requested by the job's thread scheduler, A-STEAL guarantees that the job completes in near-optimal time while utilizing at least a constant fraction of the allotted processors.

We model the job scheduler as the thread scheduler's adversary, challenging the thread scheduler to be robust to the operating environment as well as to the job scheduler's administrative policies. For example, the job scheduler might make a large number of processors available exactly when the job has little use for them. To analyze the performance of our adaptive thread scheduler under this stringent adversarial assumption, we introduce a new technique called *trim analysis,* which allows us to prove that our thread scheduler performs poorly on no more than a small number of time steps, exhibiting near-optimal behavior on the vast majority.

More precisely, suppose that a job has work T_1 and span T_∞. On a machine with P processors, A-STEAL completes the job in an expected duration of $O(T_1/\widetilde{P} + T_\infty + L \lg P)$ time steps, where L is the length of a scheduling quantum, and $\widetilde{P}$ denotes the $O(T_\infty + L \lg P)$-trimmed availability. This quantity is the average of the processor availability over all time steps except the $O(T_\infty + L \lg P)$ time steps that have the highest processor availability. When the job's parallelism dominates the trimmed availability, that is, $\widetilde{P} \ll T_1/T_\infty$, the job achieves nearly perfect linear speedup. Conversely, when the trimmed mean dominates the parallelism, the asymptotic running time of the job is nearly the length of its span, which is optimal.

We measured the performance of A-STEAL on a simulated multiprocessor system using synthetic workloads. For jobs with sufficient parallelism, our experiments confirm that A-STEAL

This research was supported in part by the Singapore-MIT Alliance and NSF Grants ACI-0324974 and CNS-0305606.
Author's address: K. Agrawal, MIT, Cambridge, MA; email: kunal@csail.mit.edu.

provides almost perfect linear speedup across a variety of processor availability profiles. We compared A-Steal with the ABP algorithm, an adaptive work-stealing thread scheduler developed by Arora et al. [1998] which does not employ parallelism feedback. On moderately to heavily loaded machines with large numbers of processors, A-Steal typically completed jobs more than twice as quickly as ABP, despite being allotted the same number or fewer processors on every step, while wasting only 10% of the processor cycles wasted by ABP.

Categories and Subject Descriptors: D.4.1 [**Operating Systems**]: Process Management; F.2 [**Analysis of Algorithms and Problem Complexity**]

General Terms: Algorithms, Performance, Theory, Experimentation

Additional Key Words and Phrases: Adaptive scheduling, adversary, span, instantaneous parallelism, job scheduling, multiprocessing, multiprogramming, parallelism feedback, parallel computation, processor allocation, randomized algorithm, thread scheduling, two-level scheduling, space sharing, trim analysis, work, work-stealing

ACM Reference Format:

Agrawal, K., He, Y., Hsu, W. J., and Leiserson, C. E. 2008. Adaptive work-stealing with parallelism feedback. ACM Trans. Comput. Syst. 26, 3, Article 7 (September 2008), 32 pages. DOI=10.1145/1394441.1394443 http://doi.acm. org/10.1145/1394441.1394443

1. INTRODUCTION

When many parallel applications share a multiprocessor machine, the system must supply a scheduling infrastructure to coordinate the multiprogrammed workload. As Feitelson mentions in his excellent survey [Feitelson 1997], schedulers for these machines can be implemented using two levels: a kernel-level *job scheduler*, which allots processors to jobs, and a user-level *thread scheduler*, which schedules the threads belonging to a given job onto the allotted processors. A job scheduler may implement either *space-sharing*, where jobs occupy disjoint processor resources, or *time-sharing*, where different jobs may share the same processor resources at different times. Either or both the thread scheduler and the job scheduler may be *adaptive* (called "dynamic" in Chiang and Vernon [1996]), where the number of processors allotted to a job may change while the job is running, or *nonadaptive* (called "static" in Chiang and Vernon [1996]), where a job runs on a fixed number of processors for its lifetime.

Prior work on thread scheduling for multithreaded jobs has tended to focus on nonadaptive scheduling [Blumofe and Leiserson 1999; Blelloch et al. 1995; Graham 1969; Brent 1974; Blelloch and Greiner 1996; Narlikar and Blelloch 1999] or adaptive scheduling without parallelism feedback [Arora et al. 1998]. For jobs whose parallelism is unknown in advance and which may change during execution, nonadaptive scheduling may waste processor cycles [Squillante 1995], because a job with low parallelism may be allotted more processors than it can productively use. Moreover, in a multiprogrammed environment, nonadaptive scheduling may not allow a new job to start, because existing jobs may already be using most of the processors. Although adaptive scheduling without parallelism feedback allows jobs to enter the system, the job still may waste processor cycles if it is alloted more processors than it can use.

The solution we present is an adaptive scheduling strategy where the thread scheduler provides parallelism feedback to the job scheduler so that when a job cannot use many processors, those processors can be reallotted to jobs that need them. Based on this parallelism feedback, the job scheduler adaptively changes the allotment of processors according to the availability of processors in the current system environment and the job scheduler's administrative policy.

The problem of how the job scheduler should partition the multiprocessor among the various jobs has been studied extensively [Deng et al. 1996; Deng and Dymond 1996; Gu 1995; Motwani et al. 1993; McCann et al. 1993; Edmonds 1999; Leutenegger and Vernon 1990; Rosti et al. 1995; Rosti et al. 1994; Yue and Lilja 2001; Martorell et al. 2000; Edmonds et al. 2003], but the administrative policy of the job scheduler is not the focus of this article. Instead, we focus on how the thread scheduler can provide effective parallelism feedback to the job scheduler without knowing the future progress of the job, the future availability of processors, or the administrative priorities of the job scheduler.

Various researchers [Deng et al. 1996; Deng and Dymond 1996; Gu 1995; McCann et al. 1993; Yue and Lilja 2001] have explored the notion of *instantaneous parallelism*,[1] the number of processors the job can effectively use at the current moment, as the parallelism feedback to the job scheduler. Although using instantaneous parallelism as parallelism feedback is simple, it can cause gross misallocation of processor resources [Sen 2004]. For example, the parallelism of a job may change substantially during a scheduling quantum, alternating between parallel and serial phases. The sampling of instantaneous parallelism at a scheduling event between quanta may lead the thread scheduler to request either too many or too few processors, depending on which phase is currently active, whereas the desirable request might be something in between. Consequently, the job may waste processor cycles or take too long to complete.

In this article, we present an adaptive thread scheduler, called A-STEAL, which provides provably good parallelism feedback. A-STEAL guarantees not to waste many processor cycles while simultaneously ensuring that the job completes quickly. Instead of using instantaneous parallelism, A-STEAL provides parallelism feedback to the job scheduler based on a single summary statistic and the job's behavior in the previous quantum. Even though A-STEAL provides parallelism feedback using the past behavior of the job, and we do not assume that the job's future parallelism is correlated with its history of parallelism, our analysis shows that A-STEAL schedules the job well with respect to both waste and completion time.

As with prior work on scheduling of multithreaded jobs [Blumofe and Leiserson 1998, 1999; Blumofe 1995; Blelloch and Greiner 1996; Blelloch et al. 1999; Fang et al. 1990; Hummel and Schonberg 1991; Narlikar and Blelloch 1999], we model the execution of a multithreaded job as a dynamically unfolding directed acyclic graph (*dag*), where each node in the dag represents a unit-time instruction, and an edge represents a precedence relationship between nodes. In this model, a *thread* is a serial chain of nodes with no branches. A node may *spawn* (create) another thread, in which case one directed edge connects the spawning node to the first node of the spawned thread and a second directed edge connects

[1]These researchers actually use the general term "parallelism," but we prefer the more descriptive term.

the spawning node to the first node of a *continuation* thread. When two threads *join*, a directed edge connects their last nodes to the first node of a succeeding thread. A node becomes *ready* when all its predecessors have been executed, and a thread becomes ready when its first node becomes ready. The *work* T_1 of the job corresponds to the total number of nodes in the dag, and the *span*[2] T_∞ corresponds to the number of nodes on the longest chain of dependencies. Each job has its own thread scheduler, which operates in an online manner, oblivious to the future characteristics of the dynamically unfolding dag.

Our scheduling model is as follows. We assume that time is broken into a sequence of equal-sized *scheduling quanta* $1, 2, \ldots,$ each consisting of L time steps. The job scheduler is free to reallocate processors between quanta. The *quantum length* L is a system configuration parameter chosen to be long enough to amortize the time to reallocate processors among the various jobs, and to perform various other bookkeeping for scheduling, including communication between the thread scheduler and the job scheduler, which typically might involve a system call. Between quanta $q-1$ and q, the thread scheduler determines its job's *desire* d_q, which is the number of processors the job wants for quantum q. The thread scheduler provides the desire, d_q to the job scheduler as its parallelism feedback. The job scheduler follows some processor allocation policy to determine the *processor availability* p_q, or the maximum number of processors to which the job is entitled for the quantum q. We assume that the job scheduler decides the availability of processors as an adversary in order to make the thread scheduler robust to different system environments and administrative policies. The number of processors the job receives for quantum q is the job's *allotment* $a_q = \min\{d_q, p_q\}$, which is the smaller of its desire and the processor availability. Once a job is allotted its processors, the allotment does not change during the quantum. Consequently, the thread scheduler must do a good job, before a quantum, of estimating how many processors it will need for all L time steps of the quantum, besides doing a good job in scheduling the ready threads on the allotted processors.

In an adaptive setting, where the number of processors allotted to a job can change during execution, both $T_1/\bar{P}$ and T_∞ impose lower bounds on the running time, where $\bar{P}$ is the mean of the processor availability during the computation. In the worst case, however, an adversarial job scheduler can prevent any thread scheduler from providing good speedup with respect to the mean availability $\bar{P}$. For example, if the adversarial job scheduler chooses a large number of processors for the job's processor availability just when the job has little instantaneous parallelism, no adaptive thread scheduler can effectively utilize the available processors on that quantum.[3]

We introduce a technique called *trim analysis* to analyze adaptive thread schedulers under these adversarial conditions. From the field of statistics, trim

[2]Also called *critical-path length* or *computation depth* in the literature.

[3]Evaluating the thread scheduler with respect to mean processor allotment, rather than mean processor availability, produces uninteresting results, because the thread scheduler can perform optimally by simply refusing to schedule a job's work in parallel. That is, the thread scheduler requests exactly 1 processor for each quantum, thereby achieving perfect linear speedup with respect to the mean processor allotment (1) while wasting 0 processor cycles.

analysis borrows the idea of ignoring a few *outliers*. A *trimmed mean*, for example, is calculated by discarding a certain number of lowest and highest values, and then computing the mean of those that remain. For our purposes, it suffices to trim the availability from just the high side. For a given value R, we define the *R-high-trimmed mean availability* as the mean availability after ignoring the R steps with the highest availability, or just *R-trimmed availability*, for short. A good thread scheduler should provide linear speedup with respect to an R-trimmed availability, where R is as small as possible.

This article proves that A-Steal guarantees linear speedup with respect to the $O(T_\infty + L \lg P)$-trimmed availability. Specifically, consider a job with work T_1 and span T_∞ running on a machine with P processors and a scheduling quantum of length L. A-Steal completes the job in an expected duration of $O(T_1/\widetilde{P} + T_\infty + L \lg P)$ time steps, where $\widetilde{P}$ denotes the $O(T_\infty + L \lg P)$-trimmed availability. Thus the job achieves linear speedup with respect to the trimmed availability $\widetilde{P}$ when the average parallelism T_1/T_∞ dominates $\widetilde{P}$. In addition, we prove that the total number of processor cycles wasted by the job is $O(T_1)$, representing at most a constant factor overhead.

