Making Impractical Implementations Practical: Observationally Cooperative Multithreading Using HLE *

Melissa E. O'Neill Christopher A. Stone
Harvey Mudd College
{oneill,stone}@cs.hmc.edu

Abstract

Observationally Cooperative Multithreading (OCM) is a new approach to shared-memory parallelism. Programmers write code using the well-understood cooperative (i.e., nonpreemptive) multithreading model for uniprocessors. OCM then allows threads to run in parallel, so long as results remain consistent with the cooperative model. OCM is easer to reason about and allows simpler coding because, conceptually, task switching only occurs at yield points, but its performance has been disappointing until now.

We show that an extremely simple-minded OCM implemention strategy (one that would normally not achieve any parallel speedup at all!) can perform remarkably well under Intel's Hardware Lock Elision (HLE), a form of hardware transactional memory. HLE is also a very good fit for OCM because HLE's limitations do not violate any aspects of OCM's parallel model.

This work suggests that when considering how to use new concurrency-control mechanisms such as HLE, we should consider not only how they may be used to replace traditional synchronization techniques that already perform fairly well, but also how they may make viable strategies that would not previously have been feasible.

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

* 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.

Keywords Observationally cooperative multithreading, cooperative multithreading, transactional memory, hardware transactional memory, parallel model, hardware lock elision.

1. Introduction

Programmers writing shared-memory programs that require irregular parallelism have traditionally had a choice between approaches that offer excellent performance but are hard to program correctly (such as lock-based strategies), or approaches that are correct but whose performance is lacking (such as software transactional memory [13, 20, 24]). Recently, however, that state of affairs has started to change—today's mainstream CPUs have taken hardware transactional memory (HTM) from an idea [9, 13] to reality. HTM has the potential to offer both good speed *and* ease of access to shared data.

But the trade-offs made by today's HTM implementations may appear not to help either of these two approaches. If we convert lock-based programs to HTM, the overheads of beginning and ending transactions may be too large to see any benefit from a transactional approach. Similarly, if we wish to use hardware transactional memory to replace software transactional memory, the limitations of the hardware-based approach can mean that we cannot throw out the STM infrastructure—it needs to remain as a fallback.

In this paper, however, we show how hardware transactional memory, even with the limitations and restrictions present in today's implementations, can be an excellent fit for some models of parallel computation. We will show that with the right model, it's possible to write elegant and simple programs for problems whose synchronization requirements cannot be easily handled by lock-based strategies *and* achieve respectable parallel speedup.

Specifically, we will examine how *Observationally Cooperative Multithreading* (OCM) [25, 26], can be implemented straightforwardly using Intel's *Hardware Lock Elision* (HLE) mechanism. OCM is inspired by traditional Cooperative Multithreading (CM) for uniprocessors, where threads run one at a time and continue until they explicitly yield control. 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 (including transactional memory and lock inference);
- Serializability, simplifying debugging and reasoning about program behavior.

When we developed OCM a few years ago, we were excited by the range of possible implementation strategies it could support, both transactional (e.g., software transactional memory) or nontransactional (e.g., by lock inference). But in practice, our performance results for realistic programs were disappointing—almost every implementation approach was slower than simple serial execution! HLE changes that state of affairs, allowing an implementation far simpler than any we had previously created (almost embarrassingly simple, in fact) but offering the potential for actual parallel speedup.

In this paper, we introduce CM and OCM for readers who may be unfamiliar with this model, and then

- Describe how to implement OCM using HLE
- Highlight the design choices in OCM that allow it to be so readily implemented using HLE
- Provide evidence that the HLE implementation of OCM can allow good parallel speedup
- Describe optimizations that can further enhance the performance of the implementation

2. Background: Cooperative Multithreading

The landscape of parallel computing is littered with attempts to finally make concurrent programming accessible to ordinary programmers. Much valuable progress has been achieved, yet the models for concurrency in widespread use today are still difficult for most programmers to use effectively (see Section 7, Related Work, for more discussion).

In contrast, Cooperative Multithreading (CM) is a well-known model for concurrency that significantly reduces opportunities for racy interactions between concurrent threads. In the CM model, at most one thread is running at any given time and control switches from one thread to another only when the currently executing thread *yields* control (or terminates). Programmers place yield statements at specific points in their code where it is safe to yield control—either to propagate changes to shared data between threads, or just to be a "good citizen" and let other threads execute.

