Proactive Work Stealing for Futures

Kyle Singer
Washington University in St. Louis
kdsinger@wustl.edu

Yifan Xu
Washington University in St. Louis
xuyifan@wustl.edu

I-Ting Angelina Lee
Washington University in St. Louis
angelee@wustl.edu

Abstract

The use of futures provides a flexible way to express parallelism and can generate arbitrary dependences among parallel subcomputations. The additional flexibility that futures provide comes with a cost, however. When scheduled using classic work stealing, a program with futures, compared to a program that uses only fork-join parallelism, can incur a much higher number of "deviations," a metric for evaluating the performance of parallel executions. All prior works assume a ***parsimonious*** work-stealing scheduler, however, where a worker thread (surrogate of a processor) steals work only when its local deque becomes empty.

In this work, we investigate an alternative scheduling approach, called ProWS, where the workers perform ***proactive*** work stealing when handling future operations. We show that ProWS, for programs that use futures, can provide provably efficient execution time and equal or better bounds on the number of deviations compared to classic parsimonious work stealing. Given a computation with T_1 work and T_∞ span, ProWS executes the computation on P processors in expected time $O(T_1/P + T_\infty \lg P)$, with an additional $\lg P$ overhead on the span term compared to the parsimonious variant. For structured use of futures, where each future is single touch with no race on the future handle, the algorithm incurs $O(PT_\infty^2)$ deviations, matching that of the parsimonious variant. For general use of futures, the algorithm incurs $O(m_k T_\infty + PT_\infty \lg P)$ deviations, where m_k is the maximum number of future touches that are logically parallel. Compared to the bound for the parsimonious variant, $O(kT_\infty + PT_\infty)$, with k being the total number of touches in the entire computation, this bound is better assuming $m_k = \Omega(P \lg P)$ and is smaller than k, which holds true for all the benchmarks we examined.

PPoPP '19, February 16–20, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-6225-2/19/02…$15.00
https://doi.org/10.1145/3293883.3295735

*CCS Concepts• **Software and its engineering** → **Scheduling**; • **Theory of computation** → *Parallel computing models*; • **Computing methodologies** → Shared memory algorithms; Parallel programming languages;

1 Introduction

The use of future constructs provides a flexible way to express parallelism. Similar to fork-join parallelism, one can ***spawn*** off a ***future task*** that executes logically in parallel with the continuation of the spawning statement. Unlike fork-join parallelism, however, the termination of a future task is not restricted to a lexical scope. Rather, the spawn statement returns a future handle that can be used to retrieve the value produced by the future task. When the handle is ***touched***, the control is blocked until the corresponding future task terminates and returns a value.

The additional flexibility of futures allows one to write a wider range of parallel programs and/or provide a higher level of parallelism beyond what can be specified using only fork-join parallelism. For instance, Blelloch and Reid-Miller [8] show that one can asymptotically reduce the span of various tree operations using parallel futures. Since its proposal in the 80s [21], future constructs have been incorporated into various task parallel languages and platforms [13–15, 18, 21, 31, 37, 42, 45], including the C++11 standard [29].

Modern task parallel platforms typically employ ***work stealing*** [9, 12, 21, 31, 48], a class of scheduler for load balancing parallel computations. Examples of such platforms include, but are not limited to, OpenMP [40], Intel TBB [27], various dialects of Cilk [17, 28, 33, 36] and Habanero [4, 13], X10 [15], and Java Fork/Join framework [32].

Work stealing utilizes a randomized distributed protocol, which admits an efficient implementation [19]. During parallel execution, each ***worker thread*** (a surrogate of a processor) maintains its own double-ended queue (***deque***) to keep track of available work. Primarily, a worker operates on its own deque locally; only when it runs out of work (i.e., its deque is emptied) does a worker become a ***thief*** and ***steal*** work from the top of a randomly chosen ***victim***'s deque.

It has been shown that such a work-stealing scheduler provides strong performance guarantees [2, 3, 10, 11]. For a given computation, let ***work*** (T_1) be the time it takes to execute on one processor; similarly, let ***span***[1] (T_∞) be the time it takes to execute on infinitely-many processors — one can also think of it as the length of a longest dependence

[1]The term span is sometimes called "critical-path length" and "computation depth" in the literature.

chain in the computation. A work-stealing scheduler can schedule the computation on P processors in expected time $T_1/P + O(T_\infty)$ [2, 3, 10, 11], which leads to linear speedups for computations containing ample parallelism.

Even though work stealing and its execution time bound apply to programs that use futures [2, 3], the flexibility of futures can incur additional costs. Specifically, prior works [1, 23, 41] show that using work stealing with futures, compared to work stealing with only fork-join parallelism, can incur a much higher number of "deviations," a better metric for evaluating the performance of parallel executions.

As articulated by Spoonhower et al. [41], the number of deviations provide a better metric for evaluating performance bounds because it is highly correlated to the additional cache misses and scheduling overheads of parallel executions. Informally, a ***deviation*** occurs during a parallel execution when a processor executes an instruction whose ordering in the instruction stream deviates from that of the serial execution. A deviation forces the scheduler to perform additional bookkeeping to keep track of the events that cause the deviations. Moreover, the number of deviations can be used to bound the extra cache misses incurred on the private caches during parallel executions (as first shown by Acar et al. [1]) — intuitively, the bound holds by considering each deviation to execute with an empty private cache.

Given a computation that employs only fork-join parallelism, Acar et al. [1] show that the expected number of deviations[2] incurred by a work-stealing scheduler is $O(PT_\infty)$. In contrast, given a computation that employs k future operations, Spoonhower et al. [41] show that the expected number of deviations incurred is $O(kT_\infty + PT_\infty)$, an additional k multiplicative factor. More recently, Herlihy and Liu [23] show that if futures are used in a *restricted* fashion one can bound the number of deviations to be $O(PT_\infty^2)$.

Although these bounds suggest that futures can indeed incur much higher overhead than strict fork-join parallelism, all these prior works assume a ***parsimonious*** work-stealing scheduler, where each worker maintains a single deque and only steals to load balance when its deque becomes empty. Due to the parsimonious nature of work stealing, each future touch can lead to $O(T_\infty)$ number of deviations, contributing to the $O(T_\infty)$ multiplicative factor in the deviation bound.

In this work we propose ***proactive*** work stealing: whenever a worker thread encounters a future touch that is not ready, it suspends the execution of its current task and tries to find something else to do. By proactively suspending the computation instead of expanding what's already on the deque, one can minimize the deviations and their corresponding scheduling overhead and cache misses.

We show that the proposed proactive work-stealing algorithm, called ProWS, can provide a comparable execution time bound to the parsimonious variant, as well as equal or better bounds on the number of deviations for programs that use futures. Given a computation that employs futures with T_1 work and T_∞ span, the proposed algorithm executes the computation on P processors in $O(T_1/P + T_\infty \lg P)$ time, which is asymptotically comparable to the parsimonious version (except for the $\lg P$ overhead on the span term). For ***structured*** use of futures, where the future is single-touch with no races on the future handle, the algorithm incurs $O(PT_\infty^2)$ number of deviations, the same as the bound for the parsimonious variant. For ***general*** use of futures, where the only restrictions are a constant number of touches per future and deadlock-freedom during one-worker execution,[3] the algorithm incurs $O(m_k T_\infty + PT_\infty \lg P)$ deviations, where m_k is the maximum number of future touches that are logically parallel. This bound is better than the bound for the parsimonious variant if $m_k = \Omega(P \lg P)$ and is smaller than k, the total number of touches in the entire computation; these assumptions hold true for all the benchmarks examined. Since proactive and parsimonious work stealing behave the same for programs that utilize only fork-join parallelism, they have the same bounds for such programs.

We have implemented a work-stealing runtime system called Cilk-F, extending the task parallel runtime system of Cilk Plus [28] to incorporate support for parallel futures scheduled using ProWS. Even though futures have been incorporated into various task parallel platforms [13–15, 18, 21, 31, 37, 42, 45], Cilk-F is the first provably efficient proactive work-stealing runtime that supports futures. An interesting design question arises when considering the implementation of the "cactus stack" [33] in supporting futures, which we discuss in more detail in Section 5. No prior stack space bound exists for parallel futures when scheduled using parsimonious work stealing. Interestingly, when scheduled using proactive work stealing, it's possible to bound the stack space, albeit loosely.

We empirically evaluate Cilk-F with nine benchmarks and compare it to Cilk Plus. For benchmarks where the use of futures does not provide additional parallelism, Cilk-F performs comparably to Cilk Plus, indicating that the additional $\lg P$ term in front of the span does not adversely impact performance in practice. For benchmarks where the use of futures provides additional benefit, Cilk-F can obtain better speedup. Nevertheless, future operations do incur higher scheduling overhead compared to pure fork-join parallelism, and we empirically analyze this overhead.

Contributions

In summary, this paper makes the following contributions:

- We propose ProWS, a proactive work-stealing algorithm for scheduling computations with futures (Section 3). Even though the algorithm is stated in terms of futures, the

[2] Acar et al. refer to deviations as drifted nodes in [1].

