The Implementation of the Cilk-5 Multithreaded Language

Matteo Frigo   Charles E. Leiserson   Keith H. Randall

MIT Laboratory for Computer Science
545 Technology Square
Cambridge, Massachusetts 02139
{athena,cel,randall}@lcs.mit.edu

Abstract

The fifth release of the multithreaded language Cilk uses a provably good "work-stealing" scheduling algorithm similar to the first system, but the language has been completely redesigned and the runtime system completely reengineered. The efficiency of the new implementation was aided by a clear strategy that arose from a theoretical analysis of the scheduling algorithm: concentrate on minimizing overheads that contribute to the work, even at the expense of overheads that contribute to the critical path. Although it may seem counterintuitive to move overheads onto the critical path, this "work-first" principle has led to a portable Cilk-5 implementation in which the typical cost of spawning a parallel thread is only between 2 and 6 times the cost of a C function call on a variety of contemporary machines. Many Cilk programs run on one processor with virtually no degradation compared to equivalent C programs. This paper describes how the work-first principle was exploited in the design of Cilk-5's compiler and its runtime system. In particular, we present Cilk-5's novel "two-clone" compilation strategy and its Dijkstra-like mutual-exclusion protocol for implementing the ready deque in the work-stealing scheduler.

Keywords

Critical path, multithreading, parallel computing, programming language, runtime system, work.

1 Introduction

Cilk is a multithreaded language for parallel programming that generalizes the semantics of C by introducing linguistic constructs for parallel control. The original Cilk-1 release [3, 4, 18] featured a provably efficient, randomized, "work-stealing" scheduler [3, 5], but the language was clumsy, because parallelism was exposed "by hand" using explicit continuation passing. The Cilk language implemented by our latest Cilk-5 release [8] still uses a theoretically efficient scheduler, but the language has been simplified considerably. It employs call/return semantics for parallelism and features a linguistically simple "inlet" mechanism for nondeterministic control. Cilk-5 is designed to run efficiently on contemporary symmetric multiprocessors (SMP's), which feature hardware support for shared memory. We have coded many applications in Cilk, including the ⋆Socrates and Cilkchess chess-playing programs which have won prizes in international competitions.

The philosophy behind Cilk development has been to make the Cilk language a true parallel extension of C, both semantically and with respect to performance. On a parallel computer, Cilk control constructs allow the program to execute in parallel. If the Cilk keywords for parallel control are elided from a Cilk program, however, a syntactically and semantically correct C program results, which we call the ***C elision*** (or more generally, the ***serial elision***) of the Cilk program. Cilk is a ***faithful*** extension of C, because the C elision of a Cilk program is a correct implementation of the semantics of the program. Moreover, on one processor, a parallel Cilk program "scales down" to run nearly as fast as its C elision.

Unlike in Cilk-1, where the Cilk scheduler was an identifiable piece of code, in Cilk-5 both the compiler and runtime system bear the responsibility for scheduling. To obtain efficiency, we have, of course, attempted to reduce scheduling overheads. Some overheads have a larger impact on execution time than others, however. A theoretical understanding of Cilk's scheduling algorithm [3, 5] has allowed us to identify and optimize the common cases. According to this abstract theory, the performance of a Cilk computation can be characterized by two quantities: its ***work***, which is the total time needed to execute the computation serially, and its ***critical-path length***, which is its execution time on an infinite number of processors. (Cilk provides instrumentation that allows a user to measure these two quantities.) Within Cilk's scheduler, we can identify a given cost as contributing to either work overhead or critical-path overhead. Much of the efficiency of Cilk derives from the following principle, which we shall justify in Section 3.

This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Grant N00014-94-1-0985. Computing facilities were provided by the MIT Xolas Project, thanks to a generous equipment donation from Sun Microsystems.

To appear in *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, June 1998.

The work-first principle: *Minimize the scheduling overhead borne by the work of a computation. Specifically, move overheads out of the work and onto the critical path.*

The work-first principle played an important role during the design of earlier Cilk systems, but Cilk-5 exploits the principle more extensively.

The work-first principle inspired a "two-clone" strategy for compiling Cilk programs. Our `cilk2c` compiler [23] is a type-checking, source-to-source translator that transforms a Cilk source into a C postsource which makes calls to Cilk's runtime library. The C postsource is then run through the `gcc` compiler to produce object code. The `cilk2c` compiler produces two clones of every Cilk procedure—a "fast" clone and a "slow" clone. The fast clone, which is identical in most respects to the C elision of the Cilk program, executes in the common case where serial semantics suffice. The slow clone is executed in the infrequent case that parallel semantics and its concomitant bookkeeping are required. All communication due to scheduling occurs in the slow clone and contributes to critical-path overhead, but not to work overhead.

The work-first principle also inspired a Dijkstra-like [11], shared-memory, mutual-exclusion protocol as part of the runtime load-balancing scheduler. Cilk's scheduler uses a "work-stealing" algorithm in which idle processors, called ***thieves***, "steal" threads from busy processors, called ***victims***. Cilk's scheduler guarantees that the cost of stealing contributes only to critical-path overhead, and not to work overhead. Nevertheless, it is hard to avoid the mutual-exclusion costs incurred by a potential victim, which contribute to work. To minimize work overhead, instead of using locking, Cilk's runtime system uses a Dijkstra-like protocol, which we call the ***THE*** protocol, to manage the runtime deque of ready threads in the work-stealing algorithm. An added advantage of the THE protocol is that it allows an exception to be signaled to a working processor with no additional work overhead, a feature used in Cilk's abort mechanism.

The remainder of this paper is organized as follows. Section 2 overviews the basic features of the Cilk language. Section 3 justifies the work-first principle. Section 4 describes how the two-clone strategy is implemented, and Section 5 presents the THE protocol. Section 6 gives empirical evidence that the Cilk-5 scheduler is efficient. Finally, Section 7 presents related work and offers some conclusions.

2 The Cilk language

This section presents a brief overview of the Cilk extensions to C as supported by Cilk-5. (For a complete description, consult the Cilk-5 manual [8].) The key features of the language are the specification of parallelism and synchronization, through the `spawn` and `sync` keywords, and the specification of nondeterminism, using `inlet` and `abort`.

```
#include <stdlib.h>
#include <stdio.h>
#include <cilk.h>

cilk int fib (int n)
{
    if (n<2) return n;
    else {
        int x, y;
        x = spawn fib (n-1);
        y = spawn fib (n-2);
        sync;
        return (x+y);
    }
}

cilk int main (int argc, char *argv[])
{
    int n, result;
    n = atoi(argv[1]);
    result = spawn fib(n);
    sync;
    printf ("Result: %d\n", result);
    return 0;
}
```

Figure 1: A simple Cilk program to compute the nth Fibonacci number in parallel (using a very bad algorithm).

