6.895 Theory of Parallel Systems Project Paper

A Space-Efficient Global Scheduler for Cilk

Jason Hickey and Tyeler Quentmeyer

Abstract

We present a new scheduling algorithm for Cilk, inspired by the work of Narlikar and Blelloch, that decreases peak memory usage. Our scheduler runs threads using Cilk's work-stealing algorithm but preempts threads before they make large memory allocations. We present our implementation of the Narlikar scheduler for Cilk and our implementation of our own Hybrid scheduler for performance comparisons with Cilk's standard work-stealing scheduler. We then briefly present performance results and discuss future work.

1 Introduction

In this paper we present a new scheduling algorithm for Cilk [**?**] that combines the space saving ideas from Narlikar and Blelloch's AsynchDF scheduling algorithm [**?**] with Cilk's fast work-stealing scheduler. Although Narlikar's algorithm benefits from a provably better space bound than Cilk, the high scheduling overhead incurred in implementing Narlikar makes it impractical to use. We apply the key space-saving idea from Narlikar to Cilk's efficient work-stealing scheduler: favor computation over memory allocations whenever possible. Although we give up Narlikar's provably good space bound, empirical results suggest that our Hybrid scheduler improves on Cilk's peak memory usage in practice.

We complete the introduction by motivating a scheduling algorithm that is aware of memory usage. Next, we give a brief overview of Narlikar's algorithm and then motivate and introduce our new Hybrid algorithm. We then explain how we implemented both the Narlikar and the Hybrid scheduler for Cilk. We conclude with a brief performance analysis and then make suggestions for future work.

1.1 Motivation

In this section, we present an example parallel program and motivate the importance of a scheduler that is aware of memory usage by examining the space used by two example schedulers. The peak memory usage of a parallel program depends on the order in which parallel threads are executed. To illustrate this, consider the following pseudo-code, where `G` is some function that performs parallel computations with minimal memory allocations:

```
example() {
    for(i = 1 to n) spawn F(i, n);
}

F(int i, int n) {
    Temp B[n];
    for(j = 1 to n) spawn G(i, j, n);
}
```

The `example` function allocates blocks of memory and then performs parallel computations using those blocks. This pattern is typical of useful parallel algorithms and is therefore worth examining carefully [**?**].

The computation diagram for `example` is shown in figure **??**a. Since each instance of `F` and their child `G`s can be executed in parallel, a parallel scheduler is free to choose the order in which threads are executed. The least amount of space any parallel scheduler can use is the amount used by the serial execution, $S_1 = n$ for our example. Consider a scheduler, *BFS*, that schedules nodes by their breadth-first ordering. Using our

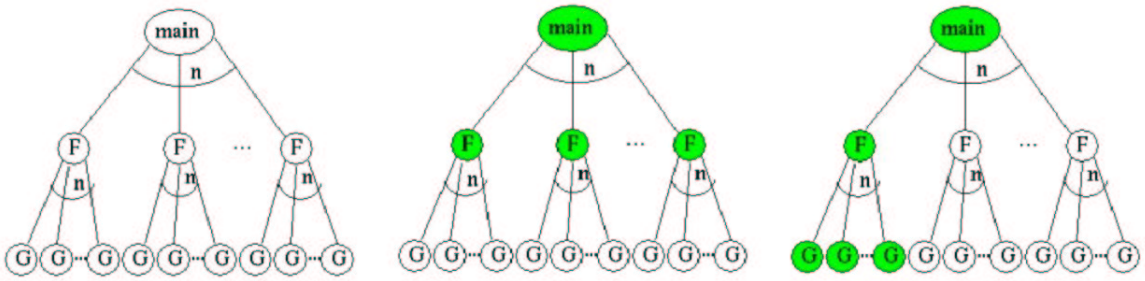


Figure 1: Computation graph for example program. Circles represent threads. Child nodes can be executed in parallel. Figure (b) shows a breadth-first schedule for an example program. Figure (c) shows a depth-first schedule for an example program. Shaded threads are executed before light threads.

example, *BFS* would schedule all n instances of `F` before any instance of `G`, as shown in figure **??**b. Since each instance of `F` allocates size n memory, *BFS* leads to peak memory usage of size n^2, S_1^2. Now consider a scheduler, *DFS*, that schedules nodes by their depth-first ordering. The peak memory usage for *DFS*'s schedule of our example, shown in figure **??**c, is just the serial memory size, S_1. A poor scheduling algorithm leads to a *quadratic* increase in peak memory usage for our example.

A program will suffer a massive performance penalty if its peak memory usage exceeds the amount of available physical memory. Adding memory is not only expensive but may be impossible due to hardware or architecture constraints. It is therefore very important to use a parallel scheduler with good peak memory usage characteristics.