As a programming model for multithreaded applications, CM has some very attractive properties. In particular, the text of the program specifies exactly where threads may be interrupted. Although CM programs may be nondeterministic, the ways in which nondeterminism can arise are relatively restricted. Further, between yields we can reason about code purely sequentially: until a thread yields, it will never see the environment being changed by other threads, nor will its changes be visible to other threads.

CM was originally conceived as a model for writing *uniprocessor* multithreaded programs, where its one-at-a-time nature presents few problems. On a multiprocessor machine, however, running one thread at a time leaves all cores but one idle. In Section 3, we'll describe OCM which keeps the *appearance* of running one thread at a time, while nevertheless allowing parallelism.

2.1 Bank Accounts Example

As an example of the CM model, let us consider the familiar example of two threads transferring funds within an array of bank accounts, provided they do not become overdrawn. Thread A moves \$50 between accounts i and j if possible, whereas Thread B loops, repeatedly moving \$5 from account x to y until x is drained. We must avoid any race conditions (e.g., money being created out of nowhere, or vanishing).

Thread A

```
// try to move $50 from i to j
if (account[i] >= 50) {
   // okay to move $50, so do it
   account[i] = account[i] – 50;
   account[j] = account[j] + 50;
}
```

Thread B

```
// move everything x to y
while (account[x] >= 5) {
   // okay to move $5, so do it
   account[x] = account[x] – 5;
   account[y] = account[y] + 5;
   yield;
}
```

Thread A always runs to completion without interruption. From the perspective of other threads, the \$50 moves atomically from accounts i to j and the total funds across all accounts stays the same. Likewise for thread B, but other threads may observe points in time where account x is partway through being drained. If we removed B's yield statement, it would perform the entire account-draining process atomically.

Adding yields elsewhere in the code would allow more interleavings, some of which would be erroneous. Placing a yield between the two assignment statements would allow people to observe moments in time when money has been taken from one account and not yet put to another. Putting a yield before both assignment statements could allow interleavings where the condition was no longer true when the assignments were executed. The explicitness of yield statements makes it relatively straightforward to reason about what could happen if we were to yield in those places.

Although CM has far fewer possible effect orderings than other concurrency models, it nevertheless allows nondeterminacy. There is no guarantee which thread will run when we yield; in fact the same thread may continue to run. In our example, nondeterminacy is possible if $i = y$ and $j = x$; the \$50 credit made by Thread A may escape being drained by Thread B if the credit happens after Thread B's loop is over.

Finally, it's worth recognizing the ways in which this code is simpler than strategies using atomic blocks or explicit locks. An explicit-locking approach requires us to be mindful of deadlock, and with atomic blocks it requires some thought to replicate Thread B's execution—we cannot place an atomic block inside the loop, nor put the loop inside an atomic block.

In both cases, experienced parallel programmers will see solutions, but these solutions lack the simplicity of the CM version.

2.2 Ants Example

In this example (based on a Clojure example proposed by Rich Hickey), a group of ants explore a 2D grid, alternating between thinking and moving towards (and consuming) any nearby food (each item of food can only feed a single ant and each ant moves randomly if there is no food in its immediate vicinity). If we allow the explorations of two arbitrary ants to occur concurrently, we must ensure that no race conditions or deadlocks occur in examining and updating the 2D grid.

In CM, the code for each ant becomes

```
while stillAlive:
  think();
  yield;
  (x, y) = grid.mostAttractiveCellNearTo(x, y);
  if (grid.hasFood(x,y))
      grid.eatFood(x,y);
  yield;
```

Notice that exploring the grid and eating the food on a grid square happen as an atomic unit because there is no yield between these two steps (we assume that none of the grid functions yield).

If we omitted the yield after think(), the execution of ants would still be interleaved but there would be half as many "chunks" to interleave. If thinking took a long time and we desired even more chunks for interleaving, the ant could yield inside think. For many CM programs, the choice of how long to go between yields may not have any bearing on its correctness, but may affect responsiveness or other performance properties.

2.3 Features Beyond yield

The examples we have seen above focus on yield, but a complete CM system will have more concurrency-control primitives than just yield. For example, the GNU Pth library for CM [5] provides a plethora of concurrency-control mechanisms, including interthread communication channels, condition variables, barriers, and mutexes.