The basic Cilk language can be understood from an example. Figure 1 shows a Cilk program that computes the nth Fibonacci number.[1] Observe that the program would be an ordinary C program if the three keywords `cilk`, `spawn`, and `sync` are elided.

The keyword `cilk` identifies `fib` as a ***Cilk procedure***, which is the parallel analog to a C function. Parallelism is created when the keyword `spawn` precedes the invocation of a procedure. The semantics of a spawn differs from a C function call only in that the parent can continue to execute in parallel with the child, instead of waiting for the child to complete as is done in C. Cilk's scheduler takes the responsibility of scheduling the spawned procedures on the processors of the parallel computer.

A Cilk procedure cannot safely use the values returned by its children until it executes a `sync` statement. The `sync` statement is a local "barrier," not a global one as, for example, is used in message-passing programming. In the Fibonacci example, a `sync` statement is required before the statement `return (x+y)` to avoid the anomaly that would occur if `x` and `y` are summed before they are computed. In addition to explicit synchronization provided by the `sync` statement, every Cilk procedure syncs implicitly before it returns, thus ensuring that all of its children terminate before it does.

Ordinarily, when a spawned procedure returns, the returned value is simply stored into a variable in its parent's frame:

[1]This program uses an inefficient algorithm which runs in exponential time. Although logarithmic-time methods are known [9, p. 850], this program nevertheless provides a good didactic example.

```
cilk int fib (int n)
{
    int x = 0;
    inlet void summer (int result)
    {
        x += result;
        return;
    }

    if (n<2) return n;
    else {
        summer(spawn fib (n-1));
        summer(spawn fib (n-2));
        sync;
        return (x);
    }
}
```

Figure 2: Using an inlet to compute the nth Fibonnaci number.

```
x = spawn foo(y);
```

Occasionally, one would like to incorporate the returned value into the parent's frame in a more complex way. Cilk provides an ***inlet*** feature for this purpose, which was inspired in part by the inlet feature of TAM [10].

An inlet is essentially a C function internal to a Cilk procedure. In the normal syntax of Cilk, the spawning of a procedure must occur as a separate statement and not in an expression. An exception is made to this rule if the spawn is performed as an argument to an inlet call. In this case, the procedure is spawned, and when it returns, the inlet is invoked. In the meantime, control of the parent procedure proceeds to the statement following the inlet call. In principle, inlets can take multiple spawned arguments, but Cilk-5 has the restriction that exactly one argument to an inlet may be spawned and that this argument must be the first argument. If necessary, this restriction is easy to program around.

Figure 2 illustrates how the `fib()` function might be coded using inlets. The inlet `summer()` is defined to take a returned value `result` and add it to the variable `x` in the frame of the procedure that does the spawning. All the variables of `fib()` are available within `summer()`, since it is an internal function of `fib()`.[2]

No lock is required around the accesses to `x` by `summer`, because Cilk provides atomicity implicitly. The concern is that the two updates might occur in parallel, and if atomicity is not imposed, an update might be lost. Cilk provides implicit atomicity among the "threads" of a procedure instance, where a ***thread*** is a maximal sequence of instructions ending with a `spawn`, `sync`, or `return` (either explicit or implicit) statement. An inlet is precluded from containing `spawn` and `sync` statements, and thus it operates atomically as a single thread. Implicit atomicity simplifies reasoning about concurrency and nondeterminism without requiring locking, declaration of critical regions, and the like.

[2]The C elision of a Cilk program with inlets is not ANSI C, because ANSI C does not support internal C functions. Cilk is based on Gnu C technology, however, which does provide this support.

Cilk provides syntactic sugar to produce certain commonly used inlets implicitly. For example, the statement `x += spawn fib(n-1)` conceptually generates an inlet similar to the one in Figure 2.

Sometimes, a procedure spawns off parallel work which it later discovers is unnecessary. This "speculative" work can be aborted in Cilk using the `abort` primitive inside an inlet. A common use of `abort` occurs during a parallel search, where many possibilities are searched in parallel. As soon as a solution is found by one of the searches, one wishes to abort any currently executing searches as soon as possible so as not to waste processor resources. The `abort` statement, when executed inside an inlet, causes all of the already-spawned children of the procedure to terminate.

We considered using "futures" [19] with implicit synchronization, as well as synchronizing on specific variables, instead of using the simple `spawn` and `sync` statements. We realized from the work-first principle, however, that different synchronization mechanisms could have an impact only on the critical-path of a computation, and so this issue was of secondary concern. Consequently, we opted for implementation simplicity. Also, in systems that support relaxed memory-consistency models, the explicit `sync` statement can be used to ensure that all side-effects from previously spawned subprocedures have occurred.

In addition to the control synchronization provided by `sync`, Cilk programmers can use explicit locking to synchronize accesses to data, providing mutual exclusion and atomicity. Data synchronization is an overhead borne on the work, however, and although we have striven to minimize these overheads, fine-grain locking on contemporary processors is expensive. We are currently investigating how to incorporate atomicity into the Cilk language so that protocol issues involved in locking can be avoided at the user level. To aid in the debugging of Cilk programs that use locks, we have been developing a tool called the "Nondeterminator" [7, 13], which detects common synchronization bugs called ***data races***.

3 The work-first principle

This section justifies the work-first principle stated in Section 1 by showing that it follows from three assumptions. First, we assume that Cilk's scheduler operates in practice according to the theoretical analysis presented in [3, 5]. Second, we assume that in the common case, ample "parallel slackness" [28] exists, that is, the average parallelism of a Cilk program exceeds the number of processors on which we run it by a sufficient margin. Third, we assume (as is indeed the case) that every Cilk program has a C elision against which its one-processor performance can be measured.

