Extended Version

Concurrent Cilk: Lazy Promotion from Tasks to Threads in C/C++

Christopher Zakian Timothy A. K. Zakian
Abhishek Kulkarni Buddhika Chamith Ryan R. Newton
Indiana University - Bloomington
{czakian, tzakian, adkulkar, budkahaw, rrnewton}@indiana.edu

Abstract

Library and language support for scheduling non-blocking tasks has greatly improved, as have lightweight (user) threading packages. However, there is a significant gap between these two developments. In previous work—and in today's software packages—lightweight thread creation incurs much larger overheads than tasking libraries, *even* on tasks that end up never blocking. This limitation can be removed. To that end, we describe an extension to the Intel Cilk Plus runtime system, *Concurrent Cilk*, where tasks are *lazily* promoted to threads. Thus Concurrent Cilk removes the overhead of lightweight thread creation on threads which end up calling no blocking operations, and Concurrent Cilk is the first system to do so for C/C++ with legacy support (standard calling conventions and stack representations). We demonstrate that Concurrent Cilk adds negligible overhead to existing Cilk programs, while its promoted threads remain more efficient than OS threads in terms of context-switch overhead and blocking communication. Further, it enables development of blocking data structures that create non-fork-join dependence graphs—which can expose more parallelism, and better supports data-driven computations waiting on results from remote devices.

Categories and Subject Descriptors D.3.2 [*Concurrent, Distributed, and Parallel Languages*]

General Terms Design, Languages, Performance, Runtimes

Keywords Work-stealing, Concurrency, Cilk

1. Introduction

Both **task-parallelism** [1, 11, 15**?**] and **lightweight threading** [20] libraries have become popular for different kinds of applications. The key difference between a task and a thread is that threads may block—for example when performing IO—and then resume again. Lightweight threading libraries usually require cooperative multitasking but can, in return, support over a million threads, which is naturally useful for applications such as servers that involve highly concurrent IO-driven computations. Tasks, on the other hand, are finite duration and do not block. Indeed this assumption is baked deeply into the implementation of libraries such as TBB (Threading Building Blocks [15]) and language extensions such as Cilk [4]. Tasks are executed on shared worker threads where blocking such a thread is a **violation of the contract** between programmer and library, which can cause subtle deadlocks, as well as a loss of parallel efficiency.

If the no-blocking-guarantee can be met, then task-parallelism libraries offer an order of magnitude lower overhead for creating parallel tasks ("many tasking" rather than "multi-threading"). Cilk [4], in particular, is well known for its low-overhead *spawn* feature where the overhead of creating a parallel *fiber* with `cilk_spawn f(x)` is as little as 2-5 times the overhead of a regular function call, `f(x)`. The key to this low-overhead is that Cilk fibers are essentially *lazily parallel*: fibers execute sequentially, exposing the continuation of the parallel call with a minimum of overhead, and lazily promoting the continuation to a parallel continuation only when *work-stealing occurs*—and even then only using shared resources, not fiber-private stacks.

As a language extension, Cilk programs do not require any additional syntactic overhead to enable parallelism; for example, there is no need to manually manage the *continuation* of a parallel call as a callback. However, because a traditional Cilk program must run even with *sequential semantics*—spawned fibers cannot serve the role of threads in the sense that they cannot be used for managing concurrent IO. That is, even continuations lazily promoted to *parallel*, are not truly *concurrent*, as they don't have their

own stacks. It is this subsequent lazy promotion that we add in Concurrent Cilk.

To the programmer, a `cilk_spawn` and a thread spawn look very similar, but current limitations require knowing *at the point of the call*, which variant will be required: will the spawned computation need to suspend, and thus require its own stack? This decision point remains even in high-level languages designed with both parallelism and concurrency in mind, which support both tasks and threads using *different* language mechanisms. For example, the Glasgow Haskell Compiler supports "sparks" (tasks) and language-level "IO threads" with different APIs [**?**].

Concurrent Cilk, on the other hand, extends the Cilk runtime interface with new primitives for *pausing* a fiber and returning a handle that will allow other fibers (or Pthreads) to unpause the computation[1], and extends the states through which a fiber is promoted with a third, fully concurrent, state:

1. Executing sequentially, continuation uninstantiated
2. Executing in parallel with continuation, shares stacks
3. Fully concurrent, private stack, able to pause/resume

That is, Concurrent Cilk initially executes fibers *sequentially*, lazily promoting them to "parallel work" during stealing, and lazily promoting them to "threads" only when necessary (Figure 5). It then becomes possible to use the `cilk_spawn` feature for *all* parallelism and concurrency, even if it is not known (or even knowable) at the point of its creation whether the fiber will need to block—for example, for a computation on a server to wait on further communications with the client, or for a ray tracer to fetch remote data to compute a particular pixel.

Previous attempts to provide blocking, lightweight fibers in C have either required changing calling conventions and breaking legacy binary support [19], or create a full [linear] call-stack for each fiber [20]. Concurrent Cilk is the first system to enable lightweight threads in C, with legacy support, and memory-use (number of stacks) proportional to *blocked* fibers, not total spawned fibers.

An the other hand, for parallel languages with specialized compiler support, and no backwards compatibility concerns (linear stacks), lazy thread spawning *has* been explored, namely in the context of Id90 [7]. (Although Id90 used only states (1) and (3) above, not the full three-state algorithm.) And yet today, Concurrent Cilk is, to our knowledge, the only threading system that uses this algorithm, even including languages like Go, Haskell, and Erlang with good lightweight threading support. Nevertheless, with the prevalence of asynchronous workflows, especially in the web-services domain, we argue that this is an idea whose time has come. It provides a better abstraction to the programmer—with a single logical spawn construct replacing careful reasoning about non-blocking tasks, shared threads, and user threads—and it is implementable even in mature systems like Intel Cilk.

In this paper, we make the following contributions:

- We present the first system for unified lightweight tasking and threading that supports C/C++ code and existing binaries. We describe the changes that are necessary to add concurrency constructs (pause/resume a parallel fiber) to a mature, commercial parallelism framework, and we argue that other many-tasking frameworks could likewise adopt lazy-promotion of tasks to threads.
- We show how to build blocking data structures (e.g. IVars, channels) on top of the core Concurrent Cilk pause/resume primitives.
- We use Linux's `epoll` mechanism to build a *Cilk IO library* that provides variants of posix routines like `read`, `write`, and `accept` which block only the current Cilk fiber, and not the OS thread.
- We evaluate Concurrent Cilk in terms of (1) additional runtime-system overhead across a set of applications (Section 6.1); (2) opportunities for improved performance by *sync elision* (Section 6.3.1); and (3) a study of injecting blocking IO in parallel applications, or, conversely, injecting parallel computation inside IO-driven server applications.

2. Background and Motivation

Cilk itself dates from 1996 [**?**]; it is a simple language extension that adds parallel subroutine calls to C/C++. Only two constructs make up the core of Cilk: `cilk_spawn` for launching parallel subroutine calls, and `cilk_sync` for waiting on outstanding calls (with an implicit `cilk_sync` at the end of each function body). For example, here is a common scheduler microbenchmark, parallel fibonnacci:

```
long parfib(int n) {
  if (n<2) return 1;
  long x = cilk_spawn parfib(n-1);
  long y =            parfib(n-2);
  cilk_sync;
  return x+y;
}
```

Logically, each `cilk_spawn` creates a virtual thread, i.e. a fiber. Cilk then multiplexes these fibers on any number of OS *worker threads*, determined at runtime. Cilk only instantiates fibers in parallel when random work-stealing occurs[2]. So running `parfib(42)` does not create stack space for half a billion fibers, rather it typically uses one worker thread for each processor or core.