[3] Prior work by Spoonhower et al. [41] assumes single-touch per future, but constant number of touches does not change their bound.

algorithm works for computations with general synchronization patterns.
- We show that ProWS provides equal or better bounds on the number of deviations than the parsimonious variant (Section 4).
- We describe the implementation of Cilk-F, the first provably efficient proactive work-stealing runtime system that supports the use of futures (Section 5). We discuss in detail how Cilk-F supports futures in the cactus stack and the resulting stack space usage bound.
- We empirically evaluate Cilk-F and show that ProWS can be implemented efficiently (Section 6).

2 Preliminaries

Fork-Join Parallelism vs. Futures. To explain the semantic distinctions between fork-join parallelism and futures, we will use syntax such as `spawn`, `sync`, `fut-create`, and `get` and assume that they operate with function instances.[4]

When a function F ***spawns*** another function G by prefixing the call with a `spawn`, the continuation of the `spawn` in F may execute in parallel with G. The `sync` keyword ensures that control cannot pass beyond the `sync` until all functions previously spawned via `spawn` have returned. If not specified explicitly, a function containing `spawn` has a `sync` inserted implicitly at the end the function, and thus all previously spawned children must return before the parent can return.

Like `spawn`, `fut-create` can be used to create parallelism. A function F may spawn off a function G representing a ***future task*** by prefixing the call to G with `fut-create`, and the continuation of F may execute in parallel with G. Unlike `spawn`, however, the termination of G is not confined to the enclosing lexical scope of the call, and the execution of a `sync` in F has no effect on it. Rather, the `fut-create` call returns a ***future handle***, which can be used later to ensure termination of G and retrieve the result of its evaluation. One can invoke `get` on the future handle, an operation referred to as the ***future touch***, which causes the control to block until the corresponding future task terminates. Implicitly, we assume that the end of a future task always executes a `put`, depositing the task's resulting value into its handle.

Given these constructs, one can derive a ***serial elision*** of the parallel code by removing `spawn` and `sync`, and replacing `fut-create` with a simple call and using the return value from the call wherever `get` is invoked.

Modeling Parallel Computations. The execution of a parallel computation can be modeled as a ***directed acyclic graph***, or ***dag*** [11], where each node represents a ***strand***, a sequence of instructions containing no parallel keywords, and the edges represents control dependencies between strands (i.e., an edge from u to v means that v cannot execute until u finishes).

[4] Although syntax for other platforms may vary, the scheduling algorithm specified in terms of the computation dag should be broadly applicable.

The execution of a `spawn` terminates the current strand, which is a ***spawn node*** with two outgoing edges, one to the first strand in the spawned child, and one to the continuation in the parent. The execution of a `sync` terminates the current strand and creates a ***sync node***, representing the continuation after the `sync`, with multiple incoming edges (one from each spawned child).

The `fut-create` keyword behaves similarly to `spawn`. It terminates the current stand, which is a ***future spawn node*** with two outgoing edges: one to the first strand in the future task, and one to the continuation of `fut-create`. A future touch, or invocation of `get`, terminates the currently executing strand and creates a ***future join node*** (or ***join node*** for short) that has two incoming edges: one from the strand that was terminated by the invocation of `get`, referred to as the ***local parent*** of the join node, and one from the last strand of the corresponding future task that executes the `put`, referred to as the ***future put node***.

For ease of description, we will refer to the edge that goes from a future spawn node to the first strand of the spawned future task as a ***create edge***. We will refer to the edge that goes from the last strand of a future task to the corresponding future join node as a ***join edge***.

For a computation that uses only `spawn` and `sync`, the resulting dag is a ***series-parallel dag (SP dag)*** [47] with a single source and a single sink that can be constructed recursively using the following rules:
- ***base case:*** a single strand (a node) is an SP dag;
- ***series composition:*** given two SP dags G_1 and G_2, compose them in series by adding an edge between the sink of G_1 and source of G_2;
- ***parallel composition:*** given two SP dags G_1 and G_2, compose them in parallel by adding a new source s and a new sink t, with edges from s to the sources of G_1 and G_2, and edges from the sinks of G_1 and G_2 to t.

When the program uses futures, the computation can be modeled as multiple independent SP dags, connected via create edges and join edges. That is, if F spawns G via `spawn`, then the SP dag of G is part of the SP dag of G. On the other hand, if F spawns G via `fut-create`, then F and G are independent SP dags, with the first strand of G being the source of a separate SP dag and the last strand of G being the sink.

Prior works on work stealing analysis typically assume that a dag consists of nodes with at most two outgoing edges (e.g., [1, 2, 11, 23, 41]). One can transform the dag by replacing a node with multiple outgoing edges into a chain of nodes, each with two outgoing edges. This becomes relevant if a future handle can be touched multiple times. Provided that the number of touches per handle is constant, however, such a transformation does not adversely impact the span.

Parsimonious work stealing. Given a program and an input, the computation dag unfolds dynamically as the program executes, and it's the job of a scheduler to determine how to best map the computation to processing cores in a way that respects the dependences specified by the parallel constructs. We say that u is an ***immediate predecessor*** of v if there is an edge from u to v and v an ***immediate successor*** of u. A node v is ***ready*** or ***enabled*** when all its immediate predecessors in the dag have executed, and only ready nodes can be executed.

With parsimonious work stealing, each worker maintains a deque that contains only ready nodes. For the most part, each worker accesses the bottom of its own deque like a stack (FILO). When a worker executes a node, the execution may enable zero, one, or two nodes (e.g., spawn node). If zero are enabled, the worker pops off the bottommost node from its deque and executes it. If one is enabled, the worker executes the enabled node. If two are enabled, a worker executes one (say the left one) and pushes the other (say the right one) onto the bottom of its deque. When a worker runs out of work, (i.e., the deque becomes empty), it turns into a ***thief*** and randomly picks a ***victim*** to ***steal*** from by removing the *topmost* (oldest) node from the victim's deque.

As customary to prior works, we shall assume that the spawned function or future task is always the left child of the spawn node and the continuation strand the right child. Thus, a ***serial (one-worker) execution*** of a computation dag follows the left-to-right depth-first traversal. This also means that we assume eager evaluation of futures, where the future task is always evaluated before the continuation of `fut-create` under serial execution.

Given a dag, the serial execution imposes a total order on the nodes. Say in this total order, v executes immediately after u. In a parallel execution, if a worker w executes v but not immediately after it executes u, then we say v incurs a ***deviation***. This could happen either because a different worker executed u or because worker w executed something else between u and v.

Types of futures. A ***structured*** use of futures imposes the following restrictions: 1) single touch, meaning that only a single `get` is invoked on each future handle, and 2) no race on a future handle, meaning that there is a directed path between a future spawn node to the local parent of its corresponding touch. Note that this restriction is the same as prior work [23] and does not preclude a future task to execute in parallel with the function that performs its touch before the `get` keyword. It simply means that the spawning of the future (which writes to the future handle) must be in series with the invocation of the corresponding `get` (which reads the future handle). A ***general*** use of futures imposes the following restrictions: each future is touched a constant number of times and all the join edges are ***forward pointing***, namely, a `fut-create` is always before its corresponding `get` in a serial execution.

3 Proactive Work-Stealing

This section describes the proactive work stealing algorithm, which we shall refer to as ProWS in the rest of the section. We will refer to the original parsimonious algorithm analyzed by Arora et al. [2] (described in Section 2) as ABP.

The main distinction between ProWS and ABP is as follows. When a worker executes a `get`, the associated future task may not be ready, so executing the `get` does not enable the subsequent future join node. With ABP, this simply falls under the case of enabling zero nodes, and the worker continues execution by popping off the bottommost node to execute next. ProWS handles the execution of `get` differently. If its future task is not ready, the worker ***suspends*** the entire deque and tries to find work elsewhere. An important consequence of such behavior is that there can be more than P deques in the system, where P is the number of workers.

In ProWS, suspended deques are still stored in a distributed fashion, thus each worker now manages a single ***active*** deque that it actively works on and a set of ***stealable*** deques that are not being actively worked on but contains ready nodes. When stealing, once a victim is chosen, a thief can steal from any deque that belongs to the victim with equal probability (including its active deque).

Data Structures Used

We shall first discuss the data structures used by the algorithm. Each deque supports the following operations:

- `popTop`: remove and return the node from the top;
- `popBottom`: remove and return a node from the bottom;
- `pushBottom`: insert a node onto the bottom;
- `pushBottomImplicit`: insert a node onto the bottom of the deque and mark the node as *suspended*; and
- `isEmpty`: return true if there are no ready nodes in the deque (but may contain one suspended future join node).

Just as in ABP, we assume that multiple workers can make calls to a deque concurrently; if more than one worker tries to pop the same element off the deque, one of them succeeds and the other one fails in a constant number of time steps.

Throughout the lifetime of a deque, it can be in one of the following four states:

- ***active:*** it is actively been worked on by a worker;
- ***suspended:*** it is suspended due to a `get` call; every node in the deque is ready, except for the bottommost node, which is the corresponding suspended future join node;
- ***resumable:*** it contains only ready nodes, but it is not actively being worked on by a worker; and
- ***muggable:*** similar to a resumable deque, except that the entire deque can be stolen and resumed.