It might seem like such mechanisms are unnecessary in a world where only one thread runs at a time, but the moment we have situations where one thread must wait for something to happen before it can continue, these mechanisms become useful. (We could instead busy-wait by yielding until the world is to our liking, but doing so would not be efficient, and if the system does not provide a yield-fairness guarantee, there is actually no guarantee that doing so would be effective either.)

It is instructive to highlight the difference between concurrency-control primitives in the CM world and those elsewhere, with CM allowing simpler constructs. In most systems that provide condition variables, a condition variable must have an associated mutex and the mutex must be held when accessing any data related to the condition to avoid race conditions (such as "lost wakeup" bugs). In contrast, in CM a condition variable does not need a mutex because its one-thread-at-a-time nature means that these kinds of race conditions can only occur if we are foolish enough to put yield in the wrong place (e.g., adding a yield between finding a buffer is full and waiting for someone to signal us to tell us that they've emptied it).

2.4 Issues with CM

There are certainly instances where relying on cooperation is inappropriate. Desktop operating systems such as Windows and MacOS formerly employed CM but have long since adopted preemptive scheduling, preventing one uncooperative process from hanging the entire system. But within a single program, a buggy thread failing to yield is no worse than an accidental infinite loop in sequential code.

But the most obvious deficiency of CM as a model for multithreaded computation is that it executes only one thread at a time! Our goal for OCM, which we describe next, is to retain the many advantages of CM but allow multicore implementations.

3. Observationally Cooperative Multithreading

Observationally Cooperative Multithreading (OCM) adopts the simple semantics of cooperative multithreading (CM) that we outlined in the previous section, but allows implementations that take advantage of multiprocessor parallelism when possible.

From a programmer's perspective, OCM and CM programs are identical. The observable behavior (final results, I/O, etc.) of a program is always consistent with a possible execution under some CM model (i.e., nonpreemptive, uniprocessor). We call this requirement *CM serializability*, and it is the fundamental property of OCM.[1] The difference between CM and OCM is that a system implementing the OCM model is free to run tasks in parallel, *if* it can do so while still guaranteeing CM serializability. Thus, CM and OCM produce the same results and have the same conceptual model, but OCM may be able to produce those results faster because it can potentially use all the cores on the machine whereas CM can only use one.

For the banking example we outlined earlier, the threads A & B can execute simultaneously if x and y are disjoint from i and j. Likewise, for the ants example, two ants exploring disjoint segments of the grid can explore simultaneously. When two threads would interfere with each other, their execution must be serialized to avoid the interference.

3.1 Implementation Agnosticism

The details of how an OCM system runs code while retaining CM serializability (and the concurrency-control mechanisms

[1] CM serializability also means that the semantics of OCM is by definition that of CM; we can immediately reuse existing formalizations of CM semantics [1].

it uses to do so) are implementation decisions, visible to users only insofar as they affect performance. Any CM implementation (e.g., GNU Pth) is trivially an OCM implementation, albeit one that fails to ever achieve any parallel speedup. Such an implementation can be useful in practice because it provides a minimum performance standard to use as a baseline.

From a parallel-execution perspective, code between any two dynamically successive yield statements executes atomically in OCM, but exactly how that atomicity is achieved is an implementation choice. Implementation techniques for OCM include lock inference and software transactional memory, allowing us to potentially compare the exact same program under radically different concurrency-control schemes.

Unfortunately, prior to this paper, the results of such comparisons did not reflect well on any of the schemes we tested. In numerous experiments, we reached the unsatisfying conclusion that in almost all situations where the inter-thread synchronization problems were interesting enough to warrant using OCM, the best-performing technique was serial execution! Thankfully, the OCM implementation that we describe in the next section actually does provide parallel speedup for meaningful programs.

4. HLE Implementation of OCM

In this section, we describe an extremely simple but extremely effective implementation technique for OCM, which owes its good performance entirely to Intel's HLE technology.

4.1 Naïve Implementation

One of the most simple-minded ways to implement OCM (besides just using a CM system) is to use a single global lock to protect all shared data. In this case, yield could be implemented as globalLock.release() immediately followed by globalLock.acquire(). Like CM, only one thread would run at a time, but different threads might execute on different cores.

