Atomic Transactions in Cilk

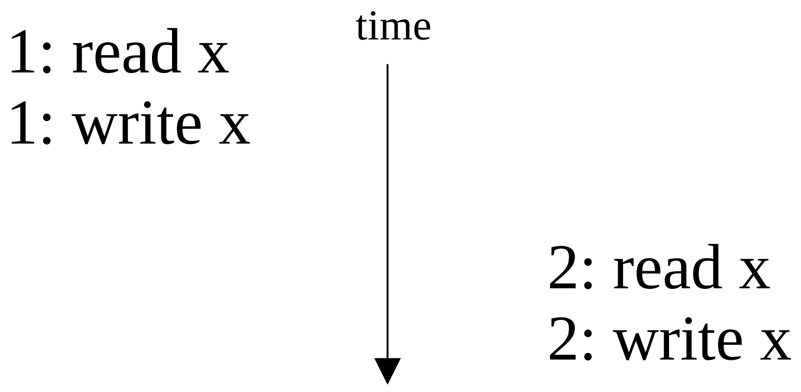
6.895 Project Presentation
12/1/03

Data Races and Nondeterminism

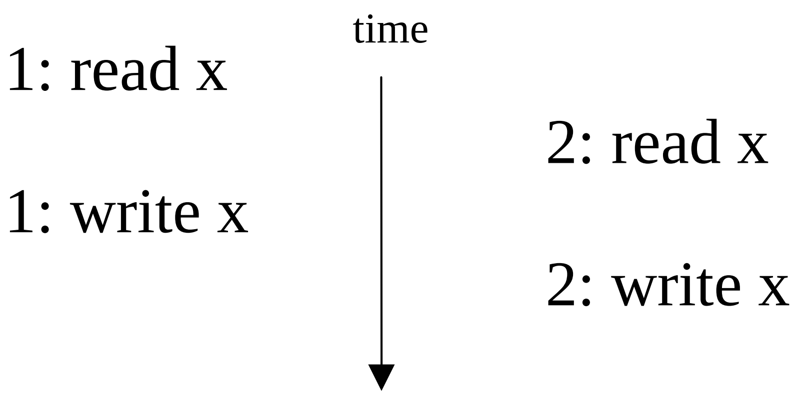
```
int x = 0;

cilk void increment() {
   x  = x + 1;
}

cilk int main() {
   spawn increment();
   spawn increment();
   sync;
   printf(''x is %d\n'', x);
   return 1;
}
```



Correct execution : x = 2



Incorrect execution: x = 1

Two Solutions to the Problem

Traditional Solution: Locks

```
cilk void increment() {
   lock(x);
   x  = x + 1;
   unlock(x);
}
```

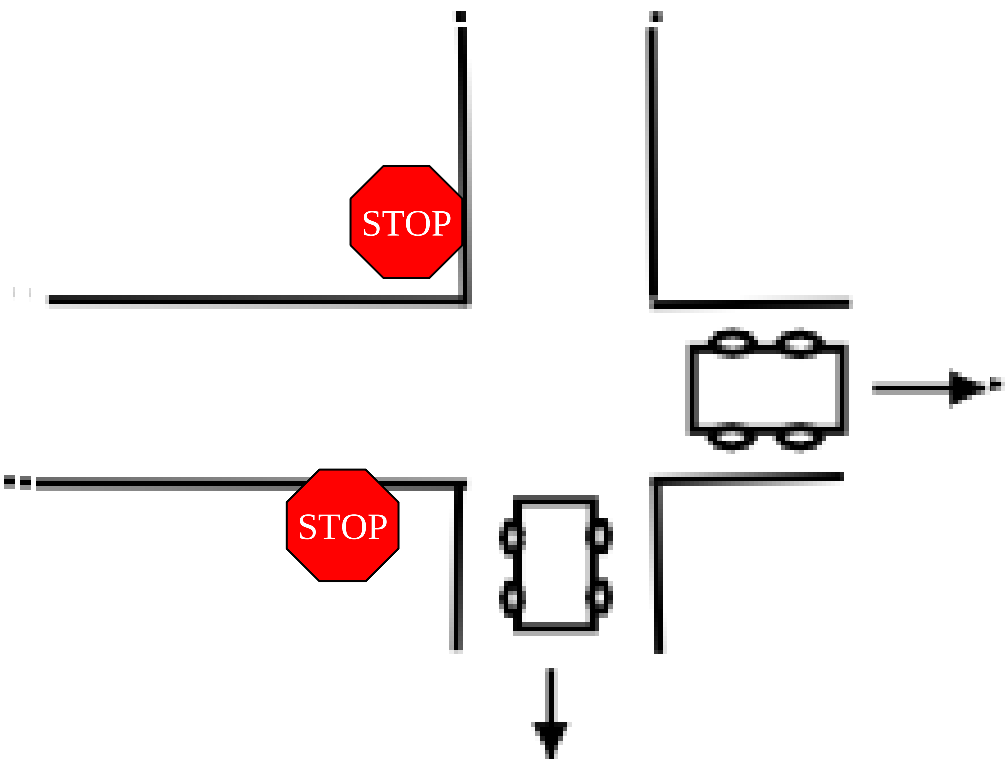
Our Solution: Transactions

```
cilk void increment() {
  xbegin
    x  = x + 1;
  xend
}
```

