**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 inversion exercise involves money transfers, with separate transfers in two threads:

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

This code is, it makes use of only one core.

Preemptive Multithreading — Familiar but Painful

In the preemptive model typical of C, C++, and Java, threads run simultaneously and interleave their steps (module or memory models). Without any coordination between threads, code can easily go wrong. For example, if i = x, neither loop by itself can overwrite 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:

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

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 yields executes atomically.

Locks are a popular choice. 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.

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0.74</td>
</tr>
<tr>
<td>12</td>
<td>0.62</td>
</tr>
<tr>
<td>18</td>
<td>0.56</td>
</tr>
<tr>
<td>24</td>
<td>0.54</td>
</tr>
<tr>
<td>30</td>
<td>0.51</td>
</tr>
<tr>
<td>36</td>
<td>0.49</td>
</tr>
<tr>
<td>42</td>
<td>0.47</td>
</tr>
<tr>
<td>48</td>
<td>0.45</td>
</tr>
</tbody>
</table>

**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.

**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:

```c
philosopher(int i):
    for iter in 1..ITERS:
        yield
        if iter in (1..ITERS):
            think()
            yield
        if iter in (1..ITERS):
            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: 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:

```c
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(t) ≡ do yield while (t), 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 monitors

For more details and a C++ library for OCM, see: http://ocm-model.org/

**Researchers**


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.