Cilk has been surprisingly successful as a language extension, recently being incorporated into the commercial Intel

[1] This handle is similar to a [parallel] *one-shot continuation*. Continuations are well studied control constructs [9, 17] and known to be sufficient to build cooperative threading (coroutines) [9] as well as blocking data structures that enable, for example, stream-processing with back-pressure.

[2] Cilk is a *work first* system, which means that the thread that executes `spawn f` will begin executing `f` immediately; it is actually the *continuation* of the spawn that is exposed for work stealing.

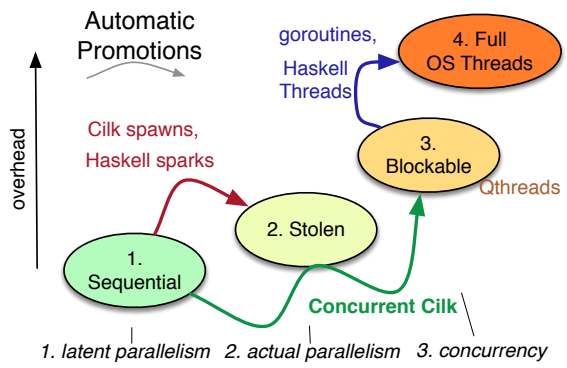

Figure 1. State transitions possible for a fiber in each of several existing systems. Increasing levels of independence require additional storage/overhead. At level (1), the fiber executes entirely within the stack of its caller. Work stealing transitions to (2) where a pre-existing system worker stack (allocated at startup) is used to execute the continutation of `f` in parallel. A blocked fiber requires additional storage for its state (3). Finally, blocking on underlying OS calls requires an OS thread (4).

C++ compiler[3], released in version 4.9 of the GNU C compiler, `gcc` [**?**], and available in `clang` as well. This success appears to be in large part due to (1) Cilk's extreme simplicity, and (2) the legacy support in Intel Cilk Plus. That is, Cilk programs can be linked with previously compiled libraries and legacy code may even call Cilk functions through function pointers.

The work-stealing model supported by Cilk has been adopted by many other C/C++ libraries (Intel TBB, Microsoft TPL, and others). Unfortunately, so has its lack of support for blocking operations within parallel tasks. None of these C/C++ runtime systems can react to a task blocking—whether on a system call or an in-memory data structure. For example, TBB blocking data structures (e.g. queues) are not integrated with TBB task scheduling; in spite of providing both these capabilities to the programmer in the same package, the scheduler *cannot* intercept blocking calls and shelve blocked tasks, keeping the worker, and thus the corresponding core, busy.

2.1 Blocking deep in a parallel application

To illustrate the problem Concurrent Cilk solves, we begin by considering adding network IO to a plain Cilk application. Take, for example, software that renders movie frames via ray tracing[4]. A rendering process runs on each machine in a compute farm, and may use all the processors/cores within that machine. Let us suppose the software evolved from a sequential application, but has been parallelized using Cilk constructs.

[3] Now called *Intel Composer* and *Cilk Plus* [1].

[4] Ray tracing follows an imaginary line from each pixel in the image into the scene to see what objects are encountered, rather than starting with the objects and drawing (rasterizing) them onto the screen.

Somewhere in our rendering application we might expect to see a parallel loop that launches a ray for each pixel we are interested in. Contemporary Cilk implementations provide a `cilk_for` drop-in replacement for `for`, which is implemented in terms of `cilk_spawn` and `cilk_sync`.

```
sort(pix_groups); // Parallel sorting and grouping.
cilk_for (i < start; i<end; i++) {
  // Task and data parallelism *within* a pix_group:
  ...cast_ray(pix_groups[i])...
}
```

We should not expect, however, that this is the only point of potential parallelism. Ray tracing is traditionally an example of irregular and *nested* data parallelism, where an individual ray may spawn others, reflect off surfaces, and bounce around the environment. Further complexity will result from attempts to optimize how pixels are sorted and grouped into chunks of rays that travel together. In a complex application, we should expect long series of function calls, deep call stacks, and parallelism introduced at multiple points during those chains of calls.

Suppose now that in this context—deeply nested inside a series of parallel and sequential function calls—we encounter a situation where the ray has left the local [virtual] space, whose textures and geometry are loaded on the current machine, and entered an adjacent area stored elsewhere in networked storage[5]. In this hypothetical rendering application, if every ray rendered had its own *Pthread* (which is impractical), then it would be fine to block that thread by directly making a network request as a system call:

```
// Deep in the stack, in the midst of rendering:
void handle_escaped(ray r, id rsrc) {
  blob f = webapi.request(rsrc);
  // Block a while here, waiting on the network...
  load_into_cache(f);
  resume_ray(r);
}
```

But if Cilk has been used to parallelize the application, the above is very dangerous indeed. First, because there is generally one Cilk worker thread per core, blocking a worker thread often *leaves a core idle*. Second, any attempts to hold locks or block on external events invalidates the traditional space and time bounds on Cilk executions [**?**]. Finally, blocking calls can deadlock the system if there are enough such calls to stall all Cilk worker threads, starving other computations that might proceed—including, potentially, the one that would unblock the others!

Attempted fix 1: avoid blocking To *avoid* blocking within a parallel task, how can the application be refactored? If the need for IO operations is discovered dynamically (as in ray tracing), there are two options: (1) fork a Pthread at the point where IO needs to occur, passing an object bundling up the

[5] Models and textures grow very large and cannot be fully loaded onto every machine. Indeed, we should expect the need to retrieve data from network storage to increase in future applications due to explosion in data sizes.

rest of the computation that needs to occur, *after* the IO completes[6]; or (2) return failure for the parallel task, wait until the parallel region is finished, then perform IO and try again (a trampoline). Because Cilk allows (strictly) nested parallelism, deferring actions until the end of a parallel region potentially requires restructuring the control-flow of the entire application—pulling all potential-IO in deeply nested contexts to the application's "outer loop".

Attempted fix 2: overprovision to tolerate blocked workers Of course, it is possible to provision additional Cilk workers, say, $2P$ or $4P$ (where P is the number of processors or cores). This would indeed hide some number of blocking operations, keeping processors from going idle, at the cost of additional memory usage and some inefficiency from over-subscription. Unfortunately, this puts the requirements on the user to understand the global pattern of blocking operations at a given point in program execution, which is especially difficult within a parallel region. Moreover, if blocked threads are interdependent on one another—for example using in-memory blocking data-structures for inter-fiber communication—then the *maximum possible* simultaneously blocked computations is key to deadlock avoidance. In general, violating the Cilk scheduler's contract (by blocking its workers) is a dangerous action that cannot be used composably or abstracted inside libraries.

Thus we argue that, if Cilk fibers must block their host threads, then it is better to create replacement worker threads on demand (as Cilk instantiates fibers on demand, upon stealing) as an integral part of the runtime system. Hence Concurrent Cilk.

3. Approach and Implementation

3.1 High-level Programming Model: IVars and friends

Concurrent Cilk follows the Cilk tradition of using a small set of powerful, composable primitives, which can then form the basis for higher-level abstractions or syntactic sugar. The core primitives for Concurrent Cilk are pause and resume on fibers, and while it is good for library implementers to directly use these primitives, most end users will prefer to use higher-level data structures. Thus we begin our exposition of the programming model using one such high-level structure—the *IVar*—as an example, and then we return to the lower level API later on in this section.