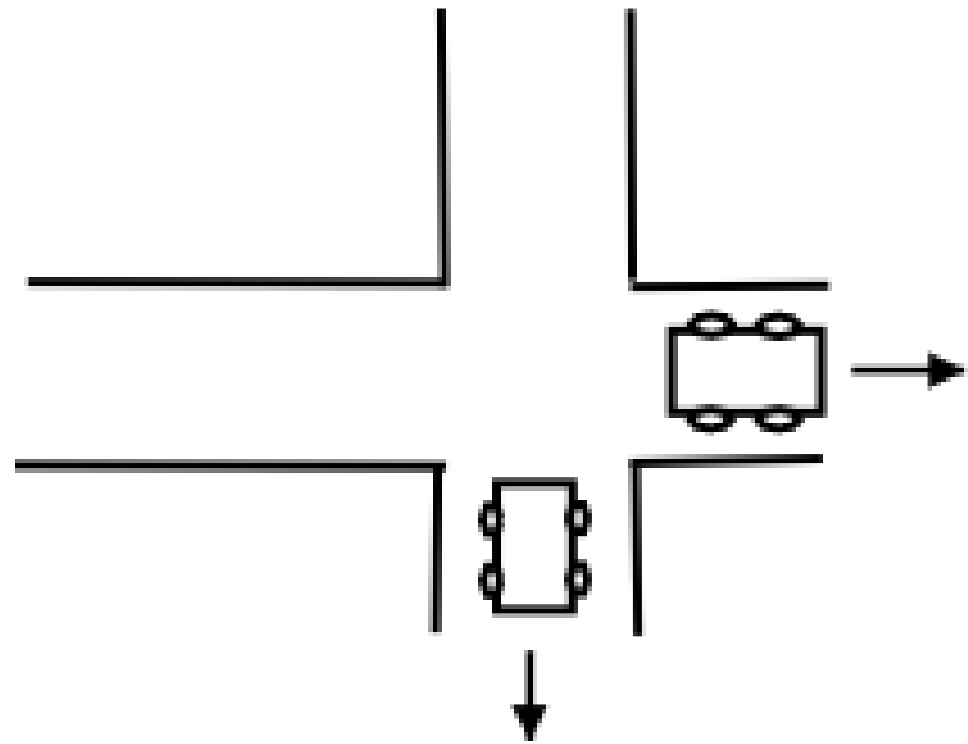
For this example, both solutions look the same. However, using transactions, to make any arbitrary section of code atomic, the programmer ideally needs only one *xbegin* and *xend*.

Locking vs. Transactions

Using Locks:



