Atomic Transactions in Cilk

Jim Sukha

12-13-03

Contents

Abstract

Programmers typically avoid data races in multi-threaded programs by using locks. Another method for enforcing atomicity which is simpler for the programmer but which requires a more complicated memory system is to specify sections of code as transactions. By modifying the Cilk compiler and implementing a runtime system in software, we have successfully constructed a system that allows programmers to specify transactions and execute them atomically. In our experiments, transactional programs ran up to 250 times slower than the corresponding implementation with locks. However, our system is flexible enough to allow experimentation with different designs in the future.

1 Introduction

1.1 Determinacy Races in Multi-Threaded Programs

When writing multi-threaded programs, correctly managing access to shared data is often a challenging task. Consider the Cilk program in Figure 1.

```
int x;

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

cilk int main(void) {
  x = 0;
  spawn increment();
  spawn increment();
  sync;
  printf("x is %d\n", x);
  return 0;
}
```

Figure 1: A simple program with a determinacy race.

During execution, the program spawns two threads that both try to increment the global variable x. We might expect x to have a final value of 2. However, in this program, there is a ***determinacy race*** on x. The statement $x = x + 1$ consists of a read and then a write to x, so the output may actually be 1 or 2, depending on how the reads and writes get interleaved. Figure 2 illustrates two interleavings that produce different outputs.

a)

Thread 1	Thread 2
read x	
write x	
	read x
	write x

b)

Thread 1	Thread 2
read x	
	read x
write x	
	write x

Figure 2: Two possible interleavings of reads and writes for the program in Figure 1. (a) After execution, the final value of x is 2. (b) After execution, the final value of x is 1.

The determinacy race in this simple program causes the execution to be non-deterministic. This bug in the code could be eliminated if we could guarantee that the write to x in one thread always completes before the read of x in the other. We want the execution of `increment()` to be ***atomic***, so that the partial execution of one instance of the function does not affect the results of the other.

Programmers typically enforce atomicity in their code by using locks to control access to shared data. In the case of `increment()`, we would simply lock x, increment x, and then unlock x. However, in more complicated code using locks is not always as straight-forward. Often programmers must use complicated locking protocols to achieve good performance while still avoiding problems such as deadlock. Writing and debugging programs with locks can be cumbersome, so this approach to enforcing atomicity is far from ideal.

1.2 Atomicity through Transactions

A different approach to enforcing atomicity in concurrent programs is to use transactions. In a program, a ***transaction*** is section of code that should be executed atomically. While transactional code is being executed, we change memory only tentatively. If, after completing the transaction code, we detect that the execution occurred atomically, then we ***commit*** the transaction by making the changes to memory permanent. Otherwise, if there is a conflict with another concurrent transaction, we ***abort*** the transaction by discarding all changes to memory.

In this framework, we execute a section of code atomically by specifying it as a transaction, and then attempting to run the transaction until it is successfully committed. In programs where we expect relatively few conflicts, this approach is, at least in theory, faster than locking the shared memory locations every time the transaction is executed.

Ideally, a programmer should be able to execute a transaction atomically by simply enclosing the relevant section of code between two keywords, `xbegin` and `xend`. Figure 3 illustrates the transaction for `increment()`.

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

Figure 3: Conversion of the code in Figure 1 into a transaction, using the keywords `xbegin` and `xend`.

Transactions are simple for the programmer, but this approach requires a more complicated memory system that can handle the abort operations. One way to support such atomic transactions is to use ***transactional memory***, a system first proposed by Herlihy and Moss [1]. In their scheme, the hardware provides machine instructions for maintaining the set of shared memory locations modified by a transaction during an execution attempt. Implementations of software transactional memory have been proposed as well [2].

1.3 Atomic Transactions in Cilk

We implemented a system that allows programmers to execute simple atomic transactions in Cilk. Users can compile programs like those shown in Figure 3 by using Cilk's compiler with the appropriate command-line option, `cilkc -atomic`.[1]

Programmers can specify transactions by enclosing the appropriate code in a ***transaction block***. `xbegin` and `xend`, the Cilk keywords that denote the beginning and end of a transaction block, obey the following rules:

- `xbegin` and `xend` are Cilk keywords, so a parse error is generated if they are encountered in any C program (any file that ends in `.c`). However, a C function in a Cilk program may contain `xbegin` and `xend`. This is useful when C functions are called from Cilk functions.
- A transaction block `xbegin ... xend` has the same semantics as a normal block, { `...` } with respect to variable scope. Declarations of local variables within a transactional block must occur before any statements.

Ideally we could handle any code inside a transaction block. However, our current implementation has the following restrictions:

[1]This feature has not been implemented yet. Compilation of user programs is currently done using a Makefile.

- Transaction blocks can only contain C code. This means spawn and sync statements are not allowed inside a transaction.
- Only functions that do not access any shared variables can be safely called inside a transaction block. A function call temporarily jumps out of the transaction.
- In a non-void function, `return` statements can not appear inside a transaction block.
- Other statements that cause jumps in or out of the transaction are not allowed. This includes `label`, `goto`, or `break` statements.
- Nested transactions are not permitted. With nested `xbegin` and `xend` keywords, only the outermost pair has any effect. The inner transactions are subsumed into the outermost transaction. Nesting by calling a function that contains a transaction inside another transaction does not work.

