Scheduling I/O Latency-Hiding Futures in Task-Parallel Platforms*

Kyle Singer† Kunal Agrawal† I-Ting Angelina Lee†

Abstract

Task parallelism research has traditionally focused on optimizing computation-intensive applications. Due to the proliferation of commodity parallel processors, there has been recent interest in supporting interactive applications. Such interactive applications frequently rely on I/O operations that require few processing cycles but may incur significant latency to complete. In order to increase performance, when a particular thread of control is blocked on an I/O operation, ideally we would like to hide this latency by using the processing resources to do other ready work instead of blocking or spin waiting on this I/O. There has been limited prior work on hiding this latency and only one result that provides a theoretical bound for interactive applications that use I/O operations.

In this work, we propose a task parallel platform that supports I/O operations using the futures abstraction and a corresponding scheduler that schedules the I/O operations while hiding their latency. We provide a theoretical analysis of our scheduling algorithm that shows our algorithm provides better execution time guarantees than prior work. We also implemented the algorithm in a practically efficient prototype library that runs on top of the Cilk-F runtime, a runtime system that supports futures within the context of the Cilk Plus language, and performed experiments that demonstrate the efficiency of our implementation.

1 Introduction

With the prevalence of multicore processors, task parallelism has become increasingly popular. With task parallelism, the programmer expresses the *logical* parallelism of the computation and lets an underlying runtime system handle the necessary load balancing and synchronizations. Modern parallel platforms that implement task parallelism include but are not limited to OpenMP [35], Intel TBB [25], various dialects of Cilk [26, 30, 31, 18] and Habanero [10, 15], X10 [17], and the Java Fork/Join framework [29].

Even though task parallelism simplifies the job of programming multicore machines, modern parallel applications such as desktop software or web services are not well supported by existing task-parallel platforms. Research on task-parallel platforms has traditionally focused on supporting applications from the domain of high-performance and scientific computing, which has very different execution characteristics from that of modern parallel software. In particular, a modern parallel application tends to incur frequent interactions with the external world, done in the form of input/output (I/O), such as obtaining user input through key strokes or mouse clicks, waiting for a data packet to arrive on a network connection, or writing output to a display terminal or network.

Existing task-parallel platforms typically implement work stealing to schedule parallel computations. In the absence of I/O operations, work stealing provides provable bounds on execution time [12, 13, 8, 9], good space usage [13], good cache locality [5, 4], and allows for an efficient implementation [21]. I/O operations are typically done via low-level system libraries (e.g., the GNU C library) or through system calls provided by the operating systems (OS). While one can directly invoke functions provided by these libraries within a task-parallel program, doing so has performance implications. In particular, the classic work-stealing scheduler does not understand the use of I/O operations nor does it account for it in the scheduling algorithm. When a ***worker thread*** — surrogate of a processing core managed by the scheduler — encounters an I/O operation, it can block for an extended period of time, leaving one of the physical cores underutilized while the worker waits for the I/O operation to complete. Thus, when a task-parallel computation includes I/O operations, a work-stealing scheduler can no longer provide the same provable guarantees.

Alternatively, one can utilize low-level system support to perform ***asynchronous*** (non-blocking) I/O operations. However, handling the asynchronous I/O completion can be complex — it involves utilizing mechanisms such as signal handling, event-driven style programming, or explicit polling, all of which require restructuring the program and can severely complicate the program control flow.

In this work, we study the problem of how to best

*Supported in part by NSF grants CCF-1527692, CCF-1733873, and CCF-1910568.

†Washington University in St. Louis

support interactions with the external world in the form of I/O operations in a task-parallel platform. Specifically, we design a work-stealing scheduler that *hides* the I/O latency — when a worker encounters blocking I/O, it simply suspends the current execution context and works elsewhere in the computation. When the I/O operation completes, some worker (not necessarily the worker that suspended it) picks up the suspended context and resumes it. To communicate to the underlying scheduler the use of latency-incurring I/O operations, the proposed task-parallel platform wraps the I/O operations with the future abstraction. As a result, both blocking and nonblocking I/O operations can be seamlessly integrated into the task-parallel programming model, and their uses are composable with other high-level parallel constructs supported by the platform. Finally, we show that the scheduler provides provably good performance bounds and empirically demonstrate the efficiency of our prototype implementation.

As far as we know, only a couple prior works study the problem of supporting I/O operations in task-parallel code. On the system side, Zakian et al. [44] extend Intel Cilk Plus [24] to provide support for a low-level library that allows a worker to suspend the current execution context upon encountering a blocking I/O operation and find something else to do. In this work, however, the proposed mechanism does not allow for provable bounds for both time and space usage due to the way workers handle active work items and blocked execution contexts.

On the scheduling algorithm front, only one prior result provides a provably efficient scheduling bound for task-parallel programs with I/O operations. Muller and Acar [34] present a cost model for reasoning about latency incurring operations (such as I/O) in task-parallel programs. In their work, given a computation with ***work*** T_1 — the total computation time on one core — and ***span***[1] T_∞— the execution time of the computation on infinitely many cores — the scheduler executes the computation in expected time $O(T_1/P + T_\infty U(1 + \lg U))$, where the U is the maximum number of latency incurring operations that are logically in parallel. Their bound is ***latency-hiding*** in that the latencies of I/O operations only appear in the span term and not the work term. If no latency-incurring operations are used, their bound is asymptotically equal to the standard work stealing bound of $O(T_1/P + T_\infty)$.

In this work, we improve the latency-hiding bound by extending a scheduling algorithm based on ProWS [36], a recently developed work-stealing scheduler that efficiently supports futures. We implement I/O operations seamlessly within task-parallel code using futures while getting nearly asymptotically optimal completion time. In particular, we were able to prove that our latency-hiding scheduler provides an execution time bound of $O(T_1/P + T_\infty \lg P)$ in expectation; this bound is independent of the number of I/O operations in the system. Compared to the standard work-stealing bound, it just has an additional term of $\lg P$ on the span term. This implies that while the standard work-stealing scheduler provides linear speedup when $T_1/T_\infty = \Omega(P)$, our scheduler provides linear speedup when $T_1/T_\infty = \Omega(P \lg P)$. ProWS has the same bound, but its original analysis does not directly apply here due to the use of latency-incurring I/O operations. We extend ProWS's analysis and performance bounds to allow for latency-incurring I/O operations using futures.

The high-level intuition on why we provide a better bound compared to prior work [34] is as follows. The work-stealing algorithm by Muller and Acar is ***parsimonious*** — a worker never steals unless it runs out of work to do. In contrast, our work-stealing algorithm based on ProWS is ***proactive*** — whenever a worker encounters a blocking I/O operation, it suspends the execution context and finds something else to do by stealing. This behavior may seem counter-intuitive since it potentially increases the number of steal attempts. It turns out, however, by carefully managing deques one can sometimes amortize the steal cost against the work term, thereby obtaining a better bound.

More importantly, the use of proactive work stealing also provides a better bound on the number of ***deviations*** [37], defined as execution points at which the parallel execution of a program differs from its sequential execution. As articulated by prior literature [4, 37], the number of deviations provides a good metric for evaluating practical performance because it is highly correlated to scheduling overheads and cache misses during parallel executions. For fork-join parallelism, one can relate the number of deviations to the number of steal attempts [4]. This relationship does not hold with parsimonious work-stealing, however, if the program uses unstructured blocking operations like futures [37], making it difficult to bound the number of deviations. The use of proactive work stealing (as in our algorithm and as in ProWS) allows one to again bound deviations using the number of steals, thereby allowing for a better bound on deviations. With proactive work stealing, ProWS and our algorithm by extension, guarantee that the expected number of deviations is $O((P \lg P + m_k)T_\infty)$ where m_k is the total number of futures that are logically in parallel.