These states are exhaustive, and a deque can only transition: 1) from active to suspended due to execution of a `get` call, 2) from suspended to resumable due to termination of the future task enabling the join node at the bottom, 3) from resumable to active if the worker who finishes the future task

has an empty deque and resumes one of the now-resumable deques suspended with the future handle; 4) from resumable to muggable after a thief steals from it once, and 5) from muggable to active when a thief mugs it and resumes its execution. Since a resumable deque transitions to muggable once it is stolen from, only its top item may be stolen before transitioning. If a thief steals into a muggable deque, it takes the entire deque and resumes its execution from the bottommost node.

The stealable deques belonging to a worker are maintained as a set. Each future handle also maintains a deque set with references to suspended deques, allowing any deques suspended with the handle to be resumed when the future task completes. A deque set supports the following operations:

- `add(`*deq*`)`: add deque *deq* into the set;
- `remove(`*deq*`)`: remove deque *deq* from the set;
- `removeRandom()`: remove and return a deque from the set, chosen uniformly at random; and
- `pickRandom()`: return a reference to a deque in the set, chosen uniformly at random (but does not remove it).

We assume that one can make concurrent calls to a given set, and an operation will finish in constant amortized time. When operating on the stealable set of a worker, the worker is always chosen uniformly at random among the P workers. Thus, the contention can be resolved in a constant number of time steps in expectation (e.g., see lemma 6 in [11]). In practice, a set can be implemented as a growable array (performing array doubling when necessary), which maintains a constant amortized insertion cost.

The Algorithm

Algorithm 1 shows the main scheduling loop for ProWS and its helper functions. Ignoring the special handling of future operations in lines 29–37, ProWS behaves the same as ABP. Each worker starts out with one active deque; it operates off the bottom of the deque (line 22 and lines 26–28) and steals when it runs out of work to do (lines 23–24). A worker, when enabling two nodes, pushes the right node (i.e., continuation) first (line 27) and then the left node (i.e., the spawned task) (line 28), which means that the left node gets executed next.

Future operations are handled differently. If the execution of this strand terminates with `get` (lines 29–35) and the corresponding future task f has not terminated, `get` enabled zero nodes. The worker then pushes the corresponding future join node *j* (the immediate successor of *n*) onto the bottom of the deque via `pushBottomImplicit` (line 31) and suspends the deque (line 32). The reference to the suspended deque is stored with the future handle of f (line 33) and the worker's active deque is set to *null* (line 34). It will be set to something else after the steal. On the other hand, if the executed strand terminates with `put`, that its corresponding future task f has terminated and all suspended deques stored with f can now be resumed (lines 36–42). At this point, if the worker executing `put` has an empty active deque, it will set its active deque to one of the suspended deques stored with the future handle and resume its execution next (lines 38–41).

The implementation of suspend is shown in lines 1–9. Since ProWS may potentially suspend many deques, it takes extra steps to ensure that the number of stealable deques

Algorithm 1: The main scheduling loop

```
1  Function suspend(deq) //w is the executing worker
2    deq.status ← SUSPENDED;
3    if dep.IsEmpty() then deq.worker ← null;
4    else
5      v ← ChooseRandomVictim() ; //can include w itself
6      v.stealable.add(deq);
7      deq.worker ← v;
8    end
9  end
10 Function setToActive(deq) //w is the executing worker
11   if deq.worker then
12     rebalanceStealables(deq.worker);
13     deq.worker.stealable.remove(deq);
14   end
15   deq.worker ← null ; //deq is not in any stealable
       set
16   deq.status ← ACTIVE;
17   if w.active is not null then freeDeque(w.active);
18   w.active ← deq;
19 end
20 while computation is not done do //w is the executing worker
21   n ← null ; //n points to next strand to execute
     //w.active points to either null or its active deque
22   if w.active is not null then n ← w.active.popBottom();
23   if n is null then
24     steal() ; //steal returns when work is found
25   else //execute n
26     left, right ← execute(n);
27     if right is not null then w.active.pushBottom(right);
28     if left is not null then w.active.pushBottom(left);
       //special case: f is a future handle
29     if n terminated with f.get() then
30       if f is not ready then
           //j is the future-join node after n
31         w.active.pushBottomImplicit(j);
32         suspend(w.active);
33         f.suspended.add(w.active);
34         w.active ← null;
35       end
36     else if n terminated with f.put() then
         //Mark every deque in f.suspended RESUMABLE
37       markSuspendedResumable(f.suspended);
38       if w.active is empty then
39         deq ← f.suspended.removeRandom();
40         setToActive(deq);
41       end
42     end
43   end
44 end
```

are roughly balanced among workers. Instead of suspending with the current worker w, it chooses a target worker v uniformly at random (which can include w itself) and suspends the deque with v. The reference to v is stored with the suspended deque so that when the deque gets resumed it can be removed from worker v's stealable set.

If the suspended deque contains no ready nodes (line 3) we don't store the deque in any worker's stealable set, as it has nothing to be stolen from. Such a deque, once gets resumed, is inserted into a stealable set of a worker chosen uniformly at random (by `markSuspendedResumable` in line 37).

Finally, a key thing to note in `setToActive` is that it invokes `rebalanceStealables` (line 12), which is invoked whenever w is about to remove a deque from v's stealable set — it randomly chooses another victim v'; if $v = v'$, w is done; otherwise w moves a stealable deque from v' to v if v' has one. Section 4 explains why we do such a rebalance.

Algorithm 2: The steal protocol

```
45 Function steal() //w is the executing worker
46   while true do //steal returns only when work is found.
47     v ← ChooseRandomVictim() ; //can include w itself
48     deq ← pickRandom(|v.active| ∪ |v.stealable|);
49     if deq is null then continue; //Nothing to steal from
         v
50     if deq.status is MUGGABLE then
51       setToActive(deq);
52       break;
53     end
54     n ← deq.popTop() ; //deq is suspended or resumable
55     if deq.isEmpty() then
56       handleEmptyDeque();
57       rebalanceStealables(v);
58     else if deq.status is RESUMABLE then
59       deq.status ← MUGGABLE
60     end
61     if n is not null then
62       if w.active is null then w.active ← newDeque();
63       w.active.pushBottom(n);
64       break;
65     end
66   end
67 end
```

Algorithm 2 shows the implementation of the steal protocol that a worker w invokes when its deque becomes empty or after it loses its deque due to suspension. The steal function performs steal attempts until w finds work successfully.

When stealing, w chooses a victim v uniformly at random (line 47, which again includes w) and chooses a deque uniformly at random among v's deques (line 48). If the chosen deque is muggable, w takes the whole deque and set it to be its active deque. Otherwise, w steals from the top (line 54). After popTop, if the deque runs out of ready nodes, it is removed from v's stealable set and possibly destroyed if there isn't even a suspended future join node at the bottom, such as in the case of resumable deque (line 56). Moreover, `rebalanceStealables` is invoked again. If the deque is resumable and not empty, it is marked as muggable (line 59). After a successful steal, w may need to allocate a new deque (lines 62 and 63).

4 Performance Bounds for ProWS

This section analyzes ProWS to show that, 1) for a computation with T_1 work and T_∞ span, it executes the computation in expected time $O(T_1/P + T_\infty \lg P)$, and 2) the number of deviations is bounded by $O(PT_\infty^2)$ for a program that uses structured futures and $O(m_k T_\infty + PT_\infty \lg P)$ for a program that uses general futures.

Before we analyze the bounds, we first show that, at any point during the execution, the set of stealable deques are roughly evenly distributed across workers, which we utilize when we discuss the bounds. We use the following lemma on the classic balls-into-bins problem, which is not hard to show (see e.g., [38, Chp. 5]):

Lemma 4.1. *When m balls are thrown independently and uniformly at random into n bins, the probability that the maximum load is more than $\frac{m}{n} + O(\lg n)$ is at most $1/n$. Similarly, the probability that the minimum load is less than $\frac{m}{n} - O(\lg n)$ is at most $1/n$.*

Lemma 4.2. *Given P workers and S number of stealable deques in the system, with probability $1 - o(1)$ each worker has at most $S/P + O(\lg P)$ deques.*

Proof Sketch. One can model the number of stealable deques per worker as the classic balls-into-bins problem, where the workers are modeled as bins and the stealable deques are modeled as ball tosses. Our process also includes muggings, however, which changes the size of the stealable sets, and thus the analysis requires additional care.

We model the entire process as two separate ball-toss processes: a ***deque-suspension*** process, where a suspended deque is modeled as a ball toss into a randomly-chosen bin (worker to leave the deque with), and the ***deque-removal*** process, where removing a deque is also modeled as a ball toss into a randomly-chosen bin (worker to remove the deque from). Then the size of a given stealable set is the number of balls resulted from the deque-suspension process minus the number of balls resulted from the deque-removal process. The upper and lower bounds on the maximum and minimum loads in Lemma 4.1 thus give us the desired bound.

