6.895 Theory of Parallel Systems Lecture 10

Cilk Implementation

Lecturer: Charles Leiserson

Lecture Summary

1. *Compiling Cilk*
   This section gives some of the implementation details and clever tricks used in compilation of Cilk.

2. *Deque Protocols*
   Description of the TH and THE protocols Cilk uses for managing the deques.

1 Compiling Cilk

In this section, we will see some implementation details of conversion of Cilk source to C post-source. We will subsequently refer to this conversion as compiling the Cilk program. We will also see the reasons behind choosing to implement the conversion in this particular way.

The Cilk compilation path

The Cilk compilation path is shown in Figure 1. The Cilk source is run through the ***cilk2c*** translator which converts the Cilk source to C post-source. This C post-source is compiled using `gcc` and the object code is linked with the ***Cilk run-time system*** to produce the native binary code.

The software components that needed to be built to implement the Cilk system were `cilk2c` and `Cilk run-time system`. The `cilk2c` translator translates the Cilk commands like `spawn` and `sync` to C instructions that call the run-time system routines and do the parallel bookkeeping. The translation process is simple, because it only needs to transform Cilk keywords, leaving the C instructions as they are. In this section, we shall discuss some implementation details of the `cilk2c` translator. Some of the mechanisms used in the `Cilk run-time system` will be discussed in the next section.

Work-First Principle

Cilk's greedy scheduler operates on the "work-first" principle: it is better to amortize overheads against the critical path than against work. Recall that for a job with T_1 work and T_∞ critical path length, the Graham Brent greedy scheduler achieves the following bound on time taken to complete the job on P processors:

$$T_P \le \frac{T_1}{P} + T_\infty \ .$$

Recall also that this bound is within a factor of 2 of optimal. Thus, a greedy scheduler with comparable performance should exhibit a running time something like :

$$T_P \le C_1 \frac{T_1}{P} + C_\infty T_\infty \ . \tag{1}$$

Where C_1 and C_∞ are constants due to the overhead in the system. For parallel programs for which we can expect linear speedups, we have $P \ll T_1/T_\infty$. Thus, the first term in Inequality (1) is much larger than the second term. Consequently, in order to optimize the performance of the scheduler, we should try to move the overhead from C_1 to C_∞.

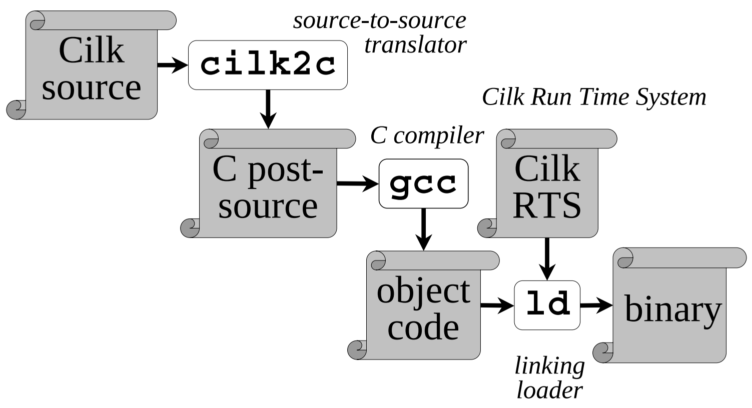


Figure 1: The compilation path for Cilk programs.

Shadow Stack

Cilk maintains a ***shadow stack*** made up of ***frames*** each of which holds all the live local variables for a single procedure. Since `cilk2c` has no control over the C stack, the code to store the local variables on the stack may duplicate some work done by the C run-time system. The overhead incurred due to this additional storage is small. In addition to storing the local variables, each frame has two extra locations ***entry*** and ***join***. The ***entry*** location holds the point at which the procedure should start executing when it is resumed in case it gets stolen and the ***join*** location holds the number of currently spawned children of the procedure.

Cilk's Compiler Strategy

The `cilk2c` converter generates two clones of each function: a ***fast clone*** and a ***slow clone***. The ***fast clone*** is the serial common case code and it is almost the same as what a C function looks like. The ***slow clone*** is the clone with the parallel book-keeping.[1]

Whenever a Cilk procedure is spawned, the fast clone is invoked after saving local live variables on the shadow stack. If a thread is stolen however, the slow clone is resumed at the other processor. Thus, a slow clone of a procedure is only executed if and when this procedure is stolen from one processor to another. which means that most of the overhead of executing the slow clone goes in the C_∞ term in Inequality (1).