Our prototype system, Cilk-L, extends Cilk-F [36], an extension of Intel Cilk Plus [24] that supports futures

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

and implements ProWS. Cilk-L defines a special type of future, called an ***IO future***, which utilizes the parallelism abstraction provided by futures to schedule I/O operations in a latency-hiding manner that is composable with the rest of the parallel constructs supported (i.e., `spawn`, `sync`, `fut-create`, and `get`). When a worker invokes an I/O operation using an IO future, a handle is returned, and the I/O operation can be done either ***synchronously*** by calling `get` on the handle immediately, or ***asynchronously***, calling `get` at a later time when the result is needed in order for the control to proceed.

We empirically evaluated Cilk-L with microbenchmarks that interleave compute-intensive kernels with I/O operations. The empirical results indicate that Cilk-L is effective at latency hiding. When we compare the execution times of Cilk-L with the "idealized" execution times (where I/O operations do not incur latency), we find that Cilk-L incurs little overhead, indicating that the I/O latencies are mostly hidden and occur in the background. In order to support I/O futures, Cilk-L necessarily needs to incorporate additional system support for scheduling I/O operations asynchronously. We also provide a detailed breakdown of overhead.

Summary of contributions

- We have developed Cilk-L, a task-parallel platform that incorporates support for scheduling I/O operations in a latency-hiding way. By utilizing the abstraction of futures, one can perform asynchronous I/O operations in task-parallel code in a way that is composable with other parallel constructs (Section 3).
- We extend the scheduling algorithm of Cilk-F to incorporate the latency-hiding cost model, and show that with I/O latency the algorithm can schedule the computation in time $O(T_1/P + T_\infty \lg P)$ on P cores, independent of the number of I/O operations active in parallel. This bound is an improvement over the prior state-of-the-art by Muller and Acar. Since $\max\{T_1/P, T_\infty\}$ is a lower bound on the execution of this program on P processors, this bound is nearly asymptotically optimal except for the $\lg P$ overhead on the span. Moreover, our algorithm provides bounds on stack space and deviations, whereas the algorithm by Muller and Acar does not (Section 4).
- We empirically evaluated Cilk-L using microbenchmarks. The empirical results indicate that Cilk-L hides I/O latency effectively and incurs little scheduling overhead in doing so (Section 5).

2 Preliminaries

This section provides the necessary background. We first discuss the syntax and semantics for the parallel control constructs supported by Cilk-F [36] and how one can represent a computation expressed with these keywords abstractly as a DAG. We then discuss how a *parsimonious* work stealing runtime schedules the computation assuming no latency-incurring operations are present.

Parallel control constructs: Cilk-F, and by extension Cilk-L, support a small set of parallel control constructs: `spawn`, `sync`, `fut-create`, and `get`.[2] These keywords operate at the level of function calls. When a function F ***spawns*** off a function G by prefixing the call with the `spawn` keyword, G may execute in parallel with the continuation of F (the statements after the `spawn`). The keyword `sync` is the counterpart of `spawn`; it indicates that control cannot pass beyond the `sync` statement until all previously spawned children have returned. In Cilk-F, there is an implicit `sync` at the end of every function, ensuring that all children spawned via `spawn` return before this function returns.

The keyword `fut-create` works in a similar fashion as `spawn`. When a function F spawns off a function G by prefixing the call with the `fut-create` call, G may execute in parallel with the continuation of F. Unlike `spawn`, however, the execution of a `sync` has no effect on `fut-create`. The control *can* pass beyond `sync` even if a function previously spawned off via `fut-create` has not returned. Moreover, a `fut-create` returns a ***handle*** h immediately, which is an object that the execution of G is associated with. The handle can later be used to ensure termination of G and retrieve its result. In particular, when G finishes execution, the last instruction is implicitly a `put` call which puts the result of G into h and marks the future as ready. By invoking `get` on the handle, the control cannot pass beyond the `get` until the execution of G terminates and the future is marked as ready.

Execution DAG: Parallel computations generated by programs written with these primitives can be represented using a directed acyclic graph (DAG). Vertices of the DAG represent a unit time computation task[3] and edges represent dependences between nodes. We make the standard assumptions: there is a single root node and the out-degree is at most 2.

We classify nodes into a few different categories. Regular nodes are simply computation nodes. A ***spawn node*** executes a `spawn` and has two children — the left child is the first node of the spawned function and the right child is the continuation node. A ***join***

[2]The keyword `cilk_for` also exists to indicate parallel loops, but it is just a syntactic sugar that translates to binary spawning of iteration space using `spawn` and `sync`.

[3]This is an assumption of convenience — longer operations can be represented as a chain of unit time operations.

node represents the continuation after a `sync` call and has multiple parents — the `sync` node itself and the last nodes of all the functions being synced. The `fut-create` keyword behaves similarly to `spawn` and generates a ***future create node*** that has two children: the left child is the first node of the future task and right is the continuation. A ***future join node*** is the node immediately after the invocation of `get` and has two parents — the `get` node (called the ***local parent***) and the ***future put node*** — the last node of the corresponding future task that puts the result of the future in the future handle.

We say that a node is ***ready*** or ***enabled*** if all its predecessors have executed. The ***work*** of the computation DAG is the total number of nodes in the DAG and is represented by T_1 — it is the total time to execute the DAG on 1 processor. The span of the weighted DAG is the longest path in the DAG and is represented by T_∞.

Parsimonious work stealing: Parsimonious work-stealing works by doing local work first. In computations with no latency-incurring operations, each worker maintains a single ***double ended-queue*** (or ***deque***) of ready nodes. For the most part, a worker operates on its deque. In particular, when a worker finishes executing a node, it may enable 0, 1 or 2 of its children. If it enables one child, the worker next executes the child. If it enables two children, it puts the right child on the deque and executes the left child. If it enables no children, it pops the bottom node from its deque and executes that node. Only when a worker runs out of work (its deque becomes empty), does it turn into a ***thief***. At this time, it randomly chooses a ***victim*** to steal work from. Upon a steal, the thief steals the ready node from the top of the victim's deque and executes it. If the victim deque has no ready nodes, then the worker tries another random steal.

3 The System Implementation

This section describes Cilk-L, a prototype system that extends Cilk-F [36] to incorporate support for performing I/O operations with latency hiding. The I/O support in Cilk-L consists of two main components: the ***IO futures*** library and runtime support for doing asynchronous I/O operations. We first discuss the programmer API for using IO futures, its implementation, and then the runtime support. Since I/O operations are typically supported via low-level system libraries and by the underlying operating system, currently Cilk-L only targets Linux platforms and utilizes various I/O related facilities from Linux.

3.1 The IO Futures Library

We use an example to illustrate the programming API provided by the library. Figure 1 shows the distributed map-and-reduce example. In this code, the function `distMapReduce` takes in five parameters: f, g, id, lo, and hi. The computation works as follows. The code obtains a set of input values from $n = hi - lo$ different network connections. The call to `openConnection` in line 6 abstracts away the sequence of steps to open a network connection, which returns a file descriptor representing the network connection once it is open. For each value x in the set, the code applies the map function $f(x)$ and then combines the resulting values from $f(x)$ using a binary reduction operation, g.

The IO futures library exposes one data type to the programmer, the handle for IO futures `io_future`, and two I/O functions, `cilk_read` and `cilk_write`, for reading from and writing to a file indicated by the file descriptor (i.e., the first argument, `fd`). In Linux, all I/O devices are presented as files, including network connections, which allows for a uniform interface for performing I/O operations [14, Chp. 10]. That means `cilk_read` and `cilk_write` work with any I/O device that can be represented as a file. The `cilk_read` and `cilk_write` functions are analogous to the Unix `read` and `write` system calls, except that they are ***asynchronous***, i.e., non-blocking. Both functions return an `io_future` handle representing the ongoing I/O operation, but the function itself does not block — the I/O is initiated and linked to the `io_future` handle, and the handle is immediately returned.

```
1  Function distMapReduce($f$, $g$, $id$, $lo$, $hi$)
2      $n \leftarrow hi - lo$;
3      if $n = 0$ then return $id$; //return identity.
4      else if $n = 1$ then
5          char $buf$[NBYTES] ; //buffer for input data.
6          $fd \leftarrow$ openConnection($lo$) //open network
             connection.
7          io_future $fut \leftarrow$ cilk_read($fd$, $buf$, NBYTES)
8          get($fut$);
9          return $f(buf)$;
10     end
11     else
12         $mid \leftarrow (lo + hi)/2$;
13         $r1 \leftarrow$ spawn distMapReduce($f$, $g$, $id$, $lo$, $mid$);
14         $r2 \leftarrow$ distMapReduce($f$, $g$, $id$, $mid$, $hi$);
15         sync;
16         return $g(r1, r2)$
17     end
18 end
```

Figure 1: Distributed map and reduce example.

