The Problem with Threads

Edward A. Lee

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-1
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.html

January 10, 2006

Acknowledgement

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF award No. CCR-0225610), the State of California Micro Program, and the following companies: Agilent, DGIST, General Motors, Hewlett Packard, Infineon, Microsoft, and Toyota.

The Problem with Threads

Edward A. Lee
Professor, Chair of EE, Associate Chair of EECS
EECS Department
University of California at Berkeley
Berkeley, CA 94720, U.S.A.
eal@eecs.berkeley.edu

January 10, 2006

Abstract

Threads are a seemingly straightforward adaptation of the dominant sequential model of computation to concurrent systems. Languages require little or no syntactic changes to support threads, and operating systems and architectures have evolved to efficiently support them. Many technologists are pushing for increased use of multithreading in software in order to take advantage of the predicted increases in parallelism in computer architectures. In this paper, I argue that this is not a good idea. Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism. Threads, as a model of computation, are wildly nondeterministic, and the job of the programmer becomes one of pruning that nondeterminism. Although many research techniques improve the model by offering more effective pruning, I argue that this is approaching the problem backwards. Rather than pruning nondeterminism, we should build from essentially deterministic, composable components. Nondeterminism should be explicitly and judiciously introduced where needed, rather than removed where not needed. The consequences of this principle are profound. I argue for the development of concurrent coordination languages based on sound, composable formalisms. I believe that such languages will yield much more reliable, and more concurrent programs.

1 Introduction

It is widely acknowledged that concurrent programming is difficult. Yet the imperative for concurrent programming is becoming more urgent. Many technologists predict that the end of Moore's Law will be answered with increasingly parallel computer architectures (multicore or chip multprocessors, CMPs) [15]. If we hope to continue to get performance gains in computing, programs must be able to exploit this parallelism.

[1]This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF award No. CCR-0225610), the State of California Micro Program, and the following companies: Agilent, DGIST, General Motors, Hewlett Packard, Infineon, Microsoft, and Toyota.

One possible technical solution is automatic exploitation of parallelism in sequential programs, through either computer architecture techniques such as dynamic dispatch, or through automatic program parallelization of sequential programs [6]. However, many researchers agree that these automatic techniques have been pushed about as far as they will go, and that they are capable of exploiting only modest parallelism. A natural conclusion is that programs themselves must become more concurrent.

If we understand why concurrent programming is so difficult, we have a better chance of solving the problem. Sutter and Larus observe [47]

> "humans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code. Even careful people miss possible interleavings among even simple collections of partially ordered operations."

Yet humans are actually quite adept at reasoning about concurrent systems. The physical world is highly concurrent, and our very survival depends on our ability to reason about concurrent physical dynamics. The problem is that we have chosen concurrent abstractions that do not even vaguely resemble the concurrency of the physical world. We have become so used to these computational abstractions that we have lost track of the fact that they are not immutable. In this paper, I argue that the difficulty of concurrent programming is a consequence of the abstractions, and that if we are are willing to let go of those abstractions, then the problem will be fixable.

An optimistic view is given by Barrosso [7], who argues that technology forces will drive computing in both servers and desktops to CMPs, and that when the technology becomes mainstream, the programming problems will somehow be solved. But we should not underestimate the challenges. What is required is, in the words of Stein, "to replace the conventional metaphor *a sequence of steps* with the notion of a community of interacting entities" [46]. This paper makes that case.

2 Threads

In general-purpose software engineering practice, we have reached a point where one approach to concurrent programming dominates all others, namely, threads. Threads are sequential processes that share memory. They represent a key concurrency model supported by modern computers, programming languages, and operating systems. Many general-purpose parallel architectures in use today (such as symmetric multiprocessors, SMPs) are direct hardware realizations of the thread abstraction.

Some applications can very effectively use threads. So-called "embarrassingly parallel" applications (for example, applications that essentially spawn multiple independent processes such as build tools, like PVM gmake, or web servers). Because of the independence of these applications, programming is relatively easy, and the abstraction being used is more like processes than threads (where memory is not shared). Where such applications do share data, they do so through database abstractions, which manage concurrency through such mechanisms as transactions. However, client-side applications are not so simple. Quoting Sutter and Larus again [47]:

> "The world of client applications is not nearly as well structured and regular. A typical client application executes a relatively small computation on behalf of a single user, so concurrency is found by dividing a computation into finer pieces. These pieces, say the user interface and program's computation, interact and share data in a myriad of ways.

> Non-homogeneous code; fine-grain, complicated interactions; and pointer-based data structures make this type of program difficult to execute concurrently."

Of course, threads are not the only possibility for concurrent programming. In scientific computing, where performance requirements have long demanded concurrent programming, data parallel language extensions and message passing libraries (like PVM [23], MPI [39], and OpenMP[1]) dominate over threads for concurrent programming. In fact, computer architectures intended for scientific computing often differ significantly from so-called "general purpose" architectures. They commonly support vectors and streams in hardware, for example. However, even in this domain, concurrent programs remain tedious to write. C and FORTRAN dominate, despite a long history of much better data parallel languages.

In distributed computing, threads are often not a practical abstraction because creating the illusion of shared memory is often too costly. Even so, we have gone to considerable lengths to create distributed computing mechanisms that emulate multithreaded programming. CORBA and .NET, for example, are rooted in distributed object-oriented techniques, where software components interact with proxies that behave as if they were local objects with shared memory. Object-orientation's data abstraction limits the extent to which the illusion of shared memory needs to be preserved, so such techniques prove reasonably cost effective. They make distributed programming look much like multithreaded programming.

Embedded computing also exploits concurrency models other than threads. Programmable DSP architectures are often VLIW machines. Video signal processors often combine SIMD with VLIW and stream processing. Network processors provide explicit hardware support for streaming data. However, despite considerable innovative research, in practice, programming models for these domains remain primitive. Designers write low-level assembly code that exploits specific hardware features, and combine this assembly code with C code only where performance is not so critical.