2 Design

We begin by giving a brief review of the Narlikar scheduling algorithm. We then motivate and describe our new Hybrid scheduler. We conclude the section by briefly comparing the bottlenecks in Narlikar and our Hybrid.

2.1 Narlikar

This section gives a brief review of Narlikar's AsyncDF scheduling algorithm. For a complete explanation of Narlikar, see [**?**]. The idea behind Narlikar is to schedule threads in parallel similar to the way they would be scheduled in a serial execution. Narlikar accomplishes this by always scheduling the P (where P is the number of processors) left-most runnable threads in the computation graph — executing threads in the order they are reached by performing a P-wide depth first search of the computation graph, as shown in figure **??**. This algorithm leads to a provable peak memory usage of $O(S_1 + PKT_\infty)$ (where K is some constant and

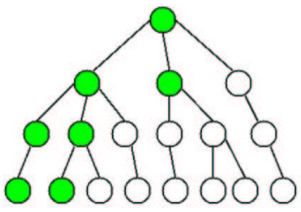


Figure 2: Narlikar schedule for example program with P=2. Shaded threads are executed before light threads.

T_∞ is the critical path length) [**?**], as compared to Cilk's bound of $O(PS_1)$ [**?**].

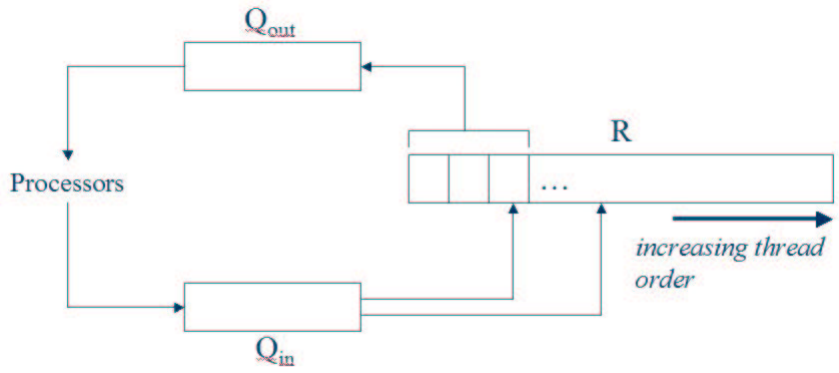


Figure 3: Data structures for the Narlikar scheduling algorithm.

The Narlikar algorithm schedules the P left-most runnable threads during a program's execution. It accomplishes this by maintaining a priority queue, R, that stores all runnable threads (a thread is a maximal segment of serial code) ordered by their depth first execution. A FIFO queue, Q_{out}, contains the threads that are scheduled to be executed next: the P leftmost threads and a special scheduler thread. Another FIFO queue, Q_{in}, contains threads that have been executed and need attention by the scheduler. Each processor sits in a loop that removes a thread from Q_{out}, executes it, and inserts it into Q_{in} after completion (i.e., on a `spawn`, `return`, et cetera). (Although Q_{in} and Q_{out} are not strictly necessary, Narlikar uses them as a performance optimization.)

Every $P + 1^{th}$ thread that a processor removes is a scheduler thread. Each time that the scheduling thread is executed is called a ***round***. The scheduler thread inserts all of the threads in Q_{in} into R and then puts the left-most P threads from R into Q_{out}. The scheduler thread is also responsible for forking threads and for removing completed threads.

The other notable point about Narlikar is that threads behave differently when they are about to make a large memory allocation. If a thread is about to allocate a block of memory larger than size K, it preempts itself and is not scheduled again by the scheduling thread for $SizeOfMemoryAllocation/K$ rounds. K represents a tunable parameter that trades of space-efficiency for time-efficiency.

2.2 Hybrid

In this section, we examine the drawbacks of Narlikar, then motivate and present our new Hybrid parallel scheduling algorithm for Cilk. Our Hybrid algorithm applies the fundamental memory saving philosophy from Narlikar to Cilk's efficient work-stealing to deliver a fast scheduler with better peak memory usage characteristics than Cilk.

Although Narlikar has a provably better space bound than Cilk, it achieves its better bound at the price of extremely high scheduling overhead. In our tests, our Narlikar implementation for Cilk performed as poorly as 400 times slower than Cilk's work-stealing algorithm. The essence of Narlikar is to drive down peak memory usage by favoring threads that perform computation over threads that allocate memory. The problem is that the way in which Narlikar favors computation over memory allocations has a high scheduling overhead for both allocation threads and for computation threads. Our Hybrid scheduler addresses this problem by only incurring scheduling overhead for threads that make large memory allocations.

2.2.1 Hybrid Scheduling Algorithm

Our Hybrid scheduling algorithm starts with Cilk's efficient distributed work-stealing algorithm. When a thread is about to make a large memory allocation, it puts itself to sleep and the processor it was running on executes a thread in its deque or performs a work-steal. Sleeping threads are woken up after a certain number of work-steal attempts. The number of work-steal attempts for which a thread sleeps is based on the size of memory the thread is about to allocate.