A call to `cilk_read` or `cilk_write` first creates an `io_future` to represent the I/O request. The corresponding data required to carry out the I/O request (such as the file descriptor fd and the buffer buf to store input) is bundled up with the `io_future`. This data bundle is inserted into a ***communication queue*** to be processed by the runtime. The `io_future` is then returned to the caller. If the user needs the result from the future or wants to ensure that it has completed, they can perform a `get` on this `io_future` handle. The continuation of `get` (the future join node) cannot execute until the I/O operation has completed.

3.2 Runtime Support to Hide I/O Latencies

At runtime startup, normally a work-stealing runtime creates P persistent threads as P workers, one per processing core. In Cilk-L, $2P$ persistent threads are created — for every worker a corresponding ***I/O thread*** is created, and this persistent thread is pinned to the same core as its worker.[4] The I/O thread is only used to process I/O requests (via the IO futures library) generated by the worker's execution of user code. Thus, in the library implementation described above, the communication queue is implemented as a lock-free single-producer/single-consumer queue and used as a means for the worker thread to communicate I/O requests to its I/O thread.

When an I/O thread runs, it dequeues items from the communication queue and attempts to perform an I/O operation as soon as it is received. If an I/O operation cannot be completed immediately (e.g., the next packet has not yet arrived on the network socket), then the I/O thread puts the request aside and processes it later when the I/O device becomes ready (e.g., it has more input to be consumed).

In order to describe how the actual mechanism works, we need to briefly discuss how I/O operations work on Linux. As mentioned earlier, any I/O device on Linux (e.g., network sockets, mice, and keyboards) can be represented as a file descriptor. Obtaining input (read) from a file descriptor is effectively copying data from the corresponding device into memory (e.g., the buf in the example). If the device is not ready to be read (e.g., the next packet has not arrived on the network channel yet), the system call `read` will block. One could mark the file descriptor with the correct flag such that the system call would simply return instead of blocking, with a return value indicating input not ready. However, in this case, we must periodically make the system call again to know when the device becomes ready.

One possibility is to periodically wake up the I/O thread and have it poll the device via non-blocking `read`. This scheme is not ideal, as a system call can be expensive. Moreover, if the device is not ready, checking would simply cause the I/O thread to take up processor cycles that could be better used by its worker working on the actual computation. Thus, we would like to avoid the periodic wake up and the unnecessary system calls. Ideally, we would like the I/O thread to simply sleep and not use any processor cycles *unless* one of the following conditions happen: (a) one of the I/O devices with pending operations becomes ready; or (b) its worker inserts a new I/O request into the communication queue.

To achieve part (a), we use the Linux `epoll` [1] facility which allows the I/O thread to monitor a set of file descriptors (an `epoll` set). Adding a file descriptor to be monitored takes $O(\lg n)$ time, where n is the number of file descriptors currently in the `epoll` set. The I/O thread can go to sleep by calling `epoll_wait`, and it will be woken up when one of the I/O devices corresponding to one of the monitored file descriptors becomes ready. Determining which file descriptors have become ready takes $O(1)$ time — adding a file descriptor to the `epoll` set registers a callback with the file's underlying system driver; this callback will move the file into a ready list and wake the monitoring thread when I/O on that device becomes possible. Once the I/O thread is woken up, it can query `epoll` to obtain the list of ready file descriptors, which allows the I/O thread to determine which pending I/O operations can continue. In summary, each I/O thread maintains its own `epoll` set. When an I/O thread receives an I/O request but the corresponding file descriptor is not ready, the I/O thread adds the file descriptor to its `epoll` set to be monitored. Once an I/O thread has processed all I/O requests in the communication queue, it goes to sleep via `epoll_wait`. Doing so achieves part (a).

One last piece of the puzzle is how to avoid having the I/O thread check the communication queue periodically and yet still allow submitted I/O requests to be processed quickly whenever they are received. We solve this by using an event wait/notify mechanism called `eventfd` provided by Linux [2]. The `eventfd` mechanism is used to create a file descriptor that can be read by an I/O thread and written to by its worker. This file descriptor can be opened with semaphore-like semantics, in which writes will increment a backing counter and reads will decrement the same counter. When used with `epoll`, a write to an `eventfd` file descriptor will cause the I/O thread to wake up whenever the backing counter is incremented from 0 to 1. By writing to an `eventfd` file descriptor associated with a communication queue whenever an I/O operation is enqueued, and

[4]If the hardware has hyperthreading enabled, Cilk-L pins them to separate hardware threads (hyperthreads) associated with the same physical core.

by symmetrically reading from the same file descriptor whenever an operation is dequeued, `epoll` can also be used to monitor the state of the communication queue. Thus, we use this combination of `eventfd` and `epoll` to achieve part (b).

By combining these mechanisms, we achieve the effect that an I/O thread takes up processor cycles only when either there is a new I/O request from the worker or when one of the previously dequeued (and unprocessed) I/O operations can be performed. When an I/O thread completes an I/O operation, it performs a `put` on the corresponding `io_future` handle. From its worker's perspective, a call to `get` can cause the current execution to suspend, but the worker will just go find something else to do. Cilk-L schedules the execution of the IO futures in the same manner Cilk-F schedules ordinary futures, which we briefly review in Section 4.

4 Algorithm and Analysis

This section describes how to represent a program with I/O operations abstractly, the high level scheduling algorithm, and the runtime analysis. For scheduling, we will use ProWS, the proactive work-stealing algorithm described by Singer et al. [36]. The algorithm by Singer et al. schedules programs with futures in a time and space efficient manner, and we will briefly describe the algorithm here for completeness. However, the analysis in [36] handles futures but not I/O operations. Here we will show how that analysis can be extended to appropriately account for I/O latencies.

4.1 Execution DAG

We extend the model from Section 2 and add weighted edges in a manner similar to [34]. In our model, I/O operations are performed within future tasks. The invocation of an I/O function (`cilk_read` and `cilk_write`) creates an `io_future`, sets up the necessary data for the I/O request, inserts the request into the communication queue, and returns (discussed in Section 3). We will call the last node of this future task before it returns the ***I/O setup node***. However, unlike in non-I/O future tasks, this future itself is not ready. The future is ready when the I/O thread executes `put` upon the I/O completion — we will call the put node of an I/O future an ***I/O put*** node. We will have a ***heavy*** edge between the I/O setup node and the corresponding I/O put node — the weight on this edge represents the amount of time elapsed between when the I/O function returns and when the I/O completes (including the time that the I/O thread takes to handle the I/O request). All other edges are ***light*** with weight of one.

We can define work and span. The work is unchanged, i.e., the total number of nodes in the DAG. Therefore, it is unaffected by the latencies on the edges. The span of the weighted DAG is the longest weighted path in the DAG and is the only parameter affected by the latencies.

Again a node is ***ready*** if all its predecessors have executed, except for the I/O put node. An I/O put node is ***suspended*** once its predecessor (the corresponding I/O setup node) finishes executing. If ℓ is the weight of the incoming edge to the put node, it remains suspended for ℓ time steps. After these ℓ time steps, it is considered to have finished executing since the I/O thread will write the result into the future handle after these ℓ time steps. This definition of suspension of a put node is simply for the ease of analysis and has no impact on the scheduler since the put node is executed by an I/O thread and not by the worker thread.