2 Project Overview

In this section, we provide an overview of the existing Cilk compiler and the changes we made to support atomic transactions. Section 3 describes in greater detail the changes made to the compiler, while Section 4 describes the memory system implemented in software. Finally, Section 5 discusses some experimental results obtained by using our system to compile some simple transactions.

We first describe how normal Cilk programs are compiled. The stages of the latest version of the compiler, `cilkc`, are illustrated in Figure 4. First, `cilk2c` transforms the input, the user's Cilk source code, into C source code.[2] The C code is then compiled by `gcc`, and the Cilk runtime libraries are linked in to produce an output binary. The Cilk runtime system implements the functions `cilk2c` added to the user's source code.

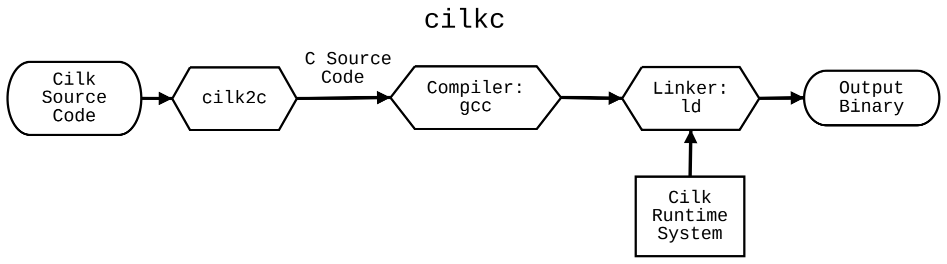

Figure 4: The stages of the current version of the Cilk compiler, `cilkc`.

Figure 5 illustrates the changes to the compiler. First, we transform the user's transactional Cilk source code using `cilk2c` with the command-line option, `-atomic`. We then link in an additional runtime library that implements the functions we added to the source code.

`cilk2c` only does a source code transformation, adding calls to functions implemented by the atomic runtime system. The interface between the two modules is the specifications for these functions. These specifications, which are described in Section 3, are not very restrictive. This gives us a lot of freedom in the design of the runtime system. In particular, if we had actual hardware transactional memory, ideally we should be able to implement the atomic runtime system using the new machine instructions without making further modifications to `cilk2c`.

3 Compiler Support

In this section we describe our changes to `cilk2c` and how `cilk2c` transforms transactional code.

[2]Technically, the user's input code gets run through a C pre-processor first before going through `cilk2c`.

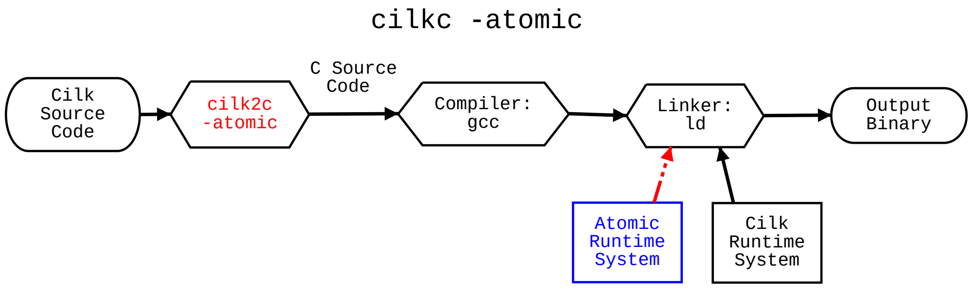

Figure 5: The stages of the Cilk compiler supporting atomic transactions, `cilkc -atomic`.

3.1 Overview of `cilk2c`

Figure 6 illustrates the stages of the original `cilk2c`. First, the lexical analyzer takes the pre-processed Cilk source code and splits it into tokens. The parser then processes the tokens and generates a Cilk abstract syntax tree (AST). The transformation module converts the Cilk AST into the corresponding C AST, and the output module generates a file containing the final C source code.

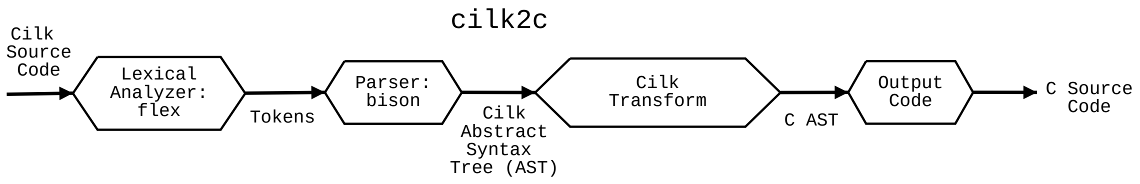

Figure 6: The stages of the existing `cilk2c` module.