The theoretical analysis presented in [3, 5] cites two fundamental lower bounds as to how fast a Cilk program can run. Let us denote by T_P the execution time of a given computation on P processors. Then, the work of the computation is T_1 and its critical-path length is T_∞. For a computation with T_1 work, the lower bound $T_P \geq T_1/P$ must hold, because at most P units of work can be executed in a single step. In addition, the lower bound $T_P \geq T_\infty$ must hold, since a finite number of processors cannot execute faster than an infinite number.[3]

Cilk's randomized work-stealing scheduler [3, 5] executes a Cilk computation on P processors in expected time

$$T_P = T_1/P + O(T_\infty) \ , \tag{1}$$

assuming an ideal parallel computer. This equation resembles "Brent's theorem" [6, 15] and is optimal to within a constant factor, since T_1/P and T_∞ are both lower bounds. We call the first term on the right-hand side of Equation (1) the ***work*** term and the second term the ***critical-path*** term. Importantly, all communication costs due to Cilk's scheduler are borne by the critical-path term, as are most of the other scheduling costs. To make these overheads explicit, we define the ***critical-path overhead*** to be the smallest constant c_∞ such that

$$T_P \leq T_1/P + c_\infty T_\infty \ . \tag{2}$$

The second assumption needed to justify the work-first principle focuses on the "common-case" regime in which a parallel program operates. Define the ***average parallelism*** as $\overline{P} = T_1/T_\infty$, which corresponds to the maximum possible speedup that the application can obtain. Define also the ***parallel slackness*** [28] to be the ratio $\overline{P}/P$. The ***assumption of parallel slackness*** is that $\overline{P}/P \gg c_\infty$, which means that the number P of processors is much smaller than the average parallelism $\overline{P}$. Under this assumption, it follows that $T_1/P \gg c_\infty T_\infty$, and hence from Inequality (2) that $T_P \approx T_1/P$, and we obtain linear speedup. The critical-path overhead c_∞ has little effect on performance when sufficient slackness exists, although it does determines how much slackness must exist to ensure linear speedup.

Whether substantial slackness exists in common applications is a matter of opinion and empiricism, but we suggest that slackness is the common case. The expressiveness of Cilk makes it easy to code applications with large amounts of parallelism. For modest-sized problems, many applications exhibit an average parallelism of over 200, yielding substantial slackness on contemporary SMP's. Even on Sandia National Laboratory's Intel Paragon, which contains 1824 nodes, the $\star$Socrates chess program (coded in Cilk-1) ran in its linear-speedup regime during the 1995 ICCA World Computer Chess Championship (where it placed second in a field of 24). Section 6 describes a dozen other diverse applications which were run on an 8-processor SMP with considerable parallel slackness. The parallelisim of these applications increases with problem size, thereby ensuring they will run well on large machines.

[3] This abstract model of execution time ignores real-life details, such as memory-hierarchy effects, but is nonetheless quite accurate [4].

The third assumption behind the work-first principle is that every Cilk program has a C elision against which its one-processor performance can be measured. Let us denote by T_S the running time of the C elision. Then, we define the ***work overhead*** by $c_1 = T_1/T_S$. Incorporating critical-path and work overheads into Inequality (2) yields

$$\begin{aligned} T_P &\leq c_1 T_S/P + c_\infty T_\infty \\ &\approx c_1 T_S/P \ , \end{aligned} \tag{3}$$

since we assume parallel slackness.

We can now restate the work-first principle precisely. *Minimize c_1, even at the expense of a larger c_∞*, because c_1 has a more direct impact on performance. Adopting the work-first principle may adversely affect the ability of an application to scale up, however, if the critical-path overhead c_∞ is too large. But, as we shall see in Section 6, critical-path overhead is reasonably small in Cilk-5, and many applications can be coded with large amounts of parallelism.

The work-first principle pervades the Cilk-5 implementation. The work-stealing scheduler guarantees that with high probability, only $O(PT_\infty)$ steal (migration) attempts occur (that is, $O(T_\infty)$ on average per processor), all costs for which are borne on the critical path. Consequently, the scheduler for Cilk-5 postpones as much of the scheduling cost as possible to when work is being stolen, thereby removing it as a contributor to work overhead. This strategy of amortizing costs against steal attempts permeates virtually every decision made in the design of the scheduler.

4 Cilk's compilation strategy

This section describes how our `cilk2c` compiler generates C postsource from a Cilk program. As dictated by the work-first principle, our compiler and scheduler are designed to reduce the work overhead as much as possible. Our strategy is to generate two clones of each procedure—a ***fast*** clone and a ***slow*** clone. The fast clone operates much as does the C elision and has little support for parallelism. The slow clone has full support for parallelism, along with its concomitant overhead. We first describe the Cilk scheduling algorithm. Then, we describe how the compiler translates the Cilk language constructs into code for the fast and slow clones of each procedure. Lastly, we describe how the runtime system links together the actions of the fast and slow clones to produce a complete Cilk implementation.

As in lazy task creation [24], in Cilk-5 each processor, called a ***worker***, maintains a ***ready deque*** (doubly-ended queue) of ready procedures (technically, procedure instances). Each deque has two ends, a ***head*** and a ***tail***, from which procedures can be added or removed. A worker operates locally on the tail of its own deque, treating it much

```
 1  int fib (int n)
 2  {
 3      fib_frame *f;                 frame pointer
 4      f = alloc(sizeof(*f));        allocate frame
 5      f->sig = fib_sig;             initialize frame
 6      if (n<2) {
 7          free(f, sizeof(*f));      free frame
 8          return n;
 9      }
10      else {
11          int x, y;
12          f->entry = 1;             save PC
13          f->n = n;                 save live vars
14          *T = f;                   store frame pointer
15          push();                   push frame
16          x = fib (n-1);            do C call
17          if (pop(x) == FAILURE)    pop frame
18              return 0;             frame stolen
19          ...                       second spawn
20          ;                         sync is free!
21          free(f, sizeof(*f));      free frame
22          return (x+y);
23      }
24  }
```

Figure 3: The fast clone generated by `cilk2c` for the `fib` procedure from Figure 1. The code for the second spawn is omitted. The functions `alloc` and `free` are inlined calls to the runtime system's fast memory allocator. The signature `fib_sig` contains a description of the `fib` procedure, including a pointer to the slow clone. The `push` and `pop` calls are operations on the scheduling deque and are described in detail in Section 5.

as C treats its call stack, pushing and popping spawned activation frames. When a worker runs out of work, it becomes a ***thief*** and attempts to steal a procedure another worker, called its ***victim***. The thief steals the procedure from the head of the victim's deque, the opposite end from which the victim is working.