An IVar is a single-assignment data structure that exists in either an empty or full state[7]. The basic interface is:

```
void           ivar_clear(ivar*);
ivar_payload_t ivar_get(ivar*);
void           ivar_put(ivar*, ivar_payload_t);
```

New IVars are stack- or heap-allocated and then set to the empty state with `ivar_clear` [8]. Get operations on an empty IVar are *blocking*—they pause the current fiber until the IVar becomes full. Once an IVar has transitioned to a full state, readers are woken so they can read and return the IVar's contents.

IVars do not allow emptying an already full IVar, but it is possible to "delete" an IVar and create a new one by clearing and reusing its memory. Further, IVars are only one representative example of a sychronization structure built with pausable fibers—*MVars* would allow synchronized emptying and refilling of the location, or a bounded queue with blocking enqueues and dequeues.

Pausing the fiber In fact, all these data structures make use of the underlying Concurrent Cilk API in the same way. As a typical example, the psuedocode for a simple `ivar_get` operation is shown in Figure 2. This is a simplified example, which we will optimize shortly, but which demonstrates our *two phase* pausing API, as follows. (In fact, later we will show how our lowest-level API is a three-phase one to enable further optimizations.)

1. `pause_fiber()` – capture the current context (`setjmp`), and begin the process of shelving the current fiber.
2. `commit_pause()` – jump to the scheduler to find other work

In between these two operations, the fiber that is just about to go to sleep has time to store a reference to itself inside a data structure. Without this step, it would not be possible for other computations to know that the fiber is asleep, and wake it. In the case of IVars, each empty IVar with blocked readers stores a pointer to a *waitlist*, which will be discussed in the next section. Further, as an implementation note, the `pause_fiber` routine must be implemented as an inline function or preprocessor macro—so that it calls `setjmp` from within the correct stack frame.

Waking the fiber The code for `ivar_put` is shown in Figure 3. Its job is simpler: attempt to compare and swap the IVar to fill it in, and retrieve the `waitlist` at the same time. If it finds the IVar already full, it errors. When it processes the `waitlist`, it uses a third Concurrent Cilk API call, which we introduce here, that has the effect of enqueing the paused fiber in a ready-queue local to the core on which it was paused.

3. `wakeup_fiber(w)` – take the worker structure, and enqueue it in the readylist

Naturally, thread wakeup and migration policies are a trade-off: depending on the relative quantity and reuse distance of the working set for the blocked computation—and the amount data communicated to it through the IVar—it

[6] In other words, this means manually converting parts of the application to *continuation passing style* (CPS).

[7] Strictly speaking, IVars differ from *futures*, in that, while both are a storage location supporting synchronization, a future binds a specific (known) producer of the value to the location at the time of its creation, whereas an IVar exports a method for filling it that can be called anywhere. Further, there are other useful variations on synchronization variables, such as MVars, which enable filling and emptying the location multiple times.

[8] Here, and in the rest of this paper, we omit the prefix `__cilkrts_` which is found in most of the symbols in CilkPlus, and our fork, Concurrent Cilk `http://github.com/[omitted for double blind]`

	pthread	CCilk same-core	CCilk diff-core
time	0.849984	0.174844	0.394509

Table 1. Time in seconds required for 100,000 ping-pongs between two fibers or threads, measured on the test platform described in Section 6. This microbenchmark compares context-switch overhead. In the Concurrent Cilk case, the ping and pong fibers communicate through a pair of IVars (that are used once and cleared). Whether the fibers end up on the same or different cores depends on random work-stealing. In the Pthread case, communication is via `pthread_cond_signal`.

could be best to wake the fiber either where it paused or where it was woken, respectively. We chose the former as our default.

3.2 Representation, efficiency, and polymorphism

In our current implementation, IVars are implemented with a single word of storage. We reserve two bits for a *tag* to represent the full/empty/paused state, and thus use macros such as `UNTAG()` and `IVAR_MASK` to operate on the tag bits. A result of this design decision is that IVars can be used to represent integers in the interval $[0, 2^{62})$, as well as pointers on the AMD 64 architecture (which allows only 48 virtual addresses). Further, because `ivar_get` inlines (while `ivar_get_slowpath` does not) the generated code to read an IVar only differs from reading a regular variable in the addition of a couple arithmetic operations and one correctly-predicted branch—notably there are no additional memory references incurred.

This *unboxed* representation of IVars exposed by the C API makes them very efficient for storing arrays of integers. However, for a more programmer-friendly, more *polymorphic*, interface we also provide a C++ IVar interface, which is templated appropriately, such that `ivar<T>` is either empty, or contains a pointer to an object of type `T`. Still, the efficiency of unboxed ivars is only possible for scalar types like 32 bit `int`s, e.g, `ivar<int>`.

One other efficiency of this IVar representation is that the "waitlist"—a linked list of fibers waiting for the value of the IVar, does not require extra storage. That is, because the IVar only has a waitlist when it is logically empty, the 62 bit free bits in its payload can be used to store a pointer to the waitlist instead. Moreover, in the full version of the psuedocode in Figure 2 (i.e., Figure 11), all cells of the `waitlist` need not even be heap allocated. Rather, they simply live on the stack frames of their respective, blocked, `ivar_get` call.

This contributes to Concurrent Cilk's improved context switch times, as compared to operating system threads (Table 1). Also, on-stack waitlists is an efficiency we can gain implementing IVars in Concurrent Cilk that is not possible in managed languages which implement IVars and MVars as heap objects only, such as Haskell [**? ?**] and Concurrent ML [**?**].

```
1  #define IVAR_SHIFT 2
2  #define TAG(p,tg) ((p << IVAR_SHIFT) | tg)
3  #define UNTAG(p)  (p >> IVAR_SHIFT)
4
5  inline ivar_payload_t ivar_get(ivar *iv)
6  {
7    // fast path -- already got a value:
8    if (IVAR_READY(*iv)) return UNTAG(*iv);
9    else return ivar_get_slowpath(iv);
10 }
11 ivar_payload_t ivar_get_slowpath(ivar *iv) {
12   // slow path -- block until a value is available
13   if (! pause_fiber()) {
14     w = get_tls_worker_fast();
15     exit = 0;
16     while(!exit) {
17       peek = *iv;
18       switch (peek & IVAR_MASK) {
19         case IVAR_FULL:
20          // Nevermind...someone filled it:
21          return UNTAG(*iv);
22
23         case IVAR_EMPTY:
24          waitlist = singleton_list(w);
25          payload = TAG(waitlist, IVAR_PAUSED);
26          exit = cas(iv, 0, payload);
27          break;
28
29         case IVAR_PAUSED:
30          waitlist = cons(w, UNTAG(peek));
31          new =  TAG(waitlist, IVAR_PAUSED);
32          exit = cas(iv, peek, new);
33          break;
34       }
35     }
36     // Register 'w' in paused list, jump to scheduler:
37     commit_pause(w);
38     ASSERT(0); //no return. heads to scheduler.
39   }
40   // <-- We jump back here only when the value is ready
41   return UNTAG(*iv);
42 }
```

Figure 2. Psuedocode for a blocking read on an IVar. In this simplified code, we elide variable declarations, memory deallocation, and some casts. We also use the *two phase* pause interface, as well as a persistent (functional) cons-list rather than a lock-free queue.

4. Another high-level interface: I/O Library

Before delving deeper into the low-level Concurrent Cilk API and scheduler implementation, we first describe another abstraction layered on top of Concurrent Cilk, one which provides another programmer-facing abstraction that is key to the goal of Concurrent Cilk: blocking I/O calls intermingled with parallel tasks.

