Automatic Mutual Exclusion

Michael Isard and Andrew Birrell
Microsoft Research, Silicon Valley

Abstract

We propose a new concurrent programming model, *Automatic Mutual Exclusion (AME)*. In contrast to lock-based programming, and to other programming models built over software transactional memory (STM), we arrange that *all* shared state is implicitly protected unless the programmer explicitly specifies otherwise. An AME program is composed from serializable atomic fragments. We include features allowing the programmer to delimit and manage the fragments to achieve appropriate program structure and performance. We explain how I/O activity and legacy code can be incorporated within an AME program. Finally, we outline ways in which future work might expand on these ideas. The resulting programming model makes it easier to write correct code than incorrect code. It favors correctness over performance for simple programs, while allowing advanced programmers the expressivity they need.

1 Introduction

Since the 1960's, system programmers have been trying to create and maintain *concurrent programs*. Initially, this desire arose as a means to allow the computer to continue useful program execution while some other part of the overall computation was unable to make progress because of a peripheral device. It also arose with the development of systems that shared the processor between multiple computations. Those who thought carefully about the problem abstracted it into the concept of multiple concurrent *threads* of execution (typically called *processes* at the time). In most analyses (CSP being an exception) the threads communicate through shared memory, and cooperate by achieving mutual exclusion on access to the memory [4]. Over time, this has become standardized as concurrent threads using shared memory and mutexes.

Unfortunately, in the real world most programmers find it difficult to use threads, shared memory, and mutexes correctly. In practice most mainstream applications tend to exhibit too little concurrency (they deadlock, or the user interface freezes during a network I/O, or they perform poorly on a multi-processor), or else they exhibit too much concurrency (by providing an incorrect answer, especially on a multi-processor, and usually non-deterministically).

A variety of problems with mutexes seem to cause much of this difficulty. One is the constraint that all the mutexes in a program have a partial order in which they are acquired, in order to prevent deadlocks. While this is a manageable constraint in systems built by small teams of programmers, it can be very difficult to meet in large systems, and extremely difficult over the long maintenance life of such systems. It is especially difficult to debug situations in which such deadlocks occur with very low probability, or where they are not formally a deadlock at all, just an unpleasantly long delay (such as waiting for a network service with a mutex held).

A second difficulty is often described as *lack of composability.* If a programmer wishes to layer an atomic operation on top of an existing abstraction (for example, adding an atomic "move" operation to move an item between two hash tables), this is difficult to do without access to the internal mutexes and mutual exclusion algorithms of the lower-level abstraction.

Finally, the existing mutex designs are fundamentally fragile: performance requirements (real or imaginary) create an incentive for the programmer to be "clever", and to minimize the use of mutexes, or their scope, or the amount of data protected by them. While this can be achieved correctly by a sufficiently clever programmer, it makes for tricky and un-maintainable programs — especially after the clever programmer has moved on to other projects.

Transactional memory (in software today, in hardware later) shows promise as an alternative to cooperation through mutexes [5]. It is generally presented to the programmer as *atomic blocks*, where the programmer delineates a region of the program to be executed as a transaction. The runtime system takes responsibility for executing the program in such a way as to have the same semantics as if the atomic blocks had been executed in some serialized order.

Serialized transactions have an appealing simplicity, and atomic blocks alleviate quite a lot of the problems of mutexes. The program no longer needs a partial order on its mutual exclusion calls, because if a conflict occurs the transaction machinery will detect it, and will almost certainly fix it by a suitable series of transaction aborts and retries. Atomic blocks also fix the composability problem: in the hash table "move" example, wrapping an atomic block around the get-insert-delete sequence will create the desired atomicity.

Unfortunately, just providing atomic blocks will not solve all our mutual exclusion problems. Fundamentally, they still require the programmer to decide what regions of the program need protection, and what data must be protected. Furthermore, the defaults are the wrong way round: when a programmer cleverly decides that it is correct to narrow a mutual exclusion range, or to exclude some data from it, the programmer does so by removing text from the source code. We believe that this makes programs harder to understand, and quite difficult to maintain. For large systems (the only ones that are really difficult in this area), this is a critical defect. Put another way: with atomic blocks, the default and simplest program is one with no atomic blocks. Unfortunately, it is also the one least likely to work correctly.