As can be seen in Figure 2 all the procedures on a processor's deque are fast clones, except possibly the procedure on the top of the deque. Moreover, a procedure can have more than one outstanding child only if it has been stolen from some other processor in which case, it would be a slow clone. Thus, fast clones can have at most one outstanding child at any time, and this child is below them on the processor's deque. Thus, when a fast clone is being executed, it has no outstanding children.

Fibonacci Example

We can see how `cilk2c` compiles programs by considering the following Cilk program for computing the Fibonacci numbers:

[1]The slow clone is only 20-30 % slower than the fast clone. Its called the slow clone just to contrast it with the fast clone.

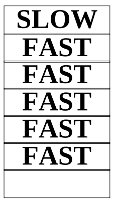


Figure 2: Shadow Stack view of any processor

```
1: cilk int proc{fib}(int n) {
2:   if (n<2) return (n);
3:   else {
4:     int x,y;
5:     x = spawn fib(n-1);
6:     y = spawn fib(n-2);
7:     sync;
8:     return(x+y);
9:   }
10:}
```

This procedure is subsequently referred to as `fib` program:

Compiling `spawn` in the fast clone

Figure 3 shows the compilation of `spawn` in line 5 of the `fib` program. In the fast clone, whenever a procedure spawns a child, all the live local variables of the parent are stored on a shadow stack. The procedure also saves the appropriate `entry` location, in this case entry number 1 on the shadow stack and increments the `join` counter.

After the `return` from the spawned function, a check is made to see if the parent is still on this processor's deque. If it is, then the parent executes just as it would in C. If the parent has been stolen, the returned value is updated in the shadow stack and the `join` counter is decremented by 1. If the value in the `join` counter is now 0 and the parent is stalled on a `sync`, then this child was the last of it's siblings to complete in which case, this processor begins executing the parent. If this child has executing siblings or the parent is not stalled, then this processor's control goes to the scheduler, and it starts work-stealing.

Compiling `sync` in the fast clone

As shown in Figure 4, in the fast clone, `sync` statement compiles to a `no-op`. This is because a fast clone has no outstanding children when it is executing.

Compiling `spawn` in a slow clone

Figure 5 shows the compilation of the `spawn` statement on line 5 in the slow clone. The slow clone of a procedure is executed whenever it is stolen from a processor and resumed on another processor. Thus, in a slow clone, we need entry points to resume the thread after each `spawn`. At the beginning of the procedure, the slow clone reads the `entry` location and a `switch` statement dispatches control to the appropriate entry point as shown in Figure 5. The live variables are then restored from the shadow stack and the execution of the program continues.

As shown in Figure 5, the segment of the program that restores the local variables is enclosed in a false conditional. This construct prevents the program from restoring the variables when it reaches this point

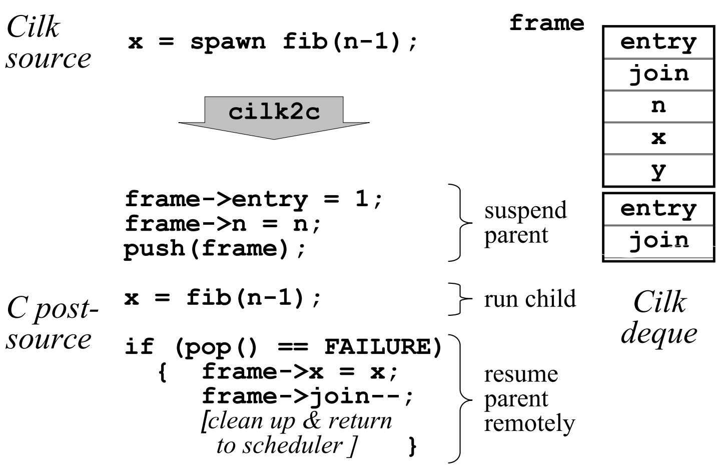


Figure 3: The compilation of `spawn` in the fast clone.

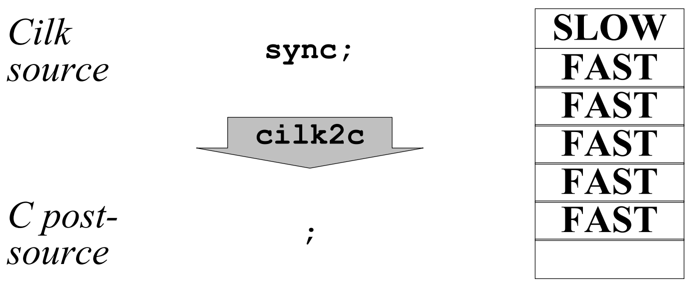


Figure 4: The compilation of `sync` in the fast clone.

during the normal sequential execution, that is when it is not being resumed after a steal. Thus, this overhead of restoring variables is only incurred when this procedure has been stolen and can be amortized against the T_∞ term in Inequality (1).