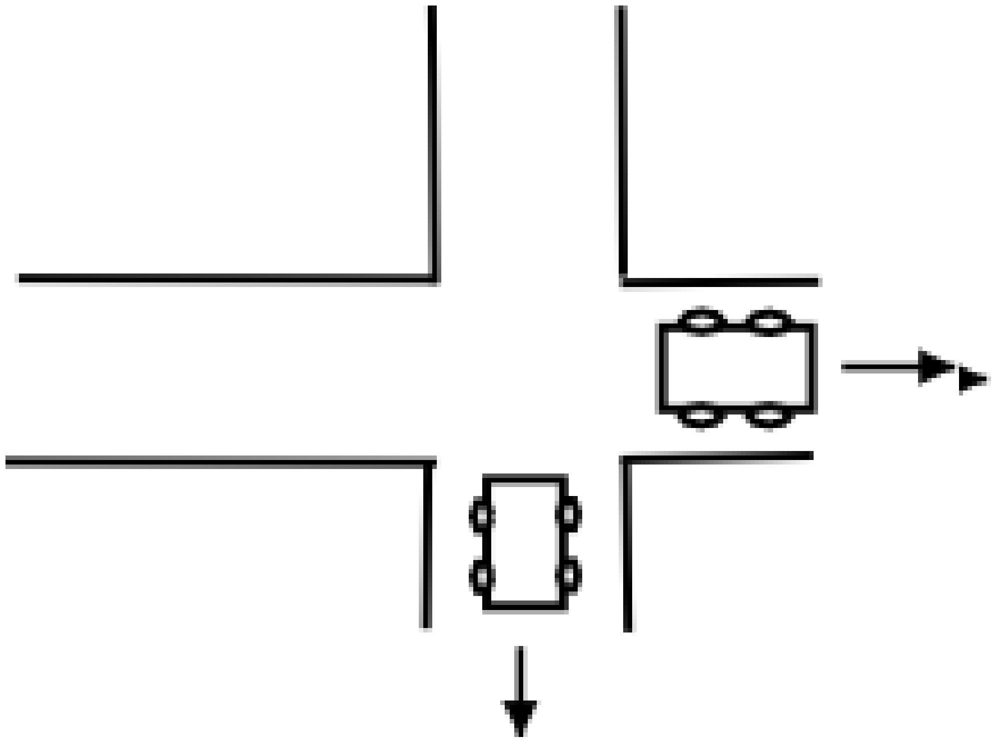
Using Transactions:



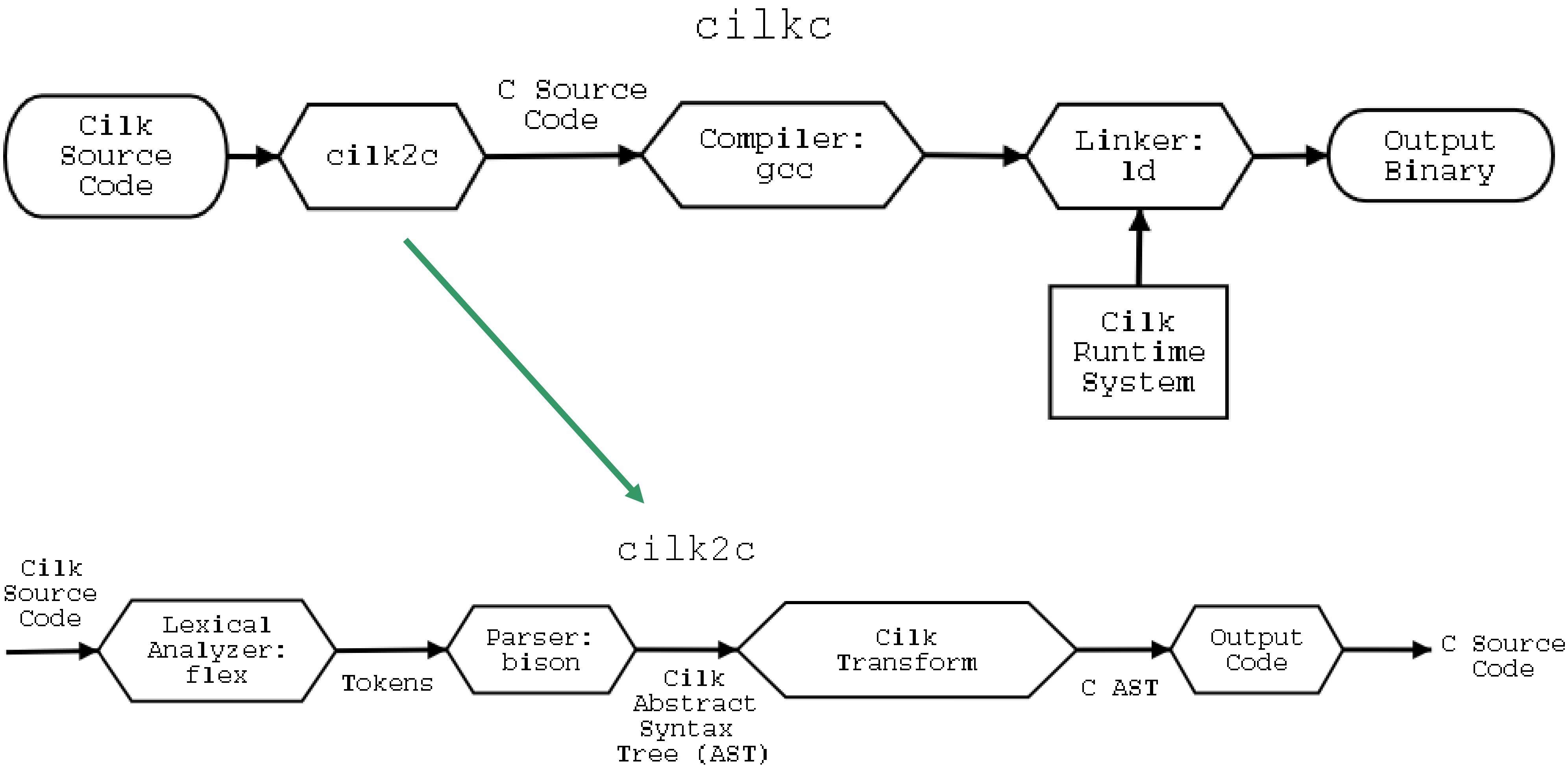
Acquiring a lock ensures that there will be no conflicts while code is executing.
With transactions, we go ahead and execute code, assuming conflicts are unlikely.