4.2 Proactive Work Stealing

We use ProWS, the proactive work stealing scheduler by Singer et al. [36], unchanged. The main difference between proactive and parsimonious work stealing is the handling of a blocked future get. In parsimonious work stealing, when a worker's current node executes a `get` and the future is not ready, the subsequent future join node is not enabled. Therefore, the current node enables 0 children and (as described in Section 2) the worker pops the next node from the bottom of the deque and continues working. The algorithm by Muller and Acar [34] is a variant of this — when a worker blocks on an I/O operation, it pops the next node off its deque and keeps working on it.

ProWS behaves differently on executing a `get` where the future handle h is not ready.[5] instead of popping the next node from its active deque d, the worker work steals. In particular, the worker (1) marks the current deque ***suspended***; (2) it randomly picks another worker to donate this suspended deque to; and (3) allocates a new active deque d' for itself and randomly work steals. When the handle h becomes ready (the future finishes), then the corresponding put node marks the deque d ***resumable*** and pushes the future join node to the bottom of d.

Therefore in ProWS, each worker p has potentially many deques. One of these is ***active*** — this is the deque the worker is currently working on. In addition, it many have many suspended and resumable deques — collectively, the suspended and resumable deques are called the worker p's ***inactive*** deques. In addition, any

[5]There are other circumstances where the execution of the node enables no other nodes, such as when a worker returns from a spawned or future function — in all these circumstances proactive work stealing behaves as the parsimonious one and pops the bottom node from its deque.

suspended deques that have no ready nodes are ***unstealable***; all other deques are ***stealable***. The reason for this distinction is that unstealable deques have no ready nodes, so stealing from them is a waste of time. Note that any resumable deques with no ready nodes are simply de-allocated. However, a suspended deque d with no ready nodes cannot be deallocated for the following reason. Deque d is suspended since some `get` executed, but the corresponding future has not completed. When this future completes, the corresponding put node will enable the future join node and push it at the bottom of d and mark it resumable. Therefore, if we deallocate it, we would not have a targeted place to push this future join node.

A steal attempt also works slightly differently compared to traditional work stealing. When work stealing, a thief first picks a random victim and then picks a random stealable deque to steal from among the deques that the victim has. If the target deque is suspended, then the worker simply steals the top node from the deque. If the deque has no more ready nodes, then this deque is marked unstealable. There are additional details on how to handle resumable deques in order to get the correct bounds on running time and deviations — however, these details do not change in our analysis and we refer the reader to [36] for those details.

The important bits from the perspective of our understanding are the following: (1) Every worker has potentially many deques: one deque is active, and there are potentially many inactive deques (either suspended or resumable and some of the suspended deques may be unstealable); and (2) due to random throws when the deques are suspended, all workers have approximately equal number of deques. We will use these two facts in the analysis.

4.3 Analysis

The analysis of ProWS with I/O operations is, to a large extent, an extension of the analysis by Singer et al. [36] (henceforth, we will call them ***SXL***) with proper accounting for latency edges. Muller and Acar do account for latency on edges, but do not use futures for I/O operations, use parsimonious work stealing, and do not rebalance deques between workers. Therefore, the running time on P processors is $O(T_1/P + T_\infty U(1 + \lg U))$, where U is the ***maximum suspension width*** — the number of I/O operations that can be pending at the same time in the DAG. There is no bound on the number of deviations. The way they handle the potential function in order to hide the latency is a little different from our method.

Here, we are using ProWS and want to get a running time bound of $O(T_1/P + T_\infty \lg P))$ and the deviation bound of $O((P \lg P + m_k)T_\infty)$ where m_k is the total number of futures logically in parallel. For the special case where all futures are I/O futures, $m_k = U$.[6] The analysis of SXL doesn't work out of the box, however, since it does not consider the latency on heavy edges. Therefore, here, we will rely on the lemmas proved in that paper, but modify the potential function in order to handle the heavy edges appropriately.

In general, in work stealing, a worker is always either working or stealing. The main point of the analysis is to bound the total number of steal attempts, say by X. Since the total work is T_1, the total running time is $(T_1 + X)/P$. In addition, bounding the total number of successful steals also gives us a bound on the total number of deviations (for proactive work stealing, though not for parsimonious work stealing).

ProWS potential function and analysis: SXL's analysis uses a potential function similar to the one used by Arora et al. [8] (henceforth called ***ABP***) to bound the number of steal attempts. The potential function there is based on the ***enabling tree*** — we say that u enables v if u is the last parent of v to execute and, in this case, we add an edge between u and v in the enabling tree. It turns out that, for technical reasons, we cannot use the enabling tree for proactive work stealing. Instead, we will use the DAG itself to decide the potential of the node.

The potential function is based on the depth of nodes in the DAG. The depth of the node u with one parent v is $d(u) = d(v) + 1$. The depth of a node with multiple parents is similar, except that we add 1 to the depth of the deepest parent. The weight of node u is $w(u) = T_\infty - d(u)$.

We say that a node u is the ***assigned*** node for deque d if d is the active deque for some worker p and p is currently executing u. The potential of a node u is defined as follows: $\Phi(u) = 3^{2w(u)-1}$ if u is assigned and $\Phi(u) = 3^{2w(u)}$ if u is ready. For technical reasons, we will say that the assigned node for deque d is at the bottom of deque d even though it cannot be stolen. The total potential of a deque d is the sum of the potential of all nodes on d including the assigned node if d is active. The total potential of the computation is the sum of the potentials of all the ready and assigned nodes on all the deques.

Some of the key results from ABP carry over with these changes in definitions.

Lemma 4.1. *The initial potential is $3^{2T_\infty - 1}$ and the*

[6]SXL provide separate deviation bounds for structured and general futures and they both carry over. If all our I/O operations are done using structured futures, then the bound is better than this — here we only provide the general bound.

final potential is 0. *In addition, the potential never increases.*

LEMMA 4.2. ***Top Heavy Deques*** *The top most node in the deque has a constant fraction of the total potential of the deque.*

The intuition is that the top of the deque contains the node that was pushed on the deque farthest in the past and, therefore, it is the shallowest node in the DAG. Since the potential decreases geometrically with the depth, this node contains most of the potential of the deque.

The following lemma is a straightforward generalization of Lemmas 7 and 8 in ABP [8]. The high-level intuition is that since the top node of each deque contains a constant fraction of its potential, if we steal and execute the top node from each deque with reasonable probability, the overall potential is likely to reduce by a constant fraction.

LEMMA 4.3. *Let Φ_i denote the potential on deques 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 of deques is at most* $\Phi(t)/4$ *with probability at least* $1/4$.

In ABP, since there are only P deques, one for each worker, this lemma shows that P random steal attempts reduce the potential by a constant factor with constant probability. However, in ProWS, there are potentially many deques. Therefore, we may need many more steal attempts to reduce the potential. In addition, it is difficult to design a way to steal from all deques with equal probability if the deques are distributed across many workers.

In ProWS, however, recall that when a deque is suspended, the worker picks a random worker and donates the deque to that worker. Therefore, even if one worker suspends many deques, it does not hold on to them — the deques are approximately evenly distributed among all workers. When a worker steals, it picks a random victim worker and then a random stealable deque from the victim. Therefore, each deque has an approximately equal chance of being a victim of a steal attempt. In particular, SXL show the following:

LEMMA 4.4. *Given P workers and D stealable deques in the system, each worker has at most* $D/P + O(\lg P)$ *stealable deques with probability at least* $1 - o(1)$.

Another insight SXL uses is that a steal attempt from a stealable deque is generally successful if it is not an active deque. Therefore, if there are many (more than $3P$) stealable deques in the system (and only P of them are active), then most processors have at least one stealable deque and most steal attempts are successful. These periods are called ***work-bounded phases*** and SXL argue that the total number of steal attempts in work-bounded phases can be bounded by $O(T_1)$ in expectation.