It is not hard to see that the workers from the deque-suspension process is chosen uniformly at random. What remains to be shown is that the same holds true for the deque-removal process. There are a couple ways a deque can disappear from a stealable set: 1) a worker takes the whole deque to resume it (lines 40 and 51); and 2) a deque becomes empty after it is stolen from (lines 55–57). In both

cases, we always invoke `rebalanceStealables`: if we are removing a deque from v, we randomly choose a victim v' to move a stealable deque to v. If v' has a stealable deque to move to v, it's as if we removed the deque from v'. If v' does not have a stealable deque, it's as if we first moved the deque to v' and then removed it. Pretending to move a deque from v to v' is ok, since v has a larger stealable set at the moment, and doing so simply balances the load from a more-loaded worker to a less-loaded one. Even though such load balancing is conditioned on v' not having any deque, doing so does not hurt the bound. □

4.1 Bound on Execution Time

Our time bound analysis follows a similar structure to the analysis done in [2] and [46]. We separately bound the number of time steps devoted to various activities: work, steal attempts, and muggings. By bounding how many time steps each activity takes, the final bound arises by summing all the time steps divided by P, the number of workers used. Obviously, the total work is bounded by T_1 time steps.

It remains to bound the number of steal attempt and mugging operations, each taking a constant number of time steps. In the original work stealing analysis by Arora et al. [2, 3], henceforth referred to as ABP, steal attempts are bounded by a potential function argument that states the following. Assuming there are P deques in the system, after $O(P)$ steal attempts, the overall potential decreases by a constant fraction. This is because, the topmost node in a deque contributes to a constant fraction of the overall potential among nodes within the deque.

More formally, the following lemma is a straightforward generalization of lemma 7 and 8 in ABP [2] which we utilize:

Lemma 4.3. *Let Φ_i denote the potential at time t and say that the probability of each deque being a victim of a steal attempt is* at least $1/X$. *Then after X steal attempts, the potential is at most $\Phi(t)/4$ with probability at least* $1/4$. □

Effectively, this lemma says that the number of steal attempts is at most $O(XT_\infty)$, since the potential function is a function of T_∞. For ABP, it is always the case that $X = P$, leading to a steal attempts bound of $O(PT_\infty)$.

The ABP analysis cannot be applied to ProWS directly, since 1) ProWS can have more than P deques in the system, and 2) a thief stealing into a muggable deque will resume the bottommost node in the deque instead of the topmost one, which may not contain sufficient amount of potential.

To resolve issue 1), we apply a similar technique to Utterback et al. [46] and divide the computation into two types of phases: a ***steal-bounded phase*** when there are at most $2P$ stealable deques, and a ***work-bounded phase*** when there are more than $2P$ stealable deques. During a steal-bounded phase, by Lemma 4.2, we know each worker has at most $O(\lg P)$ deques, leading to a steal attempts bound of $O(T_\infty P \lg P)$ by Lemma 4.3. During a work-bounded phase, the total number of deques in the system is more than $3P$. However, since there are many deques in the system distributed roughly equally among workers, steal attempts are likely to succeed, each followed by a unit of work. Thus, we can bound the steal attempts by $O(T_1)$ during a work-bounded phase. Overall, this leads to an execution time bound of $O(T_1/P + T_\infty \lg P)$.

We still need to resolve issue 2) and in addition bound the time spent on muggings. Recall that in ProWS, we enforce that every resumable deque has to be stolen from once before it becomes muggable. This may seem counter-intuitive — why not simply resume the deque from the bottom if it is already resumable? This steal-before-mug ensures that for each mugging there is a corresponding successful steal on the same deque to amortize against. Doing so prevents the worst case scenario where a deque with a high-potential node on top repeatedly becomes resumable and mugged but never stolen from. This scenario would prevent us from bounding steal attempts that lead to a successful mugging.

Thus, we can also bound the time steps spent on mugging against steals, resulting the following time bound:

Theorem 4.4. *Consider a computation with T_1 work and T_∞ span. The expected execution time is $O(T_1/P + T_\infty \lg P)$.*

4.2 Bounds on Deviations

We first define some notations. Given a computation dag G, we say that u is a ***predecessor*** of v and v is a ***successor*** of u iff there is a directed path from u to v.

We make the following assumption. Let u be a node with two outgoing edges, meaning that u can be a spawn node, a future spawn node, or a future put node. The only way for a future put node to have an out-degree of two is if the corresponding future is multi-touch, which creates a chain of put nodes, each with an out-degree of two.

Given a computational dag G, the ***sequential order*** is a total ordering of nodes in G that arises from the sequential (one-worker) execution. A ***processor order*** of a worker w is the sequence of nodes processed by w in a parallel execution of ProWS. We say $u \prec_1 v$ if u is before v and $u \prec_1 v$ if u is immediately before v in the sequential order. Similarly, we say $u \prec_w v$ if u is before v and $u \prec_w v$ if u is immediately before v in the processor order of w. Given this notation, we now formally define ***deviation***:

Definition 4.5. Let u and v be two nodes in a dag and $u \prec_1 v$. We say that v is a deviation in the parallel execution if for some worker w that executed v, we have $u \not\prec_w v$.

Given the definition of SP dags (Section 2), it's not hard to see that for every sync node v in an SP dag G, there is a corresponding spawn node u. Let $u.lchild$ denote the left child of u and $u.rchild$ denote the right child of u. Similarly, let $v.lparent$ denote the left parent of v and $v.rparent$ denote the right parent of v. Then, let G_{left} be the SP subdag that

consists of the set of nodes x in G such that there is a path from $u.lchild$ to x and from x to $v.lparent$. We say G_{left} is the SP dag ***enclosed*** by $u.lchild$ and $v.lparent$.[5] We define G_{right} symmetrically. We first show properties of the sequential and parallel executions when scheduled with ProWS.

Lemma 4.6. *Given an SP dag G enclosed by a spawn node u and a sync v, let x be a node in G_{left} and let y be a node in G_{right}. Then, $x <_1 u.rchild <_1 y <_1 v.rparent$. Moreover, $v.rparent \prec_1 v$.*

Proof Sketch. Since each future task is modeled as a separate SP dag, and a worker w performs eager evaluation on a future task when it encounters a `fut-create`, we can show that this lemma holds by inducting on the number of independent SP dags. □

Effectively, this lemma says that the sequential order of ProWS follows the depth-first left-to-right traversal of the dag. Moreover, since for either structured or general use of futures the `fut-create` of a future task f must appear before the corresponding `get` in sequential order, the sequential execution of ProWS can never suspend due to a `get`.

Now we prove lemmas about parallel executions.

Lemma 4.7. *Let v be a sync node and u its corresponding spawn node. If v is a deviation, then $u.rchild$ must be stolen.*

Proof Sketch. A similar lemma has been shown by Acar et al. [1] for pure SP dags scheduled using classic work stealing (i.e., ABP). Here, we additionally need to consider how ProWS diverges from ABP when handling futures. Let G be the SP dag enclosed by u and v and a worker w pushes $u.rchild$ onto its deque deq. By Lemma 4.6, sequentially w will not pop $u.rchild$ off the deque before executing $v.lparent$. Then, the only way for w to deviate from the sequential execution if to encounter a `get` and suspend deq. In which case, either $u.rchild$ is popped off the deque due to a successful steal, or the entire deq is mugged, in which case the worker mugging the deque will resume execution from the bottom and follow sequential order for the rest of G. □

Lemma 4.8. *If a worker w enables no children after executing the right parent of a sync node, then $w's$ deque is empty.*

Proof. Let v be the sync node and u the corresponding spawn node. By 4.7, if v is a deviation, $u.rchild$ is stolen. Consider the SP subdag G enclosed by $u.rchild$ (G's source node) and $v.rparent$ (G's sink node). Then any node in G is executed before $v.rparent$ in any execution. We know w must steal $u.rchild$ or any node in G, otherwise there is no possibility for w to process $v.rparent$. Suppose the deque is not empty after executing $v.rparent$. Let z be bottommost node on the deque. We must have z outside G since any node in G has been executed. Furthermore, $w's$ deque is empty when w performs the steal. Then everything in the deque afterwards is a descendent of $u.rchild$. So z can only be a node in a future dag spawned by G.

Worker w will turn to the future subdag immediately after executing the corresponding future spawn node. There are two ways that w can resume the execution of G: (1) the future completes, or (2) $w's$ deque is empty again and it performs a steal targeting a node in G. In both cases, z cannot be on $w's$ deque, which contradicts our supposition. □

At a high-level, we bound the number of deviations as follows. We define the notion of "traces" that divide the sequence of nodes executed by a worker based on the types of nodes. We then show that only the first node in a trace can incur a deviation. Lastly, we show that, such a node is either the direct result of a successful steal or can be amortized against a successful steal.

Definition 4.9. Consider a sequence of nodes processed by w, which we then separate into a set of ***traces***, where each trace begins with one of the following nodes: (1) a sync node, (2) a node that gets executed immediately after w performs a successful steal, and (3) a node that gets executed immediately after w performs a successful mugging.

Observation 1. *Given a node n in the dag, n can be one of the following:*

1. *n is a **regular node**: n has one child in the dag, and the child has only n as a parent;*
2. *n is a **spawn node**: n has two children in the dag, where each child has only one parent;*
3. *n is a **future put node**: n can have either one child (single touch future) or two children (chain of put nodes for multi-touch futures).*
4. *n is a **parent of a sync node**: n has one child, where the child has two parents;*
5. *n is the **local parent of a future join node**: n has one child, where the child has two parents.*

It can be seen that these types are exhaustive by enumerating all the possible combinations of the out-degree of n and in-degree of n's children. Note that we can never have a spawn node n leading to a child who is a sync node or join node — the act of invoking `get` or `sync` terminates the current strand and creates a new node with an in-degree of two. Thus, a `get`/ `sync` that immediately follows a `spawn`/ `fut-create` will have a node inserted between them.