When a procedure is spawned, the fast clone runs. Whenever a thief steals a procedure, however, the procedure is converted to a slow clone. The Cilk scheduler guarantees that the number of steals is small when sufficient slackness exists, and so we expect the fast clones to be executed most of the time. Thus, the work-first principle reduces to minimizing costs in the fast clone, which contribute more heavily to work overhead. Minimizing costs in the slow clone, although a desirable goal, is less important, since these costs contribute less heavily to work overhead and more to critical-path overhead.

We minimize the costs of the fast clone by exploiting the structure of the Cilk scheduler. Because we convert a procedure to its slow clone when it is stolen, we maintain the invariant that a fast clone has never been stolen. Furthermore, none of the descendants of a fast clone have been stolen either, since the strategy of stealing from the heads of ready deques guarantees that parents are stolen before their children. As we shall see, this simple fact allows many optimizations to be performed in the fast clone.

We now describe how our `cilk2c` compiler generates postsource C code for the `fib` procedure from Figure 1. An example of the postsource for the fast clone of `fib` is given in Figure 3. The generated C code has the same general structure as the C elision, with a few additional statements. In lines 4–5, an ***activation frame*** is allocated for `fib` and initialized. The Cilk runtime system uses activation frames to represent procedure instances. Using techniques similar to [16, 17], our inlined allocator typically takes only a few cycles. The frame is initialized in line 5 by storing a pointer to a static structure, called a signature, describing `fib`.

The first spawn in `fib` is translated into lines 12–18. In lines 12–13, the state of the `fib` procedure is saved into the activation frame. The saved state includes the program counter, encoded as an entry number, and all live, dirty variables. Then, the frame is pushed on the runtime deque in lines 14–15.[4] Next, we call the `fib` routine as we would in C. Because the `spawn` statement itself compiles directly to its C elision, the postsource can exploit the optimization capabilities of the C compiler, including its ability to pass arguments and receive return values in registers rather than in memory.

After `fib` returns, lines 17–18 check to see whether the parent procedure has been stolen. If it has, we return immediately with a dummy value. Since all of the ancestors have been stolen as well, the C stack quickly unwinds and control is returned to the runtime system.[5] The protocol to check whether the parent procedure has been stolen is quite subtle—we postpone discussion of its implementation to Section 5. If the parent procedure has not been stolen, it continues to execute at line 19, performing the second spawn, which is not shown.

In the fast clone, all `sync` statements compile to no-ops. Because a fast clone never has any children when it is executing, we know at compile time that all previously spawned procedures have completed. Thus, no operations are required for a `sync` statement, as it always succeeds. For example, line 20 in Figure 3, the translation of the `sync` statement is just the empty statement. Finally, in lines 21–22, `fib` deallocates the activation frame and returns the computed result to its parent procedure.

The slow clone is similar to the fast clone except that it provides support for parallel execution. When a procedure is stolen, control has been suspended between two of the procedure's threads, that is, at a spawn or sync point. When the slow clone is resumed, it uses a `goto` statement to restore the program counter, and then it restores local variable state from the activation frame. A `spawn` statement is translated in the slow clone just as in the fast clone. For a `sync` statement, `cilk2c` inserts a call to the runtime system, which checks to see whether the procedure has any spawned children that have not returned. Although the parallel book-

[4]If the shared memory is not sequentially consistent, a memory fence must be inserted between lines 14 and 15 to ensure that the surrounding writes are executed in the proper order.

[5]The `setjmp`/`longjmp` facility of C could have been used as well, but our unwinding strategy is simpler.

keeping in a slow clone is substantial, it contributes little to work overhead, since slow clones are rarely executed.

The separation between fast clones and slow clones also allows us to compile inlets and abort statements efficiently in the fast clone. An inlet call compiles as efficiently as an ordinary spawn. For example, the code for the inlet call from Figure 2 compiles similarly to the following Cilk code:

```
tmp = spawn fib(n-1);
summer(tmp);
```

Implicit inlet calls, such as `x += spawn fib(n-1)`, compile directly to their C elisions. An `abort` statement compiles to a no-op just as a `sync` statement does, because while it is executing, a fast clone has no children to abort.

The runtime system provides the glue between the fast and slow clones that makes the whole system work. It includes protocols for stealing procedures, returning values between processors, executing inlets, aborting computation subtrees, and the like. All of the costs of these protocols can be amortized against the critical path, so their overhead does not significantly affect the running time when sufficient parallel slackness exists. The portion of the stealing protocol executed by the worker contributes to work overhead, however, thereby warranting a careful implementation. We discuss this protocol in detail in Section 5.

The work overhead of a `spawn` in Cilk-5 is only a few reads and writes in the fast clone—3 reads and 5 writes for the `fib` example. We will experimentally quantify the work overhead in Section 6. Some work overheads still remain in our implementation, however, including the allocation and freeing of activation frames, saving state before a spawn, pushing and popping of the frame on the deque, and checking if a procedure has been stolen. A portion of this work overhead is due to the fact that Cilk-5 is duplicating the work the C compiler performs, but as Section 6 shows, this overhead is small. Although a production Cilk compiler might be able eliminate this unnecessary work, it would likely compromise portability.

In Cilk-4, the precursor to Cilk-5, we took the work-first principle to the extreme. Cilk-4 performed stack-based allocation of activation frames, since the work overhead of stack allocation is smaller than the overhead of heap allocation. Because of the "cactus stack" [25] semantics of the Cilk stack,[6] however, Cilk-4 had to manage the virtual-memory map on each processor explicitly, as was done in [27]. The work overhead in Cilk-4 for frame allocation was little more than that of incrementing the stack pointer, but whenever the stack pointer overflowed a page, an expensive user-level interrupt ensued, during which Cilk-4 would modify the memory map. Unfortunately, the operating-system mechanisms supporting these operations were too slow and unpredictable, and the possibility of a page fault in critical sections led to complicated protocols. Even though these overheads could be charged to the critical-path term, in practice, they became so large that the critical-path term contributed significantly to the running time, thereby violating the assumption of parallel slackness. A one-processor execution of a program was indeed fast, but insufficient slackness sometimes resulted in poor parallel performance.

In Cilk-5, we simplified the allocation of activation frames by simply using a heap. In the common case, a frame is allocated by removing it from a free list. Deallocation is performed by inserting the frame into the free list. No user-level management of virtual memory is required, except for the initial setup of shared memory. Heap allocation contributes only slightly more than stack allocation to the work overhead, but it saves substantially on the critical path term. On the downside, heap allocation can potentially waste more memory than stack allocation due to fragmentation. For a careful analysis of the relative merits of stack and heap based allocation that supports heap allocation, see the paper by Appel and Shao [1]. For an equally careful analysis that supports stack allocation, see [22].

