Observationally Cooperative Multithreading *

Christopher A. Stone Melissa E. O'Neill The OCM Team

Computer Science Department
Harvey Mudd College
{stone,oneill,ocm}@cs.hmc.edu

Abstract

Observationally Cooperative Multithreading (OCM) is a new approach to shared-memory parallelism. It addresses a key problem of mainstream concurrency control mechanisms—they can be prohibitively hard to reason about and debug. Programmers using OCM simply write code as if they were using the cooperative multithreading model (CM) for uniprocessors. The underlying OCM implementation then optimizes execution—running threads in parallel when possible—in such a way that the results are consistent with CM. In addition to providing easier reasoning and debugging, OCM is also highly adaptable in terms of its underlying concurrency-control mechanism. Programmers using OCM have the capability to take a finished program and choose the strategy (e.g., locks or transactions) that provides optimal performance.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming; D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed, and parallel languages

General Terms Languages, Performance

Keywords Observationally cooperative multithreading, cooperative multithreading, transactional memory, lock inference, parallel model, parallel debugging.

1. Introduction

Parallel programming is notoriously difficult; it is hard to predict all ways in which threads may interact. Synchronization code to manage these interactions can be complex and error-prone. And when bugs inevitably arise, hard-to-reproduce race conditions make debugging more difficult than in sequential code. Although there has been valuable progress in making parallel programming more accessible, popular models for parallelism are still difficult for many programmers to use effectively [3].

Inspired by Cooperative Multithreading (CM) for uniprocessors, where threads run one at a time and continue until they explicitly yield control, we propose a new model for parallel programming. *Observationally Cooperative Multithreading* (OCM) offers

- Simple semantics and syntax, taken from CM;
- Parallel execution, taking advantage of modern hardware;
- Implementation flexibility, allowing a variety of contention management methods (e.g., transactional memory, lock inference);
- Serializability, simplifying debugging and reasoning.

OCM is not an implementation mechanism, but rather an abstraction for programmers. The observable behavior of programs is consistent with execution on a uniprocessor with cooperative multithreading, even if behind the scenes threads are running simultaneously or preempting one another.

Designed to emphasize correctness over raw performance, OCM may not be suitable for all multithreaded applications. But just as many systems use garbage collection and runtime bounds checking rather than manual memory management and unsafe array accesses, we feel that there is a place for systems like OCM that provide an easier and safer path into parallel programming. And, as with garbage collection and bounds checking, there is wide scope for interesting research and design work to decrease runtime overhead.

2. Observationally Cooperative Multithreading

As with CM, under the OCM model the programmer simply specifies locations in their code where it is safe for a thread to yield control; the syntax for an OCM program is the same as for a CM program. For example, the following "banking" example of concurrent account transfers is valid in CM and in OCM.

```
# repeatedly move $5
while acct[x] >= 5:
  acct[x] = acct[x] - 5
  acct[y] = acct[y] + 5
  yield
```

```
# repeatedly move $10
while acct[i] >= 10:
  acct[i] = acct[i] - 10
  acct[j] = acct[j] + 10
  yield
```

Unlike CM, OCM is a model for *parallel* computation. A system implementing the OCM model is free to run programs in parallel, provided that the observable behavior (final results, I/O, etc.) of a program is consistent with a possible execution under some (nonpreemptive, uniprocessor) CM model. We call this requirement *CM serializability*, and it is the fundamental property of OCM.

In the above code example, CM (and hence OCM) guarantees the the comparison and updates in each interation execute atomically.[1] The two loops can execute simultaneously if `x` and `y` are disjoint from `i` and `j`. Otherwise, the loops must be interleaved. Either way produces results consistent with CM.

* 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.

SPLASH'11 Companion, October 22–27, 2011, Portland, Oregon, USA.
ACM 978-1-4503-0940-0/11/10.

[1] Obtaining equivalent behavior with block-structured `atomic` blocks would be much more awkward.

3. Implementations

Any means to execute code consistent with CM is a valid OCM implementation. We have developed several different implementations, which are available for download at http://ocm-model.org. In creating these implementations, we show that a variety of implementation strategies for OCM are feasible. Doing so also allows us to compare the tradeoffs of these different implementation strategies.

Undoubtedly the simplest implementation of the OCM model is traditional uniprocessor CM (or, for a parallel implementation, a single global lock on the CPU). Although it does not exploit multiple cores, it has value as a baseline. An OCM implementation that exploits multiple cores should outperform CM in general, but CM may be best in specific cases (e.g., for programs with massive thread contention, or on a uniprocessor machine).