The highest-level goal of this paper is to point out that the mechanism of transactional memory can be presented to the programmers with language constructs other than atomic blocks. We propose one such arrangement, but we have no doubt that others will follow.

Our proposal here is a new programming model, *Automatic Mutual Exclusion (AME)*, which we believe (hope) will cause programmers to be more likely to create applications that are concurrent, correct, and responsive, and which will remain so over the applications' life cycles. We do this by reducing the programmers' responsibility for concurrency and synchronization. By default, an AME program is correctly synchronized: if the programmer thinks not at all about mutual exclusion, there will be no data races. We then allow the programmer to take this correctly synchronized program and optimize it, by *adding* to the source code, not subtracting from it. Optimizing the concurrency behavior of an AME program requires actions where the programmer explicitly declares the places where the optimizations occur, in a way that we believe will be maintainable.

2 Asynchronous method calls

We begin by describing a simplified AME programming model that supports basic concurrent event-based programming. The remainder of the AME model is described in sections 3 and 4.

In this simplified model, running an AME program consists of executing a set of *asynchronous method calls*. The AME system guarantees that the program execution is equivalent to executing each of these calls in some serialized order (i.e., atomically). AME achieves concurrency by overlapping the execution of the calls, subject to maintaining this guarantee. The program terminates when all its asynchronous method calls have completed.

Initially, the set consists of a call of `main(…)` initiated by the AME system. Within an asynchronous method call, the program can create another asynchronous method call by executing:

```
async MethodName(MethodArguments);
```

In terms of the formal semantics of the program, the newly created asynchronous method call will be serialized after the current one. In terms of program structure, the `async` construct is reminiscent of forking a thread in thread-like systems, or posting an event in event-based systems (but subject to the serialization guarantee).

While the semantics are serialized execution, naturally the AME system will attempt to execute the set of available asynchronous method calls concurrently, within the available resources and subject to strategies (TBD!) that prevent excessive transaction aborts.

To achieve our serialization guarantee, our basic implementation is that each asynchronous method call will be executed by the AME system as an STM (or HTM) transaction, within a thread from a pool provided by the AME system. However, the semantics say nothing about transactions: if the AME system can determine that a cheaper synchronization scheme (such as mutexes, or no locking at all) will achieve the serialization guarantee for a particular program execution, it is free to use that scheme.

When a transaction initiates other asynchronous method calls, their execution is deferred until the initiating transaction commits. If the initiating transaction aborts, they are discarded. When it commits, they are made available for execution (in an indeterminate order).

Within an asynchronous method call, the program is not permitted to take actions with side-effects that the AME system cannot undo: this enables an implementation using transactional memory. In particular, I/O activity cannot occur (but we'll deal with that below).

This much of the design allows programming with concurrent non-blocking asynchronous method calls (or events). The program is correct, in that the calls can share memory without any risk of races. The calls can execute concurrently when possible, and the AME system will ensure that the result is a valid serialization of the events. The programmer has written no synchronization code.

We still need give the programmer optimization mechanisms: some control over transaction scheduling (section 2.1), and some way to split up transactions to reduce the frequency of aborts or to enable rational program structure (section 3). We also need to provide for legacy code, and for code with non-abortable side-effects such as actual I/O (section 4).

2.1 Blocking an asynchronous method

An asynchronous method may contain any number of calls to the system-supplied method:

```
  BlockUntil(<predicate>);
```

From the programmer's perspective, the code of an asynchronous method executes to completion only if all the executed calls of `BlockUntil` within the method have predicates that evaluate to `true`.

`BlockUntil's` implementation does nothing if the predicate is `true`, but otherwise it aborts the current transaction and re-executes it later (at a time when it is likely to succeed). This is like `Retry` in some systems.