Thus, although the work-first principle gives a general understanding of where overheads should be borne, our experience with Cilk-4 showed that large enough critical-path overheads can tip the scales to the point where the assumptions underlying the principle no longer hold. We believe that Cilk-5 work overhead is nearly as low as possible, given our goal of generating portable C output from our compiler.[7] Other researchers have been able to reduce overheads even more, however, at the expense of portability. For example, lazy threads [14] obtains efficiency at the expense of implementing its own calling conventions, stack layouts, etc. Although we could in principle incorporate such machine-dependent techniques into our compiler, we feel that Cilk-5 strikes a good balance between performance and portability. We also feel that the current overheads are sufficiently low that other problems, notably minimizing overheads for data synchronization, deserve more attention.

5 Implemention of work-stealing

In this section, we describe Cilk-5's work-stealing mechanism, which is based on a Dijkstra-like [11], shared-memory, mutual-exclusion protocol called the "THE" protocol. In accordance with the work-first principle, this protocol has been designed to minimize work overhead. For example, on a 167-megahertz UltraSPARC I, the `fib` program with the THE protocol runs about 25% faster than with hardware locking primitives. We first present a simplified version of the protocol. Then, we discuss the actual implementation, which allows exceptions to be signaled with no additional overhead.

[6]Suppose a procedure A spawns two children B and C. The two children can reference objects in A's activation frame, but B and C do not see each other's frame.

[7]Although the runtime system requires some effort to port between architectures, the compiler requires no changes whatsoever for different platforms.

Several straightforward mechanisms might be considered to implement a work-stealing protocol. For example, a thief might interrupt a worker and demand attention from this victim. This strategy presents problems for two reasons. First, the mechanisms for signaling interrupts are slow, and although an interrupt would be borne on the critical path, its large cost could threaten the assumption of parallel slackness. Second, the worker would necessarily incur some overhead on the work term to ensure that it could be safely interrupted in a critical section. As an alternative to sending interrupts, thieves could post steal requests, and workers could periodically poll for them. Once again, however, a cost accrues to the work overhead, this time for polling. Techniques are known that can limit the overhead of polling [12], but they require the support of a sophisticated compiler.

The work-first principle suggests that it is reasonable to put substantial effort into minimizing work overhead in the work-stealing protocol. Since Cilk-5 is designed for shared-memory machines, we chose to implement work-stealing through shared-memory, rather than with message-passing, as might otherwise be appropriate for a distributed-memory implementation. In our implementation, both victim and thief operate directly through shared memory on the victim's ready deque. The crucial issue is how to resolve the race condition that arises when a thief tries to steal the same frame that its victim is attempting to pop. One simple solution is to add a lock to the deque using relatively heavyweight hardware primitives like Compare-And-Swap or Test-And-Set. Whenever a thief or worker wishes to remove a frame from the deque, it first grabs the lock. This solution has the same fundamental problem as the interrupt and polling mechanisms just described, however. Whenever a worker pops a frame, it pays the heavy price to grab a lock, which contributes to work overhead.

Consequently, we adopted a solution that employs Dijkstra's protocol for mutual exclusion [11], which assumes only that reads and writes are atomic. Because our protocol uses three atomic shared variables T, H, and E, we call it the ***THE*** protocol. The key idea is that actions by the worker on the tail of the queue contribute to work overhead, while actions by thieves on the head of the queue contribute only to critical-path overhead. Therefore, in accordance with the work-first principle, we attempt to move costs from the worker to the thief. To arbitrate among different thieves attempting to steal from the same victim, we use a hardware lock, since this overhead can be amortized against the critical path. To resolve conflicts between a worker and the sole thief holding the lock, however, we use a lightweight Dijkstra-like protocol which contributes minimally to work overhead. A worker resorts to a heavyweight hardware lock only when it encounters an actual conflict with a thief, in which case we can charge the overhead that the victim incurs to the critical path.

In the rest of this section, we describe the THE protocol in detail. We first present a simplified protocol that uses only two shared variables T and H designating the tail and the head of the deque, respectively. Later, we extend the protocol with a third variable E that allows exceptions to be signaled to a worker. The exception mechanism is used to implement Cilk's abort statement. Interestingly, this extension does not introduce any additional work overhead.

```
        Worker/Victim                      Thief
 1  push() {                       1  steal() {
 2    T++;                         2    lock(L);
 3  }                              3    H++;
                                   4    if (H > T) {
 4  pop() {                        5      H--;
 5    T--;                         6      unlock(L);
 6    if (H > T) {                 7      return FAILURE;
 7      T++;                       8    }
 8      lock(L);                   9    unlock(L);
 9      T--;                      10    return SUCCESS;
10      if (H > T) {              11  }
11        T++;
12        unlock(L);
13        return FAILURE;
14      }
15      unlock(L);
16    }
17    return SUCCESS;
18  }
```

Figure 4: Pseudocode of a simplified version of the THE protocol. The left part of the figure shows the actions performed by the victim, and the right part shows the actions of the thief. None of the actions besides reads and writes are assumed to be atomic. For example, `T--;` can be implemented as `tmp = T; tmp = tmp - 1; T = tmp;`.

The pseudocode of the simplified THE protocol is shown in Figure 4. Assume that shared memory is sequentially consistent [20].[8] The code assumes that the ready deque is implemented as an array of frames. The head and tail of the deque are determined by two indices T and H, which are stored in shared memory and are visible to all processors. The index T points to the first unused element in the array, and H points to the first frame on the deque. Indices grow from the head towards the tail so that under normal conditions, we have $T \geq H$. Moreover, each deque has a lock L implemented with atomic hardware primitives or with OS calls.

The worker uses the deque as a stack. (See Section 4.) Before a spawn, it pushes a frame onto the tail of the deque. After a spawn, it pops the frame, unless the frame has been stolen. A thief attempts to steal the frame at the head of the deque. Only one thief at the time may steal from the deque, since a thief grabs L as its first action. As can be seen from the code, the worker alters T but not H, whereas the thief only increments H and does not alter T.

The only possible interaction between a thief and its vic-

[8]If the shared memory is not sequentially consistent, a memory fence must be inserted between lines 5 and 6 of the worker/victim code and between lines 3 and 4 of the thief code to ensure that these instructions are executed in the proper order.