The Cilk I/O library we implemented provides a way for fibers to block—not just on application-internal events like another fiber writing an IVar—but on external events such as network communication. The programmer-visible API exposed closely follows the normal posix API as shown below:

```
int cilk_read(int fd, void *buf, int len);
```

```
1  void ivar_put(ivar* iv, ivar_payload_t val)
2  {
3    new_val = TAG(val, CILK_IVAR_FULL);
4    exit = 0;
5    while (!exit) {
6      peek = *ivar;
7      switch (peek & IVAR_MASK) {
8        case CILK_IVAR_EMPTY:
9          old_val = cas(ivar, peek, new_val);
10         if (old_val == peek) exit = 1;
11         break;
12
13       // Empty but with waiting readers:
14       case CILK_IVAR_PAUSED:
15         old_val = cas(ivar, peek, new_val);
16         if (old_val != peek) break;
17         // We own the whole waitlist here:
18         waitlist = (cons_t *) UNTAG(old_val);
19         while ( waitlist != NULL ) {
20           w = waitlist->car;
21           enqueue(w->readylist, w);
22           waitlist = waitlist->cdr;
23         }
24         exit = 1;
25         break;
26
27       // Otherwise it's already full:
28       default:
29         raise("Multiple puts on IVar");
30     }
31   }
32 }
```

Figure 3. Psuedocode for a non-blocking write on an IVar.

```
int cilk_write(int fd, void *buf, int len);
int cilk_accept(int listen_fd, struct sockaddr *addr,
                socklen_t *addr_len);
void cilk_sleep(long num_microseconds);
// Can be called implicitly at process start/exit time:
int cilk_io_init(void);
void cilk_io_teardown(void);
```

Except, of course, blocking semantics are achieved, not by blocking the entire OS thread, but rather the Concurrent Cilk fiber. Our current implementation uses the Libevent library, which provides an abstraction over OS mechanisms like Linux's `epoll`. Libevent provides a programming interface for registering events with associated callbacks. It raises the abstraction level from raw `epoll` by, for example, handling the *event loop(s)* internally.

An initialization routine, `cilk_io_init`, needs to be called before calling any IO methods. This launches a new daemon thread to run the event loop, similar to other IO managers that can be found in language runtime systems [**?**]. The `cilk_accept`, `cilk_read`, `cilk_write`, and `cilk_sleep` procedures register corresponding events to the event loop before yielding the control to a different fiber by blocking on an IVar read. In this, their implementations are *all* similar to the `ivar_get` implementation shown in Figure 2. Accordingly, `ivar_put` is performed by the event callback, running on the daemon thread containing the event loop. Note, however, that we do not need to worry about running computation *on* the event loop thread (which would delay it from processing events)—`ivar_puts` are cheap and constant time, only calling `wakeup_fiber()` to resume computation. As we saw before `wakeup_fiber()` *always* resumes the fiber on the worker thread on which it went to sleep, which can never be the event loop thread.

In Section 6, we will return to the topic of the IO library as a foundation for server applications.

5. Low-level implementation and scheduler

5.1 Cilk Scheduler overview

Cilk *workers* live in a global array which is accessed during the work-stealing process. When a worker becomes starved for work, another worker is then chosen, at random, from the global array and if there is any work available to be stolen, the *thief* steals the work from the currently busy worker (victim) and does the computation on its behalf. There have been several implementations of Cilk, and other papers describe their implementation and interfaces in detail, from the early MIT versions of Cilk [6], to the binary ABI specification of Intel Cilk Plus [2]. Thus we do not go into detail here, instead outlining only the basic scheduler with an emphasis on the modern Intel Cilk Plus implementation:

1. Legacy code is supported, without recompilation and even if precompiled functions call function pointers which create parallel work. This rules out modifying calling conventions or the representation of stack frames.
2. The user creates as many OS threads as they like, (including `main()`), *any* of which may initialize a parallel call via `cilk_spawn`, in which case they become bound to the single, shared instance of the Cilk runtime system, initializing it if it is not already initialized.
3. These user threads become *workers* and participate in work-stealing, until the parallel call they entered with completes. User workers steal, but only tasks related to their original task (i.e., on their "*team*").
4. On a machine of P cores, the runtime initializes $P - 1$ *system workers* that work-steal from each-other and from user workers. All workers (up to a maximum) are accessible in $O(1)$.
5. No parallel function call from one thread can cause the completion of a parallel call from another thread to depend on it.
6. C++ exception semantics are respected, and an exception thrown in parallel emerges at the point of the `cilk_spawn` (respecting the sequential semantics in which the spawn was elided).

5.2 Adding the Concurrent Cilk Extensions

The idea of Concurrent Cilk is simple; however, the Cilk Plus runtime system is a complex and comparatively difficult to modify artifact, so implementation must proceed with care. Our basic approach is that if a Cilk worker becomes

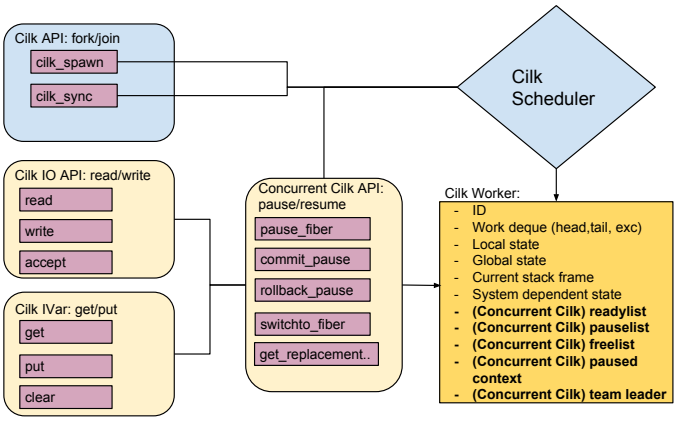

Figure 4. The architecture of the modified Concurrent Cilk runtime system. Also pictured is the included, but optional, Cilk IO library.

blocked, detach the worker from its OS thread[9] and substitute a *replacement worker* which then steals computation from any of the workers in the global worker array. When the blocking operation has finished, the worker is restored to an OS thread at the next opportunity and the replacement worker is cached for future use. In this way, all OS threads managed by the Cilk runtime are kept active. This strategy is similar to other lightweight threading systems [8, 13, 20], except in that Concurrent Cilk "threads" (fibers) start out *without* stacks of their own.

As pictured in Figure 4, most Cilk worker state is thread-local—including a stack of stealable Cilk stack frames, a linear C stack, and many book-keeping and synchronization related fields. A cache of stacks is kept both at the global and thread-local levels. To this picture, Concurrent Cilk adds three main additional fields:

1. Paused list – workers that cannot currently run
2. Ready list – workers that have unblocked and are ready for execution.
3. Free list – an additional cache of workers that previously were paused and now can be used as replacements for newly paused fibers

Each of the lists above is currently implemented as a lock-free Michael and Scott queue [14]. When the current fiber pauses, work-stealing only occurs if there are not already local fibers on the ready list. This gives a standard round-robin execution order to ready-threads.

5.3 Scheduler modifications

The additional Concurrent Cilk data structures described above are primarily touched by the pause, commit pause, and wakeup routines, and so they do not interfere with traditional Cilk programs that never block. However, there *must* be some modification of the core scheduler loop so as to be able to run work in the ready list.

