Observationally Cooperative

Melissa O’Neill
Chris Stone
lots of summer students
Multicore
Parallel programming is
Parallel programming is
Choose one, maybe
Familiar Correct
Understandable Performant
Broadly Applicable

OCM
Multicore programming for the masses

Goal: a shared-memory model that
  ★ is easy to learn and use
  ★ supports irregular problems
  ★ values correctness, ease-of-use
Race Conditions

// move $5
acct[x] = acct[x] - 5;
acct[y] = acct[y] + 5;

// move $10
acct[i] = acct[i] - 10;
acct[j] = acct[j] + 10;
Explicit Locks (?)

lock(acct[x]);
lock(acct[y]);
    // move $5
    acct[x] = acct[x] - 5;
    acct[y] = acct[y] + 5;
unlock(acct[y]);
unlock(acct[x]);

lock(acct[i]);
lock(acct[j]);
    // move $10
    acct[i] = acct[i] - 10;
    acct[j] = acct[j] + 10;
unlock(acct[j]);
unlock(acct[i]);
Atomic Blocks

atomic {
    // move $5
    acct[x] = acct[x] - 5;
    acct[y] = acct[y] + 5;
}

atomic {
    // move $10
    acct[i] = acct[i] - 10;
    acct[j] = acct[j] + 10;
}
Atomic Blocks

while (acct[x] >= 5) {
    // move $5
    acct[x] = acct[x] - 5;
    acct[y] = acct[y] + 5;
}

bool loop1;
do {
    atomic {
        loop1 = acct[x] >= 5;
        if (loop1) {
            // move $5
            acct[x] = acct[x] - 5;
            acct[y] = acct[y] + 5;
        }
    }
} while (loop1);
Atomic Blocks

```c
while (acct[x] >= 5) {
    // move $5
    acct[x] = acct[x] - 5;
    acct[y] = acct[y] + 5;
}

bool loop1;
do {
    atomic {
        atomic {
            loop1 = acct[x] >= 5;
            if (loop1) {
                // move $5
                acct[x] = acct[x] - 5;
                acct[y] = acct[y] + 5;
            }
        }
    }
} while(loop1);
```
Cooperative Multithreading (for Uniprocessors)

- Only one thread runs at a time.
- `yield` switches threads; no preemption.
Cooperative Multithreading

- Only one thread runs at a time
- **yield** statements switch threads
Cooperative

while (acct[x] >= 5) {
    // move $5
    acct[x] = acct[x] - 5;
    acct[y] = acct[y] + 5;
}

while (acct[i] >= 10) {
    // move $10
    acct[i] = acct[i] - 10;
    acct[j] = acct[j] + 10;
}
Cooperative Multithreading

while (acct[x] >= 5) {
    // move $5
    acct[x] = acct[x] - 5;
    acct[y] = acct[y] + 5;
    yield;
}

while (acct[i] >= 10) {
    // move $10
    acct[i] = acct[i] - 10;
    acct[j] = acct[j] + 10;
    yield;
}
OCM: A Model for Parallel Computation

- CM code = OCM code
- System may run threads simultaneously
- Fundamental guarantee: **CM-Serializability**
  - Result is consistent with some uniprocessor CM execution
Observationally Cooperative Multithreading

while (acct[x] >= 5) {
    // move $5
    acct[x] = acct[x] - 5;
    acct[y] = acct[y] + 5;
    yield;
}

while (acct[i] >= 10) {
    // move $10
    acct[i] = acct[i] - 10;
    acct[j] = acct[j] + 10;
    yield;
}
Let’s Try It…
A Parallel Perspective on yield

Threads primarily execute in isolation.

When a thread yields:
- Its changes are visible to the world
- Changes in the world become visible to it
Advantages of OCM

- We can reason sequentially between yields
- Fewer opportunities for deadlock
- Implementation-agnostic
That’s nice…
But how do you implement it?
You don’t need to care.
“It just works.”
You don’t need to care.

“It just works.”

In theory!
What would programmers do without OCM?
What would programmers do without OCM?

It does that, automatically!
Classic idea: Locks
Implementing OCM with Locks

- need locks for data accessed through next yield
  - Lock inference
  - Programmer annotations
- OCM is responsible for avoiding deadlock.
- Optimizations: Lazy Acquire, Eager Release
Newer idea: Atomic Transactions
Implementing OCM with STM

yield; \quad \rightarrow \quad \text{end\_transaction();}

begin\_transaction();

- One subtlety: “unreturning” from functions
Unreturning from Functions

```c
void caller() {
    callee();
    ...
    yield;
}

void callee() {
    yield;
    ...
    yield;
}
```
void caller() {
    callee();
    ...
    yield;
}

void callee() {
    yield;
    ...
    yield;
}

Solutions:

- Access the stack through STM
- Or, save and restore stack segments
Proof of Concept Implementations

- Uniprocessor CM
- Pthreads + Big Lock
- Pthreads + Big Lazy Lock
- Explicit Locks
  - Lua (proxy objects)
  - C subset (lock inference)
- Software Transactional Memory
  - Lua (TinySTM)
  - C++ (wrapper objects, TinySTM/TL2)
Example: Dijkstra’s Dining Philosophers
Traditional Philosophers

philosopher(int i):
    for iter in (1..ITERS):
        think();

        yieldUntil (isFree[i] && isFree[(i+1) % N]);

    isFree[i] = false;
    isFree[(i+1) % N] = false;
    yield;

    eat();

    isFree[i] = true;
    isFree[(i+1) % N] = true;
    yield;
philosopher(int i):
    for iter in (1..ITERS):
        think();
        yield;
        eat(fork[i], fork[(i+1) % N]);
        yield;
Speedup: Traditional & True OCM Philosophers

Number of Cores vs. Speedup

- Global Lock
- Lazy Global Lock
- C, Lock Inference
- Lua, Locks
- C, STM (TinySTM, TL2)
- Lua, STM (TinySTM)
- Global Lock
- Lazy Global Lock

Inset graph focusing on selected configurations.
Debugging and Profiling

- OCM guarantees CM-Serializability.
  - Run in parallel, record serial equivalent
  - “Replay” the trace in uniprocessor CM.

- Implemented in 2 proof-of-concept implementations.
Conclusion

- OCM appears promising
  - Simple programming model
  - Supports “irregular” problems
  - Debugging support
  - Many possible implementations

- Future Work
  - Larger benchmark suite
  - More examples
  - Better/different OCM implementations
  - Study “ease of programming”
We’d love your help!
Be our Guinea Pigs!