Determinism Should Ensure Deadlock-Freedom

Nalini Vasudevan
Columbia University, New York

Stephen A. Edwards
Columbia University, New York

Abstract

The advent of multicore processors has made concurrent programming models mandatory. However, most concurrent programming models come with two major pitfalls: non-determinism and deadlocks. By determinism, we mean the output behavior of the program is independent of the scheduling choices (e.g., the operating system) and depends only on the input behavior. A few concurrent models provide deterministic behavior by providing constructs that impose additional synchronization, but improper or out-of-order use of these constructs leads to problems like deadlocks.

In this paper, we argue for both determinism and deadlock-freedom. We propose a design of a deterministic, deadlock-free model. Any program that uses this model is guaranteed to produce the same output for a given input. Additionally, the program will never deadlock: the program will either terminate or run forever.

1 Introduction

Non-deterministic functional behavior arising from timing variability is one of the biggest problems of concurrent programming. It makes debugging nearly impossible because unwanted behavior is rarely reproducible. Re-running a non-deterministic program on the same input can produce very different behavior.

We agree with Bocchino et al. [8] that the programming environment should ensure input-output determinism. Most concurrent programming languages are not deterministic, so one program in a language may be deterministic but another program written in the same language may not be. Thus, programmers have to check for determinism on a program-by-program basis. This is not really practical because determinism checking tools either do not scale or cause too much run-time overhead.

A solution is to build a deterministic concurrent language that forces every program written in it to be deterministic. These languages provide determinism by providing additional synchronization. Although the programmer looses some flexibility in programming, his or her program is guaranteed to behave consistently.

While deterministic concurrent models are interesting, they give rise to a number of problems. For example, a deadlock may occur when tasks do not synchronize in the right order. Fortunately, deadlocks in programs written in deterministic concurrent languages are generally not hard to detect because if a program deadlocks under one interleaving of tasks it will deadlock under any other interleaving. However, static deadlock detection is not simple because of the state space explosion problem.

The problem with programs that can deadlock is that during runtime, it is not possible to differentiate when the program is still running from when it is stuck in a deadlock. Therefore, we want a run-time technique for detecting deadlocks with almost no overhead and a method to *deterministically* break the deadlock. Our ultimate goal: not only should any program produce consistent output, it should never deadlock.

In this paper, we start by discussing some of our goals in designing deterministic, deadlock-free systems. Next, in Sections 3 and 4, we examine the pros and cons of existing models. Then, we propose a new model with the desired properties of deadlock-freedom and non-determinism in Section 5. Finally, in Section 6, we iterate through other existing deterministic environments. We discuss if they provide deadlock-freedom or not and we also compare them with our model.

2 Design Considerations

Our goal in designing deterministic, deadlock-free systems is to achieve three things: efficiency, scalability, and programmer flexibility.

2.1 Efficiency

A general hypothesis is that determinism introduces performance degradation because of synchronization, and the amount of degradation depends on the type of synchronization. There are two types of synchronization: centralized and distributed. A centralized synchronization forces all tasks in a system to synchronize while a distributed synchronization forces only a subset of tasks to synchronize. Distributed methods perform better because the tasks have to wait less, but they are more susceptible to deadlocks. An out-of-order synchronization between subsets of tasks may lead to a deadlock. In contrast, deadlocks are avoided in centralized systems because all tasks are forced to synchronize at the same point.

In some cases, the programming environments are non-deterministic, but there are techniques and tools to check for determinism and deadlocks during runtime. The problem with these tools is that they add a considerable amount of overhead that reduces performance drastically.

2.2 *Scalability*

A number of programming environments provide determinism at compile time through, e.g., static verifiers and type systems. These techniques do not explicitly introduce deadlocks but they do not scale because they have to consider all possible interleavings of tasks in the program.

Among the systems that provide determinism at runtime, distributed systems are known to scale better than centralized systems in both performance and ease of implementation.