3.1 Nontrivial Lock-Based Implementations

We have developed a proof-of-concept lock-based OCM implementation as an extension to the Lua scripting language. This extension is a dynamic library loaded by the Lua interpreter, so it cannot perform static analysis to obtain the information needed for correct locking. Access to shared data is therefore mediated solely through "proxy objects" obtained through the OCM library—threads are otherwise completely separate. Because the system knows that a thread can only access shared data through proxies, and the system knows which threads are holding which proxies, the OCM scheduler can acquire and release all necessary locks on behalf of threads.

We have also implemented lock-based OCM in the form of a source-to-source translator for C with the addition of `yield` and `spawn` statements. The translator does dataflow analysis to conservatively determine which variables may be accessed in the future following each `yield` statement—those are the variables that `yield` needs to lock. This information is then used to insert calls to locking and unlocking functions using Pthreads in the necessary locations. Any `spawn` or `yield` statements are replaced with calls to library functions.

3.2 STM-based Implementations

The OCM model also permits implementations based on software transactional memory (STM), whereby all reads and writes of shared data are routed through an STM system. Each `yield` statement ends the current transaction and begins a new one, so that changes made by the current thread become visible to others.

As when investigating lock-based implementations, we began with a proof-of-concept modification to Lua. In this case, we implemented the OCM system by requiring the Lua interpreter to use the TinySTM library when accessing memory.

We have also created an STM-based OCM implementation as a C++ library using Pthreads. This library allows the programmer to indicate that certain variables are shared, which causes all accesses to those variables to be routed through the STM library. Because transactions may begin and end in different lexical scopes, our system saves and restores stack frames as necessary. Our library approach requires no changes to the underlying language, relying instead on C++ language features (overloading, templates, etc.) to make access to shared data feel natural.

4. Debugging and Performance Profiling

Although OCM dramatically reduces the potential for race conditions and deadlock compared to, say, explicit locking, it does not eliminate them.

Fortunately, reproducing bugs is far easier in OCM than in many other models, because every program execution has at least one corresponding execution under CM. If an OCM system wishes to allow reproducible debugging, it simply has to record a corresponding serial execution for that program. With that *serialization trace*, it is possible to rerun the program serially following that trace and thereby reproduce the exact sequence of interleavings that trigger the bug.

5. Conclusions and Future Work

OCM is a promising solution for shared-memory program development. It retains many of the benefits of currently existing concurrency-control systems, while mitigating their complexity. It allows the programmer to focus on the logic of the program instead of the subtleties of parallelism.

Because OCM does not require a specific implementation, an application can be written according to the OCM model and use whichever implementation is best suited for it.

To promote the broad adoption of OCM, others could implement OCM using their own concurrency-control schemes. We also hope that educators see the value in using OCM as a "kinder, gentler" form of multicore parallelism, even if they later introduce other models, like locks or transactions. In fact, OCM can serve as a springboard for subject; synchronization primitives are easy to write in OCM (e.g., `semWait(`i`)` is `while (`i` > 0) yield; --`i and `semSignal(`i`)` is `++`i), and discussions of efficient OCM implementations naturally lead to topics like transactions. We hope that our available implementations and further examples of OCM in use will provide a good starting point for these efforts.

In addition, OCM needs benchmarks that can be used to assess the performance of different concurrency control techniques and of the OCM approach as a whole. We are currently adapting the benchmarking suites STAMP and PARSEC to OCM and writing a series of examples from *The Little Book of Semaphores* [2] using OCM. We are investigating how OCM scales to larger applications, and which debugging and profiling tools prove most valuable.

6. Related Work

As a parallel model, OCM intersects with a significant portion of prior work on parallelism and concurrency. OCM is particularly closely related to Automatic Mutual Exclusion [1] where all code is atomic unless marked unsynchronized (an empty unsynchronized block corresponds to `yield`), and to work by Yi et al. [4] that explains lock-based code in terms of cooperative multithreading.

Acknowledgments

The undergraduate OCM Team included Bartholomew Broad, Kwang Ketcham, Samuel Just, Alejandro Lopez-Lago, and Joshua Peraza (2009); Sonja Bohr, Adam Cozzette, Joe DeBlasio, Julia Matsieva, Stuart Pernsteiner, and Ari Schumer (2010); and Xiaofan Fang, Sean Laguna, Stephen Levine, Jordan Librande, Stuart Pernsteiner, and Mary Rachel Stimson (2011).

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL '08*, pages 63–74, 2008.

[2] A. B. Downey. *The Little Book of Semaphores*. Green Tea Press, 2nd edition, 2008.

[3] C. J. Rossbach, O. S. Hofmann, and E. Witchel. Is transactional programming actually easier? In *PPoPP '10*, pages 47–56, 2010.

[4] J. Yi, C. Sadowski, and C. Flanagan. Cooperative reasoning for preemptive execution. In *PPoPP '11*, pages 147–156, 2011.