The core scheduler algorithm picks random victims and attempts to steal in a loop, eventually going to sleep temporarily if there is no work available. We inject checks for the extra workers in two places:

- In the stealing routine—if a first steal attempt fails, rather than moving on from a victim, we check if the victim has additional stealable work inside paused or ready workers on that same core.
- At the top of the scheduler loop—we do *not* engage in work stealing if there are already threads in the ready list prepared to run. In this way, cooperative multi-tasking is possible in which no work-stealing is performed, and control transfers directly from thread to thread as in other lightweight threading systems. To make this maximally efficient, however, in the next Section we will have to extend the pause/wakeup API from the simplified form we have seen. Preferentially handling ready (local) threads over stealable work has precedent in existing (multi-paradigm) parallel language runtimes [**?**] that prioritize user-created (explicit) concurrency over exploiting latent parallelism.

The above modifications do change how we find victims, while at the same time we retain the global (static) array of workers as it is in Intel Cilk Plus—as the starting point for all work-stealing. In Concurrent Cilk the global array represents the *active* workers, of which there are the same number in Concurrent Cilk and Cilk. To maintain this invariant, we must necessarily rotate out which workers reside in the global array. Whenever one worker pauses and activates another, that replacement becomes "on top".

In Concurrent Cilk, all paused or ready fibers may *also* have latent, stealable continuations that must be able to execute in parallel[10]. In terms of prioritizing different work sources, we conjecture that it remains best to **steal from active workers first**. Their working sets are more likely to be in a shared level of cache. For that reason we only check paused fibers when the active one yields no work.

From a software engineering perspective, leaving the global array of workers in place and fixed size enables us to avoid breaking a system wide invariant in the Cilk Plus runtime system, which would require substantial re-engineering. At the same time, by modifying work-stealing to look deeper inside the list of paused and ready workers, we retain a liveness guarantee for parallel continuations:

Guarantee: *If a physical worker thread is idle, all logically parallel work items are reachable by stealing.*

Any violation of this guarantee could greatly reduce the parallel efficiency of an application in worst-case scenarios.

[9] A Cilk worker represents a thread local state which sits on top of an OS level thread.

[10] The original proof of Cilk's space and time bounds relies on the critical path of the computation remaining always accessible in this way. Non-uniform probabilities in work-stealing are a concern to some authors of Cilk.

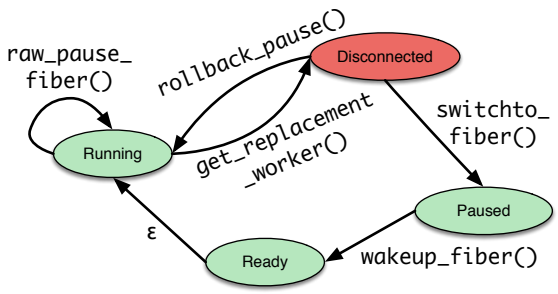

Figure 5. Transitions in the state of a worker. Disconnected is a temporary invalid state, which requires either rollback or switching to a replacement to restore the system to a good state.

5.4 Optimized pause/resume interface

Before proceeding to evaluation, there is one more implementation issue to address that can significantly improve performance. The two-phase pausing process described above (`pause_fiber()`, `commit_pause(w)`) does not specify *where* the current thread yields control to upon `commit_pause` for the simple reason that it always jumps to the scheduler. When we are round-robining threads through a given core, it is more efficient if one thread can longjump directly to the next one.

Like other library interfaces (e.g., Boost smart pointers) we provide both a convenient interface, and a more "intrusive" but performant interface, which requires that the API client assume more of the responsibility. This takes two forms. First, as promised, we enable direct `longjmp` between threads, but at the expense of replacing `commit_pause` with a multiple calls in a finer grained interface. First, a `get_replacement` function returns a pointer to the replacement rather than jumping to the scheduler. This replacement *may* enter the scheduler but it could also go directly to another thread. It becomes the client's *responsibility* to dispatch to the replacement with `switchto_fiber`:

1. `int raw_pause_fiber(jmp_buf*)`
2. `worker* get_replacement(worker*, jmp_buf*)`
3. `switchto_fiber(worker* from, worker* to)` OR `rollback_pause(worker* from, worker* to)`

The protocol is that calling (1) by itself is fine, but after calling (2), one of the options in (3) *must* be called to restore the worker to a good state (Figure 5). If the latter (`rollback_pause`) is chosen, that simply rolls back the state of the current thread and current worker to before the call sequence began at (1). The general pattern for using the fine-grained API is:

```
{
  jmp_buf ctx; // Stack allocated.
  val = raw_pause_fiber(&ctx);
  if (!val) { // Going to sleep...
    w = get_tls_worker_fast(); // Current worker.
    replacement = get_replacement(w, &ctx);
    // <- Store a pointer to w somewhere for wakeup
    if (condition())
      switchto_fiber(w, replacement);
    else
      rollback_pause(w, replacement);
  }
  // Woken up when we reach here.
}
```

In this API we can also see the second way in which we place additional obligations on the client: `raw_pause_fiber` also takes a `jmp_buf*` argument. The principle here is the same as with the IVar's waitlist—each blocked worker has a full stack, so it is possible to avoid dynamic memory allocation by making good use of this stack space, including, in this case, stack-allocating the `jmp_buf` that will enable the fiber to later resume. Thus all paused stacks store their own register context for later reenabling them after `wakeup_fiber` is called.

This optimized, fine-grained version of the pausing API is what we use to implement our current IVar and Cilk IO libraries which are evaluated in the next section. The full implementation of IVar reading and writing using this API is shown in the appendix, Figures 11 and 12.

6. Evaluation

Because Concurrent Cilk proposes a new API, it is not sufficient to run an existing suite of Cilk benchmarks. Thus to evaluate Concurrent Cilk we examine each of its (potential) pros and cons, and design an experiment to test that feature.

- Possible Con: overhead on applications that don't use Concurrent Cilk.
- Possible Pro: lower fork overhead than eager lightweight threading packages.
- Possible Pro: sync elision – express non-fork-join dependence structures
- Possible Pro: better utilization of cores
- Possible Pro: simpler programming model with uniform construct for spawning tasks and threads.

In this section, we characterize the overhead of Concurrent Cilk's extensions to the Cilk runtime through several scheduling microbenchmarks. We further compare the performance and scalability of Concurrent Cilk's blocking, context-switching and unblocking mechanisms through a performance shootout with other task runtime systems. Finally, we evaluate the performance of the Cilk IO library using an HTTP server microbenchmark.

The overhead tests in Section 6.1 and Section 6.3.1 were run on a quad socket system with Westmere Intel Xeon (E7-4830, 24M Cache) processors, each with 8 cores running at 2.13GHz, hyperthreading disabled. The compiler used was ICC version 13.0.0 on optimize level 3, on Redhat 4.4.7-3 with kernel version 2.6.32-358.0.1. For the scheduler microbenchmarks (Section 6.2) and the server experiments (Section 6.4), our experimental setup involved a cluster of 16 Dell PowerEdge R720 nodes, each equipped with two 8-core 2.6GHz Intel Xeon E5-2670 processors (16 cores

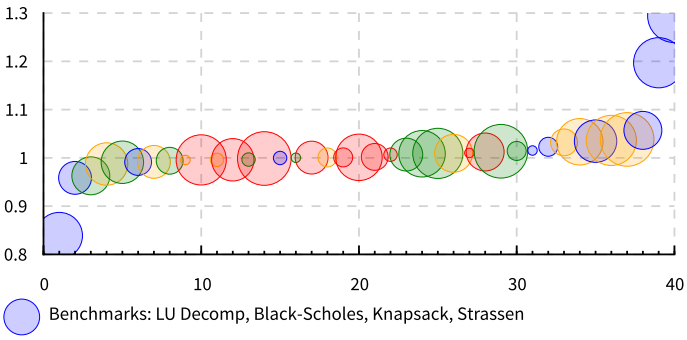

Figure 6. The overhead of adding Concurrent Cilk to the Cilk scheduler. The Y axis is the speedup/slowdown factor (the higher the better), and the X axis is the count of benchmarks. Each color represents one of the benchmarks from the set of regression tests, and for each benchmark there is a different bubble representing a different thread setting. The larger the bubble, the larger the number of threads.