Lemma 4.10. *Consider a sequence of nodes executed by a worker w during parallel execution scheduled using ProWS, which we separate into traces according to Definition 4.9. For a given trace $t = (n_1, n_2, \ldots, n_l)$, only n_1 can be a deviation.*

Proof. Let $s = (n_1, \hat{n_2}, \ldots, \hat{n_l})$ be the sequence of l nodes that starts with n_1 in sequential execution. We show that $n_i = \hat{n_i}$ for $i = 2 \ldots l$ by inducting on the length of the trace and argue that either processor w behaves exactly as sequential execution, or the trace ends.

[5] Recall that each future task is treated as its own SP dag and thus if a node x in G spawns a future task via `fut-create` none of the nodes belonging to the future task is in G.

Inductively, assume $n_i = \hat{n_i}$ for $i = 2 \ldots j-1$ and w behaves exactly the same as the sequential execution up to that point (i.e., each n_i enabled exactly the same nodes as $\hat{n_i}$). Now we consider n_j. Based on Observation 1, n_j can be one of the following. **regular node:** n_j must enable its only child and execute it next, just as in sequential execution.

spawn node: n_j must enable both children, executing the left one and pushing the right one onto the deque, just as in sequential execution.

future put node: if n_j is a put node for a single-touch future or one at the end of a put chain, then n_j can either enable nothing or the corresponding future join. If n_j enables nothing, this is the same as sequential execution, and w either pops its bottom deque (which leads to the same n_{j+1} by inductive hypothesis), or trace t ends at n_j if w's deque is empty, since the next node has to follow from either a successful steal or mugging. If n_j enables the corresponding future join node j, that means the local parent of j executed and couldn't enable j and thus pushed j onto the bottom of some (now suspended) deque. Even though n_j enabled j, note that in ProWS, it does not push j onto w's deque. Instead, it simply marks the deque as resumable.

On the other hand, n_j can be a put node for a multi-touch future that enables the next put node and may or may not enable the corresponding future join node. Enabling the next put node is exactly the same as sequential execution, and whether the corresponding future join node is enabled or not does not matter, following similar argument as above.

parent of a sync node: If executing n_j enables this sync node, then trace t ends at n_j (by the definition of the trace). If in both sequential and parallel executions, n_j enables no child, then w tries to pop a node off the bottom of its deque, which leads to the same n_{j+1} by the inductive hypothesis. On the other hand, if n_j enables no child but in sequential execution, $\hat{n_j}$ enables the sync node, then n_j must be the right parent of the sync node. Then by Lemma 4.8 w's deque must be empty and thus trace t ends at n_j.

local parent of a join node: In the sequential execution, since the local parent always enables the join node, $\hat{n}_{j+1}$ will be the join node. In the parallel execution, either n_j also enables the join node, which means $n_{j+1} = \hat{n}_{j+1}$, or it enables no child. In the latter case, w will push the join node onto the bottom of the deque and suspend the deque, which mean trace t ends at n_j because the next node has to follow from either a successful steal or mugging. □

Finally, a key theorem to bound the number of deviations:

Theorem 4.11. *Given an execution of ProWS. Let n be the number of successful steals in the execution. Then, the number of deviations is $O(n)$.*

Proof. From the definition of traces and Lemma 4.10, we know a deviation may only occur at the beginning of a trace. Each trace begins with a sync node, a stolen node, or a node processed after a successful mugging. Thus, the number of deviations is bounded by the sum of the numbers for deviated sync nodes, successful steals, and muggings.

From Lemma 4.7, we know each deviation at a sync nodes has a unique corresponding stolen node, thus we can bound the number of deviated sync nodes by the number of successful steals. Also recall that in ProWS, a resumable deque has to be stolen from successfully before it becomes muggable. Thus, the number of successful muggings is also bounded by the number of successful steals. Thus, the total number of deviations is bounded by $O(n)$, where n is the number of successful steals. □

Given Theorem 4.11, we can now bound the deviations by bounding the number of successful steals, which is less than the number of steal attempts during the computation. Recall Lemma 4.3, which effectively states that the number of steal attempts can be bounded by $O(XT_\infty)$, when a deque can be stolen into with probability at least $1/X$. We provide a bound on $1/X$ by bounding the the maximum number of stealable deques possible during the execution.

Lemma 4.12. *Given a computation that uses structured futures scheduled using ProWS, there can be at most $O(PT_\infty)$ stealable deques during execution.*

Proof. In the case of the structured use of futures, the stealable deques can only include suspended deques. Recall that a structured use of futures is restricted to single touch and no race on the handle (i.e., the future spawn node has a directed path to the local parent of the join node). Due to the former, there can exist only one suspended deque per future handle f. Moreover, due to the directed path, the continuation of the future spawn node that spawned f must be stolen in order for a corresponding `get` on f to suspend. Thus, whenever a worker w eventually executes the `put` that completes f's corresponding future task, w's deque must be empty. By Algorithm 1, w will then resume the single deque suspended with f, making it active, and thus there cannot be resumable or muggable deques in the stealable set.

Whenever a worker has to suspend a deque due to `get`, the corresponding future task f is either being actively worked on by another worker (due to eager evaluation), or f is also suspended because f itself invoked a `get` on a different future, creating a chain of suspended future tasks. Such a chain has length at most T_∞ (as any chain in the dag). Moreover, at least one worker is working on the future task at the beginning of the chain. Therefore, there can be at most $O(PT_\infty)$ suspended (stealable) deques. □

Lemma 4.13. *Given a computation that uses general futures scheduled using ProWS, there can be at most m_k stealable deques during execution.*

Proof. By Algorithm 1 lines 29–35, a deque can only suspend when encountering a `get` (future touch). When a future touch

node n causes a deque to suspend, no descendant of n can execute until the deque becomes active again. By definition, since there can be at most m_k number of future touches executing in parallel, this leads to a maximum number of m_k stealable deques at any given time. m_k. □

Finally, we can prove the following deviation bounds:

Theorem 4.14. *Given a computation that uses structured futures with span T_∞ and scheduled using ProWS on P workers, the number of deviations is $O(PT_\infty^2)$ in expectation.*

Proof. By Lemma 4.12, we know the maximum number of deques possible during execution is $O(PT_\infty)$. By Lemma 4.2, each worker can have up to $O(T_\infty + \lg P)$ deques. Thus, a deque is stolen into with probability of at least $O(\frac{1}{PT_\infty + P \lg P})$. Then by Lemma 4.3, the steal attempts across the computation is at most $O(PT_\infty^2 + PT_\infty \lg P)$, or $O(PT_\infty^2)$ assuming $T_\infty = \Omega(\lg P)$, which is likely the case. □

Theorem 4.15. *Given a computation that uses general futures with span T_∞ and scheduled using ProWS on P workers, the number of deviations is $O(m_k T_\infty + PT_\infty \lg P)$ in expectation, where m_k is the maximum number of future touches that are logically parallel.*

Proof. By Lemmas 4.13 and 4.2, we similarly derive the probability that a deque is stolen into to be at least $O(\frac{1}{m_k + P \lg P})$. Then by applying Lemma 4.3 we obtain the bound. □

5 Cilk-F: A Prototype System

This section describes Cilk-F, our prototype implementation of ProWS that extends Intel Cilk Plus [25, 28], a C/C++-based task parallel platform. An interesting design issue that arises is how to best support the "cactus stack" [22] abstraction needed by a task parallel platform. This section discusses the issue, our design choice, and a bound on the stack space based on our particular implementation.

The Cactus-Stack Abstraction

A serial language such as C [30] or C++ [43] utilizes an array-based linear stack because the activation frames of all child functions of a given parent can reuse the same stack space repeatedly. In a task parallel language, since a parent can have multiple spawned children executing simultaneously, their activation frames can no longer occupy the same space. Thus, the underlying system must maintain a ***cactus stack*** [22] abstraction that supports the stack views of multiple children that are active simultaneously, such as in Figure 1.

A natural question that arises is how to best support a cactus stack while providing a good resource usage bound. While multiple mechanisms exist to support a cactus stack, the answer to this question is nuanced, and different implementations make different tradeoffs [33]. For programs that use strictly fork-join parallelism scheduled using work stealing, the best possible bound is PS_1 [10, 11], where S_1 is the stack space needed for special execution.

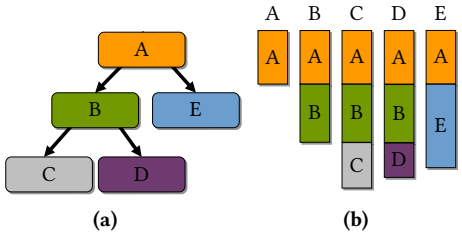

Figure 1. A cactus stack. **(a)** The invocation tree, where function A invokes B and E, and B invokes C and D. **(b)** The view of the stack by each of the five functions. In a serial execution, only one view is active at any given time. In a parallel execution, however, if some of the invocations are spawns, then multiple views may be active simultaneously.