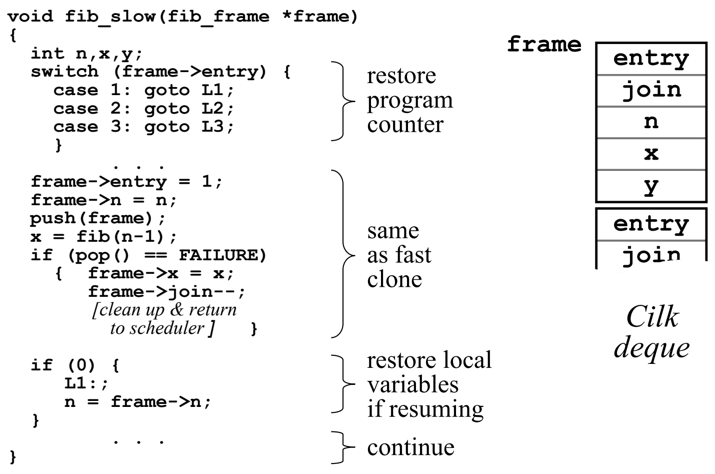
Compiling `sync` in the slow clone

When the slow clone of a procedure reaches a `sync` point, it checks whether it has any outstanding children by examining the `join` counter in its shadow stack. If this counter is 0, then there are no outstanding children and the procedure continues executing. If there are outstanding children, then the processor starts work-stealing. Figure 6 shows the example for the `fib` program.

Breakdown of Work Overhead

The efficiency of Cilk's compiling strategy can be tested by using the Fibonacci number calculation benchmark. The Figure 7 shows the performance on various single-processors machines. The breakdown of time spent on doing the various tasks is also shown in the figure.

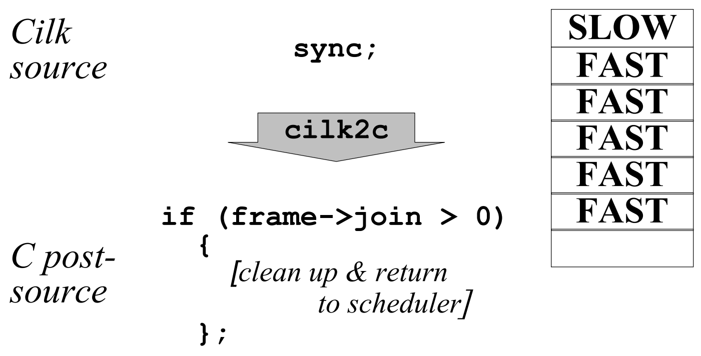
According to these benchmarks, the Cilk program is about 6 times worse than the equivalent C program in the worst case. This overhead might sound large, but the `fib` program has very little computation. Every



```
void fib_slow(fib_frame *frame)
{
  int n,x,y;
  switch (frame->entry) {
    case 1: goto L1;
    case 2: goto L2;
    case 3: goto L3;
    }
          . . .
  frame->entry = 1;
  frame->n = n;
  push(frame);
  x = fib(n-1);
  if (pop() == FAILURE)
    {  frame->x = x;
       frame->join--;
       [clean up & return
          to scheduler]     }

  if (0) {
     L1:;
     n = frame->n;
  }
          . . .
}
```

Figure 5: The compilation of `spawn` in the slow clone.



```
sync;
```

```
if (frame->join > 0)
  {
    [clean up & return
            to scheduler]
  };
```

Figure 6: The compilation of `sync` in the slow clone.

procedure just does two decrements and one addition. On the other hand it has many spawns. In general, most programs are likely to have much more computation than `fib` does and in that case the spawn overhead will be amortized against the computation, that is, for most programs, we expect that

$$Number\ of\ spawns \ll Number\ of\ C\ function\ calls \ll Number\ of\ total\ instructions\ executed.$$

The fact that *fib* is only 6 times slower in Cilk than in C is actually encouraging. Most parallel programs are actually as fast in Cilk as they are in C. In practice, for most reasonable programs, the constant C_1 in Inequality (1) is close to 1.