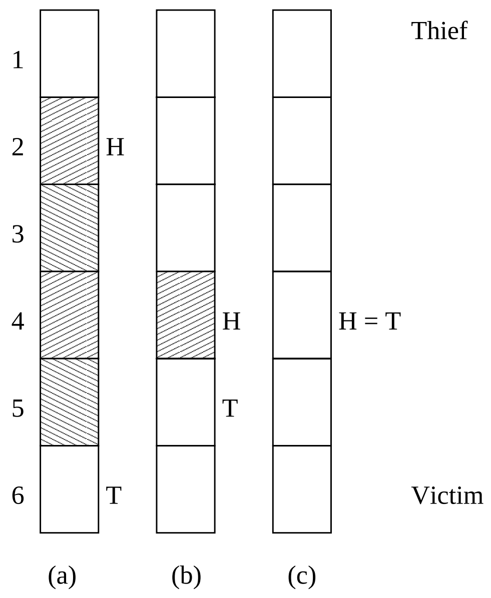

Figure 5: The three cases of the ready deque in the simplified THE protocol. A shaded entry indicates the presence of a frame at a certain position in the deque. The head and the tail are marked by T and H.

tim occurs when the thief is incrementing H while the victim is decrementing T. Consequently, it is always safe for a worker to append a new frame at the end of the deque (push) without worrying about the actions of the thief. For a pop operations, there are three cases, which are shown in Figure 5. In case (a), the thief and the victim can both get a frame from the deque. In case (b), the deque contains only one frame. If the victim decrements T without interference from thieves, it gets the frame. Similarly, a thief can steal the frame as long as its victim is not trying to obtain it. If both the thief and the victim try to grab the frame, however, the protocol guarantees that at least one of them discovers that $H > T$. If the thief discovers that $H > T$, it restores H to its original value and retreats. If the victim discovers that $H > T$, it restores T to its original value and restarts the protocol after having acquired L. With L acquired, no thief can steal from this deque so the victim can pop the frame without interference (if the frame is still there). Finally, in case (c) the deque is empty. If a thief tries to steal, it will always fail. If the victim tries to pop, the attempt fails and control returns to the Cilk runtime system. The protocol cannot deadlock, because each process holds only one lock at a time.

We now argue that the THE protocol contributes little to the work overhead. Pushing a frame involves no overhead beyond updating T. In the common case where a worker can succesfully pop a frame, the pop protocol performs only 6 operations—2 memory loads, 1 memory store, 1 decrement, 1 comparison, and 1 (predictable) conditional branch. Moreover, in the common case where no thief operates on the deque, both H and T can be cached exclusively by the worker. The expensive operation of a worker grabbing the lock L occurs only when a thief is simultaneously trying to steal the frame being popped. Since the number of steal attempts depends on T_∞, not on T_1, the relatively heavy cost of a victim grabbing L can be considered as part of the critical-path overhead c_∞ and does not influence the work overhead c_1.

We ran some experiments to determine the relative performance of the THE protocol versus the straightforward protocol in which pop just locks the deque before accessing it. On a 167-megahertz UltraSPARC I, the THE protocol is about 25% faster than the simple locking protocol. This machine's memory model requires that a memory fence instruction (membar) be inserted between lines 5 and 6 of the pop pseudocode. We tried to quantify the performance impact of the membar instruction, but in all our experiments the execution times of the code with and without membar are about the same. On a 200-megahertz Pentium Pro running Linux and gcc 2.7.1, the THE protocol is only about 5% faster than the locking protocol. On this processor, the THE protocol spends about half of its time in the memory fence.

Because it replaces locks with memory synchronization, the THE protocol is more "nonblocking" than a straightforward locking protocol. Consequently, the THE protocol is less prone to problems that arise when spin locks are used extensively. For example, even if a worker is suspended by the operating system during the execution of pop, the infrequency of locking in the THE protocol means that a thief can usually complete a steal operation on the worker's deque. Recent work by Arora et al. [2] has shown that a completely nonblocking work-stealing scheduler can be implemented. Using these ideas, Lisiecki and Medina [21] have modified the Cilk-5 scheduler to make it completely nonblocking. Their experience is that the THE protocol greatly simplifies a nonblocking implementation.

The simplified THE protocol can be extended to support the signaling of exceptions to a worker. In Figure 4, the index H plays two roles: it marks the head of the deque, and it marks the point that the worker cannot cross when it pops. These places in the deque need not be the same. In the full THE protocol, we separate the two functions of H into two variables: H, which now only marks the head of the deque, and E, which marks the point that the victim cannot cross. Whenever $E > T$, some exceptional condition has occurred, which includes the frame being stolen, but it can also be used for other exceptions. For example, setting $E = \infty$ causes the worker to discover the exception at its next pop. In the new protocol, E replaces H in line 6 of the worker/victim. Moreover, lines 7–15 of the worker/victim are replaced by a call to an ***exception handler*** to determine the type of exception (stolen frame or otherwise) and the proper action to perform. The thief code is also modified. Before trying to

Program	*Size*	T_1	T_∞	$\overline{P}$	c_1	T_8	T_1/T_8	T_S/T_8
`fib`	35	12.77	0.0005	25540	3.63	1.60	8.0	2.2
`blockedmul`	1024	29.9	0.0044	6730	1.05	4.3	7.0	6.6
`notempmul`	1024	29.7	0.015	1970	1.05	3.9	7.6	7.2
`strassen`	1024	20.2	0.58	35	1.01	3.54	5.7	5.6
*`cilksort`	$4,100,000$	5.4	0.0049	1108	1.21	0.90	6.0	5.0
†`queens`	22	150.	0.0015	96898	0.99	18.8	8.0	8.0
†`knapsack`	30	75.8	0.0014	54143	1.03	9.5	8.0	7.7
`lu`	2048	155.8	0.42	370	1.02	20.3	7.7	7.5
*`cholesky`	BCSSTK32	1427.	3.4	420	1.25	208.	6.9	5.5
`heat`	4096×512	62.3	0.16	384	1.08	9.4	6.6	6.1
`fft`	2^{20}	4.3	0.0020	2145	0.93	0.77	5.6	6.0
`Barnes-Hut`	2^{16}	124.	0.15	853	1.02	16.5	7.5	7.4

Figure 6: The performance of example Cilk programs. Times are in seconds and are accurate to within about 10%. The serial programs are C elisions of the Cilk programs, except for those programs that are starred (*), where the parallel program implements a different algorithm than the serial program. Programs labeled by a dagger (†) are nondeterministic, and thus, the running time on one processor is not the same as the work performed by the computation. For these programs, the value for T_1 indicates the actual work of the computation on 8 processors, and not the running time on one processor.

steal, the thief increments `E`. If there is nothing to steal, the thief restores `E` to the original value. Otherwise, the thief steals frame `H` and increments `H`. From the point of view of a worker, the common case is the same as in the simplified protocol: it compares two pointers (`E` and `T` rather than `H` and `T`).