We introduce the Narlikar-style notion of a ***round*** into Cilk. The next round is entered when any processor begins a work-steal attempt. (Note that keeping track of the current round does not introduce a global lock because most architectures provide an atomic increment operation.) We also introduce a global priority queue, Q, that stores sleeping threads. Threads in Q are ordered on the round in which they are to be awoken, with the earliest wake time having the highest priority.

User threads make a scheduler call before every heap allocation and deallocation and the scheduler maintains a running total of the amount of heap space used by every thread. The scheduler calculates a $SleepForRounds(TotalHeapUsage)$ function. If $SleepForRounds$ is greater than 0, the scheduler preempts the current thread and inserts it into Q with wake-up round equal to $CurrentRound + SleepForRounds$. When a thread is preempted, the processor on which that thread was running returns to the scheduler to check its local deque or perform a work-steal to acquire new work.

Sleeping threads are woken up during the work-stealing process. Cilk's work-stealing function tries work-steal attempts until new work is found. We introduce a new work-stealing function for the Hybrid scheduler. First, increment the current round number. Then, if Q has a thread that is ready to be awoken, remove it from Q and start executing it. Otherwise, do a normal work-steal attempt. The work-stealing function repeats this process until it successfully finds new work.

2.2.2 The *SleepForRounds* Function

We use $SleepForRounds(TotalHeapUsage) = \left\lfloor \frac{TotalHeapUsage}{\alpha + P\beta} \right\rfloor$ as our function to calculate how many rounds for which a thread should sleep. Comparing to Narlikar, we see that $K = \alpha + P\beta$ has a new term involving the number of processors. The motivation for this term is Cilk's distributed nature. Consider the scenario where a program has recently started and only one processor's deque has stealable work. When other processors perform a work-steal, they increment the current round number, check Q for ready work, and then do a steal-attempt. Since Cilk's steal-attempts check a random deque for work, we need to do on the order of P random steal-attempts to find the one deque that has available work. Because each of these steal-attempts increment the current round number, the scenario where a limited number of processors have work accelerates the rate at which the round counter is incremented. The β factor represents the percentage of deques, weighted by the size of the memory allocation, an idle processor should check for work before resuming a thread that does a memory allocation.

2.2.3 Preliminary Analysis And Comparison To Narlikar

The major performance bottlenecks for Narlikar are:

1. Narlikar has bad locality. Threads must preempt themselves before making a parallel function call.
2. There are heavily-used global data structures, leading to thread lock-contention.
3. Narlikar requires substantial bookkeeping to execute the P left-most threads.

Our Hybrid scheduling algorithm addresses the three bottlenecks in Narlikar, while empirically delivering lower peak memory usage than Cilk. The third Narlikar bottleneck is completely removed because we no longer try to execute the P left-most threads. The Hybrid algorithm uses Cilk's "run like a serial program until parallelism occurs" philosophy except in the case of memory allocations. We therefore do not have

Narlikar's bad locality properties in the case of computational threads. We sacrifice locality only when threads make large heap allocations.

The Hybrid algorithm has a substantially reduced dependence on global data structures. As mentioned earlier, the current round counter can be implemented without using locks and therefore is not a point of contention. The only global data structure we use is Q. We can check Q for a ready thread without locking, so Q only introduces a bottleneck when we need to insert or remove threads. The number of inserted and removed threads in Q can be amortized against the number of times a program makes large memory allocations.

In all of the potential bottlenecks of the Hybrid algorithm, we only pay a performance penalty when threads actually make large memory allocations. The Hybrid scheduler therefore achieves its goal of driving down peak memory usage with minimal time-efficiency side effects.

3 Implementation

3.1 Cilk Background

This section presents the implementation details of Cilk necessary to understand the Narlikar and Hybrid implementations. For complete details, refer to the Cilk source code, [**?**], and [**?**].

3.1.1 Data Structures

Cilk maintains a `ShadowStack`. The shadow stack is a mirror of the standard C call stack. Each entry on the `ShadowStack` is a `StackFrame`. A `StackFrame` stores up-to-date values for local variables and the entry point at which to resume execution should the function be stolen. The `StackFrame` is all of the state necessary for a processor to resume execution of a sleeping function.

A `Closure` represents a slow clone. Each `Closure` stores a `join_counter`, `return_value`, `return_size`, pointer to the clone's `StackFrame`, and `thread_status` (`running`, `suspended`, `returning`, or `ready`). It contains all of the state necessary to execute a clone in parallel and to merge results with its parent.

A `ClosureCache` is the per processor structure that stores information about the processor's currently running execution. It contains the `ShadowStack` and exception information. The `WorkerState` is the per processor structure that stores the pthread id and the processor's `ClosureCache`.