A Transaction With A Collision

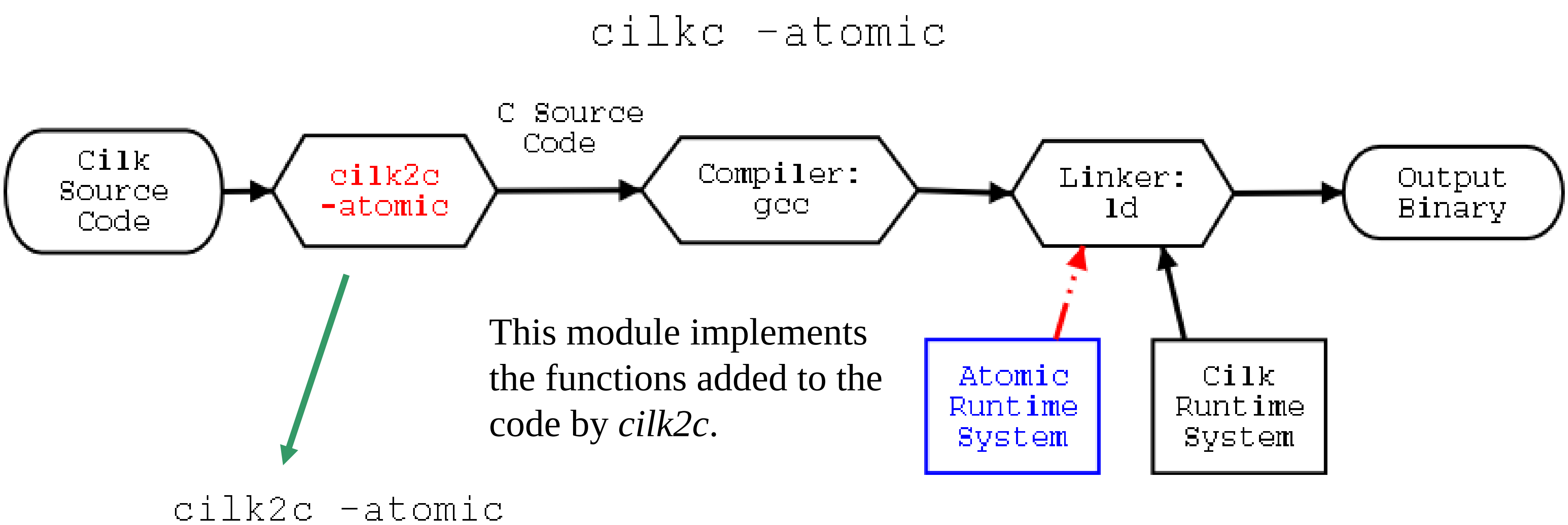
When a conflict does occur, at least one of colliding transactions must abort, restore everything back to the same state before the transaction, and then try again.