Figure 7 shows our modifications to `cilk2c`. The lexical analyzer and the parser were both altered to accept `xbegin` and `xend` as Cilk keywords. We added an atomic transform module that converts the Cilk AST with `xbegin` and `xend` into another Cilk AST that handles transactions. This Cilk AST is then converted as before into a C AST and then output to file.

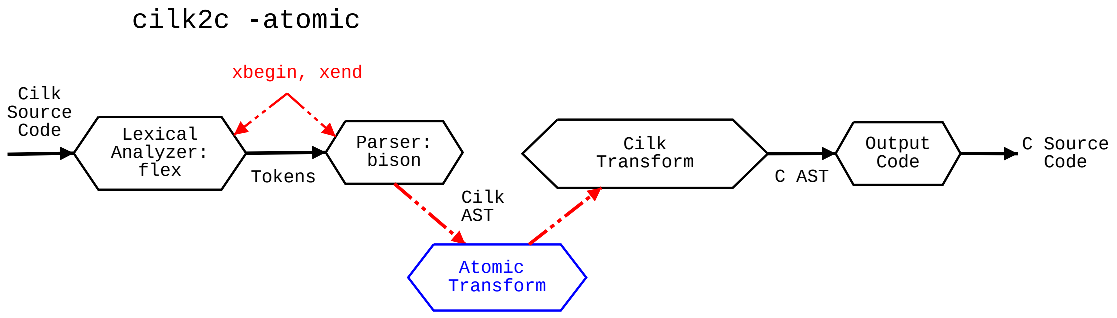

Figure 7: `cilk2c` modified to handle atomic transactions.

3.2 Code Transformation for Transactions

The code that `cilk2c` generates for a transaction can be divided into two components. First, `cilk2c` transforms `xbegin` and `xend` into code that initializes and attempts to run the transaction. `cilk2c` also

```
cilk void emptyTransaction() {
{
  {
    /** 1. Create atomic context **/
    AtomicContext *ac = createAtomicContext();

    /** 2. Attempt transaction **/
    attempt_transaction_lbl:
      initTransaction(ac);

      /** 3. Empty transaction body **/

    /** 4. Attempt commit **/
    try_commit_lbl:
      if (failedCommit(ac))
         goto failed_lbl;
      goto committed_lbl;

    /** 5. Abort transaction **/
    failed_lbl:
      doAbort(ac);
      doBackoff(ac);
      goto attempt_transaction_lbl;

    /** 6. Clean up after successful transaction **/
    committed_lbl:
      completeTransaction(ac);
      destroyAtomicContext(ac);
  }
}
```

Figure 8: Code transformation of an empty transaction block.

transforms the body of the transaction so that modifications to shared variables can be undone if an abort occurs.

3.2.1 Creation of a Transaction

We consider what happens when `cilk2c` encounters an empty transaction block.

```
cilk void emptyTransaction() {
  xbegin
     /* Empty transaction body */
  xend
}
```

The atomic transform module converts the empty block into the code shown in Figure 8. The transformed code has the following important components:

1. For each transaction, the runtime system creates an `AtomicContext` object that stores all the state information. In particular, the atomic context keeps a list of the shared memory locations accessed by the transaction and the number of times the transaction has aborted.

2. Every time we attempt a transaction, the runtime system may need to do some initialization.
3. Usually, the transformation of the body of the transaction goes here. We discuss this transformation in more detail in Section 3.2.2.
4. We try to commit the changes to memory when we successfully reach the end of a transaction. A conflict is still possible at this stage, so `failedCommit()` both tries to commit and returns `true` if we did not succeed.
5. If there is a conflict while executing the body of the transaction, we abort by jumping to this label. The function `doAbort()` restores memory back to its original state before starting the transaction, and `doBackoff()` causes the thread to wait a while before attempting the transaction again.
6. We jump to this label if we have successfully committed the changes to memory. We cleanup by destroying the `AtomicContext()` object.

Because we are relying on `label` and `goto` statements for control flow, this kind of transformation does not work if we jump in and out of the transaction body. Many of the restrictions described in Section 1.3 follow from this implementation. With additional code transformations, we may be able to support `return` statements inside a transaction. We might handle function calls by creating two versions of each function: a normal version and an transactional version. This would be similar to the fast and slow clones for Cilk functions.

3.2.2 Transformation of a Transaction Body

Inside the body of a transaction, we instrument every load and store of shared variables. This transformation is needed for detecting conflicts with other transactions and for saving the original value of variables in case we have to abort.

Cilk's profile module already successfully identifies every load and store in user code. We modified the existing profiler to implement the transformations for transactions. Inside a transaction body, a load instruction such as `x;` is transformed into

```
( ({  if (AtomicXReadFailed((const void *const) &x,
                            sizeof (x),
                            ac))
          goto failed_lbl;
   }),
  x);
```

Similarly, a store instruction such as `x = 1;` is transformed into

```
{
    int *__tempAddress = &x;
    ({ if (AtomicXWriteFailed((const void *const) __tempAddress,
                              sizeof (*__tempAddress),
                              ac))
            goto failed_lbl;}
    );
    *__tempAddress = 1;
}
```

Before every load or store, we first call a runtime system function to check if we can safely access this location. This function takes as arguments the memory address to access, the size of the location, and the atomic context of the current transaction. If there is a problem, then we jump to the appropriate label and abort the transaction. Otherwise, we continue and access the value as usual.