An interesting property of many embedded applications is that reliability and predictability are far more important than expressiveness or performance. It is arguable that this should be true in general purpose computing, but that's a side argument. I will argue that achieving reliability and predictability using threads is essentially impossible for many applications.

3 Threads as Computation

In this section, I will examine threads from a fundamental perspective, without reference to particular thread libraries or languages, and show that as a model of computation, they are serious flawed. In the next section, I consider a number of proposed fixes.

Let $\mathbb{N} = \{0, 1, 2, \cdots\}$ represent the natural numbers. Let $B = \{0, 1\}$ be the set of binary digits. Let B^* be the set of all finite sequences of bits, and

$$B^\omega = (\mathbb{N} \to B)$$

be the set of all infinite sequences of bits (each of which is a function that maps $\mathbb{N}$ into B). Following [17], let $B^{**} = B^* \cup B^\omega$. We will use B^{**} to represent the state of a computing machine, its (potentially infinite) inputs, and its (potentially infinite) outputs. Let

$$Q = (B^{**} \rightharpoonup B^{**})$$

[1] See http://www.openmp.org

denote the set of all partial functions with domain and codomain B^{**}.[2]

An *imperative machine* (A, c) is a finite set $A \subset Q$ of *atomic actions* and a *control function* $c\colon B^{**} \to \mathbb{N}$. The set A represents the atomic actions (typically instructions) of the machine and the function c represents how instructions are sequenced. We assume that A contains one *halt* instruction $h \in A$ with the property that

$$\forall\, b \in B^{**}, \quad h(b) = b.$$

That is, the halt instruction leaves the state unchanged.

A *sequential program* of length $m \in \mathbb{N}$ is a function

$$p\colon \mathbb{N} \to A$$

where

$$\forall\, n \geq m, \quad p(n) = h.$$

That is, a sequential program is a finite sequence of instructions tailed by an infinite sequence of halt instructions. Note that the set of all sequential programs, which we denote P, is a countably infinite set.

An execution of this program is a *thread*. It begins with an initial $b_0 \in B^{**}$, which represents the initial state of the machine and the (potentially infinite) input, and for all $n \in \mathbb{N}$,

$$b_{n+1} = p(c(b_n))(b_n). \tag{1}$$

Here, $c(b_n)$ provides the index into the program p for the next instruction $p(c(b_n))$. That instruction is applied to the state b_n to get the next state b_{n+1}. If for any $n \in \mathbb{N} \quad c(b_n) \geq m$, then $p(c(b_n)) = h$ and the program halts in state b_n (that is, the state henceforth never changes). If for all initial states $b_0 \in B$ a program p halts, then p defines a total function in Q. If a program p halts for some $b_0 \in B$, then it defines a partial function in Q.[3]

We now get to the core appeal that sequential programs have. Given a program and an initial state, the sequence given by (1) is defined. If the sequence halts, then the function computed by the program is defined. Any two programs p and p' can be compared. They are equivalent if they compute the same partial function. That is, they are equivalent if they halt for the same initial states, and for such initial states, their final state is the same.[4] Such a theory of equivalence is essential for any useful formalism.

These essential and appealing properties of programs are lost when multiple threads are composed. Consider two programs p_1 and p_2 that execute concurrently in a multithreaded fashion. What we mean by this is that (1) is replaced by

$$b_{n+1} = p_i(c(b_n))(b_n) \quad i \in \{1, 2\}. \tag{2}$$

[2]Partial functions are functions that may or may not be defined on each element of their domain.

[3]Note that a classic result in computing is now evident. It is easy to show that Q is not a countable set. (Even the subset of Q of constant functions is not countable, since B^{**} itself is not countable. This can be easily demonstrated using Cantor's diagonal argument.) Since the set of all finite programs P is countable, we can conclude that not all functions in Q can be given by finite programs. That is, any sequential machine has limited expressiveness. Turing and Church [48] demonstrated that many choices of sequential machines (A, c) result in programs P that can give exactly the same subset of Q. This subset is called the *effectively computable functions*.

[4]In this classical theory, programs that do not halt are all equivalent. This creates serious problems when applying the theory of computation to embedded software, where useful programs do not halt [34].

At each step n, either program may provide the next (atomic) action. Consider now whether we have a useful theory of equivalence. That is, given a pair of multithreaded programs (p_1, p_2) and another pair (p'_1, p'_2), when are these two pairs equivalent? A reasonable extension of the basic theory defines them to be equivalent if *all interleavings* halt for the same initial state and yield the same final state. The enormous number of possible interleavings makes it extremely difficult to reason about such equivalence except in trivial cases (where, for example, the state B^{**} is partitioned so that the two programs are unaffected by each others' partition).

Even worse, given two programs p and p' that are equivalent when executed according to (1), if they are executed in a multithreaded environment, we can no longer conclude that they are equivalent. In fact, we have to know about all other threads that might execute (something that may not itself be well defined), and we would have to analyze all possible interleavings. We conclude that with threads, there is no useful theory of equivalence.

Still worse, implementing a multithreaded model of computation is extremely difficult. Witness, for example, the deep subtleties with the Java memory model (see for example [41] and [24]), where even astonishingly trivial programs produce considerable debate about their possible behaviors.

The core abstraction of computation given by (1), on which all widely-used programming languages are built, emphasizes deterministic composition of deterministic components. The actions are deterministic and their sequential composition is deterministic. Sequential execution is, semantically, function composition, a neat, simple model where deterministic components compose into deterministic results.

