Transactions with Nested Parallelism

(Adding Transactions to Cilk)

Kunal Agrawal, Jeremy T. Fineman, and Jim Sukha

MIT CSAIL
TRANSACT
August 16, 2007
A Sample Cilk Program

```c
int supply1[10000];
int supply2[10000];
int N = 4;
cilk int main() {
    spawn buy_computer(supply1);
    spawn buy_computer(supply2);
    sync;
    return 0;
}
cilk void buy_computer(int* c) {
    i = rand() % (10000-N);
    for (j = 0; j < N; j++)
        spawn buy_part(c, i+j);
    sync;
}
cilk void buy_part(int* c, int i)
{
    c[i]--; }
```

A Cilk program which updates inventory after buying two computers.

Purchasing a computer decrements parts from the inventory arrays, `supply1` and `supply2`, potentially in parallel.
Cilk Runtime and Performance

```c
int supply1[10000];
int supply2[10000];
int N = 4;
cilk int main() {
    spawn buy_computer(supply1);
    spawn buy_computer(supply2);
    sync;
    return 0;
}
cilk void buy_computer(int* c) {
    i = rand() % (10000-N);
    for (j = 0; j < N; j++)
        spawn buy_part(c, i+j);
    sync;
}
cilk void buy_part(int* c, int i)
{ c[i]--; }
```

The Cilk runtime schedules the program using work-stealing. Cilk executes a computation with work $T_1$ and span (critical path) $T_\infty$ on $P$ processors in time $O(T_1 / P + T_\infty)$, w.h.p.
Transactions in Cilk?

```c
int supply1[10000];
int N = 4;

cilk int main() {
    spawn buy_computer(supply1); //x1
    spawn buy_computer(supply1); //x2
    sync; return 0;
}

cilk void buy_computer(int* c) {
    atomic {
        i = rand() % (10000-N);
        for (j = 0; j < N; j++)
            spawn buy_part(c, i+j);
        sync;
    }
}

cilk void buy_part(int* c, int i) {
    c[i]--; }
```

Suppose both computers require parts from the same inventory. Can we use transactions to ensure both calls to `buy_computer()` are atomic, even though `x1` and `x2` contain nested parallelism?
Transactions with Nested Parallelism and Nested Transactions?

```c
int supply1[10000];
int N = 4;
cilk int main() {
    spawn buy_computer(supply1); //x1
    spawn buy_computer(supply1); //x2
    sync; return 0;
}
cilk void buy_computer(int* c) {
    atomic {
        for (j = 0; j < N; j++) {
            i = rand() % 10000;
            spawn buy_part(i);
        }
    }
    sync;
}
}
cilk void buy_part(int* c, int i) {
    atomic { c[i]--; }
}
```

Suppose each computer can use more than one of the same part. Can we have parallel transactions nested inside x1 and x2?
Motivation: Library Functions

```c
int supply1[10000];
int N = 4;
cilk int main() {
    spawn buy_computer(a); // X1
    spawn buy_computer(a); // X2
    sync; return 0;
}
cilk void buy_computer(int* c) {
    atomic {
        spawn foo(c);
        sync;
    }
}
cilk void foo(int* c) {
    // spawn?
    // call?
}
```

If transactions can have nested parallelism and nested transactions, then we can compositely call some library functions written using Cilk inside a transaction without knowing their exact implementation.
XCilk Design

- We describe XCilk, a theoretical design for a software transactional memory system for Cilk which supports transactions with nested parallelism and nested transactions, both of unbounded nesting depth.

- XCilk uses an XConflict data structure to efficiently check for transaction conflicts.

- XCilk lazily cleans up memory locations on aborts.
XClk Bounds on Overhead

XClk provides a provable bound on the overhead of TM in the following special case:

- For a computation with no transaction conflicts and no concurrent readers to a shared memory location, if the computation has work $T_1$ and critical path $T_\infty$, XClk executes the computation on $P$ processors in time

  $$O(T_1 / P + PT_\infty).$$

  Linear speedup if $P = O(\sqrt{T_1 / T_\infty})$, vs. $O(T_1 / T_\infty)$ for normal Cilk.