A `Deque` maintains the ready deque of slow clones (`Closure` structures). Each processor has a globally accessible `Deque`.

3.1.2 Functionality

Cilk2c maps each user level function to two Cilk functions: the fast clone and the slow clone. The fast clone takes in the function's parameters, creates a new `StackFrame`, and pushes it on the `ShadowStack`. During execution, the fast clone saves correct values for live variables and maintains where to resume execution if the clone is stolen (the entry point) in the `StackFrame`. Fast clones return values using normal C semantics.

A function's slow clone takes in a `Closure` for the function. It then jumps to the proper entry point, restores correct values for live variables, and resumes execution. Slow clones return values by copying the return value and size into the clone's `Closure`, which is in turn copied into the correct `StackFrame` by the scheduler.

Function calls in user level code are made to fast clones. Function calls in the scheduler are made to slow clones. The scheduler calls a slow clone via a `Closure` which in turn calls fast clones that build the `ShadowStack`.

When a processor runs out of work, it performs a work-steal. Processor *Thief* steals work from processor *Victim* in three steps: First, *Thief* takes the `Closure` at the top of *Victim*'s `Deque` and puts it on its own `Deque`. Second, *Thief* removes the top `StackFrame` from *Victim*'s `ShadowStack`, converts it into a slow clone (`Closure`), and pushes the `Closure` on *Victim*'s `Deque`. Finally, *Theif* executes the stolen slow clone via

the stolen `Closure`. In this way, the `Deque` for a processor contains only the `Closure` for the slow clone it is currently executing.

3.2 Narlikar

The Narlikar algorithm is fundamentally different enough from Cilk's distributed work-stealing that implementing Narlikar using the Cilk user-level syntax required almost a complete rewrite of the Cilk code-base. Narlikar uses global queues to store threads so we removed deques. Functions are only called by the scheduler which means that the user-level call-stack is only ever at most one function deep. We therefore no longer need fast clones or the shadow stack, et cetera.

Every thread is implemented with a slow clone; therefore every `StackFrame` is associated with a `Closure` and every return uses the slow return semantics. We still use the fast clone code, but only as a constructor for `StackFrame`s.

3.2.1 Changes to `cilk2c`

The Narlikar algorithm calls for two changes in behavior in user-level functions:

1. Memory allocations require a return to the scheduler if they are large enough.

   We implement conditional preempting on a memory allocation by having the user-level function call a scheduling function `NarlikarMalloc(size)`. If `NarlikarMalloc` returns `true`, the function returns to the scheduler. Otherwise, it continues execution normally. To enable this, we changed `cilk2c` to add `if(NarlikarMalloc(size)) return;` before every call to `malloc(size);`. To enable the scheduler to resume execution right before the `malloc`, we save the live variables before calling `NarlikarMalloc` and add an entry point directly after `NarlikarMalloc` but before `malloc`.

   For example, the following code in a user-level function:

   ```
   t = malloc(1024);
   ...
   free(t);
   ```

   is transformed into the following psuedocode:

   ```
   save variables for entry point 1;
   if(NarlikarMalloc(1024))
       return;
   entry 1:
       T = malloc(1024);

   ...

   NarlikarFree(1024);
   ```

2. Parallel function calls always require a return to the scheduler.

   A parallel function call signals the end of the logical thread, so we need to return to the scheduler before every `spawn`. Although the spawned thread and the remainder of the function are both logical children of the original thread, we adopt the terminology that the spawned thread is the ***child*** and the remainder of the function is the ***parent***. The scheduler needs to be able to resume execution for both the parent and the child. To enable this, we added an entry point before every `spawn` in addition to the one that `cilk2c` adds after every `spawn`. (This also involved changing the live variable analysis to track local variables that have changed before the `spawn` in addition to variables that have changed

after the `spawn`.) Since the code following a `spawn` belongs to the parent thread, we also add a `return` after every `spawn`.

For example, the following code in a user-level function:

```
spawn fib(n-1);
```

is transformed into the following psuedocode:

```
save live variables for entry point 1;
NarlikarPreemptThreadBeforeSpawn();
return;
entry 1:
    fib(n-1);
    return;
entry 2:
```

Additionally, we modified the generated output for fast clones to use the same `SET_RESULT` semantics for return values that slow clones use. This is because all threads are slow clones and therefore their return values must be passed through the `Closure`'s `return_value` field.

3.2.2 Changes to Cilk Runtime

We had to replace almost all of the scheduling code and data structures in the Cilk Runtime to implement the Narlikar algorithm.

As mentioned earlier, we no longer need and therefore removed the `Deque` data structure. Since the call stack is always at most a single function deep, we removed the `ShadowStack` (although we still use the `StackFrame` data structure in a $1:1$ mapping to `Closure` objects). Since we always use slow clones, we removed the `ClosureCache` structure.