It turns out that there are inherent tradeoffs between space usage, time bound, and interoperability with serial binaries [33]. Known mechanisms that allows for such space efficiency (e.g., "heap-based" cactus stack [19, 20]) do not allow for interoperability with serial binaries. Cilk Plus's runtime system utilizes a strategy that allows for an efficient time bound, but exchanges a somewhat worse space bound for interoperability with serial binaries, resulting in a bound of PS_1D [33], where D is the maximum number of parallel functions nested on the stack during serial execution. Unfortunately, for a program with futures, neither space bound holds, because work stealing no longer maintains the necessary "busy-leaves" property [10].

Maintaining Cactus Stack for Futures

Cilk-F utilizes the same strategy that Cilk Plus uses, called ***stack stitching***, where a worker grabs a new stack when necessary, and the cactus stack is maintained by stitching together linear stacks. With the work-first policy and fork-join parallelism as described in Section 2, a ***linkage point***, the stitching of two linear stacks, is created only upon a successful steal, since that's when actual parallelism is realized. In Figure 1 for instance, if a worker $w1$ executes A, which spawns B, a different worker $w2$ that steals the continuation of A will grab a new stack to invoke E and remembers that its parent is A. With this strategy, the runtime maintains the invariant that only a parallel function with stealable continuations (i.e, spawn) can form linkage points and needs to know about the cactus stack; a serial function can operate with the old calling convention assuming a linear stack.

The use of futures raises unique issues with the maintenance of cactus stacks. In this example, say A is an ordinary C function which *calls* a parallel function B, and B subsequently spawns C via `fut-create`. Since the termination of a future task C is not confined within its enclosing lexical scope, without a corresponding `get`, B can return (and so can any of its ancestors) without waiting for C to complete.

The first issue that arises is that C may access resources allocated on its ancestor's stack. We won't be too concerned with this issue, since in this case, it's only appropriate that the get on C is invoked within the subcomputation of the said ancestor; otherwise it's a programming error.[6]

Second, and more importantly, the continuation of B can be stolen and resumed by a different worker, and B can subsequently return while C executes on the original worker. In this case, however, all three functions are still on the same linear stack used by the original worker, and resumption of A can clobber the execution of C.

This is a function of both the work-first policy, which allows a future task to be allocated on the same stack, and the fact that that a future allows its ancestor, which may be an ordinary C function, to resume execution while the outstanding future task uses the same stack. To get around this, whenever a worker executes a future, we opt to spawn the future task on newly allocated linear stack and let the parent (continuation of `fut-create`) reuse the stack.

Cilk-F's Cactus Space Bound

Let D be the maximum number of nested parallel functions (functions containing parallel keywords) on the stack during serial execution. Cilk-F provides the stack space bounds of $O(PT_\infty S_1 D)$ for structured futures and $O(m_k S_1 D)$ for general futures. Effectively, the maximum number of suspended deques at any given time (Lemmas 4.12 and 4.13 in Section 4) allows us to bound the stack space. Due to the proactive nature of ProWS, a deque contains only executions that correspond to a particular path in the invocation tree. For a given path, there can be at most D linkage points, each keeping S_1 stack space alive. Thus, the bound follows from the bounds on the maximum number of deques,

If we had taken the heap-based cactus stack approach, we would obtain a space bound of $O(PT_\infty S_1)$ for structured futures and $O(m_k S_1)$ for general futures, following similar arguments: the stack space taken up by the activation frames fallen on a single path in the invocation tree is at most S_1, getting rid of the D factor. However, the system would not allow for interoperability with serial binaries.

Lee et al. [33] and Yang and Mellor-Crummey [51] proposed virtual memory mechanisms to allow for a work-stealing scheduler to maintain the cactus stack abstraction in a way that bounds physical space and provides interoperability. Although their mechanisms are designed for fork-join parallelism, they can be adapted to programs with futures. Using Lee et al.'s mechanism, a space bound of $O(PT_\infty(S_1 + D))$ for structured futures and $O(m_k(S_1 + D))$ for general futures can be achieved, though the proposed VM mechanism requires special Operating System support. Mechanisms proposed by Yang and Mellor-Crummey can provide similar bounds on physical space consumption though not virtual address space. Both mechanisms require invoking system calls for each successful steal, suspension, and mugging.

[6]The future and async constructs in C++ follow similar philosophy [49].

6 Empirical Evaluation of Cilk-F

This section empirically evaluates Cilk-F, a prototype implementation of ProWS, across nine benchmarks. The use of futures provides additional flexibility in expressing parallelism, but the flexibility comes with some additional cost compared to fork-join parallelism, namely a higher number of deviations and higher cost in maintaining the cactus stack abstraction. To evaluate the overhead in using futures, we compare the execution times of benchmarks running on Cilk-F (future implementations) against Cilk Plus (fork-join implementations) and analyze its overhead. We show that ProWS can be implemented efficiently. Despite the slightly worse theoretical execution time bound compared to classic work stealing that Cilk Plus implements, Cilk-F performs comparably to Cilk Plus.

Experimental setup. We ran our experiments on an Intel Xeon E5-2665 with 16 2.40-GHz cores on two sockets. Each core has a 32-KByte L1 data cache, 32-KByte L1 instruction cache, and a 256-KByte L2 cache. There is a total of 64 GB of memory, and each socket shares a 20-MByte L3-cache. All benchmarks are compiled with LLVM/Clang 3.4.1 with `-O3 -flto`[7] running on Linux kernel version 4.4. Each data point is the average of 10 runs.

Benchmarks. We evaluate Cilk-F using nine benchmarks implemented with both fork-join parallelism and futures that fall into three different categories: A) benchmarks where futures do not provide additional benefits and both versions effectively do the same thing; B) benchmarks that can be implemented using both, but the versions with futures expose slightly higher parallelism; C) benchmarks that can only be implemented with futures.

Benchmarks under category A) include: mm (matrix multiplication, input 4k-by-4k), smm (the Strassen matrix multiplication algorithm, input 4k-by-4k), and sort (parallel mergesort, input 10^8). Benchmarks under category B) include: heartwall (adapted from the Rodinia benchmark suite [16] that tracks the movement of a mouse heart over a sequence of ultrasound images, input 104 frames), lcs (dynamic programming solving longest common subsequence, input $32k$ with base case size 512), sw (dynamic programming that implements Smith-Waterman for sequence alignment, input $2k$ with base case 32), bst (the pipelined tree merge using parallel futures from [8], input 8e6 and 4e6). The benchmark under category C) is ferret (adapted from the PARSEC benchmark suite [5] that implements a content-based similarity search on images with pipeline parallelism, with input size native. Even though ferret cannot be implemented using fork-join parallelism, it can be implemented as a pipeline program;

[7]With the exception of bst-fj and bst-gf, which were compiled with -O0 and without -flto due to a compiler bug triggered by the code.

bench	T_s	T_1	T_2	T_4	T_8	T_{16}
mm-fj	87.61	89.49	44.74 (1.96×)	22.35 (3.92×)	11.18 (7.84×)	5.59 (15.67×)
mm-sf		88.88	44.45 (1.97×)	22.22 (3.94×)	11.10 (7.89×)	5.56 (15.77×)
smm-fj	17.54	17.75	9.06 (1.94×)	4.68 (3.75×)	2.54 (6.92×)	1.38 (12.71×)
smm-sf		17.96	9.14 (1.92×)	4.73 (3.71×)	2.61 (6.71×)	1.42 (12.35×)
sort-fj	13.69	14.16	7.10 (1.93×)	3.58 (3.83×)	1.87 (7.31×)	1.04 (13.15×)
sort-sf		14.02	7.03 (1.95×)	3.55 (3.86×)	1.85 (7.39×)	1.05 (13.06×)
hw-fj	170.83	171.20	87.23 (1.96×)	45.04 (3.79×)	23.60 (7.24×)	13.57 (12.59×)
hw-sf		170.12	86.56 (1.97×)	44.77 (3.82×)	23.45 (7.28×)	13.47 (12.68×)
hw-gf		170.66	86.88 (1.97×)	44.58 (3.83×)	23.50 (7.27×)	13.35 (12.80×)
lcs-fj	8.80	9.06	4.63 (1.90×)	2.49 (3.53×)	1.60 (5.50×)	1.36 (6.47×)
lcs-sf		9.06	4.57 (1.92×)	2.42 (3.63×)	1.53 (5.76×)	1.42 (6.19×)
lcs-gf		8.84	4.54 (1.94×)	2.34 (3.75×)	1.31 (6.72×)	1.85 (4.75×)
sw-fj	14.51	12.58	6.43 (2.26×)	3.37 (4.31×)	1.83 (7.95×)	1.05 (13.78×)
sw-sf		14.47	7.27 (2.00×)	3.69 (3.93×)	1.89 (7.69×)	1.00 (14.53×)
sw-gf		13.64	6.93 (2.09×)	3.54 (4.10×)	1.82 (7.99×)	0.98 (14.74×)
bst-fj	3.31	3.41	2.42 (1.37×)	1.96 (1.69×)	1.77 (1.87×)	2.19 (1.51×)
bst-gf		3.42	2.43 (1.36×)	1.98 (1.67×)	1.79 (1.85×)	2.02 (1.63×)
fer-lp	191.54	192.72	97.62 (1.96×)	50.00 (3.83×)	26.05 (7.35×)	14.16 (13.53×)
fer-gf		190.66	96.44 (1.99×)	49.85 (3.84×)	25.43 (7.53×)	13.62 (14.07×)

Figure 2. The execution times of benchmarks in seconds, running with Cilk Plus (`fj` and `lp`) and Cilk-F (`sf` and `gf`). T_s shows the running time of the serial elision. T_p shows the running time on p processing cores, and the numbers in parenthesis are the speedups over T_s.

thus, we compare the future implementation running on Cilk-F with the pipelined version running on Cilk Plus's implementation of Piper [26], a provably efficient work-stealing scheduler that supports (linear) pipeline parallelism [34, 35]. The benchmarks are suffixed to indicate their implementation, with `fj` for fork-join parallelism, `lp` for (linear) pipeline parallelism, `sf` for structured futures, and `gf` for general futures. The serial elisions are obtained by removing parallel keywords from the `fj`/`lp` versions, but the serial elisions for the future implementations are comparable.