Threads, on the other hand, are wildly nondeterministic. The job of the programmer is to prune away that nondeterminism. We have, of course, developed tools to assist in the pruning. Semaphores, monitors, and more modern overlays on threads (discussed in the following section) offer the programmer ever more effective pruning. But pruning a wild mass of brambles rarely yields a satisfactory hedge.

To offer another analogy, suppose that we were to ask a mechanical engineer to design an internal combustion engine by starting with a pot of iron, hydrocarbon, and oxygen molecules, moving randomly according to thermal forces. The engineer's job is to constrain these motions until the result is an internal combustion engine. Thermodynamics and chemistry tells us that this is a theoretically valid way to think about the design problem. But is it practical?

To offer a third analogy, a folk definition of insanity is to do the same thing over and over again and to expect the results to be different. By this definition, we in fact require that programmers of multithreaded systems be insane. Were they sane, they could not understand their programs.

I will argue that we must (and can) build concurrent models of computation that are far more deterministic, and that we must judiciously and carefully introduce nondeterminism where needed. Nondeterminism should be explicitly added to programs, and only where needed, as it is in sequential programming. Threads take the opposite approach. They make programs absurdly nondeterministic, and rely on programming style to constrain that nondeterminism to achieve deterministic aims.

4 How Bad is it In Practice?

We have argued that threads provide a hopelessly unusable extension of the core abstractions of computation. Yet in practice, many programmers today write multi-threaded programs that work.

```
public class ValueHolder {
    private List listeners = new LinkedList();
    private int value;
    public interface Listener {
        public void valueChanged(int newValue);
    }

    public void addListener(Listener listener) {
        listeners.add(listener);
    }

    public void setValue(int newValue) {
        value = newValue;
        Iterator i = listeners.iterator();
        while(i.hasNext()) {
            ((Listener)i.next()).valueChanged(newValue);
        }
    }
}
```

Figure 1: A Java implementation of the observer pattern, valid for one thread.

Is there a contradiction here? In practice, programmers are provided with tools that prune away much of the nondeterminism. Object-oriented programming, for example, limits the visibility that certain portions of a program have into portions of the state. This effectively partitions the state space B^{**} into disjoint sections. Where programs do operate on shared portions of this state space, semaphores, mutual-exclusion locks, and monitors (objects with mutually-exclusive methods) provide mechanisms that programs can use to prune away more of the nondeterminism. But in practice, these techniques yield understandable programs only for very simple interactions.

Consider a commonly used design pattern known as the observer pattern [22]. This is a very simple and widely used design pattern. A Java implementation that is valid for a single thread is shown in figure 1.[5] This shows two methods from a class where an invocation of the `setValue()` method triggers notification of the new value by calling the `valueChanged()` method of any objects that have been registered by a call to `addListener()`.

The code in figure 1 is not thread safe, however. That is, if multiple threads can call `setValue()` or `addListener()`, then it could occur that the `listeners` list gets modified while the iterator is iterating through the list. This will trigger an exception that will likely terminate the program.

The simplest solution is to add the Java keyword `synchronized` to each of the `setValue()` and `addListener()` method definitions. The `synchronized` keyword in Java implements mutual exclusion, turning instances of this `ValueHolder` class into monitors, preventing any two threads from being in these methods simultaneously. When a synchronized method is called, the calling thread attempts to acquire an exclusive lock on the object. If any other thread holds that lock, then the calling thread stalls until the lock is released.

However, this solution is not wise because it can lead to deadlock. In particular, suppose we have an instance a of `ValueHolder` and an instance b of another class that implements the `Listener` interface. That other class can do anything in its `valueChanged()` method, including acquiring

[5] Thanks to Mark S. Miller for suggesting this example.

```
public class ValueHolder {
    private List listeners = new LinkedList();
    private int value;
    public interface Listener {
        public void valueChanged(int newValue);
    }

    public synchronized void addListener(Listener listener) {
        listeners.add(listener);
    }

    public void setValue(int newValue) {
        List copyOfListeners;
        synchronized(this) {
            value = newValue;
            copyOfListeners = new LinkedList(listeners);
        }
        Iterator i = copyOfListeners.iterator();
        while(i.hasNext()) {
            ((Listener)i.next()).valueChanged(newValue);
        }
    }
}
```

Figure 2: A Java implementation of the observer pattern that attempts to be thread safe.

a lock on another monitor. If it stalls in acquiring that lock, it will continue to hold the lock on this `ValueHolder` object while it is stalled. Meanwhile, whatever thread holds the lock it is trying to acquire might call `addListener()` on a. Both threads are now blocked with no hope of becoming unblocked. This sort of potential deadlock lurks in many programs that use monitors.

Already, this rather simple design pattern is proving difficult to implement correctly. Consider the improved implementation shown in figure 2. While holding a lock, the `setValue()` method makes copy of the listeners list. Since the `addListeners()` method is synchronized, this avoid the concurrent modification exception that might occur with the code in figure 1. Further, it calls `valueChanged()` outside of synchronized block to avoid deadlock.

It is tempting to think we have solved the problem, but in fact, this code is still not correct. Suppose two threads call setValue(). One of them will set the value last, leaving that value in the object. But listeners may be notified value changes in the opposite order. The listeners will conclude that the final value of the `ValueHolder` object is the wrong value!

Of course, this pattern can be made to work robustly in Java (I leave this as an exercise for the reader). My point is that even this very simple and commonly used design pattern has required some rather intricate thinking about possible interleavings. I speculate that most multithreaded programs have such bugs. I speculate further that the bugs have not proved to be major handicaps only because today's architectures and operating systems deliver modest parallelism. The cost of context switching is high, so only a tiny percentage of the possible interleavings of thread instructions ever occur in practice. I conjecture that most multi-threaded general-purpose applications are, in fact, so full of concurrency bugs that as multi-core architectures become commonplace, these bugs will begin to show up as system failures. This scenario is bleak for computer vendors: their next

generation of machines will become widely known as the ones on which many programs crash.