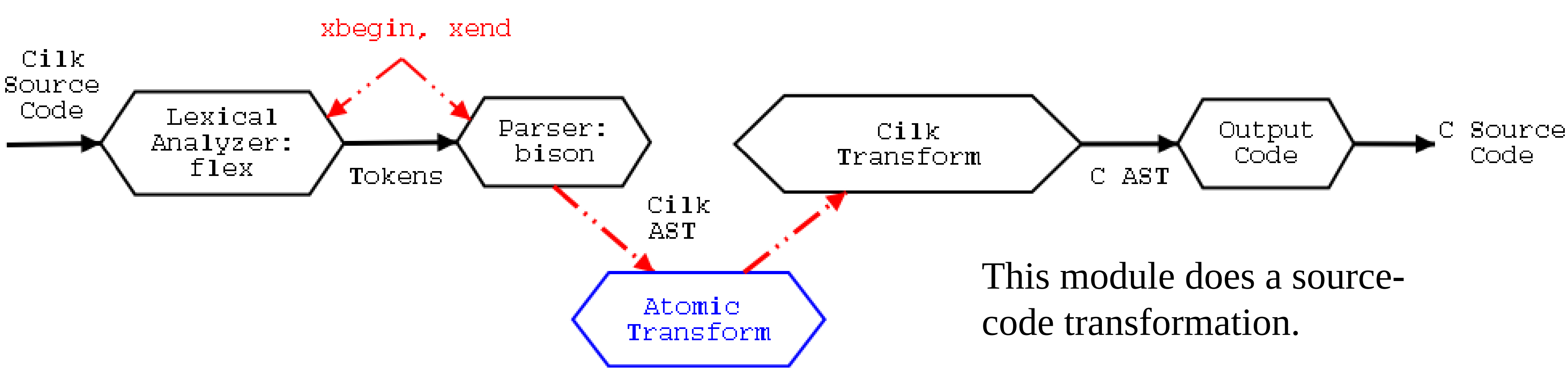
Steps of the Existing Cilk Compiler



Compiler Modified For Atomic Transactions



This module implements the functions added to the code by *cilk2c*.



This module does a source-code transformation.

Code Transformation for a Transaction

```
xbegin
   - - - -
xend
```

1.
```
Atomic Context* ac = createNewAC();
initTransaction(ac);
```

2.
```
attemptTrans:
      - - - -
   goto tryCommit;
```

3.
```
failed:
   doAbort(ac);
   doBackoff(ac);
   goto attemptTrans;
```

4.
```
tryCommit:
   if (failedCommit(ac))
     goto failed;
```

5.
```
done:
    destroyAC(ac);
```

cilk2c inserts labels and goto statements into the code for executing transactions.

1. Create atomic context for each transaction.
2. Execute main body of the transaction.
3. Handle conflicts.
4. Try to commit transaction.
5. Clean up after a successful transaction.

Inside the Body of a Transaction

Load:

```
x;
```

```
(( { if (AtomicReadFailed (&x, sizeof(x),
                           atomicContext))
        goto failed;
    }),
   x);
```

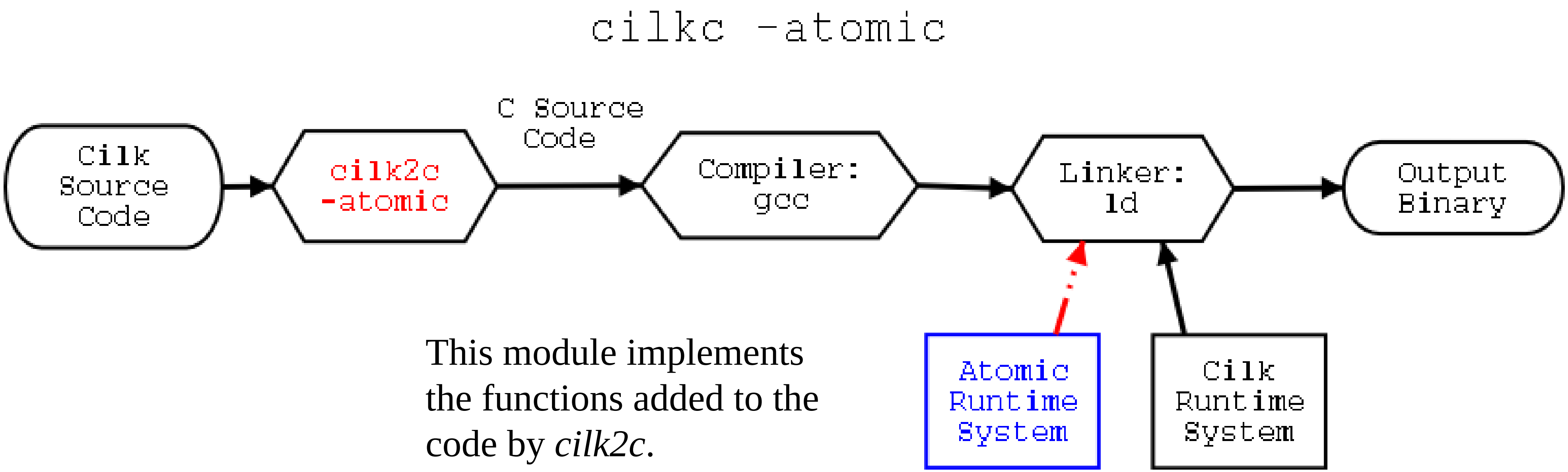
Store:

```
x = 1;
```

```
{ int *tempAddressX = &x;
   ({ if (AtomicWriteFailed(tempAddressX,
                            sizeof(*tempAddressX),
                            atomicContext))
        goto failed;
    });
    *tempAddressX = 1;
}
```