For example, a blocking read from a shared queue could be implemented as:

```
  BlockUntil(queue.Length() > 0);
  data = queue.PopFront();
```

Notice that the AME system has a lot of information available when `BlockUntil` is called with `false`: it can in principle determine what non-local memory affected the evaluation of the predicate, and it can determine when other asynchronous method calls later modify that memory. We envisage taking advantage of this to optimize the scheduling of the transaction retry.

2.2 Examples and discussion

At this point we introduce some example fragments that illustrate common concurrent idioms. By convention variables that live in the shared heap begin 'g_' (for "global"). The first example performs overlapped reading from a file, where 4 blocks are in flight at any given time (error handling is ignored for simplicity, and we elide the details of how I/O is performed within the file library, since that needs section 4):

```
  void OpenRead(FileName name) {
    File f = StartOpen(name);
    async StartRead(f);
  }

  void StartRead(File f) {
    BlockUntil(f.Opened);
    g_nextOffset = 0;
    g_nextOffsetToEnqueue = 0;
    for (int i=0; i<4; ++i) {
      ReadBlock block = new ReadBlock;
      block.offset = g_nextOffset;
      block.file = f;
      g_nextOffset += block.size;
      f.StartRead(block);
      async WaitForBlock(block);
    }
  }

  void WaitForBlock(ReadBlock block) {
    BlockUntil(block.ready &&
               g_nextOffsetToEnqueue ==
                 block.offset);
    if (block.EOF) {
      g_endOfFile = true;
    } else {
      g_queuedBlocks.PushBack(block);
      block.offset = g_nextOffset;
      g_nextOffset += block.size;
      block.file.StartRead(block);
      async WaitForBlock(block);
    }
    g_nextOffsetToEnqueue += block.size;
  }
```

The second example simulates a fragment of a computer game, implementing the logical thread of control for a particular character that is moving autonomously and interacting with its environment:

```
  void StartZombie() {
      Zombie z;
      z.Initialize();
      /* schedule the first move */
      async UpdateZombie(z);
  }

  void UpdateZombie(Zombie z) {
    Time now = GetTimeNow();
    BlockUntil(now - z.lastUpdate >
               z.updateInterval);
    z.lastUpdate = now;
    MoveAround(z);
    if (Distance(z, g_player) <
      DeathRadius) {
      KillPlayer();
    } else {
      /* schedule the next move */
      async UpdateZombie(z);
    }
  }
```

The final example illustrates a data-parallel computation that processes every item notionally "in parallel" and inserts the results into an output list whose ordering is undefined:

```
void DoBatch(Queue inQ, Queue outQ) {
  BlockUntil(inQ.Length() > 0 ||
             g_finished);
  while (inQ.Length() > 0) {
    Item i = inQ.PopFront();
    async DoItem(i, outQ);
  }
  if (!g_finished) {
    async DoBatch(inQ, outQ);
  }
}

void DoItem(Item i, Queue outQ) {
  DoSlowProcessing(i);
  /* Writing to outQ here would serialize the slow
     processing, because of contention on outQ */
  async DoOutput(i, outQ);
}

void DoOutput(Item i, Queue outQ) {
  outQ.PushBack(i);
}
```

Note that none of these examples require the programmer to make a determination of what shared state must be protected; the AME system protects the state automatically. The programmer is not tempted to consider, for example, leaving the code in `StartRead` after the `BlockUntil` call unprotected. We believe this automatic concurrency protection will be extremely valuable in complex and long-lived programming projects.

The event-based AME programming model is attractive because it makes it extremely difficult to write a "fine-grain" concurrency bug: every method call is always executed in its entirety or not at all. We also believe that the mental model of serialized execution is among the simplest concurrency abstractions for programmers to understand. Higher level races are still possible (e.g., by assuming that some state is preserved unmodified between asynchronous method calls).

3 Fragmenting an asynchronous method

With the facilities of the previous section, the programmer will inevitable be faced with a situation where a conceptually single asynchronous method call must be split up. In the simplest cases, this arises when the call creates too many memory conflicts with other calls, causing too many transaction aborts. (This is analogous to holding a mutex for too wide a range of the program.)