To better understand which constants in our theoretical analysis are due to analysis and which are inherent in the scheduler, we implemented A-Steal in a simulated multiprocessor environment. On a large variety of workloads running with a variety of availability profiles, our experiments indicate that A-Steal provides nearly perfect linear speedup when the jobs have ample parallelism. Moreover, A-Steal typically wastes less than 20% of the allotted processor cycles. We compared the performance of A-Steal with the performance of *ABP*, an adaptive scheduler proposed by Arora et al. [1998]. We ran single jobs using both A-Steal and ABP with the same availability profiles. We found that on moderately to heavily loaded machines with large number of processors, when $\bar{P} \ll P$, A-Steal completes almost all jobs about twice as quickly as ABP on average, despite the fact that ABP's allotment on any quantum is never less than A-Steal's allotment on the same quantum. In most of these job runs, A-Steal wastes less than 10% of the processor cycles wasted by ABP.

Portions of this work were previously reported in three conference papers [Agrawal et al. 2006a; Agrawal et al. 2006b, 2007]. The desire-estimation algorithm and the concept of trim analysis were introduced in Agrawal et al. [2006b] in the context of a centralized adaptive task scheduler, called A-Greedy, which is suitable for scheduling data-parallel jobs. Agrawal et al. [2006b] combined A-Greedy's desire-estimation algorithm with adaptive work-stealing, and introduced the distributed adaptive thread scheduler A-Steal with its empirical results. The theoretical results obtained by applying trim analysis to A-Steal were first reported in Agrawal et al. [2007]. The present article combines the best of these three preliminary works with new material, to make an integrated presentation of work-stealing with parallelism feedback.

The remainder of this article is organized as follows: Section 2 describes the A-Steal algorithm and Section 3 provides a trim analysis of its performance with respect to time and waste. Section 4 describes our empirical evaluation of A-Steal in a simulated environment. Section 5 describes work in adaptive and nonadaptive scheduling. Finally, Section 6 offers some concluding remarks.

2. ADAPTIVE WORK-STEALING

This section presents the adaptive work-stealing thread scheduler A-STEAL. Before the start of a quantum, A-STEAL estimates processor desire based on the job's history of utilization, to provide parallelism feedback to the job scheduler. In this section, we describe A-STEAL and its desire-estimation algorithm.

During a quantum, A-STEAL uses *work-stealing* [Blumofe and Leiserson 1999; Arora et al. 1998; Mohr et al. 1990] to schedule the job's threads on the allotted processors. A-STEAL can use any provably good work-stealing algorithm, such as that of Blumofe and Leiserson [1999] or the nonblocking one presented by Arora et al. [1998].[4] In these work-stealing thread schedulers, every processor allotted to the job maintains a double-ended queue, or *deque*, of ready threads for the job. When the current thread spawns a new thread, the processor pushes the continuation of the current thread onto the top of the deque and begins working on the new thread. When the current thread completes or blocks, the processor pops the topmost thread off the deque to work on. If the deque of a processor is empty, however, the processor becomes a *thief*, randomly picking a *victim* processor and *stealing* work from the bottom of the victim's deque. If the victim has no available work, then the steal is *unsuccessful*, and the thief continues to steal at random from the other processors until it is *successful* and finds work. At all times, every processor is either working or stealing.

2.1 Making Work-Stealing Adaptive

This work-stealing algorithm must be modified to deal with dynamic changes in processor allotment to the job, between quanta. Two simple modifications make the work-stealing algorithm adaptive:

Allotment gain. When the allotment increases from quantum $q-1$ to q, the job scheduler obtains $a_q - a_{q-1}$ additional processors. Since the deques of these new processors start out empty, all these processors immediately start stealing to get work from the other processors.

Allotment loss. When the allotment decreases from quantum $q-1$ to q, the job scheduler deallocates $a_{q-1} - a_q$ processors, whose deques may be nonempty. To deal with these deques, we use the concept of *mugging* [Blumofe et al. 1998]. When a processor runs out of work, instead of stealing immediately, it looks for a *muggable* deque, a nonempty deque that has no associated processor working on it. Upon finding a muggable deque, the thief *mugs* the deque by taking over the entire deque as its own. Thereafter, it works on the deque as if it were its own. If there are no muggable deques, the thief steals normally. Data structures can be set up between quanta so that stealing and mugging can be accomplished in $O(1)$ time [Sen 2004].

At all time steps during the execution of A-STEAL, every processor is either working, stealing, or mugging. We call the cycles that a processor spends on working, stealing, and mugging as *work-cycles*, *steal-cycles*, and *mug-cycles*,

[4]These algorithms impose some additional restrictions on the job, for example, that each node has an out-degree of at most 2. Whatever restrictions assumed by the underlying work-stealing algorithm apply to A-STEAL as well.

respectively. We assume, without loss of generality, that work-cycles, steal-cycles, and mug-cycles all take single time steps. We bound time and waste in terms of these elementary processor cycles. Cycles spent stealing and mugging are *wasted*, and the total waste is the sum of the number of steal-cycles and mug-cycles during the execution of the job.

2.2 A-STEAL's Desire-Estimation Heuristic

We now present A-STEAL's desire-estimation algorithm. To estimate the desire for the next quantum $q + 1$, A-STEAL classifies the previous quantum q as either *satisfied* or *deprived*, and either *efficient* or *inefficient.* Of the four possibilities of classification, A-STEAL only uses three: inefficient, efficient-and-satisfied, and efficient-and-deprived. Using this three-way classification, and the job's desire for the previous quantum q, it computes the desire for the next quantum $q + 1$.

To classify a quantum q as satisfied versus deprived, A-STEAL compares the job's allotment a_q with its desire d_q. The quantum q is *satisfied* if $a_q = d_q$, that is, the job receives as many processors as A-STEAL requested on its behalf from the job scheduler. Otherwise, if $a_q < d_q$, the quantum is *deprived*, because the job did not receive as many processors as A-STEAL requested.

Classifying a quantum as efficient versus inefficient is more complicated, and is based on using a *utilization parameter* δ as a threshold to differentiate between the two cases. We define the *nonsteal usage* n_q as the sum of the number of work-cycles and mug-cycles. We call a quantum q *efficient* if $n_q \geq \delta L a_q$, that is, the nonsteal usage is at least a δ fraction of the total processor cycles allotted. A quantum is *inefficient* otherwise. Inefficient quanta contain at least $(1 - \delta)La_q$ steal-cycles. Although it might seem counterintuitive for the definition of *efficient* to include mug-cycles, which, after all, are wasted, the rationale is that mug-cycles arise as a result of an allotment loss, and do not generally indicate that the job has a surplus of processors.

A-STEAL calculates the desire d_q of the current quantum q, based on the previous desire d_{q-1} and the three-way classification of quantum $q - 1$ as inefficient, efficient-and-satisfied, or efficient-and-deprived. The initial desire is $d_1 = 1$. A-STEAL uses a *responsiveness parameter* $\rho > 1$ to determine how quickly the scheduler responds to changes in parallelism.

Figure 1 shows the pseudocode of A-STEAL for one quantum. The algorithm takes as input the quantum q, the utilization parameter δ, and the responsiveness parameter ρ. It then operates as follows:

- If quantum $q - 1$ was inefficient, it contained many steal-cycles, indicating that most of the processors had insufficient work to do. Therefore, A-STEAL overestimated the desire for quantum $q - 1$. In this case, A-STEAL does not care whether quantum $q - 1$ was satisfied or deprived. It simply decreases the desire (line 4) for quantum q.
- If quantum $q - 1$ was efficient-and-satisfied, the job effectively utilized the processors that A-STEAL requested on its behalf. In this case, A-STEAL speculates that the job can use more processors. It increases the desire (line 6) for quantum q.

```
A-STEAL $(q, \delta, \rho)$
1   if $q = 1$
2       then $d_q \leftarrow 1$                  ▷ base case
3   elseif $n_{q-1} < L\delta a_{q-1}$
4       then $d_q \leftarrow d_{q-1}/\rho$       ▷ inefficient
5   elseif $a_{q-1} = d_{q-1}$
6       then $d_q \leftarrow \rho d_{q-1}$      ▷ efficient-and-satisfied
7   else $d_q \leftarrow d_{q-1}$               ▷ efficient-and-deprived
8   Report $d_q$ to the job scheduler.
9   Receive allotment $a_q$ from the job scheduler.
10  Schedule on $a_q$ processors using randomized work
    stealing for $L$ time steps.
```

Fig. 1. Pseudocode for the adaptive work-stealing thread scheduler A-STEAL, which provides parallelism feedback to a job scheduler in the form of processor desire. Before quantum q, A-STEAL uses the previous quantum's desire d_{q-1}, allotment a_{q-1}, and nonsteal usage n_{q-1} to compute the current quantum's desire d_q, based on the utilization parameter δ and the responsiveness parameter ρ.

—If quantum $q-1$ was efficient-and-deprived, the job used all the processors it was allotted, but A-STEAL had requested more processors for the job than the job actually received from the job scheduler. Since A-STEAL has no evidence as to whether the job could have used all the processors requested, it maintains the same desire (line 7) for quantum q.

3. TRIM ANALYSIS OF A-STEAL

This section uses a trim analysis to analyze A-STEAL with respect to both time and waste. Suppose that A-STEAL schedules a job with work T_1 and span T_∞ on a machine with P processors. Let ρ denote A-STEAL 's responsiveness parameter, δ the utilization parameter, and L the quantum length. We will show that A-STEAL completes the job in time $T = O\{T_1/\widetilde{P} + T_\infty + L \lg P + L \ln(1/\epsilon)\}$ with probability at least $1-\epsilon$, where $\widetilde{P}$ denotes the $O(T_\infty + L \lg P + L \ln(1/\epsilon))$-trimmed availability. This bound implies that A-STEAL achieves linear speedup on all the time steps, excluding at most $O(T_\infty + L \lg P + L \ln(1/\epsilon))$ time steps with highest processor availability. Moreover, A-STEAL guarantees that the total number of processor cycles wasted during the job's execution is $W = O(T_1)$.

We prove these bounds using a trim analysis. We label each quantum as either *accounted* or *deductible*. Accounted quanta are those with $n_q \geq L\delta p_q$, where n_q denotes the nonsteal usage. That is, the job works or mugs for at least a δ fraction of the Lp_q processor cycles possibly available during the quantum. Conversely, the deductible quanta are those where $n_q < L\delta p_q$. Our trim analysis will show that when we ignore the relatively few deductible quanta, we obtain linear speedup on the more numerous accounted quanta. We can relate this labeling to a three-way classification of quanta as inefficient, efficient-and-satisfied, or efficient-and-deprived:

—*Inefficient*. In an inefficient quantum q, we have $n_q < L\delta a_q \leq L\delta p_q$, since the allotment a_q never exceeds the availability p_q. Thus we label all inefficient quanta as deductible, irrespective of whether they are satisfied or deprived.

—*Efficient-and-satisfied*. On an efficient quantum q, we have $n_q \geq L\delta a_q$. Since we have $a_q = \min\{p_q, d_q\}$, for a satisfied quantum it follows that $a_q = d_q \leq p_q$. Despite these two bounds, we may nevertheless have $n_q < L\delta p_q$. Since we cannot guarantee that $n_q \geq L\delta p_q$, we pessimistically label the quantum q as deductible.

—*Efficient-and-deprived*. As before, on an efficient quantum q, we have $n_q \geq L\delta a_q$. On a deprived quantum, we have $a_q < d_q$ by definition. Since $a_q = \min\{p_q, d_q\}$, we must have $a_q = p_q$. Hence, it follows that $n_q \geq L\delta a_q = L\delta p_q$, and we label quantum q as accounted.

3.1 Time Analysis

We now analyze the execution time of A-STEAL by separately bounding the number of deductible and accounted quanta. Two observations provide intuition for the proof. First, each inefficient quantum contains a large number of steal-cycles, which we can expect to reduce the length of the remaining span. This observation will help us to bound the number of deductible quanta. Second, most of the processor cycles in an efficient quantum are spent either working or mugging. We will show that there cannot be too many mug-cycles during the job's execution, and thus most of the processor cycles on efficient quanta are spent doing useful work. This observation will help us to bound the number of accounted quanta.

The following lemma, proved in Lemma 11 of Blumofe and Leiserson [1999], shows how steal-cycles reduce the length of the job's span.