To the `Closure`, we added `malloc_total` to store a running total of the thread's heap usage, `sleep_for_rounds` to store the number of rounds left for which the thread should sleep because of a memory allocation, and `needs_fork` to denote if a thread has returned to the scheduler because it needs to be forked (at a `spawn`). To the `WorkerState`, we added a reference to a `Closure` to store the slow clone that each processor is currently executing.

To implement Q_{in} and Q_{out}, we made a FIFO queue with fixed-size $3P$ (see [**?**] for an explanation why this bound works) that supports parallel insert and delete operations. Both Q_{in} and Q_{out} are protected with global locks. To implement R, we made a serially accessible priority queue implemented as a doubly-linked list. R can only be accessed by the scheduling thread, which is protected by a global lock. We implemented Narlikar's arbitrary constant K as a command-line runtime parameter.

Each `pthread` calls `NarlikarWorkerMain` as its main function. In pseudo-code, `NarlikarWorkerMain` looks like:

```
void NarlikarWorkerMain() {
    while(!USE_SHARED(done)) {
        Closure *stub = ParallelRemove(qOut);
        if(stub == SCHEDULER_THREAD_MAGIC_NUMBER) NarlikarScheduler();
        else RunThread(stub);
    }
}
```

It sits in a loop until all execution is complete, removing and executing threads from `qOut`.

The function call made before a memory allocation, `NarlikarMalloc`, inserted by `cilk2c` looks like:

```
void NarlikarMalloc(Closure *stub, unsigned size) {
    stub->malloc_size += size;
    stub->sleep_for_rounds = stub->malloc_size/K;
    if(stub->sleep_for_rounds) {
        ParallelInsert(qIn, stub);
        return 1;
    }
    return 0;
}
```

If the running total of heap usage for the thread is large, preempt the current thread. Otherwise, continue normally.

The function call made before a spawn, `NarlikarPreemptThreadBeforeSpawn`, inserted by `cilk2c` looks like:

```
void NarlikarPreemptThreadBeforeSpawn(Closure *stub) {
    stub->needs_fork = 1;
    ParallelInsert(qIn, stub);
}
```

It simply marks the thread to be forked and inserts the thread into `qIn`.

The actual Narlikar scheduling function, called whenever a worker thread removes a scheduling thread from `qOut`, looks like:

```
void NarlikarScheduler() {
    Closure *stub;
    int nSchedulerThreads = 0;

    Acquire(NarlikarSchedulerLock);

    while(!QueueEmpty(qIn)) {
        stub = ParallelRemove(qIn);
        stub->inQueue = 1;
        if(stub->status == RETURNING) {
            NarlikarClosureReturn(stub);
            QueueRemove(R, stub);
        }
    }

    stub = R->Head;
    while(nScheduledThreads < P) {
        if(stub == NULL) break;

        if((stub->status == READY) && (stub->inQueue)) {
            if(stub->needs_fork) {
                Closure *forkedStub = NarlikarForkThread(stub);
                QueueInsertBefore(R, forkedStub, stub);
                stub = forkedStub;
            }

            if(stub->sleep_for_rounds) stub->sleep_for_rounds--;
            else {
                nSchedulerThreads++;
```

```
                stub->inQueue = 0;
                ParallelInsert(qOut, stub);
            }
        }

        stub = stub->next;
    }

    ParallelInsert(qOut, SCHEDULER_THREAD_MAGIC_NUMBER);

    Release(NarlikarSchedulerLock);
}
```

A user-level thread that has preempted itself before a spawn is forked into a parent and a child thread by the scheduler using the following code:

```
Closure *NarlikarForkStub(Closure *parent) {
    Closure *child = NewInitializedClosure();

    child->parent = parent;
    SetupReturnValueLocation(parent, child);
    child->frame = CopyClosureFrame(parent);

    parent->needs_fork = 0;
    parent->frame->entry++;
    parent->join_counter++;

    return child;
}
```

Before `NarlikarForkStub` is called, the parent's entry point is set to resume execution directly before the `spawn`ed function. During a fork, the parent's entry point is incremented so that it resumes after the `spawn`. The child's entry point is left unmodified so that it will resume before the `spawn`, call the `spawn`ed function's fast clone, and finally return to the scheduler.

3.3 Hybrid

The Hybrid algorithm is based on Cilk's work-stealing infrastructure so was much simpler to implement and integrate with the Cilk code-base. The Hybrid algorithm behaves just like Cilk except before a memory allocation and during a work-steal. We therefore only had to change the work-stealing code and add functionality to put threads to sleep before a `malloc`.

3.3.1 Changes to `cilk2c`

We made two changes to `cilk2c` to implement the Hybrid algorithm: we added functionality to return to the scheduler during a computation and we added functionality to conditionally sleep before a memory allocation.