Therefore, we only need worry about decreasing the potential when there are not too many stealable deques — these times are called ***steal-bounded phases***. SXL use Lemma 4.4 to argue that, during a steal-bounded phase, each stealable deque has at least $c/P \lg P$ chance of being the victim of a steal attempt (for some constant c) because no worker has more than $O(\lg P)$ stealable deques. Therefore, using Lemma 4.3, the potential of deques reduces by a constant factor after $P \lg P$ steal attempts (since unstealable deques are empty and have no potential). Given that the initial potential is $3^{2T_\infty - 1}$, the expected number of steal attempts during steal bounded phases is $O(P \lg PT_\infty)$. Therefore, considering both work- and steal-bounded phases, the total number of steal attempts is $O(T_1 + P \lg PT_\infty)$. In addition, they also separately bound the expected number of successful steals in work bounded phases by $O(m_k T_\infty)$. This allows them to bound the deviations by $O((P \lg P + m_k)T_\infty)$.

Changes to potential and analysis to handle weighted edges: We want to show the same bounds when we use futures for I/O operations. The bounds on steals in work-bounded phases carry over unchanged. In particular, the expected number of steal attempts in work-bounded phases is still T_1 and the expected number of successful steals is still $m_k T_\infty$. However, for steal bounded phases, where we rely on potential to bound the number of steal attempts, the analysis that bounds the steals does not apply directly for somewhat technical reasons.

Consider the following scenario. Some worker p with active deque d executes a `get` on an I/O future handle f and blocks since the future is not ready. It suspends this deque and steals. At some point, the I/O thread completes the I/O operation corresponding to f, executes the put, enables the future join node (say u) for f, and puts it at the bottom of d. Note that this deque's potential now suddenly increases, and our analysis strongly depends on the potential never increasing.

This is not a problem for SXL for the following reason: If a particular future join node u is not ready, then some deque must have some ancestor of u on its deque (either as a ready or an assigned node). Therefore, u does not appear on d from nowhere — some ancestor executes, this ancestor's potential is larger than u's potential and therefore, even though u becomes ready, the overall potential of the computation does not increase. In our case, no ancestor of u is ready or assigned anywhere in the system since the reason u is not ready is due to the latency on an I/O edge. This is problematic since u being enabled increases the potential of the system.

To fix this problem, we have to give potential to put nodes for I/O futures (even though they are executed by I/O threads) and handle them in a special way. In particular, recall that the only heavy edges in our DAG are between I/O setup nodes and the corresponding I/O put nodes. For I/O put nodes, we will define two notions of depth: the initial depth $id(u)$ of an I/O put node with enabling parent v (which is always an I/O setup node) is $id(u) = d(v) + 1$. The depth $d(u)$ starts out as $id(u)$ and increases on every time step while the I/O operation is pending and this I/O put node is suspended. If the weight of the heavy edge (the latency of the corresponding I/O operation) between v and u is ℓ, then u is suspended for ℓ steps. Therefore, u's final depth is $fd(u) = d(v) + \ell$. When the I/O operation completes, this put node completes.

Now consider the child node j of the I/O put node u — j is always a future join node. When deciding the depth of j, we always use u's final depth $fd(u)$. That is, if x is j's other parent (the node generated by the `get` operation), then $d(j) = \max\{fd(u), d(x)\} + 1$.

The potential of a pending put node is defined just like other ready nodes. At any time, if the depth of the put node is $d(u)$, its weight is $w(u) = T_\infty - d(u)$ and potential is $3^{2w(u)}$. However, since the depth of the node changes over time, so does its weight and potential. The total potential of the computation is the sum of the potentials of all the ready and assigned nodes on all the deques as well as the potentials of all the put nodes (which are not on any deque).

We now get back the following lemma:

Lemma 4.5. *The potential never increases.*

Proof. We only need consider the case when a future join node v is enabled by an I/O put node u. By definition, u has lower depth and thus higher potential than v. v is only enabled once u finishes. Therefore, the potential of the system does not increase. □

However, adding these put nodes creates a problem. These put nodes are not on any deque; therefore, steal attempts do not reduce the potential associated with these nodes directly. We must also now argue that the potential of I/O put nodes decreases appropriately during steal-bounded phases. This is the reason why we designed the potential of these put nodes in the funny way where their potential starts out high and reduces on every time step.

Lemma 4.6. *During steal-bounded phases, if the total potential at time i is Φ_i (including potential of assigned, ready and suspended put nodes), then after $cP \lg P$ steal attempts, for some constant c, the potential is at most $\Phi(t)/4$ with probability at least $1/4$.*

Proof. SXL already argued that the potential of deques reduces appropriately. Therefore, we only need to consider the suspended I/O put nodes. Note that if the latency of a weighted edge is ℓ, then the corresponding put node u remains suspended for ℓ time steps. Its potential starts at $3^{2(T_\infty - id(u))}$ and decreases by a factor of 1/9 on every time step. When the I/O operation completes and the future is ready, the potential of u is $3^{2(T_\infty - fd(u))}$. Since it takes at least $c \lg P \geq 1$ time steps to do $cP \lg P$ steal attempts, the potential of this put node u reduces by a large fraction during this time. This is true for all put nodes, giving the desired result. □

This lemma allows us to bound the expected number of steal attempts (and therefore, expected number of successful steals) during steal bounded phases by $P \lg PT_\infty$ — the same result as SXL. Since the expected number of steal attempts and successful steals for work-bounded phases remains unchanged, we get the same time and deviation bounds as SXL.

Theorem 4.1. *The expected number of steal attempts is $O(T_1 + P \lg PT_\infty)$. Therefore, the expected running time is $O(T_1 + P \lg PT_\infty)$. In addition, the expected number of deviations is $O((P \lg P + m_k)T_\infty)$.*

5 Empirical Evaluation

This section empirically evaluates our prototype implementation of Cilk-L using a microbenchmark `map-reduce` that closely resembles the example shown in Section 3 Figure 1. We would like to answer the following three questions in the evaluation: (1) how much benefit can one obtain from latency hiding; (2) how well can Cilk-L hide latency compared to an ***idealized*** system that hides the latency entirely and incurs zero overhead for hiding latency; and (3) how much does each mechanism used in Cilk-L to hide latency contribute to its overhead. How we measure each is explained in its respective subsections. Overall, the empirical results indicate that one can obtain substantial performance benefit from latency hiding. Cilk-L hides latency well when the application has ample parallelism. As the theory predicts, when the application has insufficient parallelism and the weighted span term (i.e., $T_\infty \lg P$) dominates, the performance of Cilk-L can lag behind the idealized version. Finally, the empirical results show that Cilk-L is lightweight, incurring minimal system overhead compared to the idealized version.

Experimental setup: We ran our experiments on a machine with two Intel Xeon Gold 6148 processors, each with 20 2.40-GHz cores, with a total of 40 cores. Each core has a 32KB L1 data cache, 32KB L1 instruction cache, and a 1MB L2 cache. Hyperthreading is enabled. Dynamic frequency scaling is disabled. Both sockets have a 27.5MB shared L3 cache, and 768GB of main memory. Cilk-L and `map-reduce` are compiled with LLVM/Clang 3.4.1 with `-O3 -flto` running on Linux kernel version 4.15. Each data point is the average of 10 runs. All data points have standard deviation less than 5% except for a handful of data points, which we note later as we explain the data.

Benchmark: We use a microbenchmark with very similar code structure to the map and reduce example (`map-reduce`) described in Section 3 (Figure 1), which is also used by Muller and Acar [34]. To allow for comparison with the system by Muller and Acar [34], we have used the same computation kernels and parameters used in their experiments, unless noted otherwise. Like Muller and Acar, we emulate 5000 remote server connections with simulated delays. At line 6 in Figure 1, rather than opening a true network connection, we used a timed file descriptor which becomes ready for I/O when the timer expires.[7] We replace the parameter f with a parallel version of the naive recursive implementation of Fibonacci with a serial base case of 25, and used it to compute the 30th Fibonacci number (`fib`). In place of calling function g (line 16), we return the sum of `r1` and `r2` modulo 1000000000.