- The XClk runtime system still works correctly in the general case, with conflicts and parallel readers.
Outline

- Definition of Conflict in XCilk
- Efficient XConflict Queries
Summary of XCilk Semantics

- XCilk performs eager conflict detection.
- Transactions in XCilk are closed-nested.
- These two conditions imply a prefix race-free execution [ALS06]. If the effects of aborted transactions can be “ignored”, then prefix race-freedom $\approx$ serializability.
XCIlik builds a computation tree as a transactional program executes. A program begins with a single root node ($X_0$).
XClk builds a computation tree as a transactional program executes. A program begins with a single root node (X₀). A spawn creates a P-node (P₁) with two S-nodes (s₁, s₂) as children. The worker then starts executing the left child.
XClk builds a computation tree as a transactional program executes. A program begins with a single root node (X₀).

A spawn creates a P-node (P₁) with two S-nodes (s₁, s₂) as children. The worker then starts executing the left child.

In XClk, as in Cilk, a worker can steal an S-node (s₂) from the deque of another worker.
XClk builds a computation tree as a transactional program executes. A program begins with a single root node ($X_0$).

A spawn creates a P-node ($P_1$) with two S-nodes ($s_1$, $s_2$) as children. The worker then starts executing the left child.

In XClk, as in Cilk, a worker can steal an S-node ($s_2$) from the deque of another worker.

An $xbegin$ creates a transactional S-node ($X_1$)
XCilk: Readsets and Writesets

Conceptually, every transaction $x$ maintains a set of locations that the transaction read from (the readset $R(x)$), and a set of locations that the transaction as written to (the writeset $W(x)$).

The root of the tree represents the world; we assume the writeset of the root contains a value for all memory locations $l$.

When a memory operation $u_1$ on a location $l$ occurs, it reads the value from the closest ancestor transaction with $l$ in its readset.

*We assume $W(x)$ is a subset of $R(x)$. 
Conceptually, every transaction $x$ maintains a set of locations that the transaction read from (the *readset* $R(x)$), and a set of locations that the transaction as written to (the *writeset* $W(x)$).

The root of the tree represents the world; we assume the writeset of the root contains a value for all memory locations $I$. When a memory operation $u_1$ on a location $I$ occurs, it reads the value from the closest ancestor transaction with $I$ in its readset.

$u_1$: write to $I$
XClk: xend

An xend commits a transaction. With closed nesting, when a transaction \( Y_1 \) commits, it conceptually merges its readset and writeset into the readset/writeset of its transactional parent \( x_1 \).
XCEilk: Conflicting write

If \( v_1 \) tries to write to \( L \), XCEilk detects a conflict, because \( x_1 \) is not an ancestor of \( v_1 \).

\[ w(X_1) = \{ L \} \]

Since \( v_1 \) conflicts with \( x_1 \), XCEilk can choose to abort \( z_1 \) immediately.

Alternatively, XCEilk can also signal an abort of \( x_1 \), wait for worker 1 to notice, and then finish \( v_1 \).
Invariant: Conflict-Free Execution

XCalk performs eager conflict detection, and guarantees that the execution is always conflict-free.

At any time, for any given memory location $i$:

1. All active transactions that have $i$ in their writeset fall along a chain.

2. All active transactions with $i$ in their readset are either along the chain or are descendants of the end of the chain.
Conflicts with the Last Writer to $\mathbb{L}$

In the case where no transactions abort, XCilk reduces conflict detection to queries checking for conflicts against the id of the last transaction to write to $\mathbb{L}$.

Let $y$ be the last transaction which last wrote to $\mathbb{L}$. $x^*$ must be an ancestor of $y$, i.e., $y$ has "merged" into $x^*$ because of transaction commits.