In our implementation, if a call to `AtomicXReadFailed()` or `AtomicXWriteFailed()` ever returns `false`, the transaction is able to safely access the memory location in question until the transaction commits or aborts. Therefore, we can safely read or write directly to the memory location in the next line. With a different implementation, we might also need to call a function, `AtomicXWritten()`, after we do a store.

The Cilk profiler code does not instrument every load and store operation, but optimizes by ignoring accesses to register variables. We also leave these loads and stores unmodified.[3]

3.2.3 Interaction between Transactions and Normal Code

A programmer might conceivably spawn one thread that modifies a variable inside a transaction and one thread that modifies the same variable in normal code. We have several choices regarding this type of conflict. One option, mentioned by Herlihy and Moss, is to simply treat the non-transactional code as a transaction that always commits. In this case, a transaction should abort if it detects a non-transactional function modifying the same variables.

To implement this alternative, we need to instrument every load and store in the program, not only those inside transactions. Again, by modifying the existing profiler code, we transform a load `x;` into

```
( AtomicRead((const void *const) &x, sizeof (x)), x);
```

Similarly, a store `x = 1;` is converted into

```
{
  int *__tempAddress = &x;
  AtomicWrite((const void *const) __tempAddress,
              sizeof (*__tempAddress));
  *__tempAddress = 1;
  AtomicWritten((const void *const) __tempAddress,
                sizeof (*__tempAddress));
}
```

This transformation is simpler because outside a transaction, there is no atomic context and no label to jump to for an abort. We call the functions `AtomicRead()` or `AtomicWrite()` to notify the runtime system that we are reading or writing to a variable. Note that we also need an `AtomicWritten()` function to notify the runtime that we have completed a write.

Unfortunately, treating non-transactional code as a transaction that always commits introduces even more overhead into user code. Instrumenting a load or a store requires one or two additional function calls. We might expect transactional code to be slow, but this alternative potentially slows down all the code in the system. Note that our system will not even work if we call any functions whose source code we can not modify (compiled libraries, for example).

The other alternative is to allow non-transactional code to function as usual. In this case the programmer is responsible for avoiding conflicts between transactional and non-transactional code. This alternative seems like a reasonable tradeoff between programmer effort and program efficiency. Although we have modified the system for the first alternative, the latter is currently the default option.

4 Runtime System

The atomic runtime module controls the behavior of our version of transactional Cilk. In this section, we briefly describe the basic principle behind our implementation and discuss some implementation details.

[3]Currently there are some cases where this optimization not correct. There is a relatively straight-forward fix for this which has not been implemented yet.

4.1 The Basic Approach

To execute a transaction atomically, we ensure that the transaction has exclusive access to all the shared memory locations it touches. We accomplish this by storing an ***owner*** for each memory location in an owner table. A transaction must first acquire ownership of a location before accessing it.

Our approach to conflict detection and resolution is straight-forward. Once a transaction gains ownership of a particular location, it does not relinquish ownership until the transaction is either committed or aborted. A transaction aborts if, at any point, it discovers that it can not gain ownership of a memory location it needs. Conceptually, this is similar to having each transaction lock all the memory locations it needs. However, there are a few important distinctions:

- A transaction tries to acquire ownership of a memory location immediately before its first access. In a simple locking scheme we need to know all the memory locations that will be modified and try to lock them all at the start of the transaction.
- Deadlock does not occur because a transaction gives up ownership of all its memory locations when it aborts.
- Livelock, however, is possible. To avoid having competing transactions that continually abort each other, we need a backoff protocol that stops aborted transactions from restarting immediately.
- We have a blocking implementation because we use locks when maintaining the owner table. However, we only need to hold a lock while modifying the owner table, not for the entire transaction.

4.2 Specification of Runtime System Functions

The code transformation done by `cilk2c` is generic enough to handle many different implementations of the runtime system. The crucial step is making sure all the runtime functions interact correctly with each other. In our implementation, the functions operate as follows:

- `Bool AtomicXReadFailed(const void* address, int size, AtomicContext* ac)`: After a call to this function, the transaction with the specified atomic context tries to get ownership of all the required memory locations. The function returns `true` if it fails to get ownership. If the transaction is accessing a location for the first time, it also saves the address of the location and the original value in its atomic context.
- `Bool AtomicXWriteFailed(const void* address, int size, AtomicContext* ac)`: This does exactly the same thing as the read function. We currently do not distinguish between reads and writes.
- `AtomicContext* createAtomicContext()`: This creates the atomic context for a transaction. Initially, the transaction owns no memory locations, has a status field of `PENDING`, and has been aborted 0 times.
- `void InitTransaction(AtomicContext* ac)`: This is not needed in our implementation. We only do initialization once per transaction, not once per attempt, so `createAtomicContext()` is sufficient.
- `Bool failedCommit(AtomicContext* ac)`: This function is called when we are trying to commit. If the status field of the atomic context is `ABORTED`, then we fail and return `true`. Otherwise, we change the field to `COMMITTED` and free all owned memory locations. In our system we read and write from the usual location in memory and copy back an old value only during an abort. Therefore on a commit, we do not need to copy final values into memory.
- `void doAbort(AtomicContext* ac)`: Because we choose to read and write from the normal memory location, an abort operation is now more expensive. During an abort, we traverse the list of owned memory locations stored in the atomic context and write the original values back to memory. We maintain ownership of a location until we write the old value back, so a data race at this step is impossible.