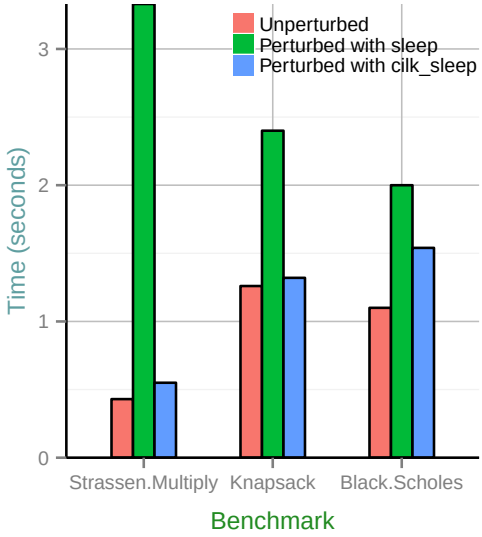

Figure 7. The effect of perturbing existing computational kernels with simulated network dependencies. We sleep on a timer (either the OS thread or using epoll through the Cilk IO library) to simulate these network dependencies. Perturbations are random, and calibrated to happen for 50% of total *CPU* time.

in total), and 32GB memory. The nodes were interconnected using 10G Ethernet connected to a Dell PowerConnect 8024F switch. The operating system used was Ubuntu Linux 12.04.5 with kernel version 3.2.0.

6.1 Overhead of Concurrent Cilk modifications

In modifying the Cilk runtime, the first principle is "do no harm"—have we incurred overhead for existing Cilk programs that do not pause fibers? In order to measure this overhead, we ran a series of existing Cilk benchmarks both with and without the Concurrent Cilk code in the runtime, scheduler loop, and work-stealing code path.

- `LU Decomp`: LU decomposition of a 2048×2048 matrix.
- `Strassen`: Strassen's algorithm for matrix multiplication on 2048×2048 matrices.
- `Black-Scholes`: Computes the Black-Scholes option pricing algorithm.
- `Knapsack`: Solve the 0-1 knapsack problem on 30 items using branch and bound.

The results of these benchmarks, as summarized in Figure 6, show that the slowdown to regular Cilk programs due to the added functionality of Concurrent Cilk, are a geometric mean of 1.1%, with all but two benchmark configurations showing no overhead throughout. Note that in this plot, each different thread setting is considered a different benchmark instance.

Finally, as a variation on these traditional benchmarks, in Figure 7, we inject simulated network IO into the middle the parallel regions of each program. This models the situations described at the outset of this paper (a ray-tracer that has to fetch network data or do RPCs). The version using the Cilk IO library is able to hide the latency of "network" operations, keeping cores busy. Here, `cilk_sleep` is provided by the Cilk IO library to block only the fiber, while keeping the core busy (just as with `cilk_read`).

What is surprising is that, on Strassen, the version that perturbs Cilk by knocking out a pthread (true `sleep` rather than `cilk_sleep`), slows down the total runtime by *more* than would be predicted based on the total volume of blocked time and compute time. The problem is that with random injection of these "network" dependencies, sometimes the blocked region increases the critical path of the program in a way parallelism does not compensate for.

6.2 Scheduling Microbenchmarks

The parallel Fibonacci algorithm (Section 1) is a widely used microbenchmark for testing scheduler overhead, being that it does very little work per spawned function. Cilk is known for its low-overhead spawns, with good constant factors and speedups on parallel Fibonacci in spite of the spawn density. Here we use this microbenchmark in two ways, to perform a shootout with or without using *first class synchronization variables*.

6.2.1 Shootout with first-class sync variables

More general than Cilk's strictly-nested, fork-join model is the class of parallel programming models with arbitrary task dependence DAGs and first-class synchronization variables (e.g., IVars, MVars, channels). After adding IVars, Concurrent Cilk joins that more general family. In this subsection—before comparing against restricted many-tasking libraries—we first examine this more expressive

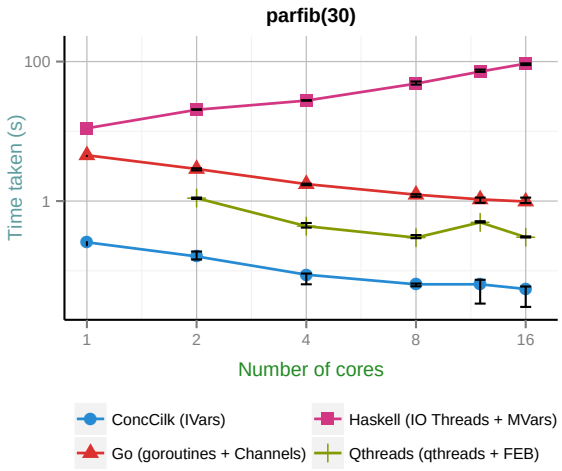

Figure 8. Error bars show min/max over three trials. A comparison of lightweight threaded runtimes with parallel fibonacci implemented by blocking on a **first-class synchronization object**: channels in the case of Go, SyncVars in Qthread, MVars in Haskell, and IVars in Concurrent Cilk. Concurrent Cilk does significantly better in this arena, because of its core strength: lazy thread creation. While each synchronization on an IVar *could* block, not all of them do, so by optimistically waiting until a blocking event to allocate resources to a thread, Concurrent Cilk is more efficient than these alternatives.

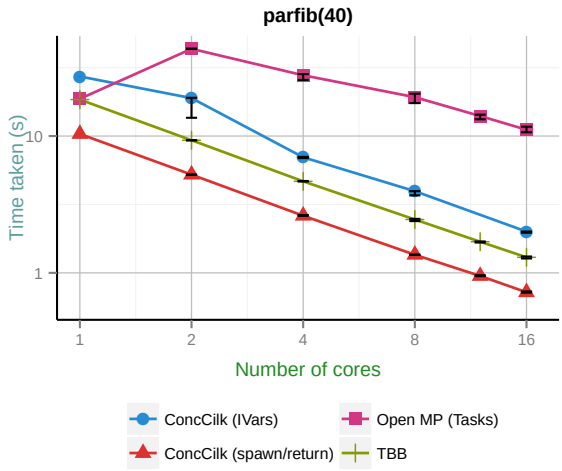

Figure 9. Restricted task-only libraries' performance on parallel fibonacci, with Concurrent Cilk (IVars) included for comparison. The IVar-based version does quite well here in spite of blocking threads on reads—it scales as well as TBB and raw Cilk (spawn/return), and outperforms Open MP. Error bars show min/max over three trials.

class of schedulers by itself. That is, we compare implementations of `parfib` in which data is returned only via first-class synchronization variables, and which every spawned computation is at least *potentially* a blockable thread. Figure 8 shows this comparison.

6.2.2 Shootout with task runtimes

Again, the best-in-class performance for low-overhead parallel function calls goes to languages and runtimes like traditional Cilk. Figure 9 shows common task-parallel libraries compared against two different implementations running on the Concurrent Cilk runtime: the first is a traditional fork-join parfib running on Concurrent Cilk using `cilk_spawn` and return results simply with `return`/`cilk_sync` rather than through IVars. The second, is the same implementation of parfib but using IVars—instead of syncs–to enforce data-dependencies.