Such a scheme should be expected to have mediocre performance. With only one thread running at a time, we can expect no parallel speedup, and thanks to the overheads of thread creation and cache effects between processors, it is quite likely that execution will run slower than it would on a single core under CM.

On the positive side, this scheme is trivial to implement. It also forms the basis for our next approach.

4.2 HLE Implementation

Intel's *Hardware Lock Elision* (HLE) technology takes existing code that uses a single global lock to ensure mutual exclusion and uses transactional atomicity to replace the lock. If two threads that wish to hold the lock do not otherwise interfere with each other, HLE may execute them simultaneously. Thus, as OCM is to CM, so HLE locks are to regular locks. In the absence of introspection about what mode the CPU is in, the only difference in behavior between HLE locks and regular locks is the performance difference, with a potential benefit from parallel execution.

Thus our HLE implementation of OCM takes the simple-minded global lock and simply makes it an HLE global lock. That's all!

HLE does not guarantee to *always* elide the lock and execute transactionally. If it cannot execute transactionally for some reason (contention or system limitations such as transaction size), it falls back performing a conventional lock acquisition.

4.2.1 Going Beyond yield

As we mentioned in Section 2.3, a complete implementation of OCM should provide a broader array of concurrency-control mechanisms than just yield to support the various scenarios in which one thread may need to wait for another. But we cannot carelessly reuse the concurrency-control mechanisms present in the system that OCM is built on (e.g., POSIX threads).

It is, however, straightforward to implement suitable concurrency-control mechanisms using system-provided primitives. One method is to associate each thread with a system mutex and condition variable that allows a thread to suspend itself and another thread to wake it up. Before sleeping, it acquires its own mutex, drops the global lock, and finally waits to be woken. To wake another thread, we are already holding the global lock (because we're running), and we may have to wait to acquire the target thread's mutex because it is still in the process of falling asleep, but eventually it will become available and then we can signal the thread to wake.

With a mechanism to put threads to sleep and wake them up again, it is straightforward to implement any desired concurrency control primitive. Our implementation provides condition variables, barriers, semaphores, and buffered communication channels.

4.3 Shrinking Atomic Regions

In our scheme as discussed so far, everything is atomic all the time, with atomicity firmly enforced via a global lock that must be held at all times (either a traditional mutex or one implemented with HLE).

Interestingly, however, the global-lock scheme can be optimized by delaying globalLock.acquire() until shared state is about to be accessed for the first time since yielding. Likewise, if the system can determine that a yield is imminent and that no more accesses to shared data will occur before the next yield is reached, it can perform the globalLock.release() action ahead of the actual yield. We call these two lock optimizations *lazy acquire* and *eager release*. Both optimizations preserve CM serializability, yet allow thread executions to overlap in parallel.

Implementing such a functionality requires a compiler to know when the program is merely accessing thread-local state and when it accesses (or may access) global state. Compilers routinely perform these kinds of analysis because it affects a wide variety of optimizations (including what can be stored in

CPU registers), so simple variations of lazy acquire and eager release require no significant new static analysis.

5. Performance

In this section, we will examine the performance of OCM implemented using HLE and with a simple-minded global lock, with and without the atomic-region–shrinking optimization.

The benchmark is the simplified variant of Rich Hickey's ant colony simulation we discussed in Section 2.2, where multiple ants explore a 2D grid. This benchmark has several useful properties. First, the synchronization demands are unpredictable—ants move randomly and must avoid interfering with each other. Second, we can vary contention in two ways, by changing the size of the grid (which affects how often shared accesses interfere) and by altering how much time ants spend "thinking" between their examinations of the grid (which affects how often shared accesses occur at all).

In all cases, we compare speedup to a serial implementation where all exploration is performed by a single thread. In this way, we realistically show how much speedup a user might expect from parallelizing serial code, rather than the speedup from running a "parallel" implementation (with all overheads it incurs) on different numbers of cores. In our approach, the serial code uses one core, and parallel code has four cores available.[2] Thus the maximum possible speedup is $4\times$, although the added complexities of *any* parallel implementation make $4\times$ speedup over serial execution unlikely. For simplicity, only four ants explore the grid, but similar results are obtained with larger numbers of ants.

We also do not favor parallel code by arranging data in a friendly way. Each square of the grid is a single byte, which leads to considerable false sharing when executed in parallel. This arrangement is particularly hostile to HLE, because false sharing will cause it to detect conflicts where none exist.