- `void doBackoff(AtomicContext* ac)`: This function causes the transaction to wait some time before executing again. We implemented a simple backoff algorithm where the thread sleeps for 2^{n-1} microseconds after an abort, with n is the number of times the transaction has previously been aborted. As we discuss in Section 5.3, there are several flaws with the current implementation.

- `void completeTransaction(AtomicContext* ac)`: In our system nothing needs to be done after we have successfully committed. We use this function to collect some runtime statistics.

- `void destroyAtomicContext(AtomicContext* ac)`: This function frees up the memory used by the atomic context.

As we briefly mentioned in Section 3.2.3, we have several options when transactional code and non-transactional code both try to access the same location. If we simply require the programmer to avoid this case, then no additional effort on our part is required. However, if we treat non-transactional code as a transaction that always commits, then we require additional runtime system functions.

- `void AtomicRead(const void* address, int size)`: When non-transactional code tries to do a read, we do nothing. This means we are only guaranteeing atomicity of a transaction with respect to other transactional code. Non-transactional code can read the intermediate value during a transaction.

- `void AtomicWrite(const void* address, int size)`: When non-transactional code wants to do a write, it grabs ownership of the memory location, labeling the owner field as the global context. If another transaction owned the memory location before, the status field of that transaction is set to `ABORTED`. This is the only case in our system when a transaction does not abort itself. If there are multiple non-transactional threads trying to access the same location, there are several global owners. We use an additional table to store the number of global owners of each memory location.

- `void AtomicWritten(const void* address, int size)`: This is called after the non-transactional code completes the write. We releases ownership of the location if there are no other non-transactional threads accessing the same location.

4.3 Data Structures and Implementation Details

We use the following data structures in our implementation:

- *Owner Table*: We store the owner of each memory location in a global hash table, using a pointer to transaction's `AtomicContext` as a key. Unfortunately, modifications to this owner hash table do not occur atomically, so we require a second hash table of locks. Although this is a blocking implementation, this method should hopefully be less expensive than locking every memory location because we only hold a lock for one access or modification of the owner table instead of for the entire transaction.

  Currently, the hash table is an array of size 1000, and memory locations are hashed as `address % 1000`. Each entry in the hash table represents a byte. This means a transaction gains ownership of one byte for every memory location that hashes to the same spot in the array.

- *Atomic Context*: The `AtomicContext` object stores a status flag that can be `PENDING`, `ABORTED`, or `COMMITTED`, and a lock that must be held when accessing this flag. The atomic context also stores a linked list of owned addresses. Each element in this list consists of two pointers: one to the actual location in memory, and one to a location that memory we allocate to store the backup value for aborts. Finally, the atomic context stores counters used for calculating backoff.

4.4 Design Rationale and Alternatives

As with the modifications made to `cilk2c`, many of the design choices were made to simplify the implementation. The primary goal was to produce a correct implementation, with performance being of secondary importance. There are several alternatives that we might adopt in future implementations:

- *Granularity of Memory Division*: Currently transactions take ownership of an single byte at a time, plus those which hash to the same location in our owner array. It might be more reasonable to divide memory into larger blocks (for example, the size of a cache-line). A large block size prevents concurrent access to locations that are close together, but a small block size increases the number of places we need to grab ownership of when accessing a large data structure.

- *Hash Table Implementation*: The size and hashing function we use for the owner table also might be chosen more carefully.

- *Backoff Algorithm and Conflict Resolution*: Exponential backoff may back off too quickly to be effective. There are several problems with the current backoff implementation, as we describe in Section 5.3. Because we are doing transactions in software, backoff is not the only available option. We might consider a scheme where conflicting transactions sleep until they are woken up by a competing transaction that successfully commits. In a practical system, some combination of these approaches might also be reasonable.

5 Experimental Results

We ran several experiments to test our implementation. All experiments were run on a machine with two 2.4 Ghz Intel Xeon processors and 1 Gb of RAM. All times reported are elapsed times measured with `/usr/bin/time`.

5.1 Correctness Testing and Timing Results

We tested the correctness of our system on four sample programs. For three of these programs, we compared the running times of three different versions of the code: the original version with a data race, a version using locks, and the version compiled using our system for atomic transactions.