Performance of Cilk-F

As Figure 2 shows, the performance of Cilk-F is comparable to the performance of Cilk Plus in all our benchmarks.

For the category A benchmarks this is to be expected; the future versions are direct conversions from fork-join parallelism by replacing `spawn` with `fut-create` and `sync` with `get`. This structure ensures that when a `get` causes a deque to suspend, that deque is always empty. Since all suspended deques are empty, they never get added to the stealable set, and thus the Cilk-F implementations have the same expected execution time as Cilk Plus, $T_1/P + O(T_\infty)$ for these benchmarks.

The category B benchmarks also have comparable results between Cilk Plus and Cilk-F, even though Cilk-F has a harsher expected time bound of $O(T_1/P + T_\infty \lg P)$. We expect different versions of the `heartwall` benchmark to perform similarly since the use of futures only provide slightly higher parallelism by some constant amount (i.e., the same work and span asymptotically). On the other hand, the Cilk-F implementations of `lcs` and `sw` both demonstrate slightly better speedups than their Cilk Plus counterparts. The future versions of `lcs` and `sw` have smaller spans than the fork-join versions: for an input size N and base case B, the fork-join versions have a span of $O(N \cdot B + \frac{N}{B} \cdot \lg \frac{N}{B})$ whereas the future versions have a span of $O(NB)$.[8] The final benchmark in category B, `bst`, does not show noticeable improvement in the Cilk-F implementation compared to the Cilk Plus one. Even though the span is reduced from $O(\lg^2 N)$ to $O(\lg N)$ (with work $O(N)$), this is not a large enough improvement to be apparent in the data we collected, especially since in order to reduce the overhead of T_1 over T_s, we had to limit the parallel recursion depth (to depth 5).

In the case of the category C, `ferret`, one would expect the linear pipeline version to outperform the general future version since the expected execution time using Piper is $T_1/P + O(T_\infty)$ [34] which is asymptotically better than that of ProWS. In this case, unlike the fork-join case, Cilk-F would incur some number of suspensions with non-empty deques. However, `ferret-gf` turns out to perform slightly better than `ferret-lp` running on Piper, likely due to the fact that Piper automatically throttles (which we set to $8P$ for P workers), whereas Cilk-F does not. Thus, `ferret-gf` can take better advantage of the parallelism in the `ferret` computation at the risk of increased memory usage.

Overhead in Using Futures

To compare the overhead of creating futures implemented in Cilk-F, we use the microbenchmark `fib`, which calculates the n^{th} Fibonacci number by recursively spawning `fib(n-1)` and `fib(n-2)`. By design, `fib` does not coarsen the base case so that the majority of the execution time is due to `spawn` or `fut-create` overhead. We implemented three separate versions of `fib`: `fib-fj` (with `spawn`/`sync`), `fib-sf` (with `fut-create` and `get`), and `-stack`, which is identical to `fib-sf` but without the overhead of switching to a new stack on every `fut-create`. This "optimization" is not generally applicable due to the issue discussed in Section 5 but still correct in this particular case (due to where `get` is invoked); we simply use it to gauge how much overhead allocating a new stack for each `fut-create` incurs.

Compared to `fib-fj`, the overhead primarily comes from two sources: the stack-switch that occurs on a `fut-create`, and synchronization between the `get` and `put` operations. By comparing the numbers between `fib-sf` and `-stack`, we see a big drop in T_1, which also helps with T_p. By removing the synchronization overhead between `get` and `put`, we see T_1 drops further from 26.78 seconds to 21.06 seconds, which is comparable to that of `fib-fj`.[9] This shows that these are

[8] This difference in span decreases as B approaches $\sqrt{N}$.

[9] Though naturally things cannot run correctly in parallel if there is not synchronization between `get` and `put`.

bench	T_1	T_2	T_4	T_8	T_{16}
fib-fj	22.05	10.84 (2.04×)	5.43 (4.06×)	2.71 (8.13×)	1.35 (16.29×)
fib-sf	34.30	17.09 (2.01×)	8.77 (3.91×)	4.36 (7.86×)	2.29 (15.00×)
-stack	26.78	13.47 (1.99×)	6.99 (3.83×)	3.46 (7.75×)	1.74 (15.41×)

Figure 3. The execution times of different versions of fib, in seconds, running with Cilk Plus (`fj`) and Cilk-F (`sf` and `-stack`). The `-stack` row shows the running times of `fib-st` on Cilk-F but removes the stack-switch upon a `fut-create`. T_p shows the running time on p processing cores and the numbers in parenthesis are the scalability over T_1. The T_s (serial elision running time) for `fib` is 2.46.

bench	fj	sf	gf
hw	10286	13419	24904
lcs	3672	4231	8197
sw	4086	4086	12068

Figure 4. The number of deviations incurred on 16 processors. The data is the maximum out of 3 runs.

the two major sources of additional overhead when using Cilk-F futures compared to Cilk Plus `spawn/sync`.

To further investigate the overhead of Cilk-F futures compared to Cilk Plus `spawn/sync`, we instrumented Cilk-F to collect the number of deviations We chose benchmarks where all three versions are available to show the number of deviations; their code structure is similar between versions although the future versions do achieve slightly more parallelism. As Figure 4 shows, matching the theory, structured futures generate many fewer deviations than general futures, but fork-join parallelism generates fewer still.

7 Related Work

Work-stealing runtime for synchronization primitives. Futures have been incorporated into many parallel platforms (e.g., [13–15, 18, 21, 31, 37, 42, 45]). Many use parsimonious work stealing [14, 18, 21, 31, 42].

Other variants of work stealing have also been implemented to support synchronization primitives that can cause suspension, but none of them provide provably efficient scheduling bounds. In variants of X10 [15, 44] and Habanero dialects [13, 24], various synchronization primitives other than futures are provided that can cause the execution to block while the executing worker's deque is not empty. In the initial release of X10 [15], little support was provided — a blocking synchronization primitive blocks the executing worker, and to compensate, the runtime spawns a new worker thread to replace the blocked worker, effectively suspending the deque. Over time, the system could be oversubscribed. Tardieu et al. [44] subsequently developed a version of X10 with better support for suspension. In their system, if a worker is blocked the worker suspends the blocked task, but uses a centralized queue to allow resumptions of suspended tasks. A similar approach is taken by the initial release of Habanero Java [13]. In a later version, Imam and Sarkar [24] describe support for suspension in Habanero Java. In their system, suspended tasks are stored with the blocking synchronization primitives (similar to how we handle futures), but once the tasks get resumed, they all get pushed onto the ready deque of the worker who unblocks them.

Work-stealing analysis with multiple deques. Researchers have proposed provably efficient work stealing schedulers where the execution allows for suspensions [39, 46]. Muller and Acar [39] studies a work stealing scheduler that hides latency of I/O operations. When a worker encounters I/O, it may suspend the currently executing task. Their scheduler is parsimonious, but due to the possibility of suspension, there can be more than P number of deques in the system. Their scheduler provides a bound of $O(T_1/P + T_\infty U(1 + \lg U))$, where U is the maximum number of parallel I/O operations. Utterback et al. propose [46] a processor-oblivious record-replayer for fork-join parallel computations that utilize locks. During replay, if a lock-acquire is not "ready" to be replayed, the executing worker suspends its current deque and steals. Our time-bound analysis takes inspiration from theirs, but we need to additionally handle muggable deques. In their system, the number of suspended deques can be unbounded, and the scheduler provides the time bound of $O(T_1/P + T_\infty \lg\lg P)$. Instead of randomly choose a victim to deposit the suspended deques, they utilize the power-of-two choices, choosing two victims and deposit it with the one with the lighter load, thereby obtaining a slightly better bound ($\lg\lg P$ in front of the T_∞ term instead of $\lg P$). We cannot apply the same strategy, since the power-of-two choices does not seem to help with bounding the minimum load [50].

Finally, with respect to space bounds, beyond what's already stated in Section 5, Blelloch et. al proposed a ***Parallel Depth-First*** (***PDF*** for short) scheduler [6, 7] that is more space efficient than work stealing. Unfortunately, a PDF scheduler is challenging to implement efficiently in practice because it requires workers to synchronize with each other at a much higher frequency.

Acknowledgements

We thank our colleagues Kunal Agrawal and Brendan Juba for their helpful suggestions. We thank the reviewers for their valuable comments and feedback. This research was supported in part by National Science Foundation under grant number CCF-1150036, CCF-1527692, and CCF-1733873.