These same computer vendors are advocating more multi-threaded programming, so that there is concurrency that can exploit the parallelism they would like to sell us. Intel, for example, has embarked on an active campaign to get leading computer science academic programs to put more emphasis on multi-threaded programming. If they are successful, and the next generation of programmers makes more intensive use of multithreading, then the next generation of computers will become nearly unusable.

5 Fixing Threads by More Aggressive Pruning

In this section, I discuss a number of approaches to solving this problem that share a common feature. Specifically, they preserve the essential thread model of computation for programmers, but provide them with more aggressive mechanisms for pruning the enormously nondeterministic behavior.

The first technique is better software engineering processes. These are, in fact, essential to get reliable multithreaded programs. However, they are not sufficient. An anecdote from the Ptolemy Project[6] is telling (and alarming). In the early part of the year 2000, my group began developing the kernel of Ptolemy II [20], a modeling environment supporting concurrent models of computation. An early objective was to permit modification of concurrent programs via a graphical user interface while those concurrent programs were executing. The challenge was to ensure that no thread could ever see an inconsistent view of the program structure. The strategy was to use Java threads with monitors.

A part of the Ptolemy Project experiment was to see whether effective software engineering practices could be developed for an academic research setting. We developed a process that included a code maturity rating system (with four levels, red, yellow, green, and blue), design reviews, code reviews, nightly builds, regression tests, and automated code coverage metrics [43]. The portion of the kernel that ensured a consistent view of the program structure was written in early 2000, design reviewed to yellow, and code reviewed to green. The reviewers included concurrency experts, not just inexperienced graduate students (Christopher Hylands (now Brooks), Bart Kienhuis, John Reekie, and myself were all reviewers). We wrote regression tests that achieved 100 percent code coverage. The nightly build and regression tests ran on a two processor SMP machine, which exhibited different thread behavior than the development machines, which all had a single processor. The Ptolemy II system itself began to be widely used, and every use of the system exercised this code. No problems were observed until the code deadlocked on April 26, 2004, four years later.

It is certainly true that our relatively rigorous software engineering practice identified and fixed many concurrency bugs. But the fact that a problem as serious as a deadlock that locked up the system could go undetected for four years despite this practice is alarming. How many more such problems remain? How long do we need test before we can be sure to have discovered all such problems? Regrettably, I have to conclude that testing may never reveal all the problems in nontrivial multithreaded code.

Of course, there are tantalizingly simple rules for avoiding deadlock. For example, always acquire locks in the same order [32]. However, this rule is very difficult to apply in practice because no method signature in any widely used programming language indicates what locks the method

[6] See http://ptolemy.eecs.berkeley.edu

acquires. You need to examine the source code of all methods that you call, and all methods that those methods call, in order to confidently invoke a method. Even if we fix this language problem by making locks part of the method signature, this rule makes it extremely difficult to implement symmetric accesses (where interactions can originate from either end). And no such fix gets around the problem that reasoning about mutual exclusion locks is extremely difficult. If programmers cannot understand their code, then the code will not be reliable.

One might conclude that the problem is in the way Java realizes threads. Perhaps the synchronized keyword is not the best pruning tool. Indeed, version 5.0 of Java, introduced in 2005, added a number of other mechanisms for synchronizing threads. These do in fact enrich the toolkit for the programmer to prune nondeterminacy. But the mechanisms (such as semaphores) still require considerable sophistication to use, and very likely will still result in incomprehensible programs with subtle lurking bugs.

Software engineering process improvements alone will not do the job. Another approach that can help is the use of vetted design patterns for concurrent computation, as in [32] and [44]. Indeed, these are an enormous help when the programmer's task identifiably matches one of the patterns. However, there are two difficulties. One is that implementation of the patterns, even with careful instructions, is still subtle and tricky. Programmers will make errors, and there are no scalable techniques for automatically checking compliance of implementations to patterns. More importantly, the patterns can be difficult to combine. Their properties are not typically composable, and hence nontrivial programs that require use of more than one pattern are unlikely to be understandable.

A very common use of patterns in concurrent computation is found in databases, particularly with the notion of transactions. Transactions support speculative unsynchronized computation on a copy of the data followed by a *commit* or *abort*. A commit occurs when it can be shown that no conflicts have occurred. Transactions can be supported on distributed hardware (as is common for databases), or in software on shared-memory machines [45], or, most interestingly, in hardware on shared-memory machines [38]. In the latter case, the technique meshes well with cache consistency protocols that are required anyway on these machines. Transactions eliminate unintended deadlocks, but despite recent extensions for composability [26], remain a highly nondeterministic interaction mechanism. They are well-suited to intrinsically nondeterminate situations, where for example multiple actors compete nondeterministically for resources. But they are not well-suited for building determinate concurrent interactions.

A particularly interesting use of patterns is MapReduce, as reported by Dean and Ghemawat [19]. This pattern has been used for large scale distributed processing of huge data sets by Google. Whereas most patterns provide fine-grain shared data structures with synchronization, MapReduce provides a framework for the construction of large distributed programs. The pattern is inspired by the higher-order functions found in Lisp and other functional languages. The parameters to the pattern are pieces of functionality represented as code rather than pieces of data.

Patterns may be encapsulated into libraries by experts, as has been done with MapReduce at Google, the concurrent data structures in Java 5.0, and STAPL in C++ [1]. This greatly improves the reliability of implementations, but requires some programmer discipline to constrain all concurrent interactions to occur via these libraries. Folding the capabilities of these libraries into languages where syntax and semantics enforce these constraints may eventually lead to much more easily constructed concurrent programs.