2 Deque Protocols

In Cilk, each processor, called a ***worker***, maintains a ***ready deque*** (doubly-ended queue) of ready Cilk procedures. Each deque has two ends, a ***head*** and a ***tail***, from which procedures can be added or removed. The worker treats the deque as a stack, pushing and popping procedures from the ***tail*** of the deque. If a worker runs out of work, it becomes a ***thief*** and tries to steal work at the head of the ready deque of a

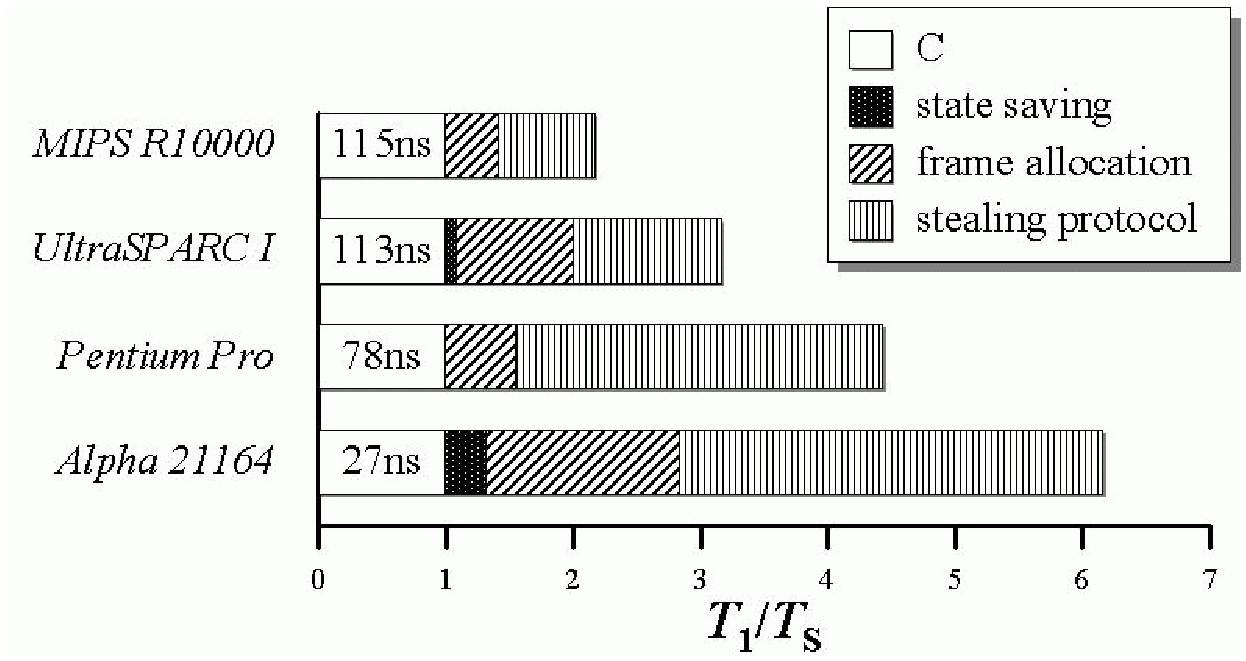


Figure 7: The breakdown of work overhead, benchmarked using *fib* on one processor.

victim processor. Work-stealing is implemented through shared-memory, and the thief and victim operate directly on the victim's ready deque. The main issue is how to resolve the potential race condition that may arise because of the concurrent accesses while at the same time, achieving good efficiency.

The protocol that Cilk adopts for managing deque make uses of three atomic shared variables T, H, and E, and it is called the THE protocol. 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.

2.1 TH Protocol

The TH protocol is an almost non-blocking protocol for managing deque, and the pseudocode is shown in Figure 8. The main idea is to minimize work overhead by moving the costs from the worker to the thief. To arbitrate among different thieves attempting to steal from the same victim, a hardware lock is used, and the overhead caused by the lock can be amortized against the critical path.

The code assumes that the deque is implemented as an array of frames. The head and tail of the deque are determined by two indices T and H. 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, and most of the time, T $\geq$ H. The worker pushes and pops frames by altering T, while the thief only increments H and does not alter T. The lock L ensures that only one thief can steal from the deque at a time.

The push operation is always safe, because it does not involve any interaction between a thief and its victim. For the pop operation, there are three cases, shown in Figure 9. When there are many thieves, only one of them will succeed in stealing the frame because of the locking mechanism. Thus we only need to consider the case of one thief.

