Transactional Processing

Peter Pirkelbauer

Computer Science
University of Alabama at Birmingham
Outline

1 Motivation

2 Transactional Processing

3 Lock Elision

4 Conclusion
Motivation
Design a sorted linked list for concurrent systems.

### Operations
- `bool contains(elem)`
- `insert(elem)`
- `erase(elem)`
- `assume garbage collection`
- Coarse-grain lock
- Fine-grain locks (per element)
Nonblocking design

- delete may be a problem
Transactional Processing
Key Idea

Transaction

- Execute multiple operations atomically

- **Operations succeeds**
  - modifications become visible when transaction commits

- **Operation fails**
  - no modification becomes visible
New Instructions

- bool tx_begin() starts a new transaction
- void tx_commit() commits a transaction
- void tx_abort() aborts a transaction

Use

```c
if (tx_begin()) {
    // transactional code
    if (something_bad_happened()) tx_abort();
    ...
    tx_commit();
}
else {
    // code was aborted
}
```
Transactional Execution

Read set
Data read during a transaction

Write set
Data written during a transaction

Example

```c
if (tx_begin()) {
    a[i] = a[i] * b[j];
    tx_commit();
}
else { ... }
```
Transactional Execution

Read set
Data read during a transaction

Write set
Data written during a transaction

Example

```c
if (tx_begin()) {
    a[i] = a[i] * b[j];
    tx_commit();
}
else { ... }
```
**Transactional Execution**

**Read set**
Data read during a transaction

**Write set**
Data written during a transaction

**Example**

```c
if (tx_begin()) {
    a[i] = a[i] * b[j];
    tx_commit();
}
else {
 ...
}
```

- **Read set:** \(i, j, a[i], b[j]\)
- **Write set:** \(a[i]\)
Concurrent Transaction

Overlap of read sets

Two transactions read from the same variable (memory)

**TX 1**

\[ x = x \times a[i]; \]

**TX 2**

\[ y = y \times a[i]; \]

Both transactions succeed (there is no interference).
Concurrent Transaction

Overlap of read sets

Two transactions read from the same variable (memory)

TX 1
\[ x = x \times a[i]; \]

TX 2
\[ y = y \times a[i]; \]

Both transactions succeed (there is no interference).
Concurrent Transaction

Overlap of read and write sets

One transaction reads, another one writes to the same variable (memory).

TX 1

\[
square = a[i] \times a[i]
\]

TX 2

\[
a[i] = a[i] + 1
\]

One transaction succeeds, the other aborts. (ensures consistent view of the data)
Concurrent Transaction

Overlap of read and write sets

One transaction reads, another one writes to the same variable (memory).

TX 1

\[ \text{square} = a[i] \times a[i] \]

TX 2

\[ a[i] = a[i] + 1 \]

One transaction succeeds, the other aborts.
(ensures consistent view of the data)
Abort handling

```c
if (tx_begin()) {
    // transactional code
    tx_commit();
}
else {
    // code was aborted (by hardware or software)
}
```

What can be done when transaction was aborted?

- retry operation (or retry transaction)
- other possibilities .. (see later slides)
if (tx_begin()) {
  // transactional code
  tx_commit();
}
else {
  // code was aborted (by hardware or software)
}

What can be done when transaction was aborted?

- retry operation (or retry transaction)
- other possibilities .. (see later slides)
Abort handling

```c
if (tx_begin()) {
    // transactional code
    tx_commit();
}
else {
    // code was aborted (by hardware or software)
}
```

What can be done when transaction was aborted?

- retry operation (or retry transaction)
- other possibilities .. (see later slides)
How would we use transactions to design operations on a linked list?
Hardware transactions and locks

No progress guaranteed

Most hardware does not guarantee progress

- Capacity limits
- OS thread context switches
- Transactional conflicts
- ...
### Solution

Provide a non-transactional fallback path (blocking or nonblocking)

### Example

```c
cnt = 0;
if (tx_begin()) {
    ...
    tx_commit();
}
else if (++cnt < MAX_RETRY) {
    // retry execution
}
else {
    // lock-based fall back path (see lock-elision)
    ...
}
```
Lock Elision
Try transactional execution of critical section. Use locks as a fallback.
**Example**

**Lock-based execution**

```c
lock(m);
// work in critical section
unlock(m);
```

**Lock-elided execution**

```c
if (tx_begin()) {
  load(m); // reads lock variable (does not update it)
}
else lock(m);
// work in critical section
if (!lock_taken(m)) tx_commit();
else unlock(m);
```

- win, if threads modify/access disjoint data in critical section
Example

Lock-based execution

lock(m);
// work in critical section
unlock(m);

Lock-elided execution

if (tx_begin()) {
  load(m); // reads lock variable (does not update it)
}
else lock(m);
// work in critical section
if (!lock_taken(m)) tx_commit();
else unlock(m);

- win, if threads modify/access disjoint data in critical section
Using the Blaze Library

```cpp
size_t counter(0);
uab::ttas_lock lock;

void test() {
    auto guard = uab::elide_guard(5, lock);
    ++counter;
}
```

- `uab::elide_guard` protects updates to the `counter`
- 5 (first argument) .. number of transactional tries
- `lock` (the fallback lock) needs to implement the `uab::elidable_lock` concept (e.g., in `locks.hpp`)
- programmers can provide an arbitrary number of locks
available at: https://gitlab.cis.uab.edu/iprogress/blaze

example at: examples/testElidableLock.cc

- compile on hardware with support for transactions
- on Intel: use compile command in file
Conclusion
Transactional memory supported on some modern CPUs
Intel, Power8

Speedup on disjoint data accesses

Transactions simplify problems, but do not solve all problems
→ software transaction memory, hybrid approaches, LFTT
Thank you!
Herlihy and Shavit: The Art of Multiprocessor Programming, 2012


Andi Kleen: Lock elision in the GNU C library, 2013.

Nakaike et al.: Quantitative Comparison of Hardware Transactional Memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8, ISCA 2015.