Higher-order patterns such as MapReduce offer some particularly interesting challenges and opportunities for language designers. These patterns function at the level of *coordination languages*

rather than more traditional *programming languages*. New coordination languages that are compatible with established programming languages (such as Java and C++) are much more likely to gain acceptance than new programming languages that replace the established languages.

A common compromise is to extend established programming languages with annotations or a few selected keywords to support concurrent computation. This compromise admits the re-use of signifcant portions of legacy code when concurrency is not an issue, but requires rewriting to expose concurrency. This strategy is followed for example by Split-C [16] and Cilk [13], which are both C-like languages supporting multithreading. In Cilk, a programmer can insert keywords "cilk" "spawn" and "sync" into what are essentially C programs. The objective of Cilk is to support dynamic multithreading in a shared-memory context, where overhead is incurred only when parallelism is actually exploited (that is, on one processor, the overhead is low).

A related approach combines language extensions with constraints that limit expressiveness of established languages in order to get more consistent and predictable behavior. For example, the Guava language [5] constrains Java so that unsynchronized objects cannot be accessed from multiple threads. It further makes explicit the distinction between locks that ensure the integrity of read data (read locks) and locks that enable safe modification of the data (write locks). These language changes prune away considerable nondeterminacy without sacrificing much performance, but they still have deadlock risk.

Another approach that puts more emphasis on the avoidance of deadlock is *promises*, as realized for example by Mark Miller in the E programming language[7]. These are also called futures, and are originally attributable to Baker and Hewitt [27]. Here, instead of blocking to access shared data, programs proceed with a proxy of the data that they expect to eventually get, using the proxy as if it were the data itself.

Yet another approach leaves the programming languages and the mechanisms for expressing concurrency unchanged, and instead introduces formal program analysis to identify potential concurrency bugs in multithreaded programs. This is done for example in Blast [28] and the Intel thread checker[8]. This approach can help considerably by revealing program behaviors that are difficult for a human to spot. Less formal techniques, such as performance debuggers like Valgrind[9], can also help in a similar way, making it easier for programmers to sort through the vast nondeterminacy of program behaviors. Although promising, both the formal and informal techniques still require considerable expertise to apply and suffer from scalability limitations.

All of the above techniques prune away some of the nondeterminacy of threads. However, they all still result in highly nondeterministic programs. For applications with intrinsic nondeterminacy, such as servers, concurrent database accesses, or competition for resources, this is appropriate. But achieving deterministic aims through nondeterministic means remains difficult. To achieve deterministic concurrent computation requires approaching the problem differently. Instead of starting with a highly nondeterministic mechanism like threads, and relying on the programmer to prune that nondeterminacy, we should start with deterministic, composable mechanisms, and introduce nondeterminism only where needed. We explore this approach next.

[7] See http://www.erights.org/

[8] See http://developer.intel.com/software/products/threading/tcwin

[9] See http://valgrind.org/

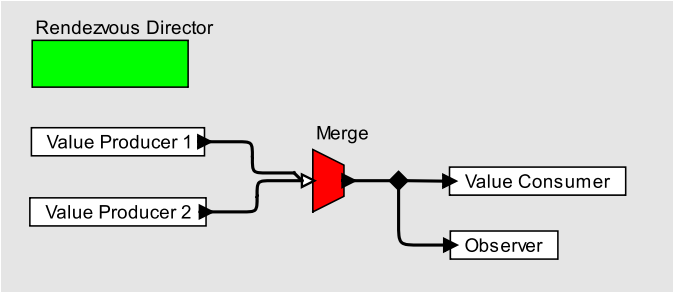

Figure 3: Observer pattern implemented in a rendezvous-based coordination language with a visual syntax.

6 Alternatives to Threads

Consider again the observer pattern shown in figures 1 and 2. This is a simple (trivial, even) design pattern [22]. An implementation should be simple. Regrettably, as shown above, it is not so easy to implement using threads.

Consider figure 3, which shows the observer pattern implemented in the "Rendezvous domain" of Ptolemy II [20]. The box at the upper left labeled "Rendezvous Director" is an annotation specifying that this diagram represents a CSP-like [29] concurrent program, where each component (represented by an icon) is a process, and communication is by rendezvous. The processes themselves are specified using ordinary Java code, so this framework is properly viewed as a coordination language (that happens to have a visual syntax). The Ptolemy II implementation of this rendezvous domain is inspired by Reo [2] and includes a "Merge" block that specifies a conditional rendezvous. In the diagram, the Merge block specifies that either of the two Value Producers can rendezvous with both the Value Consumer and the Observer. That is, there two possible three-way rendezvous interactions that can occur. These interactions can occur repeatedly in nondeterministic order.

First, note that once you understand what the icons mean, the diagram very clearly expresses the observer pattern. Second, notice that everything about the program is deterministic except the explicitly nondeterministic interaction specified by the Merge block. Were that block absent, then the program would specify deterministic interactions between deterministic processes. Third, note that deadlock is provably absent (in this case, the lack of cycles in the diagram ensures no deadlock). Fourth, note that the multiway rendezvous ensures that the Value Consumer and Observer see new values in the same order. The observer pattern becomes trivial, as it should be.

Now that we have made the trivial programming problem trivial, we can start to consider interesting elaborations. Figure 4 shows the observer pattern implemented with a "PN Director," which realizes the Kahn process networks (PN) model of concurrency [31]. In this model, each icon again represents a process, but instead of rendezvous-based interactions, the processes communicate via message passing with conceptually unbounded FIFO queues and blocking reads (streams). In the original PN model due to Kahn and MacQueen [31], the blocking reads ensure that every network defines a deterministic computation. In this case, the PN model has been augmented with a primitive called "NondeterministicMerge" that explicitly merges streams nondeterministically. Note that augmenting the PN model with such explicit nondeterminism is common for embedded software applications [18].