LEMMA 1. *If a job has r deques of ready threads, then $3r$ steal-cycles suffice to reduce the length of the job's remaining span by at least 1, with probability at least $1 - 1/e$, where e is the base of the natural logarithm.*

The next lemma shows that an inefficient quantum reduces the length of the job's span, which we will later use to bound the total number of inefficient quanta.

LEMMA 2. *Let δ denote A-STEAL's utilization parameter, and L the quantum length. With probability greater than $1/4$, A-STEAL reduces the length of a job's remaining span in an inefficient quantum by at least $(1 - \delta)L/6$.*

PROOF. Let q be an inefficient quantum. A processor with an empty deque steals only when it cannot mug a deque, and hence, all the steal-cycles in quantum q occur when the number of nonempty deques is at most the allotment a_q. Therefore, by Lemma 1, $3a_q$ steal-cycles suffice to reduce the span by 1, with probability at least $1 - 1/e$. Since the quantum q is inefficient, it contains at least $(1 - \delta)La_q$ steal-cycles. Divide the time steps of the quantum into *rounds*, such that each round contains $3a_q$ steal-cycles, except for possibly the last. Thus, there are at least $m = (1 - \delta)La_q/3a_q = (1 - \delta)L/3$ rounds.[5]

[5]Actually, the number of rounds is $m = \{(1 - \delta)L/3\}$, but we will ignore the roundoff for simplicity.

We call a round *good* if it reduces the length of the span by at least 1; otherwise, the round is *bad*. For each round i in quantum q, we define the indicator random variable X_i to be 1, if round, i is a bad round, and 0 otherwise, and let $X = \sum_{i=1}^{m} X_i$. Since we have $\{X_i = 1\} < 1/e$, linearity of expectation dictates that $\mathrm{E}[X] < m/e$. We now apply Markov's inequality [Cormen et al. 2001, p. 1111], which says that for a nonnegative random variable X, we have $\Pr\{X \geq t\} \leq \mathrm{E}[X]/t$ for all $t > 0$. Substituting $t = m/2$, we obtain $\Pr\{X > m/2\} \leq \mathrm{E}[X]/(m/2) \leq (m/e)/(m/2) = 2/e < 3/4$. Thus, the probability exceeds $1/4$ that quantum q contains at least $m/2$ good rounds. Since each good round reduces the span by at least 1, with probability greater than $1/4$, the span is reduced during quantum q by at least $m/2 = ((1-\delta)L/3)/2 = (1-\delta)L/6$. □

LEMMA 3. *Suppose that* A-STEAL *schedules a job with span T_∞ on a machine. Let ρ denote* A-STEAL *'s responsiveness parameter, δ the utilization parameter, and L the quantum length. Then, for any $\epsilon > 0$, with probability at least $1-\epsilon$, the schedule produces at most $48T_\infty/(L(1-\delta)) + 16\ln(1/\epsilon)$ inefficient quanta.*

PROOF. Let I be the set of inefficient quanta. Define an inefficient quantum q as *productive* if it reduces the span by at least $(1-\delta)L/6$, and *unproductive* otherwise. For each quantum $q \in I$, define the indicator random variable Y_q to be 1, if q is productive, and 0 otherwise. By Lemma 2, we have $\Pr\{Y_q = 1\} > 1/4$. Let the total number of productive quanta be $Y = \sum_{q \in I} Y_q$. For simplicity in notation, let $A = 6T_\infty/((1-\delta)L)$. If the job's execution contains $|I| \geq 48T_\infty/((1-\delta)L) + 16\ln(1/\epsilon)$ inefficient quanta, then we have $\mathrm{E}[Y] > |I|/4 \geq 12T_\infty/((1-\delta)L) + 4\ln(1/\epsilon) = 2A + 4\ln(1/\epsilon)$. Using the Chernoff bound $\Pr\{Y < (1-\lambda)E[Y]\} < \exp(-\lambda^2 E[Y]/2)$ [Motwani and Raghavan 1995, p. 70] and choosing $\lambda = (A + 4\ln(1/\epsilon))/(2A + 4\ln(1/\epsilon))$, we obtain

$$\begin{aligned}
&\Pr\{Y < A\} \\
&= \Pr\left\{Y < \left(1 - \frac{A + 4\ln(1/\epsilon)}{2A + 4\ln(1/\epsilon)}\right)(2A + 4\ln(1/\epsilon))\right\} \\
&= \Pr\{Y < (1-\lambda)(2A + 4\ln(1/\epsilon))\} \\
&< \exp\left(-\frac{\lambda^2}{2}(2A + 4\ln(1/\epsilon))\right) \\
&= \exp\left(-\frac{1}{2} \cdot \frac{(A + 4\ln(1/\epsilon))^2}{2A + 4\ln(1/\epsilon)}\right) \\
&< \exp\left(-\frac{1}{2} \cdot 4\ln(1/\epsilon) \cdot \frac{1}{2}\right) \\
&= \epsilon\ .
\end{aligned}$$

Therefore, if the number $|I|$ of inefficient quanta is at least $48T_\infty/((1-\delta)L) + 16\ln(1/\epsilon)$, the number of productive quanta is at least $A = 6T_\infty/((1-\delta)L)$, with probability at least $1-\epsilon$. By Lemma 2 each productive quantum reduces the span by at least $(1-\delta)L/6$, and therefore at most $A = 6T_\infty/((1-\delta)L)$ productive

A more detailed analysis can nevertheless produce the same constants in the bounds for Lemmas 3 and 6.

quanta occur during the job's execution. Consequently, with probability at least $1 - \epsilon$, the number of inefficient quanta is $|I| \leq 48T_\infty/((1-\delta)L) + 16\ln(1/\epsilon)$. □

The following technical lemma bounds the maximum value of desire.

LEMMA 4. *Suppose that* A-STEAL *schedules a job on a machine with P processors. Let ρ denote* A-STEAL*'s responsiveness parameter. Before any quantum q, the desire d_q of the job is at most ρP.*

PROOF. We use induction on the number of quanta. The base case $d_1 = 1$ holds trivially. If a given quantum $q-1$ was inefficient, the desire d_q decreases, and thus $d_q < d_{q-1} \leq \rho P$ by induction. If quantum $q-1$ was efficient-and-satisfied, then $d_q = \rho d_{q-1} = \rho a_{q-1} \leq \rho P$. If quantum $q-1$ was efficient-and-deprived, then $d_q = d_{q-1} \leq \rho P$ by induction. □

The next lemma reveals a relationship between inefficient quanta and efficient-and-satisfied quanta.

LEMMA 5. *Suppose that* A-STEAL *schedules a job on a machine with P processors. If ρ is* A-STEAL *'s responsiveness parameter, and the schedule produces m inefficient quanta, then it produces at most $m + \log_\rho P + 1$ efficient-and-satisfied quanta.*

PROOF. Assume for the purpose of contradiction that a job's execution produces $k > m + \log_\rho P + 1$ efficient-and-satisfied quanta. Recall that the desire increases by ρ after every efficient-and-satisfied quantum, decreases by ρ after every inefficient quantum, and does not change otherwise. Thus, the total increase in desire is ρ^k, and the total decrease in desire is ρ^m. Since the desire starts at 1, the desire at the end of the job is $\rho^{k-m} > \rho^{\log_\rho P+1} > \rho P$, contradicting Lemma 4. □

The following lemma bounds the number of efficient-and-satisfied quanta.

LEMMA 6. *Suppose that* A-STEAL *schedules a job with span T_∞ on a machine with P processors. Let ρ denote* A-STEAL*'s responsiveness parameter, δ the utilization parameter, and L the quantum length. Then, for any $\epsilon > 0$, with probability at least $1-\epsilon$, the schedule produces at most $48T_\infty/((1-\delta)L) + \log_\rho P + 16\ln(1/\epsilon)$ efficient-and-satisfied quanta.*

PROOF. The lemma follows directly from Lemmas 3 and 5. □

The next lemma shows that for each inefficient quantum there exists a corresponding efficient-and-satisfied quantum.

LEMMA 7. *Suppose that* A-STEAL *schedules a job on a machine. Let I and C denote the set of inefficient quanta and the set of efficient-and-satisfied quanta produced by the schedule. If ρ is* A-STEAL*'s responsiveness parameter, then there exists an injective mapping $f : I \to C$ such that for all $q \in I$, we have $f(q) < q$ and $d_{f(q)} = d_q/\rho$.*

PROOF. For every inefficient quantum $q \in I$, define $r = f(q)$ to be the latest efficient-and-satisfied quantum such that $r < q$ and $d_r = d_q/\rho$. Such a quantum always exists, because the initial desire is 1 and the desire increases only after an efficient-and-satisfied quantum. We must prove that f does not map two inefficient quanta to the same efficient-and-satisfied quantum. Assume for the sake of contradiction that there exist two inefficient quanta, $q < q'$, such that $f(q) = f(q') = r$. By definition of f, the quantum r is efficient-and-satisfied, $r < q < q'$, and $d_q = d_{q'} = \rho d_r$. After the inefficient quantum q, A-STEAL reduced the desire to d_q/ρ. Since the desire later increased again to $d_{q'} = d_q$, and the desire increases only after efficient-and-satisfied quanta, there must be an efficient-and-satisfied quantum r' in the range $q < r' < q'$ such that $d(r') = d(q')/\rho$. But then, by the definition of f, we would have $f(q') = r'$. This is a contradiction. □

We can now bound the total number of mug-cycles executed by processors.

LEMMA 8. *Suppose that* A-STEAL *schedules a job with work T_1 on a machine with P processors. Let ρ denote* A-STEAL*'s responsiveness parameter, δ the utilization parameter, and L the quantum length. Then, the schedule produces at most $((1+\rho)/(L\delta - 1 - \rho))T_1$ mug-cycles.*

PROOF. When the allotment decreases, some processors are deallocated and their deques are declared muggable. The total number M of mug-cycles is at most the number of muggable deques during the job's execution. Since the allotment reduces by at most $a_q - 1$ from quantum q to quantum $q+1$, there are $M \le \sum_q (a_q - 1) < \sum_q a_q$ mug-cycles during the execution of the job.

By Lemma 7, for each inefficient quantum q, there is a distinct corresponding efficient-and-satisfied quantum $r = f(q)$ that satisfies $d_q = \rho d_r$. By definition, each efficient-and-satisfied quantum r has a nonsteal usage $n_r \ge L\delta a_r$ and allotment $a_r = d_r$. Thus, we have $n_r + n_q \ge L\delta a_r = ((L\delta)/(1+\rho))(a_r + \rho a_r) = ((L\delta)/(1+\rho))(a_r + \rho d_r) \ge ((L\delta)/(1+\rho))(a_r + a_q)$, since $a_q \le d_q$ and $d_q = \rho d_r$. Except for these inefficient quanta and their corresponding efficient-and-satisfied quanta, any other quantum q is efficient, and hence $n_q \ge L\delta a_q$ for these quanta. Let $N = \sum_q n_q$ be the total number of nonsteal-cycles during the job's execution. We have $N = \sum_q n_q \ge ((L\delta)/(1+\rho)) \sum_q a_q \ge ((L\delta)/(1+\rho))M$. Since the total number of nonsteal-cycles is the sum of work-cycles, and mug-cycles, and the total number of work-cycles is T_1, we have $N = T_1 + M$, and hence, $T_1 = N - M \ge ((L\delta)/(1+\rho))M - M = ((L\delta - 1 - \rho)/(1+\rho))M$, which yields $M \le ((1+\rho)(L\delta - 1 - \rho))T_1$. □

LEMMA 9. *Suppose that* A-STEAL *schedules a job with work T_1 on a machine with P processors. Let ρ denote* A-STEAL*'s responsiveness parameter, δ the utilization parameter, and L the quantum length. Then, the schedule produces at most $(T_1/(L\delta P_A))(1 + (1+\rho)/(L\delta - 1 - \rho))$ accounted quanta, where P_A is mean availability on accounted quanta.*

PROOF. Let A and D denote the set of accounted and deductible quanta, respectively. The mean availability on accounted quanta is $P_A = (1/|A|)\sum_{q \in A} p_q$. Let N be the total number of nonsteal-cycles. By definition of accounted quanta, the nonsteal usage satisfies $n_q \ge L\delta a_q$. Thus, we have $N = \sum_{q \in A \cup D} n_q \ge \sum_{q \in A} n_q \ge \sum_{q \in A} \delta L p_q = \delta L |A| P_A$, and hence, we obtain

$$|A| \le N/(L\delta P_A) \, . \tag{1}$$

The total number of nonsteal-cycles is the sum of the number of work-cycles and mug-cycles. Since there are at most T_1 work-cycles on accounted quanta and, by Lemma 8, there are at most $M \le ((1+\rho)(L\delta - 1 - \rho))T_1$ mug-cycles, we have $N \le T_1 + M < T_1(1 + (1+\rho)/(L\delta - 1 - \rho))$. Substituting this bound on N into Inequality (1) completes the proof. □

We are now ready to bound the running time of jobs scheduled with A-STEAL .