5.1.1 Simple Incrementer

We start with the `increment()` function described in Section 1.1. We spawn N threads, each trying to increment the global variable x once. First, we compiled a simple version without using locks or transactions. Because the transaction is so short, when we spawn N threads one by one in a for loop, the code ends up executing serially, with no speed-up from one to two processors. To get more parallelism, and therefore more opportunities for conflict, we instead spawn `increment()` at the leaves of a binary tree. This code is shown in Figure 9, and the tree-structure is illustrated in Figure 10.

Even though the code in Figure 9 has a data race, for $N < 1000$, we still get the correct final value for x. We only start to see incorrect values for larger N. Every time two reads from x are followed by two writes to x, we miss incrementing x by one. Therefore the amount that x is less than N is a rough measure of how many conflicts occurred. Table 1 shows the final values we obtained with some large values of N. Inserting `xbegin` and `xend` keywords around `x = x + 1`, we always get the correct final value of x.

We also measured the running times of the three versions of the code for different values of N. This data is presented in Table 2. For this example, coding `increment()` using locks was only about 1.1 times slower than without locks. Using `xbegin` and `xend` is about 8 or 9 times slower on one processor. This slowdown is likely due to the overheads introduced by our code transformation. We discuss these overheads further in Section 5.2.

```
int x;
cilk void increment() {
   x = x + 1;
}
cilk void incNTimes(int n) {
  if (n > 0) {
    if (n == 1) {
      spawn increment();
      sync;
    }
    else {
      spawn incNTimes(n/2);
      spawn incNTimes(n - n/2);
      sync;
    }
  }
}
```

Figure 9: Code spawning a tree of N threads, all incrementing a global variable x.

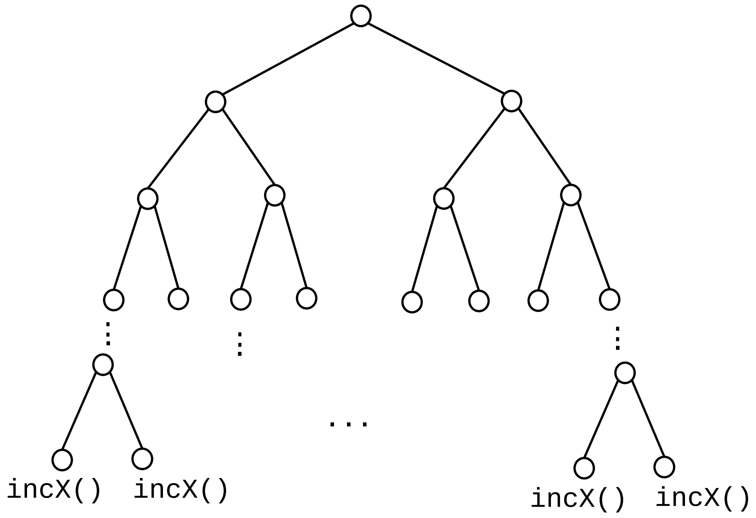

Figure 10: Tree of spawns generated by the code in Figure 9.

From the data in Table 2, we notice that using locks, we are actually getting speed-up going from one to two processors. In theory, there should be little to no parallelism in this program. This suggests the overhead of spawning `increment()` is much greater than the actual time to update x. We consider a modified version of the code, shown in Figure 11, that executes a for loop to lengthen the transaction. For $N = 100,000$, we obtained the running times shown in Table 3. The performance of our transactional system is much worse, with the code running 250 times slower than the original code on two processors. The version using locks remains about the same, being about 1.1 to 1.2 times slower than the original code.

N	Final value of x	x/N
500,000	497796	0.9956
500,000	497626	0.9952
500,000	497672	0.9953
500,000	497756	0.9955
1,000,000	992480	0.9925
1,000,000	993232	0.9932
1,000,000	993481	0.9925
1,000,000	993313	0.9933
2,000,000	1988110	0.9940
2,000,000	1988593	0.9943
2,000,000	1988561	0.9943
2,000,000	1988987	0.9945

Table 1: Final values of x after executing code in Figure 9 on 2 processors.

N	Original Version	Locking Version		Atomic Version	
	Running Time (s)	Time (s)	Ratio	Time(s)	Ratio
500,000, 1 proc.	0.38	0.41	1.1	3.13	8.2
500,000, 2 proc.	0.23	0.25	1.1	4.79	21
1,000,000, 1 proc.	0.74	0.82	1.1	6.23	8.4
1,000,000, 2 proc.	0.45	0.51	1.1	6.83	15
2,000,000, 1 proc.	1.46	1.62	1.1	12.50	8.6
2,000,000, 2 proc.	0.88	1.00	1.1	12.92	15

Table 2: Running times of different versions of `increment()` for different values of N. Computed ratios are compared to original version running time.

N	Original Version	Locking Version		Atomic Version	
	Running Time (s)	Time (s)	Ratio	Time(s)	Ratio
100,000, 1 proc.	0.15	0.18	1.2	32	213
100,000, 2 proc.	0.14	0.16	1.1	35	250

Table 3: Timings for different versions of modified increment code shown in Figure 11.