The exception mechanism is used to implement `abort`. When a Cilk procedure executes an `abort` instruction, the runtime system serially walks the tree of outstanding descendants of that procedure. It marks the descendants as aborted and signals an abort exception on any processor working on a descendant. At its next `pop`, an aborted procedure will discover the exception, notice that it has been aborted, and return immediately. It is conceivable that a procedure could run for a long time without executing a `pop` and discovering that it has been aborted. We made the design decision to accept the possibility of this unlikely scenario, figuring that more cycles were likely to be lost in work overhead if we abandoned the THE protocol for a mechanism that solves this minor problem.

6 Benchmarks

In this section, we evaluate the performance of Cilk-5. We show that on 12 applications, the work overhead c_1 is close to 1, which indicates that the Cilk-5 implementation exploits the work-first principle effectively. We then present a breakdown of Cilk's work overhead c_1 on four machines. Finally, we present experiments showing that the critical-path overhead c_∞ is reasonably small as well.

Figure 6 shows a table of performance measurements taken for 12 Cilk programs on a Sun Enterprise 5000 SMP with 8 167-megahertz UltraSPARC processors, each with 512 kilobytes of L2 cache, 16 kilobytes each of L1 data and instruction caches, running Solaris 2.5. We compiled our programs with `gcc` 2.7.2 at optimization level `-O3`. For a full description of these programs, see the Cilk 5.1 manual [8]. The table shows the work of each Cilk program T_1, the critical path T_∞, and the two derived quantities $\overline{P}$ and c_1. The table also lists the running time T_8 on 8 processors, and the speedup T_1/T_8 relative to the one-processor execution time, and speedup T_S/T_8 relative to the serial execution time.

For the 12 programs, the average parallelism $\overline{P}$ is in most cases quite large relative to the number of processors on a typical SMP. These measurements validate our assumption of parallel slackness, which implies that the work term dominates in Inequality (4). For instance, on 1024×1024 matrices, `notempmul` runs with an average parallelism of 1970—yielding adequate parallel slackness for up to several hundred processors. For even larger machines, one normally would not run such a small problem. For `notempmul`, as well as the other 11 applications, the average parallelism grows with problem size, and thus sufficient parallel slackness is likely to exist even for much larger machines, as long as the problem sizes are scaled appropriately.

The work overhead c_1 is only a few percent larger than 1 for most programs, which shows that our design of Cilk-5 faithfully implements the work-first principle. The two cases where the work overhead is larger (`cilksort` and `cholesky`) are due to the fact that we had to change the serial algorithm to obtain a parallel algorithm, and thus the comparison is not against the C elision. For example, the serial C algorithm for sorting is an in-place quicksort, but the parallel algorithm `cilksort` requires an additional temporary array which adds overhead beyond the overhead of Cilk itself. Similarly, our parallel Cholesky factorization uses a quadtree representation of the sparse matrix, which induces more work than the linked-list representation used in the serial C algorithm. Finally, the work overhead for `fib` is large, because `fib` does essentially no work besides spawning procedures. Thus, the overhead $c_1 = 3.63$ for `fib` gives a good estimate of the cost of a Cilk `spawn` versus a traditional C function call. With such a small overhead for spawning, one can understand why for most of the other applications, which perform significant work for each spawn, the overhead of Cilk-5's scheduling is barely noticeable compared to the 10% "noise" in our measurements.

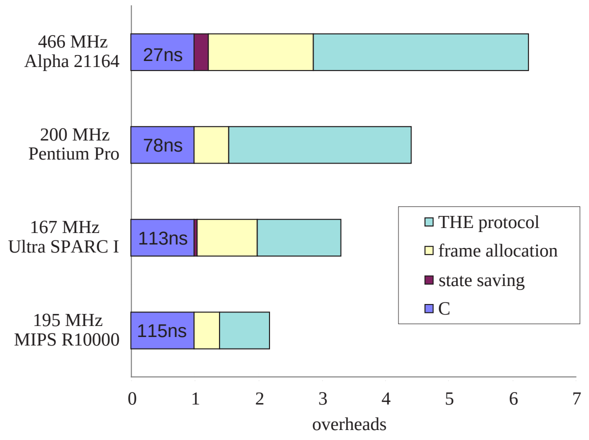

Figure 7: Breakdown of overheads for `fib` running on one processor on various architectures. The overheads are normalized to the running time of the serial C elision. The three overheads are for saving the state of a procedure before a spawn, the allocation of activation frames for procedures, and the THE protocol. Absolute times are given for the per-spawn running time of the C elision.

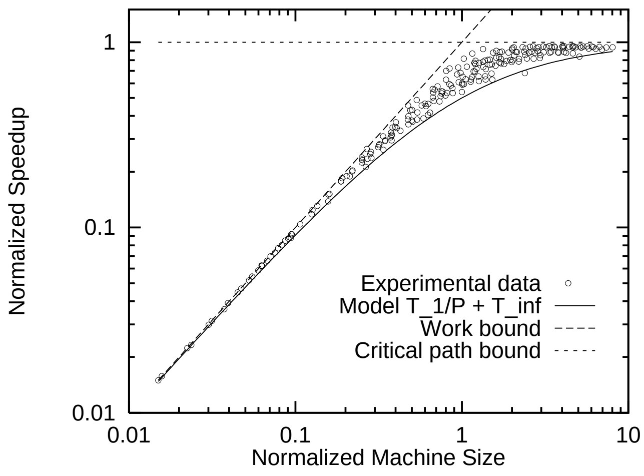

Figure 8: Normalized speedup curve for Cilk-5. The horizontal axis is the number P of processors and the vertical axis is the speedup T_1/T_P, but each data point has been normalized by dividing by T_1/T_∞. The graph also shows the speedup predicted by the formula $T_P = T_1/P + T_\infty$.