The new implementation of figure 4 has all the cited advantages of the one in figure 3 with the

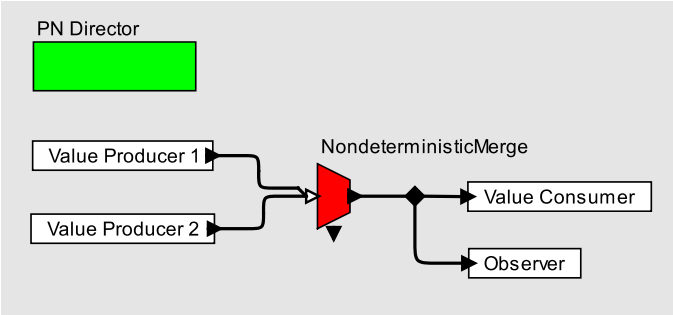

Figure 4: Observer pattern implemented using a process networks model of concurrency.

added property that the Observer need not keep up with the Value Consumer. Notifications can be queued for later processing. In a thread-based implementation, we are unlikely to ever even get to point of asking such questions, since the programmer effort to get any form of the observer pattern correct is so excessive.

A third implementation would elaborate on the nature of the nondeterminism represented by the nondeterministic merge. The principles of synchronous languages [8] could be used to ensure fairness. In Ptolemy II, the same model can be implemented with an "SR director" (synchronous/reactive), which implements a synchronous model related to Esterel [12], SIGNAL [10], and Lustre [25]. The last of these has been used successfully to design highly concurrent, safety critical software for aircraft control applications [11]. Using threads for such applications would not be wise.

A fourth implementation would focus on the timing of nondeterministic events. In Ptolemy II, a similar model using the "DE director" (discrete events), would provide a timed specification with rigorous semantics [33] related to that of hardware description languages such as VHDL and Verilog and to network modeling frameworks such as Opnet Modeler.

In all four cases (rendezvous, PN, SR, and DE), we have started with a fundamentally deterministic (and concurrent) interaction mechanism (deterministic in the sense of the computation being performed, though in the first three cases not deterministic in the sense of timing). Nondeterminism has been judiciously introduced exactly where needed. This style of design is very different from the threads style, which starts with a wildly nondeterministic interaction mechanism and attempts to prune away undesired nondeterminism.

The implementations shown in figures 3 and 4 both use Java threads. However, the programmer's model is not one of threads at all. Compared to all the techniques described in the previous section, this is closest to MapReduce, which has a similar flavor of streaming data through computations. But unlike MapReduce, it is supported in a rigorous coordination language that is sufficiently expressive to describe a wide range of interactions. In fact, two distinct coordination languages are given here, one with rendezvous semantics, and the other with PN semantics.

This style of concurrency is, of course, not new. Component architectures where data flows through components (rather than control) have been called "actor-oriented" [35]. These can take many forms. Unix pipes resemble PN, although they are more limited in that they do not support cyclic graphs. Message passing packages like MPI and OpenMP include facilities for implementing rendezvous and PN, but in a less structured context that emphasizes expressiveness rather than determinacy. A naive user of such packages can easily be bitten by unexpected nondeterminacy.

Languages such as Erlang [4] make message passing concurrency an integral part of a general-purpose language. Languages such as Ada make rendezvous an integral part. Functional languages [30] and single-assignment languages also emphasize deterministic computations, but they are less explicitly concurrent, so controlling and exploiting concurrency can be more challenging. Data parallel languages also emphasize determinate interactions, but they require low-level rewrites of software.

All of these approaches offer pieces of the solution. But it seems unlikely that any one will become mainstream. I examine the reasons for this next.

7 Challenges and Opportunities

The fact is that alternatives to threads have been around for a long time, and yet threads dominate the concurrent programming language landscape. There are, in fact, many obstacles to these alternatives taking root. Probably the most important is that the very notion of programming, and the core abstractions of computation, are deeply rooted in the sequential paradigm. The most widely used programming languages (by far) adhere to this paradigm. Syntactically, threads are either a minor extension to these languages (as in Java) or just an external library. Semantically, of course, they thoroughly disrupt the essential determinism of the languages. Regrettably, programmers seem to be more guided by syntax than semantics. The alternatives to threads that have taken root, like MPI and OpenMP, share this same key feature. They make no syntactic change to languages. Alternatives that replace these languages with entirely new syntax, such as Erlang or Ada, have not taken root, and probably will not. Even languages with minor syntactic modifications to established languages, like Split-C or Cilk, remain esoteric.

The message is clear. We should not replace established languages. We should instead build on them. However, building on them using only libraries is not satisfactory. Libraries offer little structure, no enforcement of patterns, and few composable properties.

I believe that the right answer is coordination languages. Coordination languages do introduce new syntax, but that syntax serves purposes that are orthogonal to those of established programming languages. Whereas a general-purpose concurrent language like Erlang or Ada has to include syntax for mundane operations such as arithmetic expressions, a coordination language need not specify anything more than coordination. Given this, the syntax can be noticeably distinct. The "programs" given in figures 3 and 4 use a visual syntax to specify actor-oriented coordination (where data streams through components rather than control). Although here a visual syntax is being used only for pedagogical purposes, it is conceivable that eventually such visual syntaxes will be made scalable and effective, as certain parts of UML have for object-oriented programming. Even if they are not, it is easy to envision scalable textual syntaxes that specify the same structure.

Of course, even coordination languages have been around for a long time [40]. They too have failed to take root. One reason for this that their acceptance amounts to capitulation on one key front: homogeneity. A prevailing undercurrent in programming languages research is that any worthy programming language must be general purpose. It must be, at a minimum, sufficiently expressive to express its own compiler. And then, adherents to the language are viewed as traitors if they succumb to the use of another language. Language wars are religious wars, and few of these religions are polytheistic.