If a transaction $z$ wants to perform a read from $\mathbb{L}$,

1. Determine if $x^*$ is an ancestor of $z$.

2. If no, report a conflict.
The XConflict Oracle

\[ \text{XConflict}(*, Z_1) \]

\[ \text{XConflict}(Y, Z) \] answers this query:

For any running node \( Z \),

```c
int XConflict_Oracle(Y, Z) {
    X \leftarrow Y's\ closest\ active\ ancestor\ transaction
    if (X is an ancestor of Z)
        return "no conflict"
    else
        return "conflict";
}
```

Case 1: No conflict
Case 2: Conflict

The XConflict data structure is able to answer the query of XConflict_Oracle in \( O(1) \)-time.
Outline

- Definition of Conflict in XCilk
- Efficient XConflict Queries
Sources of Overhead in XCilk

Assume we keep a history of accesses to each memory location $I$. The overhead in XCilk comes from two sources:

- Updates to the XConflict data structure / histories after a transaction $T$ commits.
- Queries to XConflict to check for conflicts on (potentially) every memory access.
Explicit Merges on Commit

Option 1: Explicit Merge
When we commit a transaction $y$ into its parent $x$, for every location $l$ in $y$’s readset and writeset, we change the id from $y$ to $x$ in $l$’s history.

Advantage: Faster queries.

On a query, the last transaction to write to $l$ is always active.
Slow Commit with Explicit Merging

Explicitly merging writesets immediately on transaction commits can blow up work by a factor proportional to the nesting depth.

For example, consider a chain of nested transactions with depth $d$, with each transaction $x_i$ accessing a different memory location $L_i$.

No nesting: $O(d)$ work.
Closed nesting: $O(d^2)$ work.
Implicit Merges

Option 2: Implicit Merge
Implicitly merge $y$ into its closest active transactional ancestor. On commit, do nothing to histories for $I$.

Advantage: Fast updates.

Sets $a$ through $i$ represent groups of transactions which have merged together.
Implicit Merges with Slow Queries?

Option 2: Implicit Merge
Implicitly merge \( y \) into its closest active transactional ancestor. On commit, do nothing to histories for \( L \).

Advantage: Fast updates.
Disadvantage: Slow queries?
If the XConflict query must walk up the tree to determine which transaction \( y \) has merged into, the query might require \( \Omega(d) \) time.

XCilk potentially performs an XConflict query on every memory access. Therefore, we need queries to take \( O(1) \) time!
The XConflict Query

For any running node Z,

```c
int XConflict_Oracle(Y, Z) {
    X ← Y’s closest active ancestor transaction
    if (X is an ancestor of Z)
        return “no conflict”
    else
        return “conflict”;
}
```

Case 1: No conflict
Case 2: Conflict

XConflict can run in O(1) time because it does not always need to find x to answer the oracle query.
Trace Construction: Updates

XClk speeds up XConflict queries, by dividing the computation tree into traces, with each trace executed by a single worker.*

In Cilk, traces are created by splitting on steals.

\[
\text{# traces} = O(\text{# steals}) \\
= O(PT) \\
\]

Thus, we can afford to acquire a global lock on steals, and perform O(1) amortized work per trace.

Nested transactions are merged together by merging traces together when traces complete.

*Trace construction is similar to the construction in [BFGL04, Fineman05], used for parallel race detection in Cilk.
Trace Construction: Queries

XCilk speeds up XConflict queries, by dividing the computation tree into traces, with each trace executed by a single worker.*

An XConflict query involves a constant number of $O(1)$-time operations at two tiers: a global tier (queries between traces), and a local tier (between tree nodes).

When there are no transaction conflicts and no parallel readers to the same memory location, XCilk performs (at most) one XConflict query per memory access.

*Trace construction is similar to the construction in [BFGL04, Fineman05], used for parallel race detection in Cilk.
XClk Performance Bound

- In the special case of a computation with no transaction conflicts and no concurrent readers to a shared memory location, XClk performs (at most) one $O(1)$-time XConflict query per memory access.
- Maintaining the XConflict data structure introduces overhead of $O(T_1 / P + PT_\infty)$.
- Therefore, the entire program runs in time $O(T_1 / P + PT_\infty)$.
- Linear speedup if $P = O(\sqrt{T_1/T_\infty})$, compared to $P = O(T_1/T_\infty)$ for normal Cilk.
Open Questions

- Can we provide any performance guarantees on programs which are conflict-free, but also allowing parallel reads to the same location? What if there are conflicts?
- Can we “garbage-collect” XCilk’s metadata (e.g., transaction ids) in a provably-efficient manner?
- Can we simplify the XCilk data structures in special but possibly “common” cases? For example, what if the nesting depth is bounded by \( d \)?