As explained by several previous authors [1,3], other cases arise that require splitting up events in a pure event-based model, producing program structure that can be unpleasant, and unstable. For example, if a previously non-blocking method call is modified to require a blocking action (e.g., a hash table is modified to use disk storage instead of main memory), the event-based style would require that the method, and all of its callers, gets split into two separate methods (a request and a response handler). This is sometimes referred to as "stack ripping".

Our solution to both of these problems is to allow an asynchronous method call to contain one or more calls to the system method `Yield`. A `Yield` call breaks a method into multiple *atomic fragments*. This is similar to breaking an atomic block into multiple adjacent blocks, except that our atomic fragments are delimited dynamically by the calls of `Yield`, not statically scoped like explicit atomic blocks.

With this enhancement, the overall execution of a program is guaranteed to be a serialization of its atomic fragments. We (intend to) implement `Yield` by committing the current transaction and starting a new one.

A `BlockUntil` call only blocks execution of the current atomic fragment (the code following the most recent `Yield()`), or equivalently, it only retries the transaction begun after the most recent `Yield`.

A `Yield` call can occur within any method, including libraries. Since `Yield` splits atomic executions, it is critical that a caller be aware of this possibility: the caller's own shared state becomes visible to other asynchronous method calls, and might be changed by other asynchronous method calls. We require that any method containing a `Yield` makes this explicit statically, by having a type signature such as the following:

```
ReturnType MethodName(Args) yields {…}
```

Any synchronous call to such a method must be decorated:

```
int foo = Method(x) yielding;
```

Of course a function that calls a `yielding` method must itself be marked `yields`.

Notice the effect on program maintenance. If a previously non-yielding library changes its implementation to use `Yield`, then the library's callers will get a compilation error because of the lack of a `yielding` annotation. The callers must then either determine that it is correct to expose their shared state (i.e., their invariants are true), or they must remove the offending call.

Here as elsewhere in the design we have chosen to require that the programmer makes explicit the places where races might occur, leaving the default case correctly synchronized, despite the inconvenience this causes.

We might be concerned that most library calls will contain *some* code path that leads to a `yielding` method. The result would be that most code would become tainted as `yields`, eliminating any useful information content from the annotation. However, there is no incentive to insert a call to `Yield` in a library method unless that method is either extremely long-running, or must block, e.g. waiting for synchronous I/O. Consequently we view the potential for a library to become tainted as a strength rather than a weakness. Some of the worst performance bugs arise from calling a method that is normally fast, but that blocks occasionally due to rare corner cases. Under our model, the programmer will know to avoid such calls or else plan for the possibility of a slow execution.

At this point we can rewrite our zombie and queue examples more concisely:

```
void RunZombie() yields {
  Zombie z;
  z.Initialize();
  do {
    Yield();
    Time now = GetTimeNow();
    BlockUntil(now - z.lastUpdate >
               z.updateInterval);
    z.lastUpdate = now;
    MoveAround(z);
    if (Distance(z, g_player) <
        DeathRadius) {
      KillPlayer();
    }
  } while (Distance(z, g_player) >=
           DeathRadius);
}

void DoQueue(Queue inQ,
       Queue outQ) yields {
  do {
    Yield();
    BlockUntil(inQ.Length() > 0 ||
               g_finished);
    while (inQ.Length() > 0) {
      Item i = inQ.PopFront();
      async DoItem(i, outQ);
    }
  } while (!g_finished);
}

void DoItem(Item i,
       Queue outQ) yields {
  DoSlowProcessing(i);
    Yield();
    outQ.PushBack(i);
  }
```

In addition to the obvious structural improvements, notice that the zombie `z` variable is now allocated on the thread's private stack. In a transactional environment thread-private variables are more efficient, and making this clear may make optimization easier.

4 Unsynchronized Fragments

There are times when automatic mutual exclusion prevents the program from doing what it needs to do. The two obvious cases are access to legacy code that doesn't use this machinery, and code with non-abortable side-effects such as I/O. To support these we allow the following:

```
unprotected { … }
```

This construct terminates the current atomic fragment (typically by committing the current transaction), then executes the inner block, then starts a new atomic fragment. Any method that uses `unprotected` must be flagged as `yields`.