1. Both the thief and the victim attempt to obtain different frames from the deque concurrently. In this case both are successful, and they do not interfere.

```
push() {
  T++;
}

pop() {
   T--;
   if (H > T) {
       T++;
       lock(L);
       T--;
       if (H > T) {
            T++;
            unlock(L);
            return FAILURE;
        }
        unlock(L);
    }
    return SUCCESS;
}
```

(a)

```
steal() {
   lock(L);
   H++;
   if (H > T) {
       H--;
       unlock(L);
       return FAILURE;
   }
   unlock(L);
   return SUCCESS;
}
```

(b)

Figure 8: **(a)** Pseudocode of the actions performed by the worker/victim in the TH protocol. **(b)** Pseudocode of the actions performed by the thief in the TH protocol.

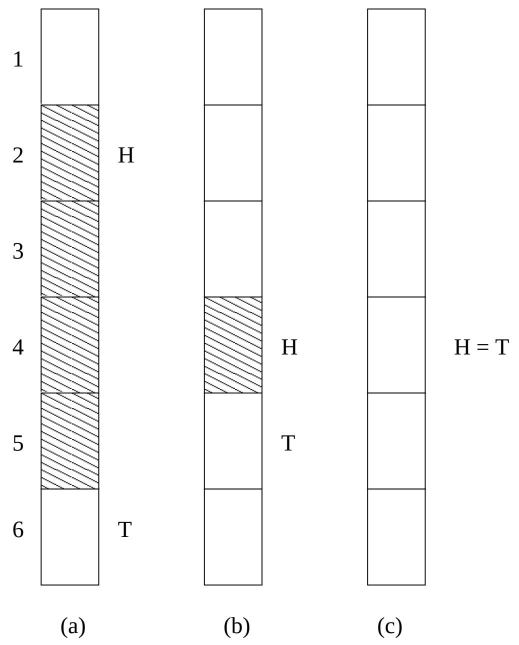


Figure 9: The three cases of the ready deque in the TH protocol. A shaded block indicates the presence of a frame at a certain position in the deque. The head and the tail are marked by T and H.

```
push() {
  T++;
}

pop() {
  T--;
  if (E > T) {
     T++;
     lock(L);
     T--;
     if (H > T) {
       T++;
       unlock(L);
       return FAILURE;
     }
     unlock(L);
   }
   return SUCCESS;
}
```

(a)

```
steal() {
  lock(L);
  E++;
  H++
   if (H > T) {
    E--;
    H++;
    unlock(L);
    return FAILURE;
  }
  unlock(L);
  return SUCCESS;
}
```

(b)

Figure 10: **(a)** Pseudocode of the actions performed by the worker/victim in the THE protocol. **(b)** Pseudocode of the actions performed by the thief in the THE protocol.

2. The deque contains only one frame. Either victim and thief will get the frame if the other is not making an attempt to obtain it. If both victim and thief try to get the frame, the protocol guarantees that at least one of them discovers that $T > H$. If the thief discovers that $T > H$, it restores H to its original value and retreats. If the victim discovers $T > H$, it will restart the protocol after acquiring the lock L, which guarantees that it will get the frame without interference from any thief unless a thief has already stolen the frame.

3. The deque is empty. A thief always fails to steal, and the victim also fails to pop the frame. The control passes to the Cilk runtime system.

The TH protocol contributes little to the work overhead. Pushing only involves updating T. Successfully popping a frame involves only 6 operations — 2 memory loads, 1 memory store, 1 decrement, 1 comparison, and 1 (predictable) conditional branch. In the case where both thief and victim simultaneously try to grab the same frame, the cost incurred by the lock can be considered as part of the critical-path overhead and does not influence the work overhead.

2.2 THE Protocol

The THE protocol extends the TH protocol to support signaling of exceptions to a worker. The index H in the TH protocol plays two roles: it marks the head of the deque, and it marks the point that the worker cannot cross when it pops. In the THE protocol, the two roles of H are separated into two variables: H, which now only marks the head of the deque, and E, which marks the point that the victim cannot cross. Figure 10 shows the pseudocode of the THE protocol.

The variable E is used to signal the worker that an exception has occurred. For example setting $E = \infty$ causes the worker to discover the exception at its next pop. At this point, it returns the control to the run time system, which can then handle the exception appropriately. Different high values of E are used for different exceptions. This mechanism is used to implement abort in Cilk. The limitation of this approach is that the worker is notified of an exception only after it returns from a `spawn` for the first time after the exception is signaled. Thus if a procedure does a lot of computation before returning, it might continue to compute for some time before discovering the `abort`. Although no bounds are placed on how long it takes for a sub-computation to stop after the exception is signaled, the exception mechanism seems to work well in practice.