Note that this graph runs a much larger input size (40 rather than 30), which is due to the fact that the multi-threading rather than multi-tasking runtimes cannot scale to nearly the same size of inputs. (In fact, they can exceed maximum-thread limits and crash!) In this plot we see that while the Concurrent Cilk/IVar implementation cannot keep up with TBB or traditional Cilk, the gap is *much* smaller than it would be with Qthreads, Go, or Haskell.

6.3 "Sync elision" and exposing parallelism

In this set of benchmarks we examine the potential effects on performance of enabling *unrestricted* program schedules not normally possible in a strictly fork-join model. The most clear-cut example of a place where scheduling is over-constrained by Cilk is when we have a producer and a consumer separated by a sync:

```
cilk_spawn producer();
cilk_sync;
cilk_spawn consumer();
```

The `producer` and `consumer` may or may not contain enough parallelism to fill the machine, but because of the `cilk_sync`, there is no possibility of pipeline parallelism between producer and consumer[11]. In Table 2, we examine a simple case of this pipeline parallelism opportunity: a sequential producer that fills an array. We see that Concurrent Cilk in this case allows simply *deleting* the `cilk_sync` statement which makes the program 37% faster. This sort of example could be generalized to more traditional stream processing by replacing the array of ivars with a bounded queue.

6.3.1 Exposing more parallelism: Wavefront

The topic of removing syncs to increase performance has received some previous attention, and in particular the Nabbit project [3] built an explicit task-DAG scheduler on top of Cilk, demonstrating its benefits on a *wavefront* benchmark.

[11] However, the specific, narrow case of linear, synchronous dataflow graphs is addressed by recent work on extending Cilk with pipeline parallelism via a new looping construct [10].

	produce/sync/consume	sync elision
time in seconds	0.6356	0.3981

Table 2. Time in seconds required to fill and then read an array of $10,000$ IVars for $1,000$ iterations. In traditional Cilk, there must be a `sync` interposed between the producer and the consumer. With Concurrent Cilk and IVars, *sync elison* is possible, introducing a benevolent race between the producer and consumer. If the consumer gets ahead, it blocks on an unavailable IVar, allowing the producer to catch up. By overlapping the producer and consumer phase, performance is improved.

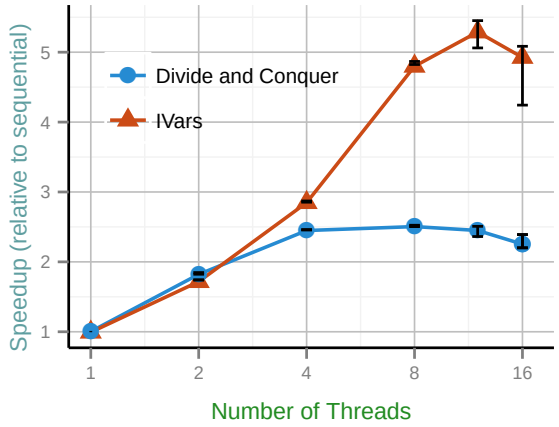

Figure 10. A wavefront algorithm ran into two modes: first, in a divide-and-conquer recursive structure that divides the matrix into quadrants, executing the NW sequentially, and the NE and SW in parallel. The second mode is to simply fork a computation for each tile, and let IVars track the inter-tile data dependencies.

Concurrent Cilk is a different tool than Nabbit in that it allows true continuation capture rather than explicit registration of callbacks (i.e., a manual form of *continuation passing style* which is a frequent source of complaints in, e.g., JavaScript web programming). In Figure 10, we can see the speedup enabled on a relatively *coarse* grained wavefront computation (16x16 matrix of inner data structures of size 512x512). In order to stress memory bandwidth, we tested a kernel with low arithmetic intensity, which reads and writes a full tile at every point in the computation. The purpose is to reproduce the effect observed in Nabbit—that more memory-reuse friendly schedules are produced by following data-dependencies than by obeying a fork-join structure. Further, because the granularity is fairly coarse, there is not much parallelism to waste in this application, and it is important to expose all of it.

6.4 Servers with per-client parallel compute

A server that performs computations on behalf of a client is an instance of *nested parallelism*:

- Parallelism between clients ("outer loop")
- Parallelism within the requested work for one client ("inner loop")

To be robust against both extremes—a single client with a large work item, and many small client requests—the Cilk approach to nested data parallelism would seem ideal. However, there's a drawback. In the server case, the outer loop includes blocking communication: to accept client connections, and then to send data to and receive data from each client.

The simplest way to program such a server is to use the same mechanism for parallelism at both levels: either `pthread_create` or `cilk_spawn`. Yet both of these implementations expose a problem. Forking too many pthreads can slow down or crash the application, whereas traditional Cilk spawns do not prevent underutilization when blocking calls are made (and blocking calls underneath a `cilk_spawn` can even be seen as a semantically incorrect contract violation).

In this experiment, we use an arbitrary parallel workload as the per-client request: compute parallel fibonnacci of 40 or 30, bottoming out to a sequential implementation below $fib(10)$, and taking about 600ms and 4ms, respectively, when executed on one core. The important thing is that there is enough work to keep all cores busy, *even* with a single concurrent client.

We consider different strategies corresponding to how the outer/inner loop is handled. Thus "Conc cilk/cilk" uses Concurrent Cilk spawns at both levels, with `cilk_accept`, `cilk_recv`, and `cilk_send` in place of the regular system calls. In contrast, "cilk/cilk" uses spawn at both levels, but regular system calls (i.e. it makes no use of Concurrent Cilk). Likewise "pthread/seq" spawns one pthread per client, but runs the inner computation sequentially. As we see in Table 3, pthread/seq is a perfectly reasonable strategy when there are enough clients. But when there is only a single client at a time, Cilk variants perform much better because they can utilize all cores even for one client. Likewise, Concurrent Cilk narrowly beats Cilk (35 vs. 32 requests per second), based on keeping all cores utilized. Of course, "pthread/pthread" cannot scale far due to limitations in OS thread scaling.

7. Related Work

In this section we consider Concurrent Cilk in the context of recent languages designed with concurrency/parallelism in mind: *e.g.* Go [8], Manticore [5], Concurrent ML [16], and Haskell. Haskell IO threads, for example, share one or more OS threads *unless* a blocking foreign function call is encountered [12], in which case more OS threads are recruited on demand. Likewise,"goroutines" in Go will share a fixed number of OS threads unless a thread makes a block-

variant	# conc clients	work-per request	throughput requests/s
pthread/seq	1	fib(40)	2.53
	4	fib(40)	9
	8	fib(40)	18
cilk/cilk	1	fib(40)	33
	4	fib(40)	33
	8	fib(40)	35
conc cilk/cilk	1	fib(40)	35
	4	fib(40)	35
	8	fib(40)	35
pthread/seq	8	fib(30)	1891
cilk/cilk	8	fib(30)	1690
conc cilk/cilk	8	fib(30)	1656
pthread/pthread	1	fib(30)	0.48
	4	fib(30)	0.12
	8	fib(30)	died

Table 3. Throughput for different numbers of clients for alternate server implementation strategies at differing server workloads.

ing call[12]. These systems implement lightweight threads in part by specializing the stack representation; for example Go uses a segmented stack representation, heap-allocating a small stack to start and growing as needed [8]. Thus, Go and Haskell (and Manticore, CML, etc) can spawn hundreds of thousands or millions of threads. Specifically, Go or Haskell can execute `parfib(30)`—using a forked thread in place of `cilk_spawn`, and a channel to communicate results back—in 4.7s and 3.1s respectively on a typical desktop[13]. This represents 1.3 million forked threads. But the programs also take 2.6Gb and 1.43Gb of memory, respectively! Also, as seen in Figure 8, Concurrent Cilk supports the same program with the same semantics (first class sync vars and suspendable threads) at much higher performance.

