Programming with Transactional Coherence and Consistency (TCC)
“all transactions, all the time”

Lance Hammond, Brian D. Carlstrom, Vicky Wong,
Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun

Stanford University
http://tcc.stanford.edu

October 11, 2004
The Need for Parallelism

Uniprocessor system scaling is hitting limits
Power consumption increasing dramatically
Wire delays becoming a limiting factor
Design and verification complexity is now overwhelming
Exploits limited instruction-level parallelism (ILP)

So chip multiprocessors are the future
Inherently avoid many of the design problems
Replicate small, easy-to-design cores
Localize high-speed signals
Exploit thread-level parallelism (TLP)
But can still use ILP within cores
But now we must force programmers to use threads
And conventional shared memory threaded programming is primitive at best . . .
Multithreaded programming requires:
Synchronization through barriers, condition variables, etc.
Shared variable access control through locks . . .

Locks are inherently difficult to use
Locking design must balance performance and correctness

*Fine-grain locking*: Lock contention
*Coarse-grain locking*: Extra overhead, more error-prone

Must be careful to avoid deadlocks or races in locking
Must not leave anything shared unprotected, or program may fail

Parallel performance tuning is unintuitive
Performance bottlenecks appear through low level events
such as: false sharing, coherence misses, . . .

Is there a simpler model with good performance?
Yes! Execute *transactions* all of the time
Programmer-defined groups of instructions within a program

*End/Begin Transaction*  Start Buffering Results

Instruction #1
Instruction #2

*End/Begin Transaction*  Commit Results Now (+ Start New Transaction)

Can *only* “commit” machine state at the *end* of each transaction

*Hardware:* Processors update state *atomically* only at a coarse granularity

*Programmer:* Transactions encapsulate and *replace* locked “critical regions”

Transactions run in a *continuous* cycle . . .
Cyclically execute code and buffer

Wait for commit permission

The TCC cycle provides commit ordering, if necessary

Process programmer-requested order on commits

Concurrency with other CPUs

Commit stores together, as a block

This provides a well-defined write ordering

For processors, all instructions within a transaction “appear” to execute atomically at transaction commit time

This provides “sequential” illusion to programmers

This enables parallelization of code

TCC is transactionally-tolerant, but requires high bandwidth
What if transactions modify the same data?
First commit causes other transaction(s) to “violate” & restart
Can provide programmer with useful (load, store, data) feedback!

Transaction A

Transaction B

Violation!

Original Code

... = X - Y

X = ...

Re-execute with new data
Write buffer (~16KB) + some new L1 cache bits in each processor

Also double buffer to overlap commit + execution

Broadcast bus or network to distribute commit packets atomically

Looping on broadcasts triggers violations, if necessary
break sequential code into *potentially* parallel transactions—
usually loop iterations, after function calls, etc.

similar to threading in conventional parallel programming, but:
- do not have to *verify* parallelism in advance
- therefore, much easier to get a parallel program running *correctly*!

then specify *order* of transactions as necessary

- *Fully Ordered*: Parallel code obeys sequential semantics
- *Partially Ordered*: Can emulate barriers and other synchronization

finally, optimize performance
- track violation feedback and commit waiting times from initial runs
- to improve and adjust the strategy
Let’s start with a simple histogram example
Counts frequency of 0–100% scores in a data array
Unmodified, runs as a single large transaction
sequential code region

```c
data = load data();
buckets[101];
i = 0; i < 1000; i++)
    buckets[data[i]]++;
buckets(buckets);
```
for transactional loop

Runs as 1002 transactions

- sequential + 1000 parallel, ordered + 1 sequential

Maintains sequential semantics of the original loop

```
data = load data();
buckets[101];
for (i = 0; i < 1000; i++)
    buckets[data[i]]++;
buckets(buckets);
```
Unordered Loops

Programming with TCC

_**for unordered** transactional loop

Programmer/compiler must _verify_ that ordering is not required

- no loop-carried dependencies
- loop-carried variables are _tolerant_ of out-of-order update (like histogram buckets)

Removes sequential dependencies on loop commit

Allows transactions to finish out-of-order

Useful for load imbalance, when transactions vary dramatically in length

```c
*data = load data();
t, buckets[101];
_for_unordered (i = 0; i < 1000; i++)

buckets[data[i]]++;```

Conventional parallelization requires explicit locking
Programmer must manually define the required locks
Programmer must manually mark critical regions
Even more complex if multiple locks must be acquired at once
Completely eliminated with TCC!