A Artifact Appendix

The Cilk-F implementation of ProWS is open source and available at https://github.com/wustl-pctg/ProWS or through Zenodo (DOI: 10.5281/zenodo.2227457). Instructions on building and running all relevant code are contained in the artifact's README.md file. Please send bug reports and any feedback to our github repository in order to help us continue to improve the project.

References

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. 2000. The Data Locality of Work Stealing. In *Proc. of the 12th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA 2000)*. 1–12.

[2] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 1998. Thread Scheduling for Multiprogrammed Multiprocessors. In *10th Annual ACM Symposium on Parallel Algorithms and Architectures*. 119–129.

[3] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 2001. Thread Scheduling for Multiprogrammed Multiprocessors. *Theory of Computing Systems* (2001), 115–144.

[4] Rajkishore Barik, Zoran Budimlić, Vincent Cavè, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Yonghong Yan, Yisheng Zhao, and Vivek Sarkar. 2009. The Habanero Multicore Software Research Project. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, Orlando, Florida, USA, 735–736.

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*. ACM, 72–81.

[6] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. 1995. Provably Efficient Scheduling for Languages with Fine-Grained Parallelism. In *7th Annual ACM Symposium on Parallel Algorithms and Architectures*. 1–12.

[7] Guy E. Blelloch, Phillip B. Gibbons, Yossi Matias, and Girija J. Narlikar. 1997. Space-Efficient Scheduling of Parallelism with Synchronization Variables. In *9th Annual ACM Symposium on Parallel Algorithms and Architectures*. 12–23.

[8] Guy E. Blelloch and Margaret Reid-Miller. 1997. Pipelining with futures. In *SPAA*. ACM, 249–259.

[9] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1996. Cilk: An Efficient Multithreaded Runtime System. *JPDC* 37, 1 (1996), 55–69.

[10] Robert D. Blumofe and Charles E. Leiserson. 1994. Scheduling Multithreaded Computations by Work Stealing. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*. 356–368.

[11] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *JACM* 46, 5 (1999), 720–748.

[12] F. Warren Burton and M. Ronan Sleep. 1981. Executing Functional Programs on a Virtual Tree of Processors. In *FPCA*. ACM, 187–194.

[13] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: the new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ '11)*. 51–61.

[14] Rohit Chandra, Anoop Gupta, and John L. Hennessy. 1994. COOL: An Object-Based Language for Parallel Programming. *IEEE Computer* 27, 8 (Aug. 1994), 13–26.

[15] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 519–538.

[16] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54.

[17] John S. Danaher, I-Ting Angelina Lee, and Charles E. Leiserson. 2008. Programming with exceptions in JCilk. *Science of Computer Programming* 63, 2 (Dec. 2008), 147–171.

[18] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. 2010. Implicitly Threaded Parallelism in Manticore. *Journal of Functional Programming* 20, 5-6 (Nov. 2010), 537–576. https://doi.org/10.1017/S0956796810000201

[19] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. ACM, 212–223.

[20] Seth Copen Goldstein, Klaus Erik Schauser, and David Culler. 1995. Enabling Primitives for Compiling Parallel Languages. In *Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*. Troy, New York.

[21] Robert H. Halstead, Jr. 1985. Multilisp: A Language for Concurrent Symbolic Computation. *ACM TOPLAS* 7, 4 (Oct. 1985), 501–538.

[22] E. A. Hauck and B. A. Dent. 1968. Burroughs' B6500/B7500 stack mechanism. *Proceedings of the AFIPS Spring Joint Computer Conference* (1968), 245–251.

[23] Maurice Herlihy and Zhiyu Liu. 2014. Well-structured Futures and Cache Locality. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. ACM, Orlando, Florida, USA, 155–166. https://doi.org/10.1145/2555243.2555257

[24] Shams Imam and Vivek Sarkar. 2014. Cooperative Scheduling of Parallel Tasks with General Synchronization Patterns. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*. Springer-Verlag New York, Inc., New York, NY, USA, 618–643. https://doi.org/10.1007/978-3-662-44202-9_25

[25] Intel 2013. Intel® Cilk™ Plus. https://www.cilkplus.org.

[26] Intel 2014. Piper: Experimental Language Support for Pipeline Parallelism in Intel® Cilk™ Plus. https://www.cilkplus.org/piper-experimental-language-support-pipeline-parallelism-intel-cilk-plus.

[27] Intel Corporation 2012. *Intel(R) Threading Building Blocks*. Intel Corporation. Available from http://software.intel.com/sites/products/documentation/doclib/tbb_sa/help/index.htm.

[28] Intel Corporation 2013. *Intel® Cilk™ Plus Language Extension Specification, Version 1.1*. Intel Corporation. Document 324396-002US. Available from http://cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_2.htm.

[29] ISOIEC14882 2012. ISO/IEC 14882:2011(E) Information technology — Programming Languages — C++. Third Edition, 2012-02-14.

[30] Brian W. Kernighan and Dennis M. Ritchie. 1988. *The C Programming Language* (second ed.). Prentice Hall, Inc.

[31] David A. Kranz, Robert H. Halstead, Jr., and Eric Mohr. 1989. Mul-T: A High-Performance Parallel Lisp. In *PLDI*. ACM, 81–90.

[32] Doug Lea. 2000. A Java fork/join framework. In *ACM 2000 Conference on Java Grande*. 36–43.

[33] I-Ting Angelina Lee, Silas Boyd-Wickizer, Zhiyi Huang, and Charles E. Leiserson. 2010. Using Memory Mapping to Support Cactus Stacks in Work-Stealing Runtime Systems. In *PACT*. ACM, 411–420.

[34] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Jim Sukha, and Zhunping Zhang. 2013. On-the-Fly Pipeline Parallelism. In *Proceedings of the 25th Annual ACM Symposium on Parallelism in Algorithms and Architectures*. 140–151.

[35] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Jim Sukha, and Zhunping Zhang. 2015. On-the-Fly Pipeline Parallelism. *ACM Transactions on Parallel Computing* 2, 3, Article 17 (Sept. 2015), 42 pages. https://doi.org/10.1145/2809808

[36] Charles E. Leiserson. 2010. The Cilk++ Concurrency Platform. *J. Supercomputing* 51, 3 (2010), 244–257.

[37] Li Lu, Weixing Ji, and Michael L. Scott. 2014. Dynamic Enforcement of Determinism in a Parallel Scripting Language. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, Edinburgh, United Kingdom, 519–529.

[38] Michael Mitzenmacher and Eli Upfal. 2017. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis* (2nd ed.). Cambridge University Press, New York, NY, USA.

[39] Stefan K. Muller and Umut A. Acar. 2016. Latency-Hiding Work Stealing: Scheduling Interacting Parallel Computations with Work Stealing.

In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '16)*. ACM, Pacific Grove, California, USA, 71–82.

[40] OpenMP 4.0 2013. *OpenMP Application Program Interface, Version 4.0.*

[41] Daniel Spoonhower, Guy E. Blelloch, Phillip B. Gibbons, and Robert Harper. 2009. Beyond Nested Parallelism: Tight Bounds on Work-stealing Overheads for Parallel Futures. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures (SPAA '09)*. ACM, Calgary, AB, Canada, 91–100. https://doi.org/10.1145/1583991.1584019

[42] Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. 2008. Space Profiling for Parallel Functional Programs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*. ACM, Victoria, BC, Canada, 253–264.

[43] Bjarne Stroustrup. 2000. *The C++ Programming Language* (third ed.). Addison-Wesley, Boston, MA.

[44] Olivier Tardieu, Haichuan Wang, and Haibo Lin. 2012. A Work-stealing Scheduler for X10's Task Parallelism with Suspension. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New Orleans, Louisiana, USA, 267–276.

[45] Sağnak Taşırlar and Vivek Sarkar. 2011. Data-Driven Tasks and Their Implementation. In *Proceedings of the 2011 International Conference on Parallel Processing (ICPP '11)*. IEEE Computer Society, Taipei City, Taiwan, 652–661.

[46] Robert Utterback, Kunal Agrawal, I-Ting Angelina Lee, and Milind Kulkarni. 2017. Processor-Oblivious Record and Replay. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. ACM, Austin, Texas, USA, 145–161. https://doi.org/10.1145/3018743.3018764

[47] Jacobo Valdes. 1978. *Parsing Flowcharts and Series-Parallel Graphs*. Ph.D. Dissertation. Stanford University. STAN-CS-78-682.

[48] Mark T. Vandevoorde and Eric S. Roberts. 1988. WorkCrews: An Abstraction for Controlling Parallelism. *International Journal of Parallel Programming* 17, 4 (1988), 347–366.

[49] Anthony Williams. 2012. *C++ COncurrency in Action*. Manning Publications Co.

[50] Weiyu Xu and A. Kevin Tang. 2011. A Generalized Coupon Collector Problem. *Journal of Applied Probability* 48, 4 (2011), 1081–1094. http://www.jstor.org/stable/23066444

[51] Chaoran Yang and John Mellor-Crummey. 2016. A Practical Solution to the Cactus Stack Problem. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '16)*. ACM, Pacific Grove, California, USA, 61–70.