We now present a breakdown of Cilk's serial overhead c_1 into its components. Because scheduling overheads are small for most programs, we perform our analysis with the `fib` program from Figure 1. This program is unusually sensitive to scheduling overheads, because it contains little actual computation. We give a breakdown of the serial overhead into three components: the overhead of saving state before spawning, the overhead of allocating activation frames, and the overhead of the THE protocol.

Figure 7 shows the breakdown of Cilk's serial overhead for `fib` on four machines. Our methodology for obtaining these numbers is as follows. First, we take the serial C `fib` program and time its execution. Then, we individually add in the code that generates each of the overheads and time the execution of the resulting program. We attribute the additional time required by the modified program to the scheduling code we added. In order to verify our numbers, we timed the `fib` code with all of the Cilk overheads added (the code shown in Figure 3), and compared the resulting time to the sum of the individual overheads. In all cases, the two times differed by less than 10%.

Overheads vary across architectures, but the overhead of Cilk is typically only a few times the C running time on this spawn-intensive program. Overheads on the Alpha machine are particularly large, because its native C function calls are fast compared to the other architectures. The state-saving costs are small for `fib`, because all four architectures have write buffers that can hide the latency of the writes required.

We also attempted to measure the critical-path overhead c_∞. We used the synthetic `knary` benchmark [4] to synthesize computations artificially with a wide range of work and critical-path lengths. Figure 8 shows the outcome from many such experiments. The figure plots the measured speedup T_1/T_P for each run against the machine size P for that run. In order to plot different computations on the same graph, we normalized the machine size and the speedup by dividing these values by the average parallelism $\overline{P} = T_1/T_\infty$, as was done in [4]. For each run, the horizontal position of the plotted datum is the inverse of the slackness $P/\overline{P}$, and thus, the normalized machine size is 1.0 when the number of processors is equal to the average parallelism. The vertical position of the plotted datum is $(T_1/T_P)/\overline{P} = T_\infty/T_P$, which measures the fraction of maximum obtainable speedup. As can be seen in the chart, for almost all runs of this benchmark, we observed $T_P \le T_1/P + 1.0T_\infty$. (One exceptional data point satisfies $T_P \approx T_1/P + 1.05T_\infty$.) Thus, although the work-first principle caused us to move overheads to the critical path, the ability of Cilk applications to scale up was not significantly compromised.

7 Conclusion

We conclude this paper by examining some related work.

Mohr *et al.* [24] introduced lazy task creation in their implementation of the Mul-T language. Lazy task creation is similar in many ways to our lazy scheduling techniques. Mohr *et al.* report a work overhead of around 2 when comparing with serial T, the Scheme dialect on which Mul-T is based. Our research confirms the intuition behind their methods and shows that work overheads of close to 1 are achievable.

The Cid language [26] is like Cilk in that it uses C as a base language and has a simple preprocessing compiler to convert parallel Cid constructs to C. Cid is designed to work in a distributed memory environment, and so it employs latency-hiding mechanisms which Cilk-5 could avoid. (We are working on a distributed version of Cilk, however.) Both Cilk and Cid recognize the attractiveness of basing a parallel language on C so as to leverage C compiler technology for high-performance codes. Cilk is a faithful extension of C, however, supporting the simplifying notion of a C elision and allowing Cilk to exploit the C compiler technology more readily.

TAM [10] and Lazy Threads [14] also analyze many of the same overhead issues in a more general, "nonstrict" language setting, where the individual performances of a whole host of mechanisms are required for applications to obtain good overall performance. In contrast, Cilk's multithreaded language provides an execution model based on work and critical-path length that allows us to focus our implementation efforts by using the work-first principle. Using this principle as a guide, we have concentrated our optimizing effort on the common-case protocol code to develop an efficient and portable implementation of the Cilk language.

Acknowledgments

We gratefully thank all those who have contributed to Cilk development, including Bobby Blumofe, Ien Cheng, Don Dailey, Mingdong Feng, Chris Joerg, Bradley Kuszmaul, Phil Lisiecki, Alberto Medina, Rob Miller, Aske Plaat, Bin Song, Andy Stark, Volker Strumpen, and Yuli Zhou. Many thanks to all our users who have provided us with feedback and suggestions for improvements. Martin Rinard suggested the term "work-first."

References

[1] Andrew W. Appel and Zhong Shao. Empirical and analytic study of stack versus heap cost for languages with closures. *Journal of Functional Programming*, 6(1):47–74, 1996.

[2] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Puerto Vallarta, Mexico, June 1998. To appear.

[3] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1995.

[4] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 1996.

[5] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 356–368, Santa Fe, New Mexico, November 1994.

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

[7] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Puerto Vallarta, Mexico, June 1998. To appear.

[8] Cilk-5.1 (Beta 1) Reference Manual. Available on the Internet from `http://theory.lcs.mit.edu/~cilk`.

[9] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1990.

[10] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 164–175, Santa Clara, California, April 1991.

[11] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[12] Marc Feeley. Polling efficiently on stock hardware. In *Proceedings of the 1993 ACM SIGPLAN Conference on Functional Programming and Computer Architecture*, pages 179–187, Copenhagen, Denmark, June 1993.

[13] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–11, Newport, Rhode Island, June 1997.

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

[15] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, March 1969.

[16] Dirk Grunwald. Heaps o' stacks: Time and space efficient threads without operating system support. Technical Report CU-CS-750-94, University of Colorado, November 1994.

[17] Dirk Grunwald and Richard Neves. Whole-program optimization for time and space efficient threads. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 50–59, Cambridge, Massachusetts, October 1996.

[18] Christopher F. Joerg. *The Cilk System for Parallel Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January 1996.

[19] Robert H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[20] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[21] Phillip Lisiecki and Alberto Medina. Personal communication.

[22] James S. Miller and Guillermo J. Rozas. Garbage collection is fast, but a stack is faster. Technical Report Memo 1462, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1994.

[23] Robert C. Miller. A type-checking preprocessor for Cilk 2, a multithreaded C language. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1995.

[24] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[25] Joel Moses. The function of FUNCTION in LISP or why the FUNARG problem should be called the environment problem. Technical Report memo AI-199, MIT Artificial Intelligence Laboratory, June 1970.

[26] Rishiyur Sivaswami Nikhil. Parallel Symbolic Computing in Cid. In *Proc. Wkshp. on Parallel Symbolic Computing, Beaune, France, Springer-Verlag LNCS 1068*, pages 217–242, October 1995.

[27] Per Stenström. VLSI support for a cactus stack oriented memory organization. *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences, volume 1*, pages 211–220, January 1988.

[28] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.