The typical pattern for a region of the program that wants to perform I/O (probably by calling legacy libraries, but perhaps by calling the kernel directly, or even by writing to device registers) will be as follows. Within an atomic fragment, the program decides on the details of the I/O operations and stores them on a thread-local queue. It then uses `unprotected` to execute code that extracts the requests from the thread-local queue and passes them into the actual I/O system. Typically, this dance will be performed within some AME library code or wrappers. The library calls that perform synchronous I/O will consequently be marked as `yields`, as expected.

However, an asynchronous I/O library (such as the `StartOpen` and `StartRead` methods used in the example in Section 2.2) does not need to have its methods marked as `yields`, because the library internally defers the actual I/O calls by invoking them through asynchronous method calls.

Notice that since this particular pattern of using `unprotected` touches only thread-local and non-transacted memory, it is not subject to the issues of privatization common in transactional memory designs when interacting with non-transactional code. Other uses of `unprotected` might have privatization problems, and we are considering making them illegal.

5 Discussion

This proposal is not about inventing fundamentally new semantics for concurrency. Primarily, we are exploring a new way to present our existing mechanisms to the programmer. We intend that the programmer can write straightforward code first, and achieve a good level of concurrency with little risk of races. As the program develops, it can be optimized to achieve better concurrency, using mechanisms that flag the places where concurrency has been improved at the risk of introducing races. We encourage correctness first, performance second, and maintainability always. We support a non-blocking event-driven style, or a blocking procedural style with `Yield`, or any convenient combination.

The design is intended for real programs (though limited by today's STM performance). We intend to handle programs that perform disk and network I/O and that interact with the user. We intend to be able to handle large systems, with long lifetimes.

One way to view this design is to compare it with either single-threaded cooperative multi-tasking systems, or with single-threaded event based systems such as the JavaScript side of AJAX. Those programming models are similar to AME, except that AME can execute the program with real concurrency, utilizing real multi-processors — with very little extra effort from the programmer.

There are numerous research questions that we see arising from the AME proposal. Most obviously, we need to implement AME, develop some real experience, and determine whether it is in fact useful. We have a good STM implementation available to us, and we plan to modify our compiler to support the AME extensions instead of explicit atomic blocks.

Critical to AME performance is the scheduling of transactions so as to minimize the amount of work that will be aborted. We are hopeful that `BlockUntil` will help in this (more so than a simple `Retry` statement).

There are optimizations available in some cases of `BlockUntil`, especially where it occurs as a guard at the start of an atomic fragment (for example, we might not need to use transactions for those cases). We are considering whether to restrict its use to just those situations.

Transactional memory designs will continue to be impractical until the system can optimize by eliminating transactional overhead for memory accesses that are in fact thread-private. We have some tentative but incomplete ideas for doing this.

Our `Yield` operation could be enhanced by having the programmer specify a subset of the transactional variables that should be modifiable during the yield. This would enhance correctness by allowing the system to report an error if some other transaction attempts to modify the non-modifiable state.

Overall, we are excited by the possibility of the AME ideas. We believe they make it much more likely that programmers will create correct, efficient, and maintainable concurrent programs.

6 References

There are many important works in this area. We have assembled the 20 that we found most useful on a web page [4]. Below, we cite only the ones most specific to this paper.

1. Adya, A. et al "Cooperative Task Management without Manual Stack Management", Proc. Usenix 2002 Annual Technical Conference, June 2002.
2. Bacon, D. et al "The 'Double-Checked Locking is Broken' Declaration", http://tinyurl.com/1rja, viewed December 2006.
3. Von Behren, R et al "Why Events Are A Bad Idea", Proc. 9th Workshop on Hot Topics in Operating Systems, May 2003.
4. Birrell, A. "A Selected Bibliography of Concurrency", http://birrell.org/andrew/concurrency/, December 2006.
5. Harris, T. "Composable Memory Transactions", Proc. 10th Symposium on Principles and Practice of Parallel Programming, June 2005.