cilk2c transforms every load and store. The extra code around each load/store detects if a conflict has occurred and backs up the original values in case we have to abort.

Atomic Runtime System

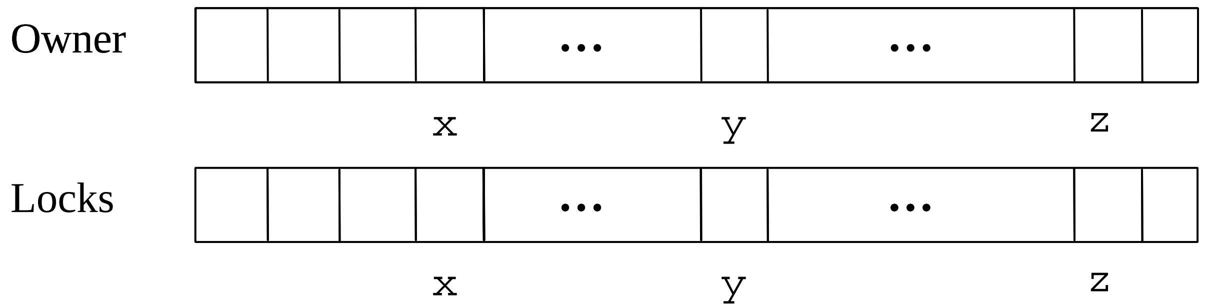


For every memory location that has been accessed by a currently executing transaction, the runtime system keeps track of:

1. *Owner*: the transaction that is allowed to access the location .
2. *Backup Value*: the value to put back in case of an abort.

How ~~fast~~ slow are transactions in software?

- We have the overhead of creating/destroying a transaction.
- We have to make a function call with each load/store.
- Unfortunately, to ensure operations on the owner array occur atomically, we use locks.