A key development, however, has broken the ice. UML, which is properly viewed as a family of languages, each with a visual syntax, is routinely combined with C++ and Java. Programmers are

starting to get used to using more than one language, where complementary features are provided by the disjoint languages. The programs in figures 3 and 4 follow this spirit, in that they specify diagrammatically large-grain structure, which is quite orthogonal to fine-grain computation.

Concurrency models with stronger determinism than threads, such as Kahn process networks, CSP, and dataflow, have also been available for a long time. Some of these have led to programming languages, such as Occam [21] (based on CSP), and some have led to domain-specific frameworks, such as YAPI [18]. Most, however, have principally been used to build elaborate process calculi, and have not had much effect on mainstream programming. I believe this can change if these concurrency models are used to define coordination languages rather than replacement languages.

There are many challenges on this path. Designing good coordination languages is no easier than designing good general-purpose languages, and is full of pitfalls. For example, it is easy to be trapped by the false elegance of a spare set of primitives. In general purpose languages, it is well known that seven primitives are sufficient [48], but no serious programming language is built on these primitives.

Figure 5 shows two implementations of a simple concurrent computation. In the upper program, an adaptation of an example from [3], successive outputs from Data Source 1 and Data Source 2 are interleaved deterministically, appearing in alternating order at the Display block. This rather simple functionality, however, is accomplished in a rather complex way. In fact, it is a bit of puzzle to understand how it works.[10] The lower program, however, is easily understood. The Commutator block performs a rendezvous with each of its inputs in top-to-bottom order, and thus accomplishes the same interleaving. Judicious choice of "language" primitives enable simple, direct, and deterministic expressions of deterministic aims. The upper model uses nondeterministic mechanisms, albeit more expressive ones, to accomplish deterministic aims. It is thus much more obtuse.

Of course, coordination languages need to develop scalability and modularity features analogous to those in established languages. This can be done. Ptolemy II, for example, provides a sophisticated, modern type system at the coordination language level [49]. Moreover, it offers a preliminary form of inheritance and polymorphism that is adapted from object-oriented techniques [35]. A huge opportunity exists in adapting the concept of higher-order functions to coordination languages to offer constructs like MapReduce at the coordination language level. Some very promising early work in this direction is given by Reekie [42].

A more challenging, long-term opportunity is to adapt the theory of computation to provide better foundations for concurrent computations. Although considerable progress has been made in this direction, I believe there is much more to be done. In section 3, sequential computation is modeled as functions mapping bit sequences into bit sequences. A corresponding concurrent model has been explored in [36], where instead of a function

$$f\colon B^{**} \to B^{**}$$

concurrent computation is given as a function

$$f\colon (T \to B^{**}) \to (T \to B^{**}).$$

[10]The Barrier actor ensures that all its inputs participate in the same rendezvous interaction. The Merge actor was explained before. The Buffer actor, which has capacity one, is initially empty, and thus willing to rendezvous with its input. Once it becomes full, it is only willing to rendezvous with its output. All together, this model allows either of two multiway rendezvous interactions to occur, in alternating order. The first involves both Data Source actors, the Buffer, and the Display. The second involves only the Buffer and the Display.

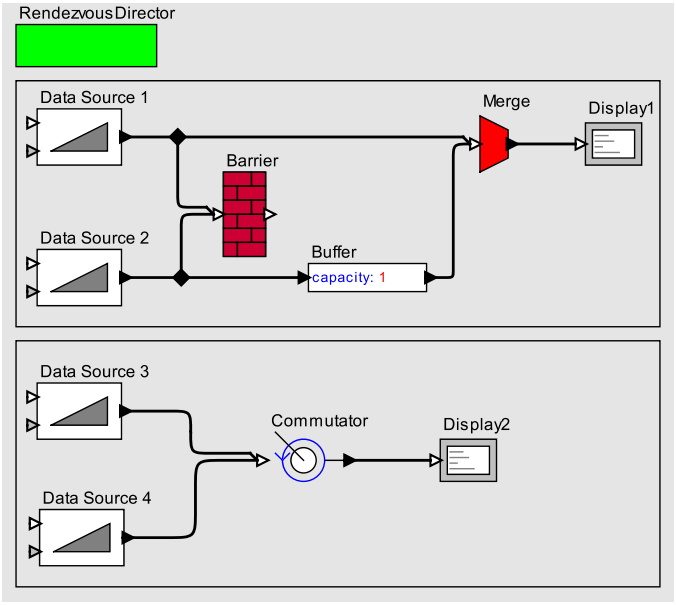

Figure 5: Two ways to accomplish deterministic interleaving using rendezvous.

In the above, T is a partially or totally ordered set of *tags*, where the ordering can represent time, causality, or more abstract dependency relations. A computation viewed in this way maps an evolving bit pattern into an evolving bit pattern. This basic formulation has been shown adaptable to many concurrent models of computation [9, 14, 37].

8 Conclusion

Concurrency in software is difficult. However, much of this difficulty is a consequence of the abstractions for concurrency that we have chosen to use. The dominant one in use today for general-purpose computing is threads. But non-trivial multi-threaded programs are *incomprehensible to humans*. It is true that the programming model can be improved through the use of design patterns, better granularity of atomicity (e.g. transactions), improved languages, and formal methods. However, these techniques merely chip away at the unnecessarily enormous nondeterminism of the threading model. The model remains intrinsically intractable.

If we expect concurrent programming to be mainstream, and if we demand reliability and predictability from programs, then we must discard threads as a programming model. Concurrent programming models can be constructed that are much more predictable and understandable than threads. They are based on a very simple principle: *deterministic ends should be accomplished with deterministic means*. Nondeterminism should be judiciously and carefully introduced where needed, and should be explicit in programs. This principle seems obvious, yet it is not accomplished by threads. Threads must be relegated to the engine room of computing, to be suffered only by expert technology providers.