2.3 *Flexibility and Ease of Use*

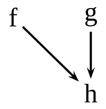
Most deterministic programming models provide determinism by imposing a number of restrictions. Most type systems require programmers to explicitly annotate the program. Static verifiers do not force any restrictions on the program, but they do not scale with flexible programs and give false positives as results. Our goal is to achieve a balance between performance, scalability and programmer flexibility.

3 Kahn Networks

One of the early deterministic programming models is the Kahn Network. A Kahn Network [9] is composed of a set of communication processes that may send and receive on channels. Each communication channel connects a single sending process with a single receiving process. The communication structure of a system is therefore a directed graph whose nodes are processes and whose arcs are channels. There is no shared data; processes communicate only through channels. The receiver process is blocking: it waits until the sender writes the data. The receiver cannot choose to wait based on whether the data is available or not. This property makes the model deterministic. The sender is non-blocking; it writes to one end of the channel and the receiver reads from the other end. The channel is implemented as an unbounded buffer.

```
void f(out a)
{
  for (;;) {
    // send 1 on channel a
    send a = 1;
  }
}

void g(out b)
{
  for(;;) {
    // send 0 on channel b
    send b = 0;
  }
}
```



```
void h(in a, in b) {
  int j;
  for (int i = 0; i++; )
    if (i%2)
      j = recv a;
      // j is now 1
    else
      j = recv b;
      // j is now 0
}

main() {
   chan int a, b;
   // Run f, g, h in parallel
   f(a) par g(b) par h(a, b);
}
```

Figure 1: Kahn processes and their network

Figure 1 shows a Kahn processes and its corresponding network. *f*, *g*, and *h* are three parallel tasks created by the *par* construct in *main*. The two producer tasks *f* and *g* send values 1 (on channel *a*) and 0 (on channel *b*) respectively. Task *h* receives the values from channels *a* and *b* into variable *j*, which sees an alternating stream of 1's and 0's.

In Figure 1, suppose *f* runs faster than *g* or *h*. Here, the buffers of channel *a* fill quickly. However, *h* will not be able to receive the data as quickly as *f* sends. Therefore, there will be an accumulation of data on the channel. This is not a problem in Kahn's model, because the channel acts as an infinite queue between the producer and the consumer. Unfortunately, an infinite bound is impossible to implement in practice. In the next session, we see how this problem can be avoided by adding some restrictions to Kahn Networks.

4 SHIM Networks

SHIM [6] eliminates Kahn's infinite buffers by replacing them with 0-place buffers. The sender and the receiver use rendezvous communication and thus both are blocking; both have to wait for each other to communicate. SHIM, thereby combines the determinism of Kahn Networks with Hoare's CSP [7] to provide bounded determinism. Like Kahn Networks, the tasks in SHIM run asynchronously but synchronize during communication. Tasks can be created dynamically. There is no shared data.

The SHIM program in Figure 2 creates two tasks, *f*

```
void f(in a) {        void g(out b) {      main() {
  a = 3;                b = 5;               chan int c;
  recv a;               send b;              f(c) par g(c);
  // a is now 5         // b is 5            // c is 5
}                     }                    }
```

Figure 2: Example of a SHIM program

and *g*, and runs them in parallel. The *par* statement blocks until both *f* and *g* terminate. *c* is a channel and both *a* and *b* are incarnations of *c*. *g* takes *c* by *out* (reference); any modification of *b* is therefore reflected in main's *c*. *f* takes *c* by *in* (value), and hence *f* maintains a local copy of *c*. Suppose *f* wants to receive the updated value, then it explicitly calls *recv* on *a*. This statement synchronizes with the *send b* of *g* to exchange values.

In Figure 2, even if *g* runs faster than *f*, it has to wait for *f* to synchronize on channel *c*. Therefore, there is some performance degradation. But in practice SHIM programs do not face a significant bottleneck because of the structure of the programs. Future work would be to implement channels as k-bounded buffers.