5.1.2 Random Array Increments

To test the system on transactions that affect multiple memory locations, we altered the transaction to increment random elements in an array, as shown in Figure 12.

Obtaining meaningful running times for this particular example is somewhat difficult. We discovered that for $N = 1,000,000$, the original code ran almost 7 times slower on two processors than on one. This is probably because the `rand()` function acquires a lock. Also, coding a version of this function with locks is more difficult. To get the correct result using an array of locks, we can generate `v` randomly, lock position `v`, increment, and then unlock position `v`. However, technically, if we want the entire transaction to appear atomic, we need to generate all the random locations `v` first and then grab locks in order to prevent deadlock.

5.1.3 Struct Manipulation

The next test example, shown in Figure 13, involves manipulation of a global struct. The timing data for $N = 1,000,000$ is presented in Table 4. The slowdown of our system is not as bad as with the modified

```
void increment() {
  int i;
  for (i = 0; i < 100; i++) {
     x = x + 1;
     x = x - 1;
  }
  x = x + 1;
}
```

Figure 11: The `increment()` function modified to require more computation.

```
int x[10];
cilk void increment() {
  int j, v;
  for (j = 0; j < 5; j++) {
     v = rand() % 10;
     x[v] = x[v] + 1;
     x[v] = x[v] - 1;
  }
  v = rand() %10;
  x[v] = x[v] + 1;
}
```

Figure 12: Code that accesses multiple random locations in an array, and increments one of them.

increment code, but not as good as with the original increment code.

N	Original Version	Locking Version		Atomic Version	
	Running Time (s)	Time (s)	Ratio	Time(s)	Ratio
1,000,000, 1 proc.	0.78	0.84	1.1	17	22
1,000,000, 2 proc.	0.48	0.56	1.2	34	71

Table 4: Timings for struct manipulation example (code in Figure 13).

```
typedef struct {
  int x;
  int* y;
  char z;
} TestStruct;
cilk void structInc(TestStruct* sPtr) {
  sPtr->x = (sPtr->x) + 1;
  *(sPtr->y) = *(sPtr->y) + 1;
  sPtr->z = (sPtr->x)%26 + ’a’;
}
```

Figure 13: Code that accesses a global struct.

```
cilk int main(int argc, char* argv[]) {
  int sum, sum2;

  spawn pushNTimes(N/2);
  spawn pushNTimes(N-N/2);
  sync;

  sum = spawn popNTimes(N/2);
  sum2 = spawn popNTimes(N - N/2);
  sync;

  printf("Final sum: %d \n", sum + sum2);
}
```

Figure 14: Code that concurrently does pushes and pops on a stack. Each push and pop constitutes a separate transaction.

5.1.4 Linked List Manipulation

Our final test example, shown in Figure 14, involves concurrently pushing and popping elements from a linked-list implementation of a stack. We first push N numbers onto the stack, and then pop them off and verify that the sum is correct. For different values of N, we obtained the running times shown in Table 5. In all but one case, the atomic version of the program was less than 40 times slower than the version using locks.

N	Locking Version Running Time (s)	Atomic Version Time(s)	Atomic Version Ratio
100,000, 1 proc.	0.06	2.3	38
100,000, 2 proc.	0.10	2.9	29
500,000, 1 proc.	0.32	12	38
500,000, 2 proc.	0.50	13	26
1,000,000, 1 proc.	0.63	27	43
1,000,000, 2 proc.	1.13	27	24

Table 5: Running times for inserting N items into a linked list (code in Figure 14). Ratio is running time for the atomic version compared to the locking version.

5.2 Transaction Overhead

It is clear from our timing data that using `xbegin` and `xend` is significantly slower than simply using locks. We ran several tests to try to estimate some of the overheads introduced by the code transformation and by the calls to our runtime functions.

5.2.1 Creating a Transaction

We create and destroy an atomic context even for the empty transaction illustrated in Figure 8. We compared the running times of the two functions shown in Figure 15. The empty function loop in (a) took 0.34 seconds to run, while the empty transaction took 25 seconds to run. This suggests that creating a transaction is approximately 74 times as expensive as a normal function call.

```
#define N 50000000
void doNothing() {
}
cilk int main(int argc, char* argv[]) {
  int i;
  for (i = 0; i < N; i++) {
    doNothing();
  }
  return 0;
}
```

(a)

```
#define N 50000000
void emptyTransaction() {
  xbegin
  xend
}
cilk int main(int argc, char* argv[]) {
  int i;
  for (i = 0; i < N; i++) {
    emptyTransaction();
  }
  return 0;
}
```

(b)

Figure 15: Code to test overhead of creating a transaction. The code in (a) is an empty function call, while the code in (b) is an empty transaction.