Figure 1(a) shows the performance when running the simulation with approximately equal time spent between thinking and grid exploration. The x-axis shows the grid size; contention decreases with larger grid sizes. With four ants exploring an 8×8 grid, and each ant examining its immediate eight cells, we should expect significant contention and we do see that (in fact, almost 20% of transactions fail due to conflicts). Nevertheless, both HLE implementations outperform the serial implementation. In contrast, the simplistic global lock fails badly, causing a significant slowdown over the serial code. The optimized versions apply the ideas we discussed in Section 4.3 so that the global lock is not held while thinking. This optimization improves the performance of both schemes, but is not enough to make the global-lock scheme outperform serial execution.

[2] When we performed our tests, Intel's HLE implementations were restricted to four-core systems (arguably the most common CPU configuration). Our tests were run on a desktop machine with an Intel "Haswell" Core i7-4770 CPU, running at 3.40GHz with Turbo-boost disabled. In future work we expect to test a server-class machine with a larger number of cores.

The optimized global lock scheme is about the best we could hope for with a simple locking strategy, but some readers may believe that a fine-grained locking scheme would perform better. Even if we were to assume that better strategies were possible with fine-grained locking, or clever lock-free approaches, these approaches have the downside of being *much* more complex.

In Figures 1(b) and 1(c), the ants do more computing between accesses to shared state, so accesses to shared state are less likely to clash (whether or not they actually interfere). The performance of the serial code and the simplistic global lock remains the same, but all the other schemes improve their level of parallel speedup. In particular, the global lock that is released while thinking finally performs better than serial execution, but not as well as its HLE counterparts.

These results indicate that across a space of parallel programs with different access patterns, HLE can offer useful parallel speedup to extremely simple concurrency control strategies. Using the OCM model and HLE, we can parallelize programs with very modest effort and strong certainty about their correctness, and yet achieve noticable parallel speedup.

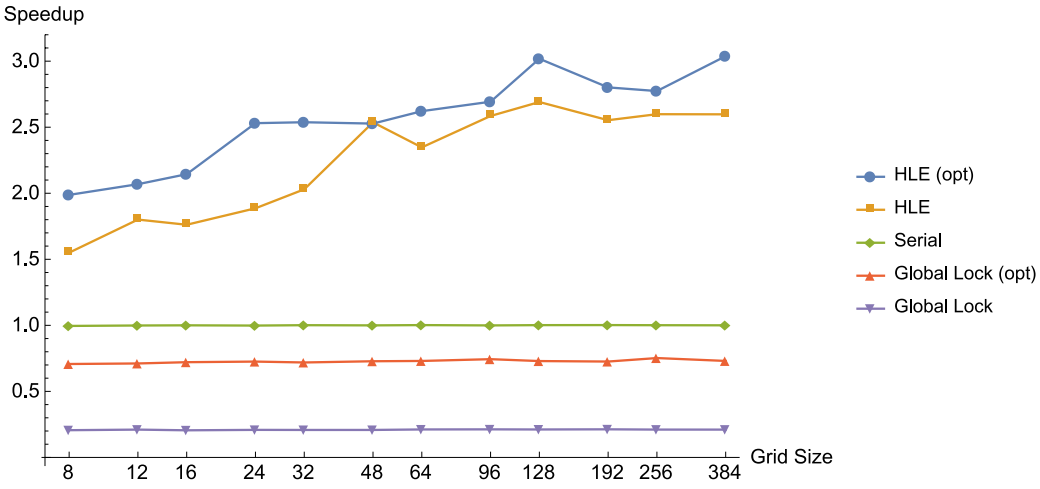
6. Conclusions and Future Work

OCM provides a straightforward way to understand and express problems that require irregular parallelism. The code is easy to write and easy to reason about, but until now, many programs written for this model have not seen useful parallel speedup. Hardware transactional memory changes that state of affairs, allowing even a very simple implementation of OCM to offer good performance.

