2011 HMC NSF CISE SHF RUI CS Summer Research

Observationally Cooperative Multithreading

Motivation

Modern software uses threads to take advantage of multicore processors, but coordination between threads is difficult. OCM is a new programming model for shared memory concurrency, designed to make *correct* multithreaded programs easier to write.

Existing Models

A classic concurrency exercise involves money transfers, with separate transfers in two threads:

```
while (account[i] >= $1):
    take $1 from account[i]
    put $1 in account[j]
```

```
while (account[x] >= $1):
    take $1 from account[x]
    put $1 in account[y]
```

Preemptive Multithreading — Familiar but Painful

In the *preemptive* model typical of C, C++, and Java, threads run simultaneously and interleave their steps (modulo memory models). Without any coordination between threads, code can easily go wrong. For example, if `i == x`, neither loop by itself can overdraw the shared source, but their combination can.

To prevent unwanted interference between threads, programmers use *locks* or *monitors* to enforce mutual exclusion. But it's hard to get the right locks, at the right time, in the right order to prevent deadlock. (Good lock-based code for this simple loop is far too ugly to show here!) Further, lock-based code doesn't compose; atomic deposit and withdrawal operations generally cannot be combined into an atomic transfer.

Cooperative Multithreading — Nice but Uniprocessor

In *cooperative* multithreading, one thread runs at a time, and gets the whole computer until it voluntarily yields control. For example:

```
while (account[i] >= $1):
    take $1 from account[i]
    put $1 in account[j]
    yield
```

```
while (account[x] >= $1):
    take $1 from account[x]
    put $1 in account[y]
    yield
```

Here neither thread can be interrupted between checking the source account and performing the transfer, so no account can become overdrawn. But intuitive as this code is, it makes use of only one core.

The OCM Model

Observationally Cooperative Multithreading (OCM) *schedules cooperative programs on a multiprocessor, without changing the answers.* Cooperative threads can safely run in parallel when not writing/reading the same data; if the OCM system can identify threads that do not conflict, it can take advantage of multiprocessor hardware. From the programmer's perspective:

- Code from each `yield` to the (dynamically) next executes atomically, allowing sequential reasoning.
- At `yield` points, local changes are exported to the outside world, and external changes are imported.

Research Opportunity: Implementation

OCM does not specify an implementation. Any concurrency control technique can be used behind the scenes, as long as code between `yield`s executes atomically.

Locks ensure exclusive access to data. We can ensure atomicity by having each `yield` lock shared data that might be accessed before the next `yield`. Locks can be inferred by static analysis or from programmer annotations.

What the programmer writes:	What actually happens:
`yield`	`release_all_locks()` `acquire_locks(a,b)`
`tmp = a` `b = tmp + 1`	`tmp = a` `b = tmp + 1`
`yield`	`release_all_locks()` `acquire_locks(...)`

Software Transactional Memory (STM) systems let programmers create *transactions*, which track reads and writes. When transactions conflict, they undo and retry; otherwise they commit with atomic behavior. We can have each `yield` end the current transaction and begin the next.

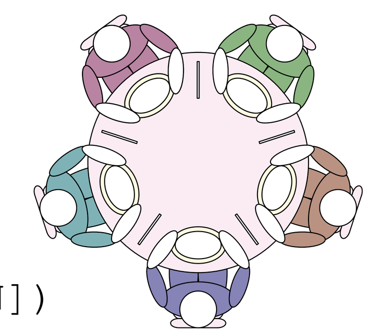
What the programmer writes:	What actually happens:
`yield`	`end_transaction()` `begin_transaction()`
`tmp = a` `b = tmp + 1`	`tmp = stm_read(a)` `stm_write(b, tmp + 1)`
`yield`	`end_transaction()` `begin_transaction()`