References

[1] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: An adaptive, generic parallel C++ library. In *Wkshp. on Lang. and Comp. for Par. Comp. (LCPC)*, pages 193–208, Cumberland Falls, Kentucky, 2001.

[2] F. Arbab. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[3] F. Arbab. A behavioral model for composition of software components. *L'Object*, to appear, 2006.

[4] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams. *Concurrent programming in Erlang*. Prentice Hall, second edition, 1996.

[5] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: a dialect of Java without data races. In *ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 35 of *ACM SIGPLAN Notices*, pages 382–400, 2000.

[6] U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, 1993.

[7] L. A. Barroso. The price of performance. *ACM Queue*, 3(7), 2005.

[8] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[9] A. Benveniste, L. Carloni, P. Caspi, and A. Sangiovanni-Vincentelli. Heterogeneous reactive systems modeling and correct-by-construction deployment. In *EMSOFT*. Springer, 2003.

[10] A. Benveniste and P. L. Guernic. Hybrid dynamical systems theory and the signal language. *IEEE Tr. on Automatic Control*, 35(5):525–546, 1990.

[11] G. Berry. The effectiveness of synchronous languages for the development of safety-critical systems. White paper, Esterel Technologies, 2003.

[12] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[13] 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 *ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*, ACM SIGPLAN Notices, 1995.

[14] J. R. Burch, R. Passerone, and A. L. Sangiovanni-Vincentelli. Notes on agent algebras. Technical Report UCB/ERL M03/38, University of California, November 2003.

[15] M. Creeger. Multicore CPUs for the masses. *ACM Queue*, 3(7), 2005.

[16] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. v. Eicken, and K. Yelick. Parallel programming in Split-C. In *Supercomputing*, Portland, OR, 1993.

[17] B. A. Davey and H. A. Priestly. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[18] E. A. de Kock, G. Essink, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. Kruijtzer, P. Lieverse, and K. A. Vissers. Yapi: Application modeling for signal processing systems. In *37th Design Automation Conference (DAC'00)*, pages 402–405, Los Angeles, CA, 2000.

[19] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Sixth Symposium on Operating System Design and Implementation (OSDI)*, San Francisco, CA, 2004.

[20] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(2), 2003.

[21] J. Galletly. *Occam-2*. University College London Press, 2nd edition, 1996.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[23] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine — A Users Guide and Tutorial for Network Parallel Computing*. MIT Press, Cambridge, MA, 1994.

[24] A. Gontmakher and A. Schuster. Java consistency: nonoperational characterizations for Java memory behavior. *ACM Trans. Comput. Syst.*, 18(4):333–386, 2000.

[25] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1319, 1991.

[26] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *ACM Conference on Principles and Practice of Parallel Programming (PPoPP)*, Chicago, IL, 2005.
[27] J. Henry G. Baker and C. Hewitt. The incremental garbage collection of processes. In *Proceedings of the Symposium on AI and Programming Languages*, volume 12 of *ACM SIGPLAN Notices*, pages 55–59, 1977.
[28] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *15th International Conference on Computer-Aided Verification (CAV)*, volume 2725 of *Lecture Notes in Computer Science*, pages 262–274. Springer-Verlag, 2003.
[29] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), 1978.
[30] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3), 1989.
[31] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing*. North-Holland Publishing Co., 1977.
[32] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, Reading MA, 1997.
[33] E. A. Lee. Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, 7:25–45, 1999.
[34] E. A. Lee. What's ahead for embedded software? *IEEE Computer Magazine*, pages 18–26, 2000.
[35] E. A. Lee and S. Neuendorffer. Classes and subclasses in actor-oriented design. In *Conference on Formal Methods and Models for Codesign (MEMOCODE)*, San Diego, CA, USA, 2004.
[36] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on CAD*, 17(12), 1998.
[37] X. Liu. Semantic foundation of the tagged signal model. Phd thesis, EECS Department, University of California, December 20 2005.
[38] H. Maurice and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, San Diego, California, United States, 1993. ACM Press.
[39] Message Passing Interface Forum. MPI2: A message passing interface standard. *International Journal of High Performance Computing Applications*, 12(1-2):1–299, 1998.
[40] G. Papadopoulos and F. Arbab. Coordination models and languages. In M. Zelkowitz, editor, *Advances in Computers - The Engineering of Large Systems*, volume 46, pages 329–400. Academic Press, 1998.
[41] W. Pugh. Fixing the Java memory model. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 89–98, San Francisco, California, United States, 1999. ACM Press.
[42] H. J. Reekie. Toward effective programming for parallel digital signal processing. Ph.D. Thesis Research Report 92.1, University of Technology, Sydney, 1992.
[43] H. J. Reekie, S. Neuendorffer, C. Hylands, and E. A. Lee. Software practice in the Ptolemy project. Technical Report Series GSRC-TR-1999-01, Gigascale Semiconductor Research Center, University of California, Berkeley, April 1999.
[44] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture - Patterns for Concurrent and Networked Objects*. Wiley, 2000.
[45] N. Shavit and D. Touitou. Software transactional memory. In *ACM symposium on Principles of Distributed Computing*, pages 204–213, Ottowa, Ontario, Canada, 1995. ACM Press.
[46] L. A. Stein. Challenging the computational metaphor: Implications for how we think. *Cybernetics and Systems*, 30(6), 1999.
[47] H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7), 2005.
[48] A. M. Turing. Computability and λ-definability. *Journal of Symbolic Logic*, 2:153–163, 1937.
[49] Y. Xiong. An extensible type system for component-based design. Ph.D. Thesis Technical Memorandum UCB/ERL M02/13, University of California, Berkeley, CA 94720, May 1 2002.