We added the ability to return to the scheduler at will by adding a mechanism similar to exceptions. The scheduler signals that a processor should stop its current execution and return to the scheduler by setting the `hybrid_return_to_scheduler` field of the `WorkerState` to true. User code checks this value after every spawn and conditionally returns down the C call-stack to the scheduler. To do this, we modified the macro inserted by `cilk2c` after every spawn (CILK2C_XPOP_FRAME_RESULT and CILK2C_XPOP_FRAME_NORESULT) to include the code

```
if(WorkerState->hybrid_return_to_scheduler)
    return;
```

Since `cilk2c` already adds an entry point after every `spawn`, introducing this conditional return has no functional side effects.

To conditionally put the current thread to sleep before a memory allocation, we modified `cilk2c` to insert `HybridMalloc(size)` before each call to `malloc(size)`. We will see in the runtime section that `HybridMalloc` uses the `hybrid_return_to_scheduler` field to signal to the user-level code that it needs to return to the scheduler so that it can be put to sleep. Once in the scheduler, we actually put a thread to sleep by artificially increasing its `join_counter` and simulating a `sync`. Therefore, we also add a check on `hybrid_return_to_scheduler`, an entry point, and a `sync` before the call to `malloc(size)`.

For example, the following code in a user-level function

```
t = malloc(1024);
...
free(t);
```

is transformed into the following psuedocode:

```
save variables for entry point 1;
HybridMalloc(1024);
if(WorkerState->HybridReturnToScheduler)
    return;
entry 1:
sync:
entry 2:
    T = malloc(1024);

...

HybridFree(1024);
```

3.3.2 Changes to Cilk Runtime

We were able to use existing mechanisms in Cilk to implement most of the Hybrid algorithm. Our changes to the runtime are therefore relatively minor.

As we mentioned in the `cilk2c` modifications section, we added a `hybrid_return_to_scheduler` field to the `WorkerState` object. We added a `malloc_size` field to the `StackFrame` object. We added a global time variable protected by a lock. We implemented α and β as command-line runtime parameters. We implemented the priority Q as a heap protected by a lock.

The function call made before a memory allocation, `HybridMalloc`, inserted by `cilk2c` looks like:

```
void HybridMalloc(unsigned size) {
    StackFrame *CurrentFrame = GetCurrentlyExecutingStackFrame();
    unsigned SleepForRounds;

    CurrentFrame->malloc_total += size;
    SleepForRounds = ComputeNRoundsToSleep(CurrentFrame->malloc_total);

    if(SleepForRounds > 0) HybridPreemptThread(SleepForRounds);
}
```

The `StackFrame` for the current thread maintains a running total of the heap usage for the current thread. If the scheduler decides the current thread should be put to sleep, it calls `HybridPreemptThread` and signals the current execution to return to the scheduler.

We preempt a thread by artificially incrementing its `join_counter` and adding a `sync` immediately following points in the code where a thread can be put to sleep. In this way, when the thread reaches the `sync`, it will necessarily suspend until the scheduler resumes the thread by decrementing its `join_counter` and reactivating the thread. This scheme exploits Cilk's existing infrastructure to suspend and resume threads. Unfortunately, Cilk only maintains a `join_counter` for and only knows how to suspend and resume slow clones. In the general case, the currently executing function is a fast clone. We therefore have to work our way from the top of the `ShadowStack` down and convert each fast clone on the stack into a slow clone, adding the new `Closure`s to the `Deque` along the way. Finally, since we have put everything on the C call-stack on the `Deque`, we need to clear the C call-stack so that we do not duplicate execution. This is why we added the `hybrid_return_to_scheduler` field. The psuedo-code for `HybridPreemptThread` is as follows:

```
void HybridPreempThread(unsigned SleepForRounds) {
    Closure *t;

    while(ShadowStack is not empty) {
        t = MakeClosureFromStackFrame(GetBottomStackFrame());
        PopStackFrameFromShadowStack();
        PushClosureOnDeque(t);
    }

    t->join_counter++;
    InsertInPriorityQueue(Q, t, CurrentTime + SleepForRounds);
    WorkerState.hybrid_return_to_scheduler = 1;
}
```

Our last modification to the Cilk runtime is for the work-stealing step. We replaced the work-steal attempt functionality with the `HybridDoSteal` function as follows:

```
Closure *HybridDoSteal() {
    Closure *t = NULL;

    CurrentTime++;
    if(LowestPriority(Q) < CurrentTime) {
        t = RemoveLowestPriority(Q);
        t->join_counter--;
        if(!((t->join_counter == 0) && (t->status == SUSPENDED)))
            t = NULL;
    }

    if(t == NULL) t = NormalWorkStealAttempt();

    return t;
}
```

The line that checks to ensure that the `join_counter` is 0 and that the `Closure`'s status is SUSPENDED is synonymous with how a slow clone returns to its parent. When a slow clones returns, if it is its parent's last outstanding child, it is the child's responsibility to reactivate the parent. In this way, putting a thread to sleep is exactly like calling a spawn that does not return for some number of rounds.