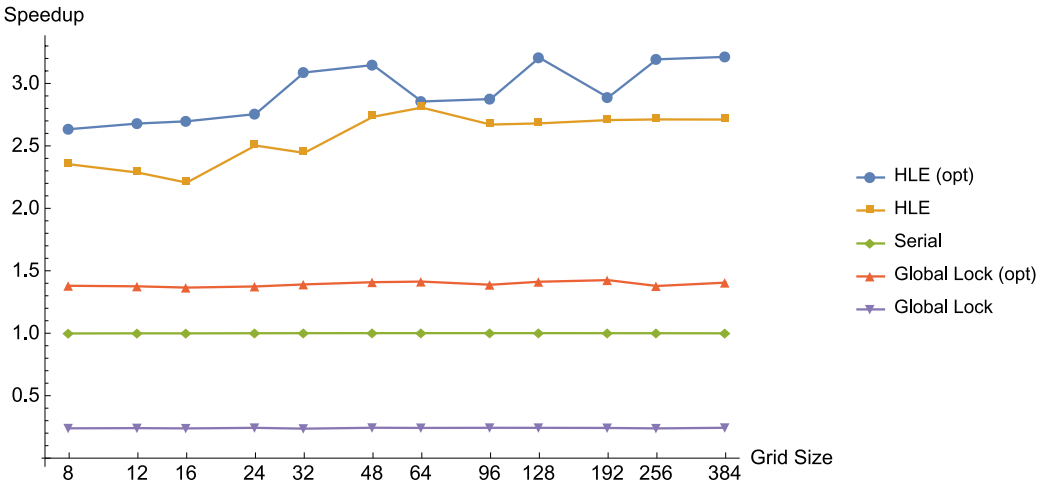
OCM is not intended to replace carefully written custom synchronization schemes. Rather, we see it as a "kinder gentler" form of multicore parallelism that is highly suited to education, prototyping, and other areas where coding complexity concerns trump absolute performance. Our original expectations for OCM were that it would never be the fastest technique, but it could often be the easiest one. The HLE performance results we've shown in this paper show that, at least *sometimes*, performance can actually be extremely promising.

A key reason why HLE fits OCM so well is that OCM doesn't expose the concept of transactions or transaction failure to the user. As such, the limitations of HLE (e.g., limits on the length of transactions, some instructions, including system calls, not being executed transactionally) pose no problems for our model, merely causing some additional serialization.

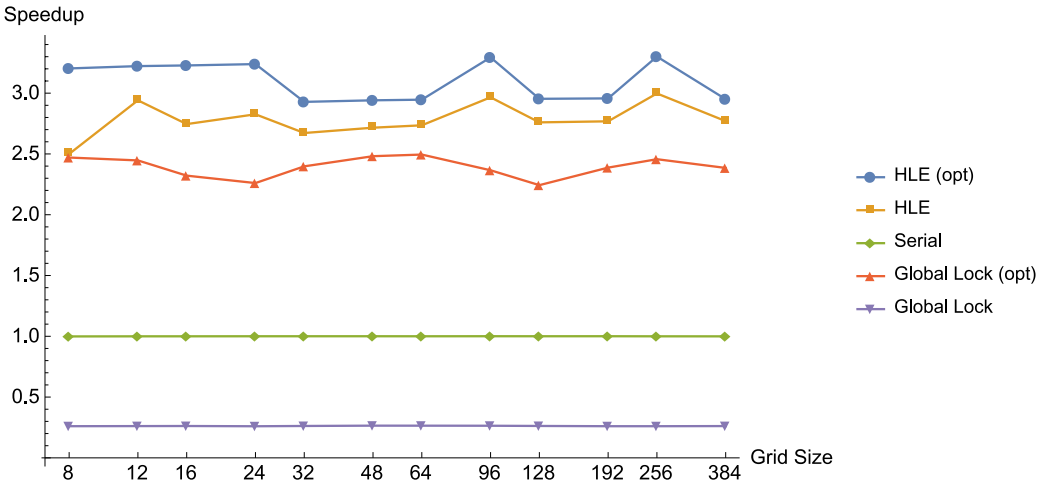
We've shown that an embarrassingly simple implementation scheme can offer good performance, but there are plenty of opportunities for improvement. For example, it isn't hard to record information about yield points to determine whether they usually execute transactionally or not, and skip transactional execution when it will always abort. These kinds of improvements are a worthy topic for future exploration.



(a) Equal thinking and exploring.



(b) Two-to-one thinking vs exploring.



(c) Four-to-one thinking vs exploring.

Figure 1. Performance of the Ants Benchmark.

We also hope that this example inspires others to consider whether other previously embarrassingly bad strategies for parallel coordination become not just feasible, but performant, under HLE.