THEOREM 10. *Suppose that* A-STEAL *schedules a job with work T_1 and span T_∞ on a machine with P processors. Let ρ denote* A-STEAL*'s responsiveness parameter, δ the utilization parameter, and L the quantum length. For any $\epsilon > 0$, with probability at least $1 - \epsilon$,* A-STEAL *completes the job in*

$$T = \frac{T_1}{\delta \widetilde{P}}\left(1 + \frac{1+\rho}{L\delta - 1 - \rho}\right) + O\left(\frac{T_\infty}{1-\delta} + L\log_\rho P + L\ln(1/\epsilon)\right) \qquad (2)$$

time steps, where $\widetilde{P}$ is the $O(T_\infty/(1-\delta) + L\log_\rho P + L\ln(1/\epsilon))$-trimmed availability.

PROOF. The proof is a trim analysis. Let A be the set of accounted quanta, and let D be the set of deductible quanta. The overall number of time steps is thus at most $L(|A| + |D|)$. Lemmas 3 and 6 show that there are at most $|D| = O(T_\infty/((1-\delta)L) + \log_\rho P + \ln(1/\epsilon))$ deductible quanta with high probability, since efficient-and-satisfied quanta, and inefficient quanta, are deductible. Hence there are at most $L\,|D| = O(T_\infty/(1-\delta) + L\log_\rho P + L\ln(1/\epsilon))$ time steps in deductible quanta with high probability. We have that $P_A \ge \widetilde{P}$, since the mean availability on the accounted time steps (we trim the $L\,|D|$ deductible steps) must be at least the $O(T_\infty/(1-\delta) + L\log_\rho P + L\ln(1/\epsilon))$-trimmed availability (we trim the $O(T_\infty/(1-\delta) + L\log_\rho P + L\ln(1/\epsilon))$ steps that have the highest availability). From Lemma 9, the number of accounted quanta is bounded by $|A| = (T_1/(L\delta P_A))(1 + (1+\rho)/(L\delta - 1 - \rho))$. Combining the two parts, the desired time bound follows. □

COROLLARY 11. *Suppose that* A-STEAL *schedules a job with work T_1 and span T_∞ on a machine with P processors. Let ρ denote* A-STEAL*'s responsiveness parameter, δ the utilization parameter, and L the quantum length. Then,* A-STEAL *completes the job in expected time* $\mathrm{E}[T] = O(T_1/\widetilde{P} + T_\infty + L\lg P)$*, where $\widetilde{P}$ is the $O(T_\infty + L\lg P)$-trimmed availability.*

PROOF. Straightforward conversion of a high-probability bound to expectation, together with setting δ and ρ to suitable constants. □

The analysis leading to Theorem 10 and its corollary makes two assumptions. First, we assume that the scheduler knows exactly how many steal-cycles have occurred in the quantum. Second, we assume that the processors can find the muggable deques instantaneously. We now relax these assumptions and show that they do not adversely affect the asymptotic running time of A-STEAL.

A scheduling system can implement the counting of steal-cycles in several ways that impact our theoretical bounds only minimally. For example, if the number of processors in the machine P is smaller than the quantum length L, then the system can designate one processor to collect all the information from the other processors at the end of each quantum. Collecting this information increases the time bound by a multiplicative factor of only $1 + P/L$. As a practical matter, one would expect that $P \ll L$, since scheduling quanta tend to be measured in tens of milliseconds and processor cycle times in nanoseconds or less, and thus the slowdown would be negligible. Alternatively, one might organize the processors for the job into a tree structure so that it takes $O(\lg P)$ time to collect the total number of steal-cycles at the end of each quantum. The tree implementation introduces a multiplicative factor of $1 + (\lg P)/L$ to the job's execution time, an even less significant overhead.

The second assumption, that it takes constant time to find a muggable deque, can be relaxed in a similar manner. One option is to mug serially; that is, while there is a muggable deque, all processors try to mug according to a fixed linear order. This strategy could increase the number of mug-cycles by a factor of P in the worst case. If $P \ll L$, however, this change again does not affect the running time bound by much. Alternatively, to obtain a better theoretical bound, we could use a counting network [Aspnes et al. 1994] with width P to implement the list of muggable deques, in which case each mugging operation would consume $O(\lg^2 P)$ processor cycles. The number of accounted steps in the time bound from Lemma 9 would increase slightly to $(T_1/(\delta\widetilde{P}))/(1 + (1+\rho)\lg^2 P/(L\delta - 1 - \rho))$, but the number of deductible steps would not change.

3.2 Waste Analysis

The next theorem bounds the waste, which is the total number of mug- and steal-cycles.

THEOREM 12. *Suppose that* A-STEAL *schedules a job with work T_1 on a machine with P processors. Let ρ denote* A-STEAL*'s responsiveness parameter, δ the utilization parameter, and L the quantum length. Then,* A-STEAL *wastes at most*

$$W \le \left(\frac{1+\rho-\delta}{\delta} + \frac{(1+\rho)^2}{\delta(L\delta - 1 - \rho)}\right) T_1 \qquad (3)$$

processor cycles in the course of computation.

PROOF. Let M be the total number of mug-cycles, and let S be the total number of steal-cycles. Hence, we have $W = S + M$. Since Lemma 8 bounds M, we only need to bound S, which we do using an accounting argument based on whether a quantum is inefficient or efficient. Let S_{ineff} and S_{eff}, where $S = S_{\text{ineff}} + S_{\text{eff}}$, be the numbers of steal-cycles on innefficient and efficient quanta, respectively.

Inefficient quanta. Lemma 7 shows that every inefficient quantum q with desire d_q has a distinct corresponding efficient-and-satisfied quantum $r = f(q)$ with desire $d_r = d_q/\rho$. Thus the steal-cycles on quantum q can be amortized against the nonsteal-cycles on quantum r. Since quantum r is efficient-and-satisfied, its nonsteal usage satisfies $n_r \ge L\delta a_q/\rho$, and its allocation is $a_r = d_r$. Therefore, we have $n_r \ge L\delta a_r = L\delta d_r = L\delta d_q/\rho \ge L\delta a_q/\rho$. Let s_q be the number of steal-cycles in quantum q. Since there are La_q processor cycles in the quantum, we have $s_q \le La_q \le \rho n_r/\delta$, that is, the number of steal-cycles in the inefficient quantum q is at most a ρ/δ fraction of the nonsteal-cycles in its corresponding efficient-and-satisfied quantum r. Therefore, the total number of steal-cycles in all inefficient quanta satisfies $S_{\text{ineff}} \leq (\rho/\delta)(T_1+M)$.

Efficient quanta. On any efficient quantum q, the job has at least $L\delta a_q$ work- and mug-cycles and at most $L(1-\delta)a_q$ steal-cycles. Summing over all efficient quanta, the number of steal-cycles on efficient quanta is $S_{\text{eff}} \leq ((1-\delta)/\delta)(T_1+M)$.

The total waste is therefore $W = S + M = S_{\text{ineff}} + S_{\text{eff}} + M \leq (T_1+M)(1+\rho-\delta)/\delta + M$. Since Lemma 8 provides $M < T_1(1+\rho)/(L\delta - 1 - \rho)$, the theorem follows. $\square$

3.3 Interpretation of the Bounds

If the utilization parameter δ and responsiveness parameter ρ are constants, the bounds in Equation (2) and Inequality (3) can be simplified somewhat as follows:

$$\begin{aligned} T &= \frac{T_1}{\delta\widetilde{P}}(1+O(1/L)) + O\left(\frac{T_\infty}{1-\delta} + L\log_\rho P + L\ln(1/\epsilon)\right), \\ W &= \left(\frac{1+\rho-\delta}{\delta} + O(1/L)\right)T_1. \end{aligned} \tag{4}$$

This reformulation allows us to more easily see the tradeoffs due to the setting of the δ and ρ parameters.

In the time bound, as δ increases toward 1, the coefficient of $T_1/\widetilde{P}$ decreases toward 1, and the job comes closer to perfect linear speedup on accounted steps. The number of deductible steps increases at the same time, however. Moreover, as δ increases and ρ decreases, the completion time increases and the waste decreases. The utilization parameter δ may lie between 80% and 95%, and the responsiveness parameter ρ can be set between 1.2 and 2.0. The quantum length L is a system configuration parameter, which might have values in the range 10^3 to 10^5.

To see how these settings affect the waste bound, consider the waste bound as comprising two parts, where the waste due to steal-cycles is $S \leq ((1+\rho-\delta)/\delta)T_1$ and the waste due to mug-cycles is $M = O(1/L)T_1$. We can see that the waste due to mug-cycles is just a tiny fraction compared to the work T_1. Thus these bounds indicate that adaptive scheduling with parallelism feedback can be achieved without imposing much overhead when adding to or removing processors from jobs.

Most of the waste comes from steal-cycles, where S is generally less than $2T_1$ for typical parameter values. The analysis of Theorem 12 shows, however, that the number of steal-cycles in efficient steps is bounded by $((1-\delta)/\delta)T_1$, which is a small fraction of S. Thus, most of the waste comes from the steal-cycles in inefficient quanta. Our analysis assumes that the job scheduler is an adversary, creating as many inefficient quanta as possible. Of course, job schedulers are generally not adversarial. Thus, in practice, we expect the waste to be a much smaller fraction of T_1 than our bounds. exp describes experiments that confirm this intuition.

4. EXPERIMENTAL EVALUATION

To evaluate the performance of A-STEAL empirically, we built a discrete-time simulator using DESMO-J [DESMOJ 1999]. Some of our experiments benchmarked A-STEAL against ABP [Arora et al. 1998], an adaptive thread scheduler that does not supply parallelism feedback to the job scheduler. This section describes our simulation setup and the results of the experiments.

We conducted four sets of experiments on the simulator with synthetic jobs. Our results are summarized below:

—The *time experiments* investigated the performance of A-STEAL on over 2300 job runs. A linear-regression analysis of the results provides evidence that the coefficients on the number of accounted and deductible steps are considerably smaller than the upper bounds provided by our theoretical bounds. A second linear-regression analysis indicates that A-STEAL completes jobs on average for at most twice the optimal number of time steps, which is the same bound provided by offline greedy scheduling [Graham 1969; Brent 1974].

—The *waste experiments* are designed to measure the waste incurred by A-STEAL in practice, and compare the observed waste to the theoretical upper bounds. Our experiments indicate that the waste is almost insensitive to the parameter settings and is a tiny fraction (less than 10%) of the work for jobs with high parallelism.

—The *time-waste experiments* compare the completion time and waste of A-STEAL, with ABP [Arora et al. 1998] by running single jobs with predetermined availability profiles. These experiments indicate that on large machines, when the mean availability $\bar{P}$ is considerably smaller than the number P of processors in the machine, A-STEAL completes jobs faster than ABP, while wasting fewer processor cycles than ABP. On medium-sized machines, when $\bar{P}$ is of the same order as P, ABP completes jobs slightly faster than A-STEAL, but it still wastes many more processor cycles than A-STEAL.