4 Performance Results

We implemented the `example` function from the introduction to compare the performance of our two new Cilk schedulers to Cilk's work-stealing scheduler. For `G`, the parallel computation that does not allocate

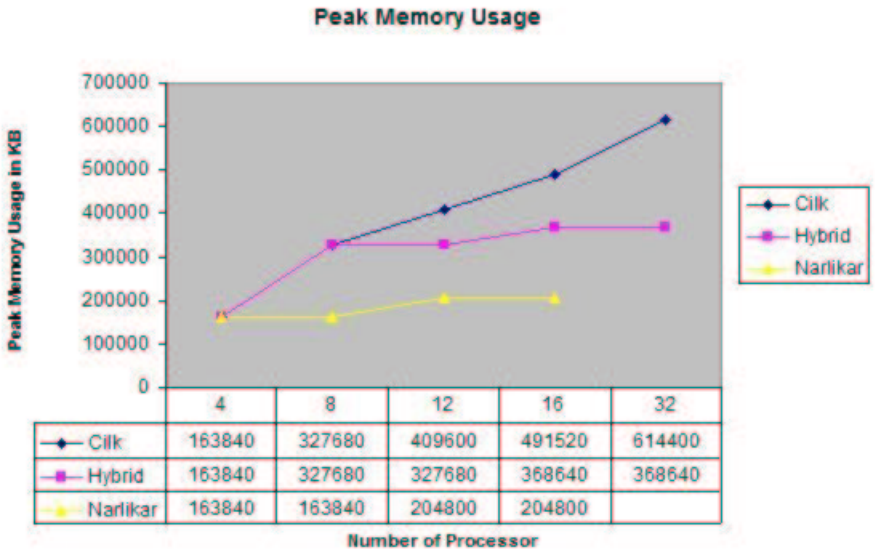
memory, we computed a fibonacci sequence. In pseudo-code:

```
main() {
    StartTimer();
    for(i = 1 to 25) spawn TestFunction();
    sync;
    StopTimer();
    ReportPeakMemoryUsageAndTiming();
}

TestFunction() {
    void *TestAllocation = malloc(sizeof(int)*10M);
    UpdateMemoryUsageTracker(sizeof(int)*10M);
    spawn fib(30);
    sync;
    free(TestAllocation);
    UpdateMemoryUsageTracker(-sizeof(int)*10M);
}
```

This example program is in some sense ideal because it allows us to compare all of the interesting properties of our schedulers at once. It follows `example`'s pattern and therefore allows us to compare peak memory usage. The `fib` function has high parallelism and low computational complexity, allowing us to obtain a rough measure of how much scheduling overhead is imposed on threads that do not allocate memory.

We used a 32-processor SGI with 16 gigabytes of physical memory to run performance tests. Work-stealing Cilk and our Narlikar-Work-Stealing Hybrid Cilk are non-deterministic (Narlikar is deterministic ignoring hardware glitches and operating system factors) so we repeatedly ran our test program to obtain average performance values. We chose and fixed suitable values for K, α, and β for all tests. Our results for peak memory usage are summarized in figure **??** and our results for time efficiency are summarized in figure **??**.



	4	8	12	16	32
Cilk	163840	327680	409600	491520	614400
Hybrid	163840	327680	327680	368640	368640
Narlikar	163840	163840	204800	204800	

Figure 4: Peak memory usage of the example program.

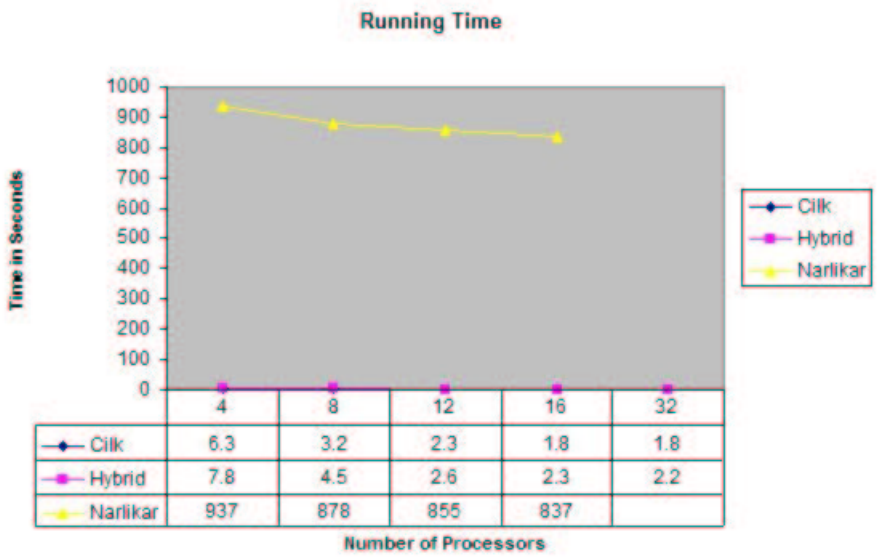


Figure 5: Running time of the example program.