7. Related Work

We are not the first to see the CM model as a generally desirable model for programmers [8, 28]. In recent years, it has been suggested most often to combine the efficiency and simplicity advantages of sequential event-driven code with a more natural programming model [6, 29]. There is still debate about the relative overheads of events and cooperative threads, but there is no doubt that CM can provide an attractive and natural model for systems programming.

Rossbach et al. [18] found that undergraduate students in a systems class had more difficulty understanding the transactional concurrency model than they did the coarse-grained locking model. However, when they completed an assignment using both the locking and transactional models, many more students made errors with explicit locking than with transactions. From this we see that both the transactional and locking models have weaknesses when it comes to ease of understanding and ease of use.

Skillicorn and Talia [22] observed that "conscious human thinking appears to us to be sequential, so that there is something appealing about software that can be considered in a sequential way", that "models ought to be as abstract and simple as possible", and that "[g]enerally, easy-to-use tools with clear goals, even if minimal, are preferable to complex ones that are difficult to use." These views are echoed by Sadowski and Kurniawan [19]. By making a firm commitment to sequential cooperative semantics, we believe that the OCM model makes significant progress towards these goals.

Yi et al. [30] use the easy-to-reason-about properties of CM to advocate for programs with explicitly marked yields, but in their approach yield is not a statement but an annotation for programs that already include traditional synchronization. Their approach allows the system to detect (and flag as an error) program executions interleave in ways not explicitly allowed by its yields. Our approach and theirs are compatible—with our OCM implementation, a serial program can be made parallel with yield statements, and then if higher performance is desired, explicit synchronization can be added while retaining the yields as annotations.

As a parallel model, OCM intersects with a significant portion of prior work on parallelism and concurrency. There is a vast literature describing parallel models, concurrency-control mechanisms, debugging techniques, and so forth that could be compared to OCM, but we cannot hope to do them all justice here. Thus, we must restrict our discussion to those techniques that we feel are of most interest because they parallel, influence, or counterpoint OCM in particularly significant ways.

7.1 HTM and Global Locks

One of the most widely known instances of global locking are the "global interpreter locks" used by scripting language interpreters such as Python and Ruby. Riley and Zilles [17] were one of the first authors to discuss the value of hardware transactional memory in eliding global locks for interpreters, improving Python, although their work relied on a simple simulator. Various authors have further developed the idea [4, 27] of using HTM to elide Python's global lock. Most recently, Odaira and colleagues [14] showed fairly good speedup for an implementation of Ruby where HTM was used to eliminate the global lock. But whereas the global lock in Python and Ruby was designed to protect shared data representing the core state of the interpreter and relates to the interpreter runtime (and is thus often highly contended), our use of a global lock is far less specific—exactly what the global lock protects depends entirely on the user program. In the degenerate case, the lock protects nothing at all.

7.2 Other Models of Concurrency

There are, of course, many other models for parallel programming, including monitors [10] and Java synchronized methods [7], communicating sequential processes [11], Threading Building Blocks [16], Transactions with Isolation and Concurrency [23], Cilk [3] and OpenMP [15] to name just a few. We cannot compare each in detail here, but to the extent that they provide particular scheduling policies or ways to create new threads (e.g., parallel for loops), they may be transferrable to an OCM context. However, one model deserve specific comparison with OCM.

Automatic Mutual Exclusion AME [2, 12] is a variant of software transactional memory. Like OCM, AME makes the assumption that all code should be executed atomically by default, but provides nonatomic escape hatch in the form of unprotected blocks. Consequently, atomic code is dynamically delimited by the execution of unprotected blocks. An AME system could easily be used to implement OCM (yield corresponds to an empty unprotected block [2]), and AME has already engendered work on the denotational semantics of uniprocessor cooperative multithreading [1].

While we have found the AME work inspiring, there are two ways in which OCM intentionally differs from AME.

First, although nonempty unprotected blocks can improve performance, but they can also be a source of bugs and semantic surprises [21]. By avoiding this feature, OCM sacrifices some performance to offer a simpler and safer model.

AME also exposes the underlying STM implementation. Its blockUntil operator permits users to roll back and retry in the middle of an atomic transaction, allowing code that is impossible without run-time tracking and undoing side-effects. AME's approach cannot be implemented using only HLE because HLE sometimes executes atomic sections nontransactionally.