5.1 The Benefit of Latency Hiding

To evaluate the benefit of latency-hiding, we compare Cilk-L with the baseline system Cilk-F, a ProWS scheduler that provide the same provably efficient time and deviation bounds that supports futures but does not hide I/O latency (i.e., a worker encountering an I/O function blocks until the I/O operation completes).[8] Specifically, we compare to two different versions of Cilk-F: one uses the same number of workers (Cilk-F) and one uses twice as many workers so as to oversubscribe the system (Cilk-F (O)) and let the underlying OS perform scheduling to hide latency. The Cilk-L version of `map-reduce` uses IO futures to hide latencies whereas the two versions of Cilk-F execute the baseline code that simply uses a blocking `read`.[9] We have also compared Cilk-L with the state-of-the-art latency-hiding work stealing scheduler proposed by Muller and Acar [34], denoted as ParWS (parsimonious work stealing). However, note that the comparison to ParWS is not strictly an apples-to-apples comparison, as their implementation is done in a variant of Parallel ML that implements parsimonious work stealing [39] whereas our implementation is C/C++-based. Nonetheless, since their implementation is the only other existing task-parallel scheduler that supports latency-hiding I/O operations with provably efficient execution time bound, we thus include the comparison for completeness.

We ran `map-reduce` with simulated I/O latencies of 1, 50, and 100 milliseconds. Figure 2 shows the speedup of Cilk-L compared to the one-worker execution time of running the baseline version of `map-reduce` on Cilk-F. Unlike what was observed by Muller and Acar [34], we do see some advantage to using Cilk-L to hide I/O latency at 1 millisecond. When the latency is short, however, one could benefit from simply using the oversubscription strategy (i.e., Cilk-F(O)). The Cilk-F(O) outperforms Cilk-L and even the Ideal at 1-millisecond latency because it utilizes all available hyperthreads to do work. Its performance slows once the computation incurs cross-socket communication (i.e., $P > 20$). We did separate experiments and found that Cilk-L breaks even / outperforms the oversubscription strategy at a latency of 7 milliseconds or higher for all P tested using `map-reduce`.

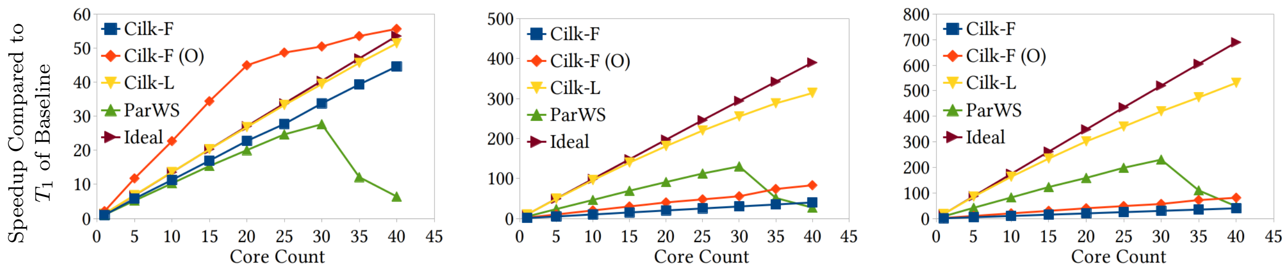
By the time latency hits 50 milliseconds, there is a more pronounced advantage to using the I/O functions provided by Cilk-L. With $P = 1$, we already see a speedup greater than $9\times$, which increases to over $313\times$ at $P = 40$. On the other hand, oversubscribing Cilk-F achieves about $2P\times$ speedup. This pattern continues as we increase the latency to 100 milliseconds, reaching speedups greater than $530\times$.

We also ran `map-reduce` using the Parallel ML implementation of Muller and Acar [34] and plotted the speedup relative to their Parallel ML baseline application. Although this is not quite an apples-to-apples comparison, as there is inherent overhead when using functional languages, the significant increase in speedup for Cilk-L compared to ParWS for $P \leq 20$ is still worth noting. For $P > 20$ it is an especially unfair comparison, as even the Parallel ML baseline application used by Muller and Acar [34] achieves poor speedup once processor socket communication overhead makes memory management very costly for Parallel ML. For ParWS, the standard deviations for $P \geq 35$ (for all latency configurations) are quite high, ranging from $16 - 30\%$.

[7]This functionality is provided by the Linux `timerfd` [3].

[8]Cilk-F extends Cilk Plus to support futures. Singer et al. [36] empirically evaluated Cilk-F and showed that it performs comparably to Cilk Plus.

[9]The code executed by Cilk-F and Cilk-F (O) contains only `spawn` and `sync` since `read` is used in place of IO futures.



(a) Latency = 1 millisecond (b) Latency = 50 milliseconds (c) Latency = 100 milliseconds

Figure 2: Speedups, compared to the one-core running time (T_1) of the respective baselines, of `map-reduce` running in ParWS, Cilk-F, Cilk-F (O), and Cilk-L with latencies of 1, 50, and 100 milliseconds. Ideal is the Cilk-F implementation that makes calls to `read` that returns immediately. For Ideal, Cilk-F, Cilk-F (O), and Cilk-L, we computed the speedup against the T_1 in Cilk-F. We computed the speedup for the ParWS against its corresponding baseline implemented in Parallel ML that is not shown. The x-axis shows the core counts (P) and the y-axis shows the speedup. Cilk-F (O) oversubscribes the system by using $2P$ workers instead of P, for P number of cores (i.e. every core has two workers pinned, one per hyperthread context). Cilk-L pins one worker thread and one I/O thread per core, each on its own hyperthread context.

	latency	T_1	T_5	T_{10}	T_{15}	T_{20}	T_{25}	T_{30}	T_{35}	T_{40}
	Ideal	32.87 (1.00×)	6.58 (1.00×)	3.29 (1.00×)	2.20 (1.00×)	1.65 (1.00×)	1.32 (1.00×)	1.11 (1.00×)	0.95 (1.00×)	0.83 (1.00×)
`n=30`	*1 ms*	32.42 (0.99×)	6.51 (0.99×)	3.28 (1.00×)	2.20 (1.00×)	1.67 (1.01×)	1.34 (1.01×)	1.13 (1.02×)	0.98 (1.03×)	0.87 (1.04×)
`b=25`	*50 ms*	32.54 (0.99×)	6.63 (1.01×)	3.39 (1.03×)	2.33 (1.06×)	1.80 (1.09×)	1.48 (1.12×)	1.27 (1.15×)	1.13 (1.19×)	1.04 (1.24×)
	100 ms	32.68 (0.99×)	6.71 (1.02×)	3.52 (1.07×)	2.45 (1.12×)	1.91 (1.16×)	1.60 (1.21×)	1.37 (1.24×)	1.21 (1.28×)	1.08 (1.30×)
	Ideal	369.19 (1.00×)	73.85 (1.00×)	36.93 (1.00×)	24.62 (1.00×)	18.47 (1.00×)	14.79 (1.00×)	12.32 (1.00×)	10.57 (1.00×)	9.25 (1.00×)
`n=35`	*1 ms*	362.73 (0.98×)	72.59 (0.98×)	36.32 (0.98×)	24.22 (0.98×)	18.19 (0.98×)	14.56 (0.98×)	12.15 (0.99×)	10.42 (0.99×)	9.13 (0.99×)
`b=15`	*50 ms*	362.89 (0.98×)	72.69 (0.98×)	36.42 (0.99×)	24.31 (0.99×)	18.27 (0.99×)	14.67 (0.99×)	12.24 (0.99×)	10.51 (0.99×)	9.22 (1.00×)
	100 ms	362.99 (0.98×)	72.76 (0.99×)	36.49 (0.99×)	24.42 (0.99×)	18.38 (1.00×)	14.74 (1.00×)	12.34 (1.00×)	10.61 (1.00×)	9.31 (1.01×)

Figure 3: The execution times, in seconds, of `map-reduce` with different Fibonacci parameters running on Ideal and Cilk-L with different latencies (ms = milliseconds). Two sets of Fibonacci parameters are tested; `n` shows the input size and `b` shows the serial base case size. The values in parentheses are overheads relative to the corresponding T_P time of *Ideal*, which runs `map-reduce` on Cilk-F with zero latency and incurs no system overhead for latency hiding (same as the `Ideal` label shown in Figure 2).

