Atomic Transactions in Cilk

6.895 Project Presentation
12/1/03
Data Races and Nondeterminism

```cilk
int x = 0;
cilk void increment() {
    x = x + 1;
}
cilk int main() {
    spawn increment();
    spawn increment();
sync;
    printf(''x is %d
'', 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 \texttt{xbegin} and \texttt{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

cilk2c

Cilk Source Code → cilkc

C Source Code → Compiler: gcc

Linker: ld → Output Binary

Cilk Runtime System

Cilk Source Code → cilk2c

Lexical Analyzer: flex → Tokens

Parser: bison → Cilk Abstract Syntax Tree (AST)

Cilk Transform → C AST → Output Code → C Source Code
Compiler Modified For Atomic Transactions

cilkc -atomic

This module does a source-code transformation.

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

cilkc -atomic

This module does a source-code transformation.
Code Transformation for a Transaction

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.

\[
\text{Atomic Context} \* ac = \text{createNewAC}() ;
\text{initTransaction} (ac) ;
\]

\[
\text{attemptTrans:}
\begin{align*}
\text{ - - - - }
\end{align*}
\text{goto tryCommit;}
\]

\[
\text{failed:}
\begin{align*}
\text{doAbort} (ac) ;
\text{doBackoff} (ac) ;
\text{goto attemptTrans;}
\end{align*}
\]

\[
\text{tryCommit:}
\begin{align*}
\text{if (failedCommit} (ac))
\text{goto failed;}
\end{align*}
\]

\[
\text{done:}
\begin{align*}
\text{destroyAC} (ac) ;
\end{align*}
\]

\textit{cilk2c} inserts labels and goto statements into the code for executing transactions.
Inside the Body of a Transaction

Load:

\[
\begin{align*}
\texttt{\{ \{ \text{if} (\text{AtomicReadFailed} (\&x, \text{sizeof}(x), \text{atomicContext}))} \\
\texttt{x; goto \text{failed};} \\
\texttt{\}}, \}
\end{align*}
\]

Store:

\[
\begin{align*}
\texttt{\{ \texttt{int \*tempAddressX = \&x;} \\
\texttt{\{ \texttt{if} (\text{AtomicWriteFailed}(\text{tempAddressX, sizeof(\text{tempAddressX}), \text{atomicContext}}))} \\
\texttt{\quad \texttt{goto \text{failed};}} \\
\texttt{\}}, \}
\end{align*}
\]

\texttt{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

cilk -atomic

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 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.

```
<table>
<thead>
<tr>
<th>Owner</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
<td>y</td>
<td>z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Locks</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
<td>y</td>
<td>z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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

```c
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 $n = 10,000,000$:

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

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

On \( n = 10,000,000 \):

```cilk
int x = 0;

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

<table>
<thead>
<tr>
<th></th>
<th>Running time (s)</th>
<th>Final x</th>
<th>Transactions Aborted</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 processor</td>
<td>11.6 s</td>
<td>10,000,000</td>
<td>-</td>
</tr>
<tr>
<td>2 processors</td>
<td>29.9 s</td>
<td>7,192,399</td>
<td>-</td>
</tr>
<tr>
<td>1 proc, with Cilk_lock</td>
<td>14.2 s</td>
<td>10,000,000</td>
<td>-</td>
</tr>
<tr>
<td>2 proc, with Cilk_lock</td>
<td>34.9 s</td>
<td>10,000,000</td>
<td>-</td>
</tr>
<tr>
<td>1 proc, atomic</td>
<td>605 s</td>
<td>10,000,000</td>
<td>0</td>
</tr>
<tr>
<td>2 proc, atomic</td>
<td>612 s</td>
<td>10,000,000</td>
<td>2</td>
</tr>
</tbody>
</table>

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

\[
\text{int } x[10];
\]

\[
cilk \text{ void inc}\_X() \{ \\
\text{int } j = 0; \\
\text{int } i = \text{rand()} \% 10; \\
\text{for } (j = 0; j < 100; j++) \{ \\
\quad i = \text{rand()} \% 10; \\
\quad x[i] = x[i] + 1; \\
\quad x[i] = x[i] - 1; \\
\}\text{ x[i] = x[i] + 1; }
\}
\]

\[
\text{Max \# times a transaction was aborted: } 24
\]

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Transactions} & \text{Running time (s)} & \text{Sum } x[i] & \text{Aborted / Total Aborts} \\
\hline
1 \text{ processor} & 2.2 \text{ s} & 100,000 & - \\
\hline
2 \text{ processors} & 30 \text{ s} & 99,987 & - \\
\hline
1 \text{ proc, with Cilk_lock} & 3.1 \text{ s} & 100,000 & - \\
\hline
2 \text{ proc, with Cilk_lock} & 32.1 \text{ s} & 100,000 & - \\
\hline
1 \text{ proc, atomic} & 15.9\text{s} & 100,000 & 0/0 \\
\hline
2 \text{ proc, atomic} & 16.4 \text{s} & 100,000 & 6/53 \\
\hline
\end{array}
\]
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: 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

---

Owner

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>...</th>
<th>1</th>
<th>...</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td>z</td>
</tr>
</tbody>
</table>

Normal Memory

<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>...</th>
<th>6</th>
<th>...</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td>z</td>
</tr>
</tbody>
</table>

Atomic Contexts

#1

- Status: COMMITTED
- Owned Addresses: 5

#2

- Status: ABORTED
- Owned Addresses: 1