MultiMLton—a whole program compiler for a parallel dialect of SML—is a recent system which employs a lazy thread creation technique called *parasitic threads* [18]. These leverage relocatable stack frames to execute forked threads immediately inside the callers stack, moving them lazily only if necessary. This technique is effective, but not applicable to C/C++ where stack frames are non-relocatable.

8. Conclusions and Future Work

We've shown how, even with the constraint of legacy language support (C/C++ with linear stacks) and the complications of a mature parallel runtime system (Cilk Plus), lazy thread creation can still be an appealing prospect. Implementing it for Cilk Plus required only a couple points of contact with the existing scheduler code. Most of the complexity falls in higher level libraries, such as our IVar and Cilk IO libraries.

In future work, we plan to continue building high-level concurrent data structures and control constructs on top of the simple pause/resume fiber interface. As we saw in Section 6, IVars are already sufficient to speed up some programs with data-driven control-flow in a non-fork-join topology, and the Cilk IO library is sufficient to build server applications that mix concurrency and implicit parallelism.

[12] A difference between these two is that Haskell manages regular IO centrally on a pool of system threads, a design based on [**?**]. Thus 1000 threads that call `printf` will result in 1000 OS threads in Go [**?**], but not in Haskell.

[13] Using all four cores of an Intel Westmere processor (i5-2400 at 3.10GHz), 4Gb memory, Linux 2.6.32, GHC 7.4.2 and Go 1.0.3.

References

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

[2] Intel Cilk Plus Application Binary Interface Specification. `https://www.cilkplus.org/sites/default/files/open_specifications/CilkPlusABI_1.1.pdf`.

[3] K. Agrawal, C. Leiserson, and J. Sukha. Executing task graphs using work-stealing. In *IPDPS*, pages 1–12, Apr. 2010.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30:207–216, August 1995.

[5] M. Fluet, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Manticore: a heterogeneous parallel language. In *2007 workshop on Declarative aspects of multicore programming*, DAMP '07, pages 37–44, New York, NY, USA, 2007. ACM.

[6] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, May 1998.

[7] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy threads: Implementing a fast parallel call. *J. of Parallel and Distributed Computing*, 37(1):5–20, 1996.

[8] Google. The Go Programming Language. `https://golang.org`.

[9] C. T. Haynes, D. P. Friedman, and M. Wand. Obtaining coroutines with continuations. *Computer Languages*, 11(3.4):143–153, 1986.

[10] I. Lee, T. Angelina, C. E. Leiserson, T. B. Schardl, J. Sukha, and Z. Zhang. On-the-fly pipeline parallelism. In *Proceedings of the 25th ACM symposium on Parallelism in algorithms and architectures*, pages 140–151. ACM, 2013.

[11] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. *SIGPLAN Not.*, 44:227–242, Oct. 2009.

[12] S. Marlow, S. P. Jones, and W. Thaller. Extending the haskell foreign function interface with concurrency. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 22–32. ACM, 2004.

[13] S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore haskell. In *International Conference on Functional Programming*, ICFP '09, pages 65–78, New York, NY, USA, 2009. ACM.

[14] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

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

[16] J. H. Reppy. Concurrent ml: Design, application and semantics. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 165–198. Springer, 1993.

[17] T. Rompf, I. Maier, and M. Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. *SIGPLAN Not.*, 44:317–328, Aug. 2009.

[18] K. Sivaramakrishnan, L. Ziarek, R. Prasad, and S. Jagannathan. Lightweight asynchrony using parasitic threads. In *Workshop on Declarative Aspects of Multicore Programming*, DAMP '10, pages 63–72, New York, NY, USA, 2010. ACM.

[19] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. *SIGOPS Oper. Syst. Rev.*, 37(5):268–281, Oct. 2003.

[20] K. B. Wheeler, R. C. Murphy, and D. Thain. Qthreads: An api for programming with millions of lightweight threads. In *In IPDPS*, pages 1–8. IEEE, 2008.

Appendix

This appendix includes full code listings for `ivar_put` and `ivar_get` using the optimized, three-stage pausing API.

```
inline ivar_payload_t ivar_get(ivar *iv)
{
  worker *w, *replacement;
  unsigned short exit = 0;
  uintptr_t val;
  jmp_buf ctx;
  uintptr_t volatile peek;
  ivar_payload_t my_payload;

  cons_t my_waitlist_cell, *cur_cell;

  //fast path -- already got a value.
  //---------------------------------
  if (IVAR_READY(*ivar)) return UNTAG(*ivar);

  //slow path -- must block until a value is available.
  //---------------------------------
  val = (uintptr_t) raw_pause_fiber(&ctx);

  if (! val) {
    // Get current worker from thread-local storage:
    w = get_tls_worker_fast();
    my_waitlist_cell.car = w;
    my_payload = TAG(&my_waitlist_cell, CILK_IVAR_PAUSED);
    replacement = commit_pause(w, &ctx);

    while (!exit) {
      peek = *ivar;
      switch (peek & IVAR_MASK) {
        case CILK_IVAR_EMPTY:
          my_waitlist_cell.cdr = NULL;
          exit = cas(ivar, 0, my_payload);
          break;
        case CILK_IVAR_PAUSED:
          cur_cell = (cons_t *)UNTAG(peek);
          my_waitlist_cell.cdr = cur_cell;
          exit = cas(ivar, peek, my_payload);
          break;
        case CILK_IVAR_FULL:
          // nevermind... someone filled it.
          roll_back_pause(w, replacement);
          return UNTAG(*ivar);
          break; //go around again
      }
    }

    // Thread local array operation, no lock needed:
    switchto_fiber(w, replacement);
    ASSERT(0); // Unreachable.
  }
  return UNTAG(*ivar);
}
```

Figure 11. Full, optimized implementation of blocking IVar reads.

```
void ivar_put(ivar* iv, ivar_payload_t val)
{
  worker *w;
  unsigned short exit = 0;
  ivar_payload_t new_val = TAG(val, CILK_IVAR_FULL);
  ivar_payload_t old_val;
  ivar_payload_t volatile peek;
  cons_t *waitlist = NULL;

  while (!exit) {
    peek = *ivar;
    switch (peek & IVAR_MASK) {
      case CILK_IVAR_PAUSED:
        old_val = cas(ivar, peek, new_val);
        if (old_val != peek) break;
        // We own the whole waitlist here:
        waitlist = (cons_t *) UNTAG(old_val);
        while ( waitlist != NULL ) {
          w = waitlist->car;
          enqueue(w->readylist, (ELEMENT_TYPE) w);
          waitlist = waitlist->cdr;
        }
        exit = 1;
        break;
      case CILK_IVAR_EMPTY:
        old_val = cas(ivar, peek, new_val);
        if ((*ivar & IVAR_MASK) == CILK_IVAR_FULL)
          exit = 1;
        break;
      default:
        raise("Attempted␣multiple␣puts␣on␣an␣IVar");
    }
  }
}
```

Figure 12. Full, optimized implementation of non-blocking IVar writes.