5.2 Cilk-L's Proximity to Ideal

Now we evaluate how close Cilk-L is to an "idealized" version at hiding I/O latencies. We obtain the ***Ideal*** measurement by running Cilk-F with a timed file descriptor with zero delay (i.e., in place of `cilk_read` the benchmark invokes a `read` that returns immediately).[10] Moreover, since it is run with Cilk-F, it does not incur any overhead of setting up IO futures, `epoll`, or waking up and context switching to I/O threads. That is, the idealized version incurs neither latency nor the overhead for hiding latency.

Figure 2 also includes the idealized version (labeled as "Ideal") in the plots. As we can see, when the latency is large Cilk-L lags behind Ideal when running `fib` of 30 with a serial base case of 25. We suspect that this is because the benchmark running on Cilk-L does not have sufficient parallelism and becomes span-dominated (i.e., the span term $T_\infty \lg P$ dominates the work term T_1/P in the time bound). By measuring `fib` of 30 running on one worker, the total amount of work is about 7 milliseconds, which is small compared to the 100 millisecond latency. Note that the computation running on Cilk-F variants do not have the same issue, as the I/O latency is incurred both on the work and span terms (or, in the Ideal case, there is zero latency). Thus, the parallelism profile is not the same when running on Cilk-L versus on Cilk-F variants. This shows up as Cilk-L lagging behind Ideal, because Ideal still achieves near-linear scalability whereas Cilk-L does not (scalability computed as T_P over T_1 running on the same platform).

To verify this, we also run both Ideal and Cilk-L with additional parameters, i.e., computing the Fibonacci number with different input size (`n`) and serial base case (`b`). Figure 3 shows the raw execution times of the Ideal and Cilk-L with different latencies. The first four rows are the times used to plot Figure 2. With

[10] Technically we used a 1 nanosecond latency, which is the smallest latency one could specify with the timed file descriptor on Linux, but it effectively causes the read to become ready immediately.

input of 30 and base case of 25, the overhead (T_P of Cilk-L divided by T_P of Ideal) starts out small and increases as P gets larger. With input of 35 and base case of 15, the computation has sufficient parallelism to hide the latency fully, and we see the discrepancy between Ideal and Cilk-L disappears. We also note that, for data points where Cilk-L runs `fib` of 30 with $P \geq 35$, we see higher standard deviations, between $6 - 10\%$.

5.3 Cilk-L's Overhead in Latency-Hiding

The use of IO futures in Cilk-L has some inherent overhead: (1) setting up and tearing down IO futures, (2) invoking the `epoll` mechanism, which has its inherent system call overheads, and (3) waking up and context switching into I/O threads. These overheads likely contribute to both the less desirable performance compared to oversubscribing Cilk-F when the latency is small and the additional overhead comparing to the ideal version. To figure out how much overhead is contributed by each source, we measure different versions of Cilk-L and compare that to Ideal (Cilk-F running `map-reduce` where `read` returns immediately). The `+future` version is similar to `Ideal` except that we added back the overhead of using IO futures (`fut-create`, placing the result into future handles, and `get`). Building on the *+future* version, the *+epoll* version then adds back the overhead of using `epoll`. Finally, the *+IO Thread* version adds back the overhead of using a separate thread to handle the I/O operations.[11] Note that since the latency is effectively zero, the I/O thread will be woken up only once per request when the request is inserted into the communication queue. Figure 4 shows the comparison. The empirical results show that the overhead from the use of futures is negligible. The overhead from `epoll` and the I/O thread are comparable, but both are small.

6 Related Work

Interesting use of futures: Since its proposal [22], the use of futures has been incorporated into various task parallel platforms [16, 28, 22, 17, 38, 20, 42, 15, 32]. Futures are typically used as a high-level synchronization construct to allow parallel tasks to coordinate with one another in a way that is more flexible than pure fork-join parallelism.

Researchers have proposed interesting uses of futures. Blelloch and Reid-Miller [11] used futures to generate non-linear pipelines. Using futures to pipeline the split and merge of binary trees, they developed a parallel algorithm of tree merge with better span than a fork-join parallel marge algorithm. Surendran and Sarkar [40] proposed using futures to automatically parallelize pure function calls in programs and developed the corresponding compiler analyses. Kogan and Herlihy [27] proposed ***linearizable futures*** that allow a concurrent data structure to be shared among threads via the use of futures and formalized the correctness guarantees for such uses. Milman et al. [33] proposed an algorithm for a batched lock-free queue using futures and proved correctness guarantees for the batched queue. By exploiting the semantic requirements of a queue, they can optimize the batched operations.

Prior work has used futures as an abstraction for I/O operations [19] in server software. However, their work does not address how the use of futures are scheduled nor what kind of performance bounds the scheduler can provide.

Supporting blocking synchronization primitives: Researchers have also proposed runtime schedulers for scheduling programs with blocking synchronizations. For instance, Agrawal et al. [7] proposed a work-stealing runtime system for ***helper locks*** — where when a worker tries to acquire a lock that is not available, it tries to help complete the critical section that is currently holding the lock. They proved that this scheduler was efficient if large critical sections had sufficient internal parallelism.

X10 [17] and Habanero [15] variants support blocking synchronization primitives such as conditional blocks, clocks, and phasers. Most of these implementations do not have provably efficient performance bounds. Initially, in X10 [17] and Habanero Java, synchronization primitives (e.g., conditional atomic blocks or barriers) may cause the worker to simply block, and the runtime compensates by creating a new worker thread to replace the blocked worker. Later, Tardieu et al. [41] proposed better compiler and runtime support for X10 for suspending a task blocked on synchronizations. However, the suspended tasks are stored in a centralized queue. For Habanero Java, [23] describes an alternative strategy: when suspended tasks become resumable, they are pushed onto the deque of the worker that executed the operation to unblock the tasks.

As mentioned in Section 1, Zakian et al. [44] extend Intel Cilk Plus [24] to provide support for a low-level library which allows a worker to suspend the current execution context upon encountering a blocking I/O operation and find something else to do. Due to how the deques with the suspended execution context are handled, however, their system does not provide a provably efficient execution time bound. In their system, the multiple suspended contexts (deques) are stored with the worker which suspended them, causing the deques

[11] Cilk-F is effectively a stripped-down version of Cilk-L that removes all mechanisms to support latency-hiding. We obtain each version by incrementally adding back each mechanism that incurs the overhead.

overhead	T_1	T_5	T_{10}	T_{15}	T_{20}	T_{25}	T_{30}	T_{35}	T_{40}
Ideal	32.87 (1.00×)	6.58 (1.00×)	3.29 (1.00×)	2.20 (1.00×)	1.65 (1.00×)	1.32 (1.00×)	1.11 (1.00×)	0.95 (1.00×)	0.83 (1.00×)
+future	31.58 (0.96×)	6.32 (0.96×)	3.16 (0.96×)	2.11 (0.96×)	1.58 (0.96×)	1.27 (0.96×)	1.06 (0.96×)	0.91 (0.96×)	0.80 (0.96×)
+epoll	32.88 (1.00×)	6.58 (1.00×)	3.29 (1.00×)	2.20 (1.00×)	1.65 (1.00×)	1.32 (1.00×)	1.10 (1.00×)	0.95 (1.00×)	0.83 (1.00×)
+IO Thread	32.38 (0.99×)	6.50 (0.99×)	3.27 (0.99×)	2.20 (1.00×)	1.66 (1.01×)	1.35 (1.02×)	1.13 (1.02×)	0.97 (1.03×)	0.86 (1.03×)

Figure 4: The execution times, in seconds, of `map-reduce` with various configurations of Cilk-L with no I/O latency (i.e. reads do not block). The overheads are relative to the corresponding T_P time of Ideal, which uses Cilk-F without latency-hiding.

among workers to potentially become extremely imbalanced. Moreover, a thief stealing into a victim always checks the active deque first before stealing into deques with suspended execution contexts. In addition, once a suspended execution context becomes ready to be resumed, the deque holding the context may become unstealable, but the worker who owns the deque is busy working on something else and cannot resume it in a timely manner. Thus, a high potential work item may have little or no chance to be stolen into despite many steal attempts.