—The *utilization experiments* compare the utilization of A-STEAL and ABP when many jobs with varying characteristics are using the same multiprocessor resource. The experiments provide evidence that on moderately to heavily loaded large machines, A-STEAL consistently provides a higher utilization than ABP for a variety of job mixes.

4.1 Simulation Setup

Our Java-based discrete-time simulator, which was implemented using DESMO-J [DESMOJ 1999], implements four major entities—processors, jobs, thread schedulers, and job schedulers. The simulator tracks their interactions in a two-level scheduling environment. We modeled jobs as dags, which are executed by the thread scheduler. When a job is submitted to the simulated multiprocessor system, an instance of a thread scheduler is created for the job. The job scheduler allots processors to the job, and the thread scheduler simulates the execution of the job using work-stealing. The simulator operates in discrete time steps: a processor can complete either a work-cycle, steal-cycle, or

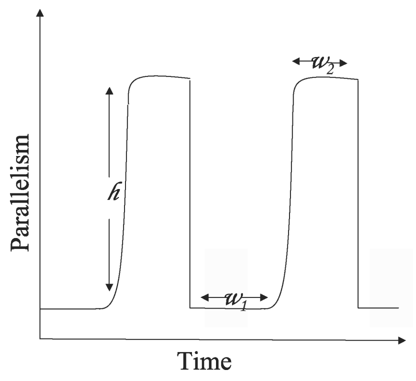


Fig. 2. The parallelism profile (for 2 iterations) of the jobs used in the simulation.

mug-cycle during each time step. We ignored the overheads due to the reallocation of processors in the simulation.

We tested synthetic multithreaded jobs with the parallelism profile shown in Figure 2. Each job alternates between a serial phase of length w_1 and a parallel phase (with h-way parallelism) of length w_2. The average parallelism of the job is approximately $(w_1 + hw_2)/(w_1 + w_2)$. By varying the values of w_1, w_2, h, and the number of iterations, we can generate jobs with different work, span, and frequency of the change of the parallelism.

In the time-waste experiments and the utilization experiments, we compared the performance of A-STEAL with that of another thread scheduler, ABP [Arora et al. 1998], an adaptive thread scheduler that does not provide parallelism feedback to the job scheduler. In these experiments, ABP is always allotted all the processors available to the job. ABP uses a nonblocking implementation of work-stealing and always maintains P deques. When the job scheduler allots $a_q = p_q$ processors in quantum q, ABP selects a_q deques uniformly at random from the P deques, and the allotted processors start working on them. Arora et al. [1998] prove that ABP completes a job in expected time

$$T = O(T_1/\bar{P} + PT_\infty/\bar{P}) , \tag{5}$$

where $\bar{P}$ is the average number of processors allotted to the job by the job scheduler. Although they provide no bounds on waste, one can prove that ABP may waste $\Omega(T_1 + PT_\infty)$ processor cycles in an adversarial setting.

We implemented three kinds of job schedulers: profile-based, equipartitioning [McCann et al. 1993], and dynamic equipartitioning [McCann et al. 1993]. A profile-based job scheduler was used in the first four sets of experiments, and both equipartitioning and dynamic equipartitioning job schedulers were used in the utilization experiment. An *equipartitioning* (EQ) job scheduler simply allots the same number of processors to all the active jobs in the system. Since ABP provides no parallelism feedback, EQ is a suitable job scheduler for ABP's scheduling model. *Dynamic equipartitioning* (DEQ) is a dynamic version of the equipartitioning policy, but it requires parallelism feedback. A DEQ job scheduler maintains an equal allotment of processors to all jobs with the constraint that no job is allotted more processors than it requests. DEQ is compatible with A-STEAL's scheduling model, since it can use the feedback provided by A-STEAL to decide the allotment.

For the first three experiments—time, waste, and time-waste—we ran a single job with a predetermined *availability profile*: the sequence of processor availabilities p_q for all the quanta while the job is executing. For the *profile-based* job scheduler, we precomputed the availability profile, and during the simulation, the job scheduler simply used the precomputed availability for each quantum. We generated three kinds of profiles:

—*Uniform profiles.* The processor availabilities in these profiles follow the uniform distribution in the range from 1 to the maximum number P of processors in the system. These profiles represent near-adversarial conditions for A-STEAL, because the availability for one quantum is unrelated to the availability for the previous quantum.

—*Smooth profiles.* In these profiles, the change of processor availabilities from one scheduling quantum to the next follows a standard normal distribution. Thus the processor availability is unlikely to change significantly over two consecutive quanta. These profiles attempt to model situations where new arrivals of jobs are rare, and the availability changes significantly only when a new job arrives.

—*Practical profiles.* These availability profiles were generated from the workload archives [Feitelson 2005] of various computer clusters. We computed the availability at every quantum by subtracting the number of processors that were being used at the start of the quantum from the number of processors in the machine. These profiles are meant to capture the processor availability in practical systems.

A-STEAL requires certain parameters as input. The responsiveness parameter is $\rho = 1.5$ for all the experiments. For all experiments except the waste experiments, the utilization parameter is $\delta = 0.8$. We varied δ in the waste experiments. The quantum length L represents the time between successive reallocations of processors by the job scheduler, and is selected to amortize the overheads due to communication between the job scheduler and the thread scheduler and to the reallocation of processors. In conventional computer systems, a scheduling quantum is typically between 10 and 20 milliseconds. Our experience with the Cilk runtime system [Supercomputing Technologies Group 2001] indicates that a steal/mug-cycle takes approximately 0.5 to 5 microseconds, suggesting that the quantum length L should be set to values between 10^3 and 10^5 time steps. Our theoretical bounds indicate that as long as $T_\infty \gg L \log P$, the length of L should have little effect on our results. Due to the performance limitations of our simulation environment, however, we were unable to run very long jobs: most have span in the order of only a few thousand time steps. Therefore, to satisfy the condition that $T_\infty \gg L \log P$, we set $L = 200$.

4.2 Time Experiments

The running-time bounds proved in analysis, though asymptotically strong, have weak constants. The time experiments were designed to investigate what constants would occur in practice and how A-STEAL performs compared to an optimal scheduler. We performed linear-regression analysis on the results of 2331 job runs using many availability profiles as decided earlier to answer these questions.

Our first time experiment uses the bounds in Equation (2) as a simple model, as in the study Blumofe et al. [1996]. Assuming that equality holds and disregarding smaller terms, the model estimates performance as

$$T \approx c_1 T_1/\widetilde{P} + c_\infty T_\infty, \tag{6}$$

where $c_1 > 0$ is the *work overhead* and $c_\infty > 0$ is the *span overhead*. When $\delta = 0.8$, $\rho = 1.5$, and $L = 200$, the coefficients for the asymptotic bounds in Equation (2) turn out to be $1.26 < c_1 < 1.27$ and $c_\infty = 480$, but a direct analysis of expectation can improve the bound on span overhead to $c_\infty = 60$. Since the span overhead c_∞ is large, the bound indicates that A-STEAL may not provide linear speedup except when $T_1/T_\infty \gg 60\widetilde{P}$. Moreover, on accounted time steps, A-STEAL might not provide perfect linear speedup, since the work overhead is $1.26 > 1$.

In practice, however, we should not expect these large overheads to materialize. First, our analysis is focused on asymptotic bounds and use bounding techniques such as Markov's inequality and Chernoff bounds, which are not necessarily tight. Second, our analysis assumes that the job completes only the minimum number of work-cycles in each quantum, specifically, 0 on a deductible quantum and $\delta L a_q$ on an accounted quantum with allotment a_q.

Our first linear-regression analysis fits the running time of the 2331 job runs to Equation (6). The trimmed mean $\widetilde{P}$ of a job run is computed as the average processor availability of all accounted steps during the execution of the job. The least-squares fit to the data to minimize relative error, yields $c_1 = 0.960 \pm 0.003$ and $c_\infty = 0.812 \pm 0.009$, with 95% confidence. The R^2 correlation coefficient of the fit is 99.4%. Since $c_\infty = 0.812 \pm 0.009$, on average the jobs achieved linear speedup when $T_1/T_\infty \gg \widetilde{P}$. In addition, since we have $c_1 = 0.960 \pm 0.003$, A-STEAL achieves almost perfect linear speedup on the accounted steps. The fact that $c_1 < 1$ stems from the fact that jobs performed some work during the deductible steps.

We performed a second set of regression tests on the same set of jobs to compare the performance of A-STEAL with an optimal scheduler. We fit the job data to the curve

$$T = \hat{c}_1 T_1/\bar{P} + \hat{c}_\infty T_\infty. \tag{7}$$

The analysis yields $\hat{c}_1 = 0.992 \pm 0.003$ and $c_\infty = 0.911 \pm 0.008$ with an R^2 correlation coefficient of 99.4%. Both $T_1/\bar{P}$ and T_∞ are lower bounds on the job's running time, and thus an optimal scheduler requires at least $\max\{T_1/\bar{P}, T_\infty\} \geq (T_1/\bar{P} + T_\infty)/2 \geq (\hat{c}_1 T_1/\bar{P} + \hat{c}_\infty T_\infty)/2$ time steps, since $\hat{c}_1 < 1$ and $\hat{c}_\infty < 1$. Consequently, on average A-STEAL completed the jobs within at most twice the time of an optimal scheduler.

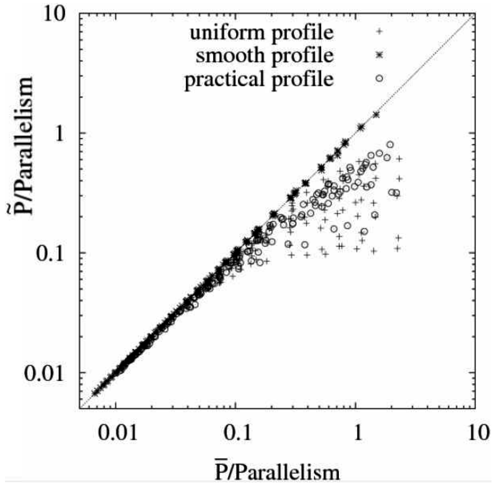


Fig. 3. Comparing the (true) mean availability $\bar{P}$ with the trimmed availability $\widetilde{P}$ using three availability profiles. Each data point represents a job execution for which the mean availability and trimmed availability were measured. These values were normalized by dividing by the parallelism T_1/T_∞ of the job. When the parallelism satisfies $T_1/T_\infty > 5\bar{P}$, the experiments indicate that for all profiles, the trimmed availability is a good approximation of the mean availability. All these experiments used $\delta = 0.8$ and $\rho = 1.5$.

Equations (6) and (7) both predict performance with high accuracy, and yet $\widetilde{P}$ and $\bar{P}$ can diverge significantly. To resolve this paradox, we compared $\widetilde{P}$ and $\bar{P}$ on the job runs. Figure 3 shows a graph of the results, where $\widetilde{P}$ and $\bar{P}$ are each normalized by dividing by the parallelism T_1/T_∞ of the job. The diagonal line in the figure is the curve $\widetilde{P} = \bar{P}$.

If a job has parallelism $T_1/T_\infty > 5\bar{P}$ (data points on the left), the experiment indicates that for all three kinds of availability profiles, we have $\widetilde{P} \approx \bar{P}$. In this case, we have $T_1/\widetilde{P} \approx T_1/\bar{P}$ and $T_1/\bar{P} \gg T_\infty$, which implies that the first terms in Equations (6) and (7) are nearly identical and dominate the running time. On the other hand, if a job has small parallelism (data points on the right), the values of $\widetilde{P}$ and $\bar{P}$ diverge and the divergence depends on the availability profile used. In this region, however, the running time is dominated by the span T_∞, and thus, the divergence of $\widetilde{P}$ and $\bar{P}$ has little influence on the running time.

4.3 Waste Experiments

Our theoretical analysis shows that the waste exhibited by A-Steal is at most $O(T_1)$. The constant hidden in the O-notation depends on the parameter settings. In our first waste experiment, we varied the value of the utilization parameter δ to determine the relationship between the waste and the setting of δ. For our second experiment, we investigated whether the waste incurred by a job depends on the job's parallelism.