```c
int* data = load_data();
int i, buckets[101];
LOCK_TYPE bucketLock[101];
for (i = 0; i < 101; i++)
    LOCK_INIT(bucketLock[i]);
for (i = 0; i < 1000; i++) {
    LOCK(bucketLock[data[i]]);
    buckets[data[i]]++;
    UNLOCK(bucketLock[data[i]]);
```
An alternative transactional API **forks** off transactions
Allows creation of essentially arbitrary transactions

*An example*: Main loop of a processor simulator
Fetch instructions in one transaction
Fork off parallel transactions to execute individual instructions

```c
pc = INITIAL PC;
code = i fetch(PC);
(opcode != END CODE)
fork(execute, &opcode,
    SEQ, 1, 1);
ment PC(opcode, &PC);
de = i fetch(PC);
```
We parallelized several sequential applications:
From SPEC, Java benchmarks, SpecJBB (1 warehouse)
Divided into transactions using looping or forking APIs

Trace-based analysis
Generated execution traces from sequential execution
Then analyzed the traces while varying:
- number of processors
- interconnect bandwidth
- communication overheads

Simplifications
Results shown assume infinite caches and write-buffers
but we track the amount of state stored in them…
fixed one instruction/cycle
Would require a reasonable superscalar processor for this rate
Initial parallelizations had mixed results
Some applications speed up well with “obvious” transactions
Others don’t . . .
Unordered loops can provide some benefit
Eliminates excess “waiting for commit” time from \textit{load imbalance}
Eliminate spurious violations using *violation feedback*

Privatize associative reduction variables or temporary buffers

Remaining violations from *true* inter-transaction communication
Large transactions can be split *between* critical regions
For early commit & communication of shared data (equake)
For reduction of work lost on violations (SPECjbb)
Merging small transactions can also be helpful
Reduces the number of commits per unit time
Often reduces the commit bandwidth (avoids repetition)
Overall Results

Programming with TCC

- Speedups very good to excellent across the board
- And achieved in hours or days, not weeks or months

Scalability varies among applications
- Low commit BW apps work in board-level and chip-level MPs
- High commit BW apps require a CMP
Conclusions

TCC eases parallel programming
Transactions provide easy-to-use atomicity
eliminates many sources of common parallel programming errors
Parallelization mostly just dividing code into transactions!
us programmer doesn’t have to verify parallelism

TCC eases parallel performance optimization
Provides direct feedback about variables causing communication
implifies elimination of communication
Unordered transactions can allow more speedup
Splitting and merging transactions simpler than adjusting locks
Programmers can parallelize aggressively
ome infrequently violating dependencies can be ignored

TCC provides good parallel performance
TCC
“all transactions, all the time”

More info at: http://tcc.stanford.edu
**Merged Loop Iterations**

```c
data = load_data();
buckets[101];
for n (i = 0; i < 1000; i++; 20)
  buckets[data[i]]++;;
buckets(buckets);
```

- `t_for_n` transactional loop
- Group transactions to lower startup/commit overhead
- Runs as 52 transactions
  - sequential + 50 parallel, ordered + 1 sequential
Phase Control of Transactions

Unordered for purely parallel code
Fully ordered to specify “sequential” tasks
Partially ordered to insert synchronization like barriers

Diagram:
- Unordered Transactions
- Barrier = Phase Transition
- Parallelized Sequential Code

Table:

<table>
<thead>
<tr>
<th>CPU 0</th>
<th>CPU 1</th>
<th>CPU 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>

Time
Only a few KB of write buffering needed

...set by the “natural” transaction sizes in applications...occasional overflow can be handled by “committing” early...reasonable for on-chip buffers...
Another issue is broadcast bandwidth

If data is sent with commit, to avoid broadcast saturation:
- Needs about 16 bytes/cycle/IPC @ 32p with whole modified lines
- Needs only about 8 bytes/cycle/IPC @ 32p with dirty data only

High, but feasible on-chip
Most parallel applications are tolerant of limited BW
SPECjbb shows some server-code “noise” speedup variation
Snooping requirements are quite reasonable
Significantly less than 1 address/cycle on most systems

Address-only commits could reduce BW requirements
Only broadcast addresses for an invalidation-based protocol
Send full packets only to memory
Needs only about 1–2 bytes/cycle/IPC @ 32p
Future Work

Further API adjustments
Are more API extensions useful or necessary?
Are there better ways to expose transactions to programmers?

What can be automated?
Choice of loops
Ordered or unordered loops
Proper degree of transaction merging
Localizable vs. communication-required data

API for optimization hints?

Runtime system and library issues
Building transaction-friendly OS and library routines
Execution-driven evaluation of TCC