Acknowledgments

A number of undergraduates from Harvey Mudd College have worked on the OCM project, from developing implementations to porting benchmarks to testing usability. Specifically, we would like to thank Joe Agajanian, Savannah Baron, Sonja Bohr, Bartholomew Broad, John Brooks, Chloe Calvarin, Adam Cozzette, Michael DeBlasio, Xiaofan Fang, Alec Griffith, Ki Wan Gkoo, Alexander Gruver, Samuel Just, Kwang Ketcham, Joseph Klonowki, Sean Laguna, Joshua Landgraf, Stephen Levine, Jordan Librande, Alejandro Lopez-Lago, Julia Matsieva, Joshua Peraza, Stuart Pernsteiner, John Phillpot, Ari Schumer, Mary-Rachel Stimson, Alice Szeliga and Jesse Werner. Their efforts and frustrations showed us that for a wide variety of benchmarks and implementation techniques, true parallel speedup seemed to be a remarkably difficult goal to attain.

References

[1] M. Abadi and G. Plotkin. A model of cooperative threads. In *POPL '09*, pages 29–40, 2009.

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

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *PPoPP '95*, pages 207–216, 1995.

[4] C. Blundell, A. Raghavan, and M. M. Martin. Retcon: Transactional repair without replay. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 258–269, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0053-7.

[5] R. S. Engelschall. Gnu Pth—the GNU portable threads. June 2006. URL `http://www.gnu.org/software/pth`.

[6] J. Fischer, R. Majumdar, and T. Millstein. Tasks: language support for event-driven programming. In *PEPM '07*, pages 134–143, 2007.

[7] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java™ Language Specification (3rd Edition)*. Addison-Wesley Professional, 2005.

[8] A. Gustafsson. Threads without the pain. *Queue*, 3(9):34–41, 2005.

[9] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05*, pages 48–60, 2005.

[10] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[11] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[12] M. Isard and A. Birrell. Automatic mutual exclusion. In *11th USENIX Workshop on Hot Topics in Operating Systems (HotOS XI)*, pages 1–6, 2007.

[13] J. R. Larus and R. Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2007.

[14] R. Odaira, J. G. Castanos, and H. Tomari. Eliminating global interpreter locks in ruby through hardware transactional memory. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 131–142, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2656-8.

[15] OpenMP Architecture Review Board. OpenMP Application Program Interface, version 3.0. May 2008.

[16] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.

[17] N. Riley and C. Zilles. Hardware tansactional memory support for lightweight dynamic language evolution. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 998–1008, New York, NY, USA, 2006. ACM. ISBN 1-59593-491-X.

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

[19] C. Sadowski and S. Kurniawan. Heuristic evaluation of programming language features: Two parallel programming case studies. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '11, pages 9–14, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1024-6.

[20] N. Shavit and D. Touitou. Software transactional memory. In *Fourteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*, pages 204–213, 1995.

[21] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in STM. In *PLDI '07*, pages 78–88, 2007.

[22] D. B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, 1998.

[23] Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. Transactions with isolation and cooperation. In *OOPSLA '07*, pages 191–210, 2007.

[24] M. F. Spear. Lightweight, robust adaptivity for software transactional memory. In *22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '10)*, pages 273–283, 2010.

[25] C. A. Stone, M. E. O'Neill, and The OCM Team. Observationally cooperative multithreading. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 205–206, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0942-4.

[26] C. A. Stone, M. E. O'Neill, S. A. Bohr, A. M. Cozzette, M. J. DeBlasio, J. Matsieva, S. A. Pernsteiner, and A. D. Schumer. Observationally Cooperative Multithreading. *ArXiv e-prints*, 1502.05094, Feb. 2015 (originally written in 2011).

[27] F. Tabba. Adding concurrency in python using a commercial processor's hardware transactional memory support. *SIGARCH Computer Architecture News*, 38(5):12–19, Apr. 2010. ISSN 0163-5964.

[28] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *9th USENIX Workshop*

on Hot Topics in Operating Systems (HotOS IX), pages 19–24. USENIX Association, 2003.

[29] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *SOSP '03*, pages 268–281, 2003.

[30] J. Yi, C. Sadowski, and C. Flanagan. Cooperative reasoning for preemptive execution. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 147–156, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0119-0.