The proof of Theorem 12 shows that the number of processor cycles wasted by a job is $((1-\delta)/\delta)T_1$ on efficient quanta and approximately $(\rho/\delta)T_1$ on inefficient quanta. Substituting $\delta = 0.8$ and $\rho = 1.5$, A-Stealcould waste as many as $0.25T_1$

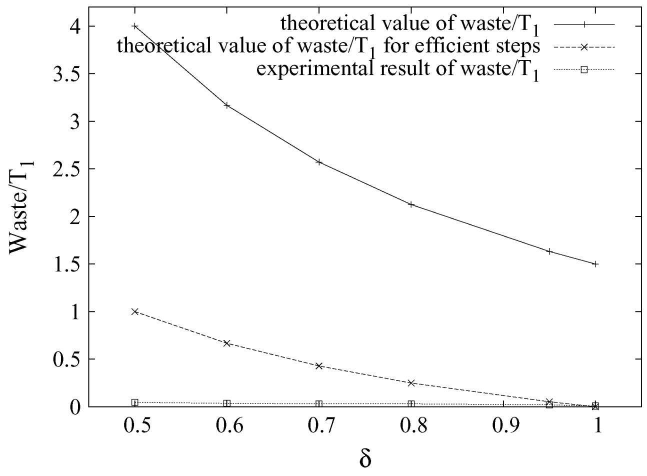


Fig. 4. Comparing the theoretical and practical waste (normalized by T_1) using A-STEAL for various values of the utilization parameter δ. The top line shows the total theoretical waste, the next line shows the theoretical waste on efficient quanta, and the bottom line shows the observed waste. The observed waste appears to be almost insensitive to the value of δ and is much smaller than the theoretical waste.

processor cycles on efficient quanta, and as many as $1.875T_1$ processor cycles on inefficient quanta. Since this analysis assumes that the job scheduler is an adversary and that the job completes the minimum number of work-cycles in each quantum, we did not expect these constants to materialize in practice.

We measured the waste for 300 jobs, most of which had parallelism $T_1/T_\infty > 5\bar{P}$, for $\delta = 0.5, 0.6, \ldots, 1.0$. The job runs used many availability profiles drawn equally from the three kinds. Figure 4 shows the average of waste normalized by the work T_1 of the job. For comparison we plotted the normalized theoretical bound Inequality (4) for the total waste and the normalized bound $((1-\delta)/\delta)T_1$ for the waste on efficient quanta. As the figure shows (although the curve is barely distinguishable from the x-axis), the observed waste is less than 10% of the work T_1 for most values of δ, and is considerably less than what the theoretical bounds predicted. Moreover, the waste seems to be quite insensitive to the particular value of δ.

We also ran an experiment to determine whether parallelism has an effect on waste. The bound in Inequality (4) does not depend on the parallelism T_1/T_∞ of the job, but only on the work T_1. For the 2331 job runs used in the time experiments, we measured the waste versus parallelism. Since waste is insensitive to δ, all jobs used the value $\delta = 0.8$. Figure 5 graphs the results. As can be seen in the figure, the higher the parallelism, the lower the waste-to-work ratio. The reason is that when the parallelism is high, the job can usually use most of the available processors without readjusting its desire. When the parallelism is low, however, the job's desire must track its parallelism closely to avoid waste. This situation is where A-STEAL is most effective, as the job pushes the theoretical waste bounds to their limit.

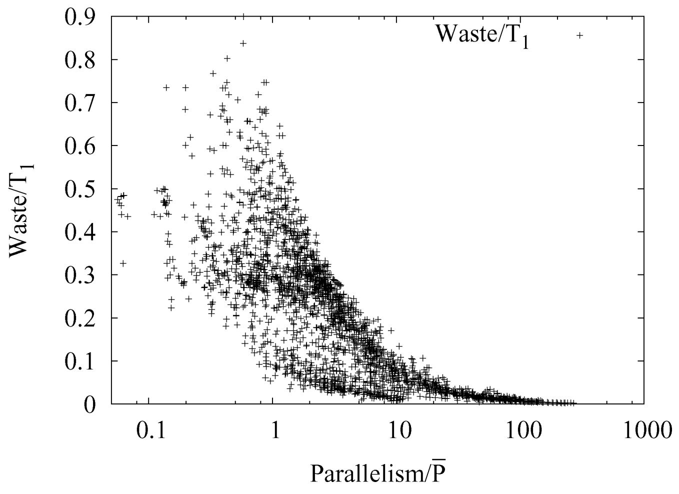


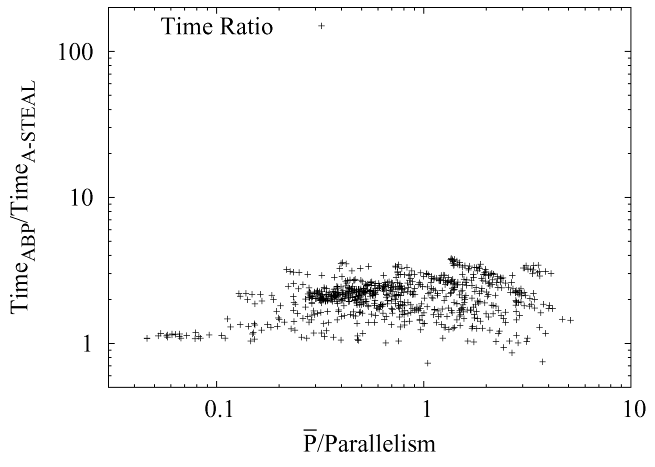
Fig. 5. How waste varies with parallelism. When $T_1/T_\infty > 10\bar{P}$, that is, the job's parallelism significantly exceeds the average availability, the observed waste is only a tiny fraction of the work T_1. For jobs with small parallelism, the waste showed a large variance but never exceeded the work T_1 in any of our runs. The utilization parameter was $\delta = 0.8$ for all job runs.

4.4 Time-Waste Experiments

The time-waste experiments were designed to compare A-STEAL with ABP, an adaptive thread scheduler with no parallelism feedback. For our first experiment, we ran A-STEAL and ABP to execute 756 job-runs on a simulated machine with $P = 512$ processors. Each head-to-head run used one of two practical availability profiles, one with $\bar{P} = 30$ and one with $\bar{P} = 60$. We measured the time and waste of A-STEAL and ABP for each run. Our second experiment was similar, but it used only $P = 128$ processors in the simulated machine over 330 job runs. Whenever the availability exceeded 128, which was not often, we chopped the availability to 128.

Figure 6 shows the ratio of ABP to A-STEAL with respect to both time and waste as a function of the mean availability $\bar{P}$, normalized by dividing by the parallelism T_1/T_∞. This experiment shows that A-STEAL completed jobs about twice as fast as ABP, while wasting only about 10% of the processor cycles wasted by ABP. Not surprisingly, A-STEAL wastes fewer processor cycles than ABP, since A-STEAL uses parallelism feedback to limit possible excessive allotment. Paradoxically, however, A-STEAL completes jobs faster than ABP, even though A-STEAL's allotment in every quantum is at most that of ABP, which is always allotted all the available processors.

ABP's slow completion is due to how ABP manages its ready deques. In particular, ABP has no mechanism for increasing and decreasing the number r of ready deques, and it maintains $r = P$ deques throughout the execution. Randomized work-stealing algorithms require $\Theta(r)$ steal-cycles to reduce the length of the span by 1 in expectation. Consequently, if r is large, each steal-cycle becomes less effective, and the job's progress along its span slows. Thus if the job has small or moderate parallelism (data points on the right), the span dominates the running time. If the job has large parallelism (data points on the



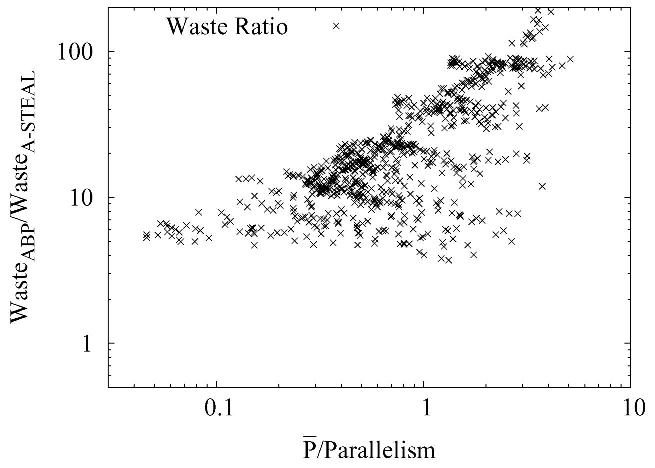


Fig. 6. Comparing the time and waste of A-Steal against ABP when $P = 512$ and $\bar{P} = 30, 60$. In this experiment, where P exceeds $\bar{P}$ by a significant margin, A-Steal completes jobs about twice as fast as ABP while wasting less than 10% of the processor cycles wasted by ABP.

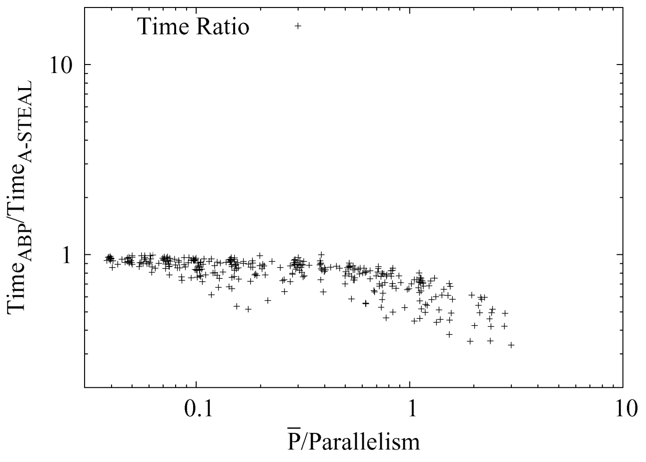
left), however, the impact is less. In contrast, A-Steal continues to make good progress along the span, regardless of parallelism, by reducing the number of deques according to its allotment.

This paradox can also be understood by using the model from Equation (6) for A-Steal, and an analogous model based on Equation (5) for ABP. Let us consider three cases:

—$T_1/T_\infty < \bar{P} \ll P$ (data points on the right): Whereas A-Steal completes the job in $\Theta(T_\infty)$ time, ABP requires $\Theta(PT_\infty/\bar{P})$ time.

—$\bar{P} < T_1/T_\infty \ll P$ (data points in the middle): A-Steal provides linear speedup since $T_1/T_\infty > \widetilde{P}$, but ABP does not, since $T_1/T_\infty \ll P$.

—$P < T_1/T_\infty$ (data points on the left): Both provide linear speedup in this range.



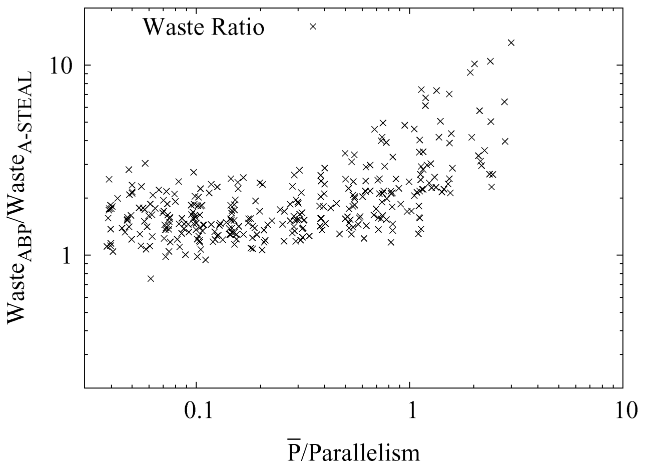


Fig. 7. Comparing the time and waste of A-Steal against ABP when $P = 128$ and $\bar{P} = 30, 60$. In this experiment, where P and $\bar{P}$ are closer in magnitude, A-Steal runs slightly slower than ABP, but it still tends to waste fewer processor cycles than ABP.