5.2.2 Instrumenting Loads and Stores

To estimate the overhead generated by instrumenting every load and store, we compare the execution times of the two programs shown in Figure 16. The two are identical except that in (a), x is a local variable, so the loads and stores to x won't be instrumented. The program in (a) took about 0.25 seconds to run, compared to 11.6 seconds for the code in (b). This is approximately 46 times worse. If we insert an additional statement $x = x + 1$ in the for loop of each transaction, the two programs take about 0.48 and 22.5 seconds, respectively, giving us a ratio of about 47.

```
#define N 10000000
cilk int main(int argc, char* argv[]) {
  int i;
  xbegin
    int x;
    x = 0;
    for (i = 0; i < N; i++) {
      x = x + 1;
    }
  xend
}
```

(a)

```
#define N 10000000
int x;
cilk int main(int argc, char* argv[]) {
  int i;
  xbegin
    x = 0;
    for (i = 0; i < N; i++) {
      x = x + 1;
    }
  xend
}
```

(b)

Figure 16: Code to test overhead of instrumenting loads and stores. The access to x is instrumented for the code in (b), but not (a).

5.3 Backoff Algorithm Tests

For the simple incrementer, with our system, we got no speedup with two processors. One reason is that our system introduces enough overhead that the time to increment x now dominates the overhead of the spawn and the function call. However, when we ran the modified code in Figure 11, the statistics collected by our runtime system highlighted other problems with our backoff implementation. On this example, our system reported that typically only one or two different transactions were aborted, and the worst-case transaction was aborted about 25 to 30 times. Essentially, one transaction is being aborted over and over again, and no other transactions are being attempted.

Our current backoff algorithm simply puts the thread to sleep. Since we have not changed the Cilk scheduler, this does not allow any other threads to run. Ideally, we want to postpone running an aborted transaction and instead run one that does not conflict. Unfortunately, implementing this would possibly require us to modify the Cilk runtime system. Note that to prevent space usage from growing arbitrarily, we need to limit the number of transactions that we attempt to run at once. Otherwise, with N copies of `increment()`, if the first processor was running `increment()`, we might spawn a copy of the function on the 2nd processor, abort the transaction, spawn another copy, abort that one, and so forth, until the copy on the first processor finished.

Our backoff algorithm is implemented by calling the function `usleep()`, which in theory sleeps for the specified number of microseconds. However, as Figure 17 shows, the resolution of this function is not very good.

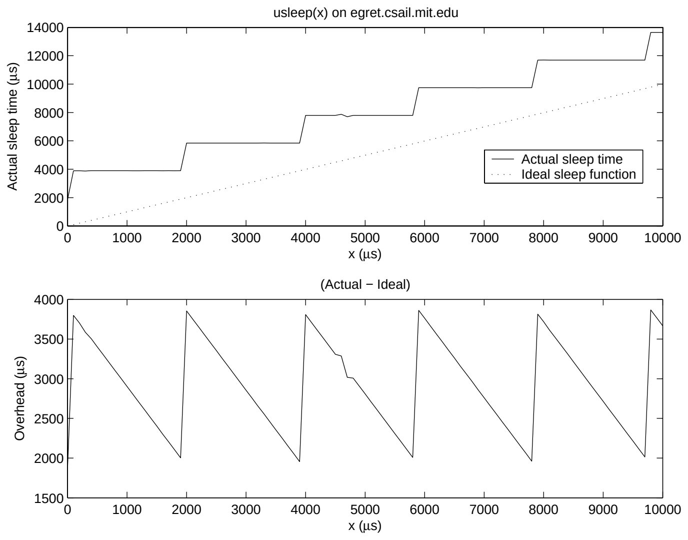

Figure 17: Actual sleep time vs. `x` for `usleep(x)`.

The sleep time was estimated by using `gettimeofday()`.[4] We see that there seems to be an overhead in the function of about 2 to 4 ms. Also, the actual sleep time seems to be rounded to the nearest 2 ms, generating a stair-case type plot. This suggests the shortest period we can sleep is about 4 ms, and we can only go up in increments of 2 ms.

Since our backoff algorithm starts by calling `usleep(1)` and doubling the argument each time, we need to abort about 11 times or so before we get beyond 2 ms. Sleeping for a millisecond is much longer than the time that our example transactions take to run, so this method of doing backoff is problematic.

6 Conclusions

We have successfully modified Cilk to implement a system that can correctly compile simple atomic transactions. As expected, this system is slow, with our transactional programs running anywhere from 8 to 250 times slower than the corresponding implementation with locks. Much of this slowdown can be attributed to

[4]By making consecutive calls to `gettimeofday`, we estimate the resolution of this function to be about 1 μs.

the overhead of creating a transaction and instrumenting loads and stores, although our backoff algorithm also needs to be improved.

There are several options for future work:

- We still need to make several changes to `cilkc` compiler to make the system more complete and more user-friendly.
- We would like to extend the compiler so it has fewer restrictions on `xbegin` and `xend`. The most useful changes would probably be to allow `return` statements and some types of function calls within transactions.
- The implementation of transactional backoff in our system needs to be improved.
- Ideally, we would like to find an implementation of the runtime system that has less overheads than the current system.

Although our system is slow, our implementation of the runtime system in software is flexible enough facilitate future experimentation between different design choices. Hopefully, this work can help us gain a better understanding of the issues in transactional memory systems.

References

[1] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.

[2] Nir Shavit and Dan Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.