Comparing Implementation Techniques

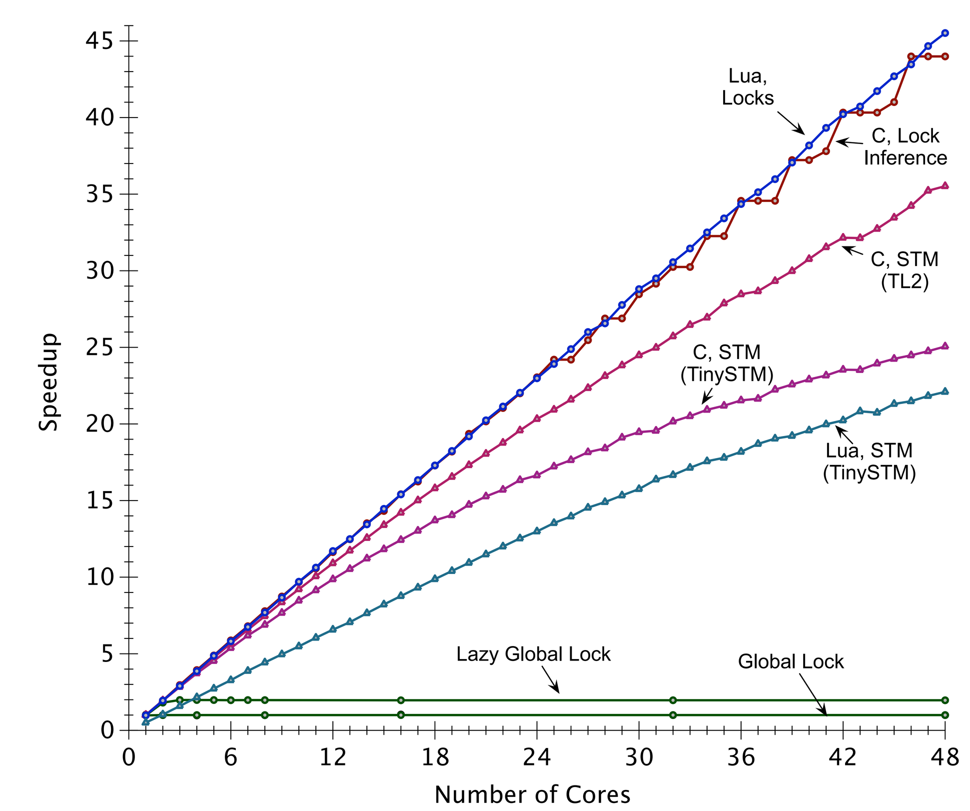
OCM prototypes have been implemented for C/C++ and Lua, both via locks and via STM. OCM algorithms should work on any underlying implementation, so we can use OCM to compare concurrency control techniques.

Consider Dijkstra's classic concurrency problem. There are `N` philosophers at a round table, alternately thinking and eating. Between each pair is a utensil, and philosophers need the closest two utensils to eat. The simplest OCM solution is:

```
philosopher(int i):
    for iter in (1..ITERS):
        think()
        yield
        eat(fork[i],fork[(i+1)%N])
        yield
```



This code is trivially correct if scheduled cooperatively (one philosopher eating at a time). OCM systems should realize that philosophers manipulating different utensils can run simultaneously. How well did our OCM prototypes exploit the available parallelism?



Research Opportunity: Usability

Is OCM a viable parallel programming model? We have solved common parallel programming puzzles in OCM and are porting performance benchmarks (e.g., STAMP, PARSEC). User studies are in progress.

Nondeterministic parallel programs are difficult to debug, but OCM behavior is always valid cooperative behavior. We can (and do) record buggy executions and replay them as necessary with a cooperative schedule implementation.

Research Opportunity: Design

What might we want from OCM, besides `yield` and a method of creating threads? Cooperative multithreading (i.e., OCM semantics) is compatible with locks, barriers, message-passing, and many other traditional mechanisms. Of course these can often be *explained* simply in terms of OCM, e.g., barriers:

```
barrier():
    ++waiting
    do yield while (waiting != NUM_THREADS)
```

The semantics is clear, and naturally leads to discussions of more efficient implementations.

Two other constructs seem particularly valuable:

- `yieldUntil(c)` ≡ `do yield while (!c)`, i.e., Hoare's conditional critical regions. As a primitive, cleverer implementations may be possible.
- `canYield`. A `yield` performed to be a "good citizen," rather than because we actively desire to switch threads. We have seen speedups of up to 300× by letting OCM decide dynamically whether or not to `yield`.

Further Information

OCM was inspired by the STM-focused Automatic Mutual Exclusion system of Abadi et al., and is complementary to work by Yi et al. explaining code with explicit locks in terms of cooperative multithreading.

For more details and a C++ library for OCM, see:

http://ocm-model.org/

Researchers

Students: Xiaofan Fang, Sean Laguna, Stephen Levine, Jordan Librande, Stuart Pernsteiner, and Mary Rachel Stimson.
Faculty: Melissa O'Neill and Christopher Stone.
Previous work by: Sonja Bohr, Bartholomew Broad, Adam Cozzette, Joe DeBlasio, Sam Just, Kwang Ketcham, Alejandro Lopez-Lago, Julia Matsieva, Josh Peraza, and Ari Schumer.

This material is based upon work supported by the National Science Foundation under Grant No. CCF-0917345. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.