Since ABP performed relatively poorly when P was large compared to $\bar{P}$, our second experiment investigated the results when P is closer to $\bar{P}$. Figure 7 shows the results on 330 job-runs on a simulated machine, with $P = 128$. In this case, when the jobs' parallelism is large compared to $\bar{P}$, both ABP and A-Steal perform about the same with respect to both time and waste. As the parallelism gets closer to $\bar{P}$, ABP performs slightly better than A-Steal with respect to time, and slightly worse with respect to waste. Since $\bar{P} \approx P$, the two models coincide, and ABP and A-Steal perform comparably. Therefore, on small machines, where the disparity between $\bar{P}$ and P cannot be very great, the advantage of parallelism feedback is diminished, and ABP may yet be an effective thread-scheduling algorithm.

4.5 Utilization Experiments

The utilization experiments compared A-Stealwith ABP on a large server where many jobs are running simultaneously and jobs arrive and leave dynamically. We implemented job schedulers to allocate processors among various jobs: dynamic equipartitioning [McCann et al. 1993] for A-Steal, and equipartitioning [Tucker and Gupta 1989] for ABP. We simulated a 1000-processor machine for about 10^6 time steps, where jobs had a mean interarrival time of 1000 time steps. We compared the utilization provided by A-Steal and ABP over time.

It was unclear to us what distribution the parallelism and the span should follow. Although many workload models for parallel jobs have been studied [Sevcik 1994; Feitelson 1996; Downey 1998; Cirne and Berman 2001; Lublin and Feitelson 2003], none appears to apply directly to multithreaded jobs. Some studies [Leland and Ott 1986; Harchol-Balter and Downey 1997; Harchol-Balter 1999] claim that the sizes of Unix jobs follow a heavy-tailed distribution. Lacking a well-recognized guideline, we decided to try various distributions, and as it turned out, our results were fairly insensitive to which we chose.

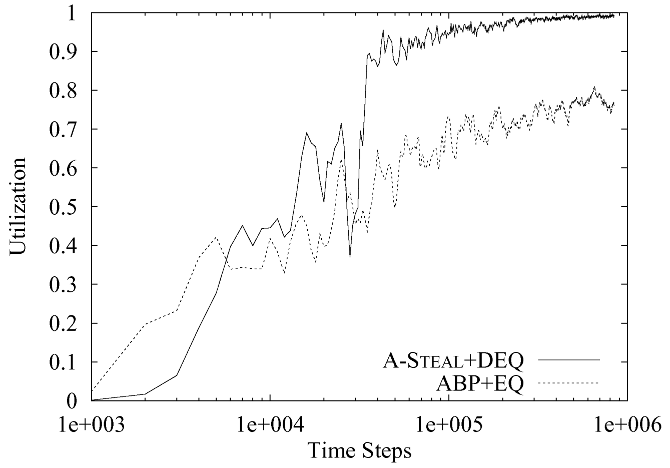
We considered nine sets of jobs using three distributions on each of the parallelism and the span. The means of the distributions were chosen so that jobs arrive faster than they complete and the load on the machine progressively increases. Thus we were able to measure the utilization of the machine under various loads. The three distributions we explored were the following:

—*Uniform distribution (U)*. The span is picked uniformly from the range 1,000 to 99,000. The parallelism is generated uniformly in the range [1, 80].

—*Heavy-tailed distribution 1 (HT1)*. We used a Zipf's-like [Zipf 1949] heavy-tailed distribution, where the probability of generating x is proportional to $1/x$. In our experiments, the distribution for parallelism has mean value 36, and the distribution for span has mean value 50,000.

—*Heavy-tailed distribution 2 (HT2)*. In this distribution, the probability of generating x is proportional to $1/\sqrt{x}$. In our experiments, the distribution for parallelism has mean value 36, and the distribution for span has mean value 50,000.

Of the nine possible sets of jobs, we ran six experiments using parallelism and span drawn from U/U, U/HT1, HT1/U, HT1/HT1, HT2/U, and HT2/HT2. For all these experiments, the comparison between A-Steal+DEQ and ABP+EQ followed the same qualitative trends. We broke time into intervals of 2000 time steps and measured the utilization—the fraction of processor cycles spent working—for each interval. Figure 8 shows the utilization as a function of time (log-scale) for the U/U experiment at the top, and for HT1/HT1 on the bottom. As can be seen in both figures, ABP+EQ starts out with a higher utilization, since A-Steal+DEQ initially requests just one processor. Before 10% of the simulation has elapsed, however, A-Steal+DEQ overtakes ABP+EQ with respect to the utilization, and then consistently provides a higher utilization. Although the figure does not show it, the mean completion time of jobs



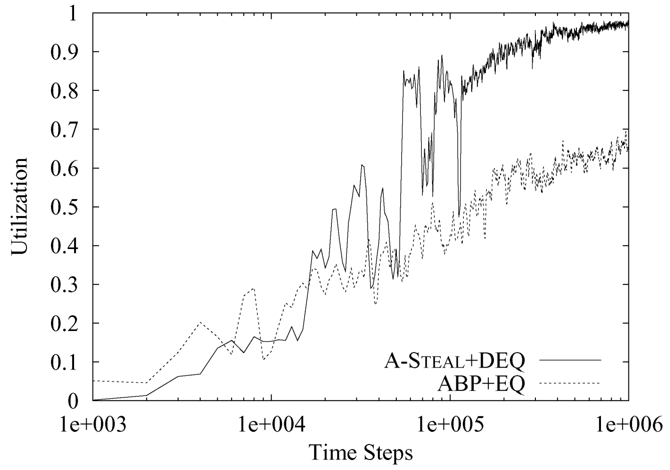


Fig. 8. Comparing the utilization over time of A-STEAL+DEQ and ABP+EQ. In the top figure, both the span and the parallelism follow the uniform distribution, and in the bottom figure, they follow the HT1 distribution.

under ABP+EQ is nearly 50% less than those under A-STEAL+DEQ, for both these distributions.

5. RELATED WORK

This section discusses related work on adaptive and nonadaptive schedulers for multithreaded jobs. Work in the area has centered on either job schedulers or on nonadaptive thread schedulers. We start by discussing nonadaptive work-stealing schedulers. We then discuss empirical and theoretical work on adaptive thread schedulers. Finally, we give a brief summary of research on adaptive job schedulers.

Work-stealing has been used as a heuristic since Burton and Sleep's research [Burton and Sleep 1981] and Halstead's implementation of Multilisp [Halstead

1984]. Many variants have been implemented since then [Mohr et al. 1990; Halbherr et al. 1994; Finkel and Manber 1987], and it has been analyzed in the context of load balancing [Rudolph et al. 1991], backtrack search [Karp and Zhang 1988], and so forth. Blumofe and Leiserson [1999] proved that the work-stealing algorithm is efficient with respect to time, space, and communication for the class of *fully strict* multithreaded computations. Arora et al. [1998] extended the time bound result to arbitrary multithreaded computations. Various researchers [Hendler et al. 2006; Hendler and Shavit 2002; Chase and Lev 2005] have since simplified and improved the memory allocation for deques for ABP. In addition, Acar et al. [2000] showed that work-stealing schedulers are efficient with respect to cache misses for jobs with *nested parallelism.* Variants of work-stealing algorithms have been implemented in many systems [Blumofe et al. 1995; Frigo et al. 1998; Blumofe and Papadopoulos 1999], and empirical studies show that work-stealing schedulers are scalable and practical [Frigo et al. 1998; Blumofe and Papadopoulos 1998].

Adaptive thread scheduling without parallelism feedback has been studied in the context of multithreading, primarily by Blumofe and his coauthors [Blumofe and Lisiecki 1997; Blumofe and Park 1994; Arora et al. 1998; Blumofe and Papadopoulos 1998]. In this work, the thread scheduler uses randomized work-stealing strategy to schedule threads on available processors, but does not provide feedback about the job's parallelism to the job scheduler. The work in Blumofe and Lisiecki [1997], and Blumofe and Park [1994] addresses networks of workstations where processors may fail, or join and leave a computation while the job is running, showing that work-stealing provides a good foundation for adaptive thread scheduling. In theoretical work, Arora et al. [1998] showed that the ABP thread scheduler provably completes a job in an expected time of $O(T_1/\bar{P} + PT_\infty/\bar{P})$. Blumofe and Papadopoulos [1998] performed an empirical evaluation of ABP showing that on an eight-processor machine, ABP provides almost perfect linear speedup for jobs with reasonable parallelism. In all these experiments, the job parallelism T_1/T_∞ is much greater than eight.

Adaptive thread scheduling with parallelism feedback has been studied empirically in Timothy B. Brecht [1996], Song [1998], and Sen [2004]. These researchers use a job's history of processor utilization to provide feedback to dynamic equipartitioning job schedulers. Their studies use different strategies for parallelism feedback, and all report better system performance with parallelism feedback than without, but it is not apparent which of their strategies is best.

In contrast to adaptive thread schedulers, adaptive job schedulers have been studied extensively, both empirically [McCann et al. 1993; Yue and Lilja 2001; Chiang and Vernon 1996; Parsons and Sevcik 1995; Eager et al. 1989; Ghosal et al. 1991; Nguyen et al. 1996a, 1996b; Sevcik 1989 and theoretically Gu 1995; Deng and Dymond 1996; Motwani et al. 1993; Edmonds 1999; Edmonds et al. 2003; Bansal et al. 2004]. McCann et al. [1993] studied many different job schedulers and evaluated them on a set of benchmarks. They also introduced the notion of dynamic equipartitioning, which gives each job a fair allotment of processors, while allowing processors that cannot be used by a job to be reallocated to other jobs. Gu [1995] proved that dynamic equipartitioning with instantaneous parallelism feedback is four-competitive with respect to makespan for batched jobs with multiple phases, assuming that the parallelism of the job remains constant during the phase and that the phases are relatively long compared with the length of a scheduling quantum. Deng and Dymond [1996] proved a similar result for mean response time for multiphase jobs, regardless of their arrival times. Song [1998] proved that a randomized distributed strategy can implement dynamic equipartitioning.

6. CONCLUSIONS

This research has used the technique of trim analysis to limit a powerful adversary, enabling us to analyze adaptive schedulers with parallelism feedback. The idea of ignoring a few outliers while calculating averages is often used in statistics to ignore anomalous data points. For example, teachers often ignore the lowest score while computing a student's grade, and in the Olympic Games, the lowest and the highest scores are sometimes ignored when computing an athlete's average. In theoretical computer science, when an adversary is too powerful, we sometimes make statistical assumptions about the input to render the analysis tractable, but statistical assumptions may not be valid in practice. Trim analysis may prove itself of value for analyzing such problems.

A-STEAL, as presented, uses full information about the previous quantum to estimate the desire of the current quantum. Collecting perfect information might become difficult as the number of processors becomes larger, especially if the number of processors exceeds the quantum length. A-STEAL only estimates the desire, however, and therefore approximate information should be enough to provide feedback. We are currently studying the possibility of using sampling techniques to estimate the number of steal-cycles, instead of counting the exact number.

Our empirical studies provide evidence that A-STEAL performs better than ABP when the machine has a large number of processors and has many jobs running on it. The reason is that A-STEAL uses parallelism feedback and the mugging mechanism to reclaim abandoned deques. One can imagine implementing ABP, which does not use parallelism feedback, but which does use a mugging mechanism. Although adding a mugging mechanism to ABP may not improve its performance theoretically, such a modification to ABP might improve its performance as a matter of practice. We are currently studying ABP with this modification in order to evaluate the importance of parallelism feedback itself in adaptive work-stealing.

ACKNOWLEDGMENTS

We thank the members of the Supercomputing Technologies group at MIT CSAIL for helpful discussion and comments. Siddhartha Sen, also of MIT CSAIL, contributed to the implementation of an adaptive scheduler for Cilk, which inspired this research.

REFERENCES

ACAR, U. A., BLELLOCH, G. E., AND BLUMOFE, R. D. 2000. The data locality of work stealing. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA).* 1–12.