Work-stealing schedulers with multiple deques per worker: Various work-stealing runtime systems have used multiple deques per worker for different reasons. The runtime system for helper locks [7] (discussed above) used multiple deques per worker. When a worker is blocked on a lock, it is only allowed to work on the critical section that is holding the lock (assuming this critical section has internal parallelism) and does so by allocating another deque specifically for this critical section. Therefore, in a program with nested locks with nesting depth d, workers could have as many as d deques each. However, the scheduler is designed so that each worker can steal from at most one deque of each of the other workers. In a similar vein, Agrawal et al. [6] proposed the Batcher runtime system to handle parallel programs that access shared data structures. In this case, workers can be working on either the program work or the data structure work, and these types of work are kept on different deques. But again, at any given time a worker steals randomly among P deques.

Porridge [43] is a processor-oblivious record-and-replay system for dynamic multithreaded programs using work stealing. Porridge allows multithreaded program with locks to be executed on some number of processors while recording all the happens-before relationship between critical sections. The execution can later be replayed on a different number of processors, but in a way that guarantees the same happens-before relationships. During record, the vanilla work-stealing algorithm that just blocks on an unavailable lock can be used. However, during replay, a vanilla work-stealing scheduler can lead to deadlocks. Therefore, if a critical section C tries to acquire a lock and can not acquire it since the critical section with a happens-before edge to C has not finished, the processor must find something else to do. The runtime system there also uses proactive work-stealing and achieves similar bounds.

7 Conclusion

In order to support modern desktop and server software, I/O operations should be supported as a fundamental component of task parallel platforms. In this paper, we show how one may incorporate I/O operations into a task parallel platform seamlessly with efficient scheduling to hide I/O latencies. In particular, our platform Cilk-L provides a programming API for performing I/O operations that works harmoniously with existing parallel control constructs. In addition, the underlying runtime system efficiently schedules both the computation and the I/O operations to provide nearly optimal execution time guarantees and a bound on the number of deviations. We achieve this by using the proactive work stealing scheduler recently developed for scheduling computations with futures. Empirical evaluation of our prototype system shows that I/O can be supported efficiently with effective latency hiding.

References

[1] Linux programmer's manual epoll(7). `http://man7.org/linux/man-pages/man7/epoll.7.html`, 2019. Accessed in January 2019.

[2] Linux programmer's manual eventfd(2). `http://man7.org/linux/man-pages/man2/eventfd.2.html`, 2019. Accessed in January 2019.

[3] Linux programmer's manual timerfd_create(2). `http://man7.org/linux/man-pages/man2/timerfd_create.2.html`, 2019. Accessed in January 2019.

[4] Umut Acar, Guy E. Blelloch, and Robert Blumofe. The data locality of work stealing. *Theory of Computing Systems*, 35(3):321–347, 2002.

[5] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *Proc. of the 12th ACM Annual Symp. on Parallel*

Algorithms and Architectures (SPAA 2000), pages 1–12, 2000.

[6] Kunal Agrawal, Jeremy T. Fineman, Kefu Lu, Brendan Sheridan, Jim Sukha, and Robert Utterback. Provably good scheduling for parallel programs that use data structures through implicit batching. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 84–95, New York, NY, USA, 2014. ACM.

[7] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. Helper locks for fork-join parallel programming. In *15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 245–256, 2010.

[8] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *10th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, 1998.

[9] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, pages 115–144, 2001.

[10] 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. 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, pages 735–736, Orlando, Florida, USA, 2009. ACM.

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

[12] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 356–368, November 1994.

[13] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, 1999.

[14] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Pearson, USA, 3rd edition, 2015.

[15] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 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, pages 51–61, 2011.

[16] Rohit Chandra, Anoop Gupta, and John L. Hennessy. COOL: An object-based language for parallel programming. *IEEE Computer*, 27(8):13–26, August 1994.

[17] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, 2005.

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

[19] Marius Eriksen. Your server as a function. *SIGOPS Oper. Syst. Rev.*, 48(1):51–57, May 2014.

[20] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly threaded parallelism in manticore. *Journal of Functional Programming*, 20(5-6):537–576, November 2010.

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

[22] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM TOPLAS*, 7(4):501–538, October 1985.

[23] Shams Imam and Vivek Sarkar. Cooperative scheduling of parallel tasks with general synchronization patterns. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*, pages 618–643, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[24] Intel® Cilk™ Plus. `https://www.cilkplus.org`, 2013.

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

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

[27] Alex Kogan and Maurice Herlihy. The future(s) of shared data structures. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 30–39, Paris, France, 2014. ACM.

[28] David A. Kranz, Robert H. Halstead, Jr., and Eric Mohr. Mul-T: A high-performance parallel Lisp. In *PLDI*, pages 81–90. ACM, 1989.

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

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

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

[32] Li Lu, Weixing Ji, and Michael L. Scott. 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, pages 519–529, Edinburgh, United Kingdom, 2014. ACM.

[33] Gal Milman, Alex Kogan, Yossi Lev, Victor Luchangco, and Erez Petrank. BQ: A lock-free queue with batching. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, pages 99–109, Vienna, Austria, 2018. ACM.

[34] Stefan K. Muller and Umut A. Acar. 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, pages 71–82, Pacific Grove, California, USA, 2016. ACM.

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

[36] Kyle Singer, Yifan Xu, and I-Ting Angelina Lee. Proactive work stealing for futures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 257–271, New York, NY, USA, 2019. ACM.

[37] Daniel Spoonhower, Guy E. Blelloch, Phillip B. Gibbons, and Robert Harper. 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, pages 91–100, Calgary, AB, Canada, 2009. ACM.

[38] Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. Space profiling for parallel functional programs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 253–264, Victoria, BC, Canada, 2008. ACM.

[39] Daniel John Spoonhower. *Scheduling Deterministic Parallel Programs*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 2009.

[40] Rishi Surendran and Vivek Sarkar. Automatic parallelization of pure method calls via conditional future synthesis. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 20–38, New York, NY, USA, 2016. ACM.

[41] Olivier Tardieu, Haichuan Wang, and Haibo Lin. 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, pages 267–276, New Orleans, Louisiana, USA, 2012. ACM.

[42] Sağnak Taşırlar and Vivek Sarkar. Data-driven tasks and their implementation. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 652–661, Taipei City, Taiwan, 2011. IEEE Computer Society.

[43] Robert Utterback, Kunal Agrawal, I-Ting Angelina Lee, and Milind Kulkarni. Processor-oblivious record and replay. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 145–161, Austin, Texas, USA, 2017. ACM.

[44] Christopher S. Zakian, Timothy A. Zakian, Abhishek Kulkarni, Buddhika Chamith, and Ryan R. Newton. Concurrent Cilk: Lazy promotion from tasks to threads in c/c++. In *Revised Selected Papers of the 28th International Workshop on Languages and Compilers for Parallel Computing -*

Volume 9519, LCPC 2015, pages 73–90, Raleigh, NC, USA, 2016. Springer-Verlag.