The peak memory usage characteristics are as expected: Cilk is the worst, Narlikar is the best, and the Hybrid is somewhere in the middle. Cilk work-stealing uses more memory as we increase the number of processors. Narlikar uses close to the serial execution. The Hybrid peak memory usage is somewhere in between Narlikar and Cilk, but looks more like Narlikar, the serial memory execution, as we increase the number of processors. The peak memory usage for both Narlikar and the Hybrid can be characterized by the problem at hand, S_1, whereas Cilk has a multiplicative factor of P. The Hybrid's peak memory usage characteristics improve, and therefore its improvement over Cilk grows, as we move to machines with more processors.

As you can see, Narlikar has significantly higher scheduling overhead than both Cilk work-stealing and the Hybrid scheduler. In our example, Narlikar is as much as 400 times slower than Cilk. It also suffers from little parallel speedup because of contention for global data structures. Although the Hybrid is slower than Cilk, its performance is on the order of Cilk's (note that our example program exhibits the worst-case time-efficiency because `fib` is almost pure scheduling overhead). Additionally, if our test case were expanded to exhaust physical memory, Cilk would swap to disk more than the Hybrid, making the Hybrid's performance substantially better than Cilk's. For the target problem that uses more memory than is physically available, the Hybrid scheduler looks to be faster than Cilk.

5 Conclusion

Our Hybrid scheduler achieves its goal of minimizing peak memory usage while maintaining reasonable running times. The peak memory usage for the Hybrid is empirically on the order of Narlikar's peak memory usage. More importantly, the peak memory usage of the Hybrid is independent of the number of processors being used, contrasted with Cilk's linear dependence on the number of processors. Finally, our Hybrid scheduler achieves its memory efficiency without making unreasonable sacrifices in running time. Our example program runs roughly 30% slower with our Hybrid implementation than with Cilk, making it not impractical to use. Given more time it is reasonable to expect that our Hybrid implementation could be improved to be even closer in speed to Cilk.

We successfully implemented the Narlikar scheduler for Cilk. We also designed and implemented a new Hybrid scheduler for Cilk. Finally, we showed that our Hybrid algorithm has enough desirable characteristics to merit further investigation.

5.1 Future Work

Although we have designed and implemented our Hybrid scheduling algorithm, there is still a substantial amount of work that needs to be done:

1. We need a theoretical analysis of the peak memory usage for our Hybrid algorithm. Given Cilk's randomized distributed nature, an analysis would hopefully yield a high-probability argument for a bound on memory usage. A simple approach that would give Cilk's memory bound would be to limit the size of Q to a factor of P.

2. More performance testing needs to be done. We need to run a suite of real parallel programs on larger machines. We also need to carefully probe the space of constants for both the Narlikar and the Hybrid scheduler (empirically examine the performance for different values of K, α, and β).

3. We need to carefully examine the *SleepForRounds* function. Although we gave a brief justification for our choice, it was mostly arbitrary. Choosing a good *SleepForRounds* function may greatly improve the performance of our Hybrid scheduler.

4. Although the end goal of our work was not implementing Narlikar, we need to improve our implementation of Narlikar for Cilk to ensure the reliability of our performance results. Our Narlikar implementation is complete and correct but may be able to be substantially improved. For example, better performance might come from finer grained locks or from experimenting with additional code to "group threads for locality" [**?**].

5. We need to optimize the implementation of our Hybrid scheduler. As a first step, we need an analysis of where the bottlenecks in our implementation are. From there we could try to remove bottlenecks and allow the Hybrid performance to approach Cilk's. We could also use a faster lock-free implementation for the priority queue Q.

6. Finally, we would like to experiment with variations of our Hybrid algorithm. For example, instead of using a single global queue, we could use distributed sleeping queues much like the way deques are now organized.

References

[1] Robert D. Blumofe and Charles E. Leiserson. Space-Efficient Scheduling of Multithreaded Computations. *25th Annual ACM Symposium on the Theory of Computing (STOC '93),* May 16-18 1993, San Diego, California, pp. 362-371.

[2] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. *1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, June 1998.

[3] Girija J. Narlikar and Guy E. Blelloch. Space-Efficient Scheduling of Nested Parallelism. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(1), January 1999.

[4] Supercomputing Technology Group, Massachusetts Institute of Technology. *Cilk 5.3.2 Reference Manual*, November 2001. Available on the World Wide Web at URL "`http://supertech.lcs.mit.edu/cilk`".