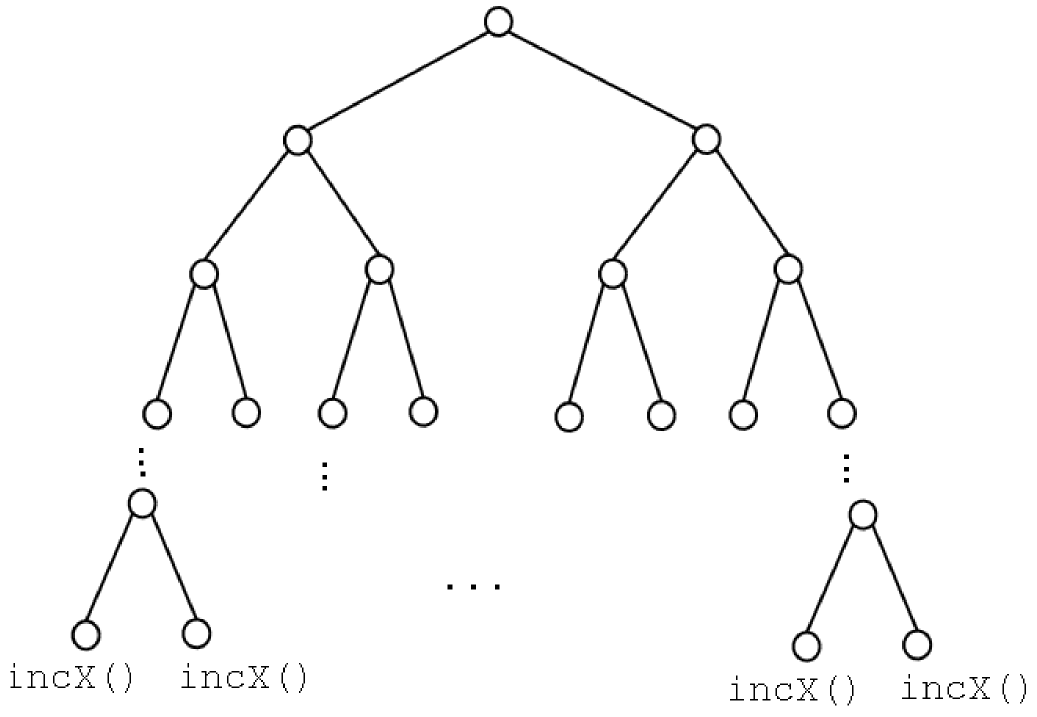
- Ideally, we would have hardware support for the runtime system.

An Experiment

```
int x = 0;

cilk void incX() {
   x = x + 1;
}

cilk void incrementTest(int n) {
  if (n > 0) {
     if (n == 1) {
       incX();
     }
     else {
       spawn incrementTest(n/2);
       spawn incrementTest(n-n/2);
       sync;
     }
  }
}
```



Preliminary Results

On n = 10,000,000:

	Running time (s)	Final x	Correct?	Transactions Aborted / Total Aborts
1 processor	7.4 s	10,000,000	Y	-
2 processors	8.6 s	9,938,893	N	-
1 proc, with Cilk_lock	8.1 s	10,000,000	Y	-
2 proc, with Cilk_lock	9.8 s	10,000,000	Y	-
1 proc, atomic	25.8 s	10,000,000	Y	0
2 proc, atomic	25.7 s	10,000,000	Y	4657/6712

In last case, max # times a transaction was aborted: 8

A Longer Transaction:

```
int x = 0;

cilk void incX() {
   int j = 0;
   for (j = 0; j < 100; j++) {
      x = x + 1;
      x = x - 1;
   }
   x = x + 1;
}
```

On n = 10,000,000:

	Running time (s)	Final x	Transactions Aborted
1 processor	11.6 s	10,000,000	-
2 processors	29.9 s	7,192,399	-
1 proc, with Cilk_lock	14.2 s	10,000,000	-
2 proc, with Cilk_lock	34.9 s	10,000,000	-
1 proc, atomic	605 s	10,000,000	0
2 proc, atomic	612 s	10,000,000	2

Max # times a transaction was aborted: 30

Conclusion

- Options for further work:
  - Test more complicated transactions.
  - Modify *cilkc* to be more user-friendly and portable.
  - Improve runtime system.
  - Experiment with different backoff schemes.
  - More testing!
- We have a version of Cilk which can successfully compile and execute simple transactions atomically.

A Transaction with Random Memory Accesses

```
int x[10];

cilk void incX() {
   int j = 0;
   int i = rand() % 10;
   for (j = 0; j < 100; j++) {
      i = rand() %10;
      x[i] = x[i] + 1;
      x[i] = x[i] - 1;
   }
   x[i] = x[i] + 1;
}
```

n = 100,000:

	Running time (s)	Sum x[i]	Transactions Aborted / Total Aborts
1 processor	2.2 s	100,000	-
2 processors	30 s	99,987	-
1 proc, with Cilk_lock	3.1 s	100,000	-
2 proc, with Cilk_lock	32.1 s	100,000	-
1 proc, atomic	15.9s	100,000	0/0
2 proc, atomic	16.4 s ????	100,000	6/53

Max # times a transaction was aborted: 24

A Correct Execution Sequence

```
int x = 5;
int y = 0;
int z = 1;

cilk void foo() {
  xbegin
    x = x + 1;
    y = x;
  xend
}

cilk void bar() {
  xbegin
    z = 42;
    y = y + 1;
  xend
}

cilk int main() {
  spawn foo();
  spawn bar();
  sync;
}
```