AGRAWAL, K., HE, Y., HSU, W. J., AND LEISERSON, C. E. 2006a. Adaptive task scheduling with parallelism feedback. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

AGRAWAL, K., HE, Y., AND LEISERSON, C. E. 2006b. An empirical evaluation of work stealing with parallelism feedback. In *Proceedings of the 2006 IEEE International Conference on Distributed Computing Systems (ICDCS'06)*.

AGRAWAL, K., HE, Y., AND LEISERSON, C. E. 2007. Adaptive work stealing with parallelism feedback. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (IPDPS '07)*.

ARORA, N. S., BLUMOFE, R. D., AND PLAXTON, C. G. 1998. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 119–129.

ASPNES, J., HERLIHY, M., AND SHAVIT, N. 1994. Counting networks. *J. ACM 41*, 5, 1020–1048.

BANSAL, N., DHAMDHERE, K., KONEMANN, J., AND SINHA, A. 2004. Non-clairvoyant scheduling for minimizing mean slowdown. *Algorithmica 40*, 4, 305–318.

BLELLOCH, G., GIBBONS, P., AND MATIAS, Y. 1999. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM 46*, 2, 281–321.

BLELLOCH, G. E., GIBBONS, P. B., AND MATIAS, Y. 1995. Provably efficient scheduling for languages with fine-grained parallelism. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 1–12.

BLELLOCH, G. E. AND GREINER, J. 1996. A provable time and space efficient implementation of NESL. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*. 213–225.

BLUMOFE, R. D. 1995. Executing multithreaded programs efficiently. Ph.D. Thesis. Massachusetts Institute of Technology.

BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. 1995. Cilk: an efficient multithreaded runtime system. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 207–216.

BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. 1996. Cilk: an efficient multithreaded runtime system. *J. Parallel Distrib. Comput. 37*, 1, 55–69.

BLUMOFE, R. D. AND LEISERSON, C. E. 1998. Space-efficient scheduling of multithreaded computations. *SIAM J. Comput. 27*, 1 (Feb.), 202–229.

BLUMOFE, R. D. AND LEISERSON, C. E. 1999. Scheduling multithreaded computations by work stealing. *J. ACM 46*, 5, 720–748.

BLUMOFE, R. D., LEISERSON, C. E., AND SONG, B. 1998. Automatic processor allocation for work-stealing jobs. unpublished.

BLUMOFE, R. D. AND LISIECKI, P. A. 1997. Adaptive and reliable parallel computing on networks of workstations. In *Proceedings of the USENIX 1997 Annual Technical Conference (USENJX'97)*. pp. 133–147.

BLUMOFE, R. D. AND PAPADOPOULOS, D. 1998. The performance of work stealing in multiprogrammed environments. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pp. 266–267.

BLUMOFE, R. D. AND PAPADOPOULOS, D. 1999. Hood: a user-level threads library for multiprogrammed multiprocessors. Tech. Rep., University of Texas at Austin.

BLUMOFE, R. D. AND PARK, D. S. 1994. Scheduling large-scale parallel computations on networks of workstations. In *Proceedings of the IEEE International Symposium on High Performance Distributed Computing (HPDC'94)*, pp. 96–105.

BRENT, R. P. 1974. The parallel evaluation of general arithmetic expressions. *J. ACM 21*, 2, 201–206.

BURTON, F. W. AND SLEEP, M. R. 1981. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture (FPCA'81)*. 187–194.

CHASE, D. AND LEV, Y. 2005. Dynamic circular work-stealing deque. In *Proceedings of the ACM symposium on Parallelism in Algorithms and Architectures*. 21–28.

Chiang, S.-H. and Vernon, M. K. 1996. Dynamic vs. static quantum-based parallel processor allocation. In *Proceedings of the International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*. 200–223.

Cirne, W. and Berman, F. 2001. A model for moldable supercomputer jobs. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (IPDPS'01)*. IEEE Computer Society, Washington, DC, USA, pp. 50–59.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. 2001. *Introduction to Algorithms* (Second ed). The MIT Press and McGraw-Hill.

Deng, X. and Dymond, P. 1996. On multiprocessor system scheduling. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pp. 82–88.

Deng, X., Gu, N., Brecht, T., and Lu, K. 1996. Preemptive scheduling of parallel jobs on multiprocessors. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '96)*, pp. 159–167. Society for Industrial and Applied Mathematics.

DESMOJ. 1999. DESMO-J: a framework for discrete-event modelling and simulation. http://asi-www.informatik.uni-hamburg.de/desmoj/.

Downey, A. B. 1998. A parallel workload model and its implications for processor allocation. *Cluster Comput. 1*, 1, 133–145.

Eager, D. L., Zahorjan, J., and Lozowska, E. D. 1989. Speedup versus efficiency in parallel systems. *IEEE Trans. Comput. 38*, 3, 408–423.

Edmonds, J. 1999. Scheduling in the dark. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing (STOC'99)*, pp. 179–188.

Edmonds, J., Chinn, D. D., Brecht, T., and Deng, X. 2003. Non-clairvoyant multiprocessor scheduling of jobs with changing execution characteristics. *J. Sched. 6*, 3, 231–250.

Fang, Z., Tang, P., Yew, P.-C., and Zhu, C.-Q. 1990. Dynamic processor self-scheduling for general parallel nested loops. *IEEE Trans. Comput. 39*, 7, 919–929.

Feitelson, D. 2005. Parallel workloads archive. http://www.cs.huji.ac.il/labs/parallel/workload/.

Feitelson, D. G. 1996. Packing schemes for gang scheduling. In *Proceedings of the International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, D. G. Feitelson and L. Rudolph, Eds. Vol. 1162, pp. 89–110. Springer.

Feitelson, D. G. 1997. Job scheduling in multiprogrammed parallel systems (extended version). Tech. Rep., IBM Research Report RC 19790 (87657) 2nd Revision.

Finkel, R. and Manber, U. 1987. DIB—A distributed implementation of backtracking. *Trans. Progr. Lang. 9*, 2 (Apr.), 235–256.

Frigo, M., Leiserson, C. E., and Randall, K. H. 1998. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, pp. 212–223.

Ghosal, D., Serazzi, G., and Tripathi, S. K. 1991. The processor working set and its use in scheduling multiprocessor systems. *IEEE Trans. Softw. Eng. 17*, 5, 443–453.

Graham, R. L. 1969. Bounds on multiprocessing anomalies. *SIAM Journ. Appl. Math. 17*, 2, 416–429.

Gu, N. 1995. Competitive analysis of dynamic processor allocation strategies. Masters Thesis. York University.

Halbherr, M., Zhou, Y., and Joerg, C. F. 1994. MIMD-style parallel programming with continuation-passing threads. In *Proceedings of the International Workshop on Massive Parallelism: Hardware, Software, and Applications*.

Halstead Jr., R. H. 1984. Implementation of Multilisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming (LFP'84)*. Austin, Texas, pp. 9–17.

Harchol-Balter, M. 1999. The effect of heavy-tailed job size. distributions on computer system design. In *Proceedings of the Conference on Applications of Heavy Tailed Distributions in Economics*.

Harchol-Balter, M. and Downey, A. B. 1997. Exploiting process lifetime distributions for dynamic load balancing. *ACM Trans. Comput. Syst. 15*, 3, 253–285.

Hendler, D., Lev, Y., Moir, M., and Shavit, N. 2006. A dynamic-sized nonblocking work stealing deque. *Distrib. Comput. 18*, 3, 189–207.

HENDLER, D. AND SHAVIT, N. 2002. Non-blocking steal-half work queues. In *Proceedings of the Annual Symposium on Principles of Distributed Computing*, 280–289.

HUMMEL, S. F. AND SCHONBERG, E. 1991. Low-overhead scheduling of nested parallelism. *IBM J. Res. Develop. 35*, 5-6, 743–765.

KARP, R. M. AND ZHANG, Y. 1988. A randomized parallel branch-and-bound procedure. In *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC'88)*. 290–300.

LELAND, W. AND OTT, T. J. 1986. Load-balancing heuristics and process behavior. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. 54–69.

LEUTENEGGER, S. T. AND VERNON, M. K. 1990. The performance of multiprogrammed multiprocessor scheduling policies. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. 226–236.

LUBLIN, U. AND FEITELSON, D. G. 2003. The workload on parallel supercomputers: modeling the characteristics of rigid jobs. *J. Parallel Distrib. Comput. 63*, 11, 1105–1122.

MARTORELL, X., CORBALÁN, J., NIKOLOPOULOS, D. S., NAVARRO, N., POLYCHRONOPOULOS, E. D., PAPATHEODOROU, T. S., AND LABARTA, J. 2000. A tool to schedule parallel applications on multiprocessors: the NANOS CPU manager. In *Proceedings of the International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, 87–112.

MCCANN, C., VASWANI, R., AND ZAHORJAN, J. 1993. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Trans. Comput. Syst. 11*, 2, 146–178.

MOHR, E., KRANZ, D. A., AND HALSTEAD, JR., R. H. 1990. Lazy task creation: a technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Symposium on LISP and Functional Programming (LFP'90)*, pp. 185–197.

MOTWANI, R., PHILLIPS, S., AND TORNG, E. 1993. Non-clairvoyant scheduling. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'93)*, pp. 422–431.

MOTWANI, R. AND RAGHAVAN, P. 1995. *Randomized Algorithms* (1st Ed). Cambridge University Press.

NARLIKAR, G. J. AND BLELLOCH, G. E. 1999. Space-efficient scheduling of nested parallelism. *ACM Trans. Prog. Lang. Syst. 21*, 1, 138–173.

NGUYEN, T. D., VASWANI, R., AND ZAHORJAN, J. 1996a. Maximizing speedup through self-tuning of processor allocation. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS'96)*, pp. 463–468.

NGUYEN, T. D., VASWANI, R., AND ZAHORJAN, J. 1996b. Using runtime measured workload characteristics in parallel processor scheduling. In *Proceedings of the International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pp. 155–174.

PARSONS, E. W. AND SEVCIK, K. C. 1995. Multiprocessor scheduling for high-variability service time distributions. In *Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)*. 127–145.

ROSTI, E., SMIRNI, E., DOWDY, L. W., SERAZZI, G., AND CARLSON, B. M. 1994. Robust partitioning schemes of multiprocessor systems. *Perform. Eval. 19*, 2-3, 141–165.

ROSTI, E., SMIRNI, E., SERAZZI, G., AND DOWDY, L. W. 1995. Analysis of non-work-conserving processor partitioning policies. In *Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)*, pp. 165–181.

RUDOLPH, L., SLIVKIN-ALLALOUF, M., AND UPFAL, E. 1991. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 237–245.

SEN, S. 2004. Dynamic processor allocation for adaptively parallel jobs. Masters Thesis. Massachusetts Institute of Technology.

SEVCIK, K. C. 1989. Characterizations of parallelism in applications and their use in scheduling. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pp. 171–180.

SEVCIK, K. C. 1994. Application scheduling and processor allocation in multiprogrammed parallel processing systems. *Perform. Eval. 19*, 2-3, 107–140.

SONG, B. 1998. Scheduling adaptively parallel jobs. Masters Thesis. Massachusetts Institute of Technology.

Squillante, M. S. 1995. On the benefits and limitations of dynamic partitioning in parallel computer systems. In *Proceedings of the 9th International Parallel Processing Symposium (IPPS'95)*, pp. 219–238.

Supercomputing Technologies Group. 2001. *Cilk 5.3.2 Reference Manual*. MIT Laboratory for Computer Science.

Timothy B. Brecht, K. G. 1996. Using parallel program characteristics in dynamic processor allocation policies. *Perform. Eval. 27-28*, 519–539.

Tucker, A. and Gupta, A. 1989. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP'89)*. 159–166.

Yue, K. K. and Lilja, D. J. 2001. Implementing a dynamic processor allocation policy for multiprogrammed parallel applications in the Solaris$^{\text{TM}}$ operating system. *Concurrency Computat. Pract. Exper. 13*, 6, 449–464.

Zipf, G. K. 1949. *Human Behavior and the Principle of Least Effort*. Addison-Wesley.

Received February 2007; revised July 2008; accepted October 2007