1: read x
1: write x
1: read x
1: write y
1: commit

time ↓

2: write z
2: read y
2: write y
2: commit

A Successful Transaction

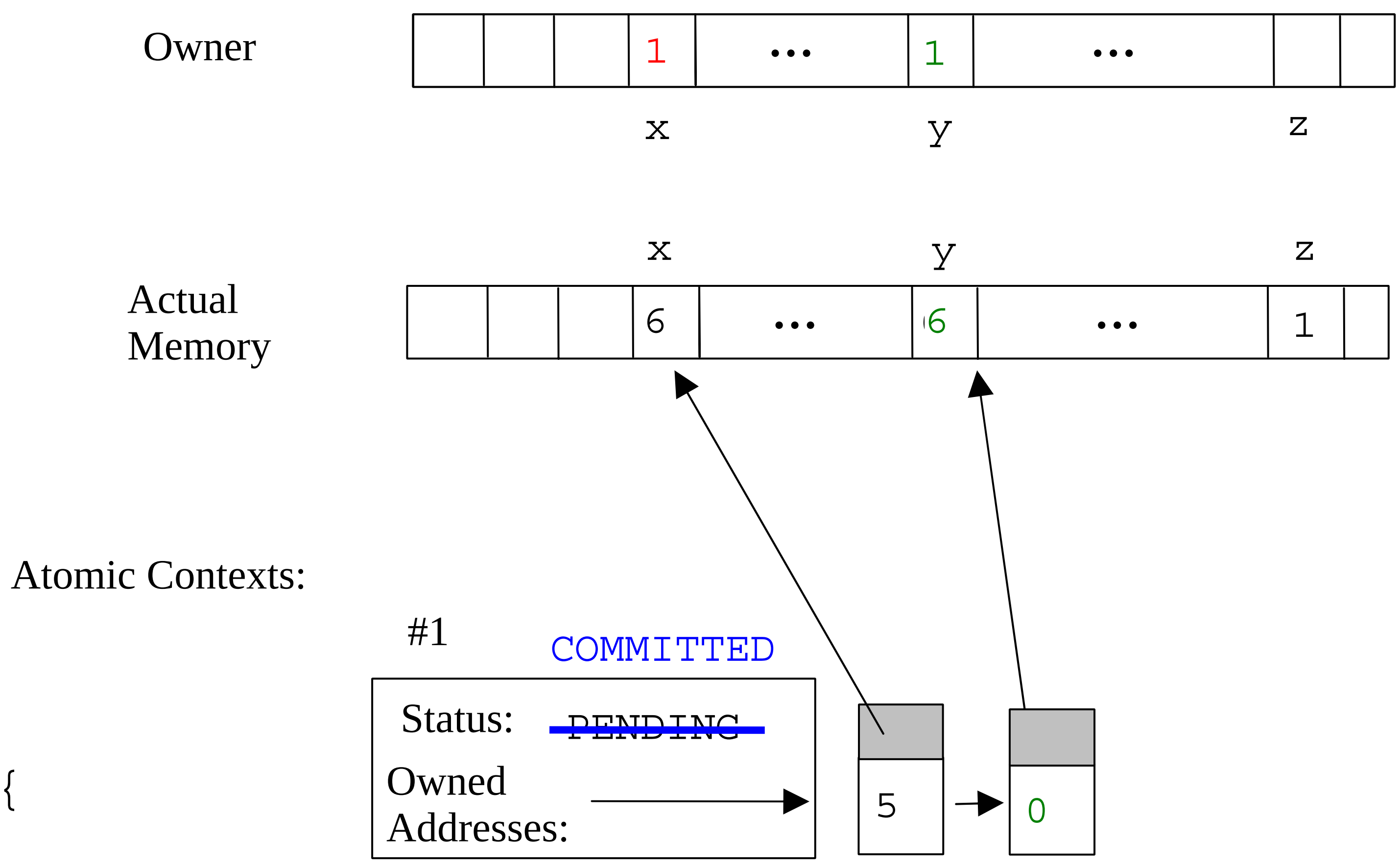
1: read x

1: write x

1: read x

1: write y

1: commit




```
cilk void foo() {
  xbegin
    x = x + 1;
    y = x;
  xend
}
```

Conflicting Transactions

1: read x
1: write x
1: read x
1: write y

2: write z
2: read y
2: abort

1: commit