The SHIM model prohibits any variable from being passed by reference (*out*) to more than one task at a time and this makes it impossible for a task to modify another task's copy of a variable through a simple assignment. Only an *out* variable can act as a sender; pass-by-*in* channels are always receivers. The compiler rejects programs that do not follow this rule. Also, channels cannot be aliased or passed within complex data structures like structures.

To make communication deterministic, a send or receive forces the task sharing the channel to synchronize on either *send* or *recv*, with at most one task acting as a sender and at most one task acting as a receiver. If the receiver terminates before receiving, then the sender does not wait and therefore performs a dummy write to the channel and advances. Similarly, if the sender terminates, then the receiver does not wait and receives the previous value on the channel.

Although SHIM solves the infinite buffer problem, it can deadlock. Consider a program in Figure 3. Task *f*'s *send a* waits for a matching *recv a* from task *g*; task *g*'s *send b* waits for a matching *recv b* from task *f*. The two tasks *f* and *g* wait infinitely for each other: a deadlock.

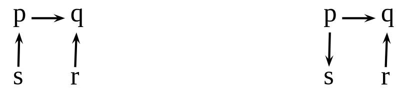
5 Deadlock-free SHIM Networks

A SHIM network is deterministic but not deadlock-free. However, the deadlocks are reproducible [13]; for some input, a deadlock that occurs under one schedule will occur under any schedule.

When a set of tasks deadlock in SHIM, we propose to use a "magic wand" that wakes up the deadlocked tasks

```
void f(out a, in b) {
  // Wait for a recv a in task g
  send a = 1;                        main() {
  recv b; // Unreached                 chan int c;
}                                      f(a, b) par g(b, a);
                                     }
void g(out b, in a) {
  // Wait for a recv b in task f
  send b = 2;
  recv a; // Unreached
}
```

Figure 3: A SHIM program that deadlocks: *f* and *g* each wait for the other on different channels.



(a) A possible SHIM network

(b) An **impossible** SHIM network. *p* has two outgoing edges

Figure 4: Possible and impossible configurations of tasks in the SHIM model

and tells them to proceed with their executions without waiting for matching communications from peers because we have proven they will never occur. At this point, all the deadlocked tasks synchronize to break the deadlock and continue their executions. Before any deadlock, the execution of the SHIM program will be deterministic because of the property of the SHIM model. The deadlock breaking step is deterministic because it just advances all the deadlocked tasks. The program is deterministic after the deadlock is broken because the remaining statements are executed normally following the SHIM principle. Therefore, we still maintain determinism even after introducing a run-time deadlock breaker to the basic SHIM model.

To remove deadlocks, we maintain a dependency graph during runtime. The vertices of the graph represent tasks. When a task *p* calls *send* on a channel, it waits for a peer task *q* to perform a matching *recv* on the same channel. If task *q* is also ready to communicate, then the two tasks rendezvous and the communication is successful. On the other hand, if task *q* is not ready and doing some other work, then task *p* indicates that it is waiting by adding an edge from *p* to *q* in the dependency graph. Then, *p* checks if there is a path from *q* leading back to itself. If there is a cycle, then the program has a deadlock. For a SHIM program, the cycle detection algorithm is inexpensive because each task can block on at most one channel at a time. It follows that there is at most one outgoing edge from any task *p* (Figure 4). Consequently, our cycle finding algorithm takes time linear in the number of tasks.

Since every task updates edges originating from its vertex in the shared dependency graph, the addition of edges by two tasks can be done concurrently because no two tasks would ever add the same edge (i.e., an edge with the same end vertices).

Two or more tasks can check for a cycle concurrently and at least one task in the deadlock will detect a cycle. This is because every task adds the edge first and then checks for a cycle. If a cycle is found, then the first task to detect a cycle clears the cycle by removing the edges in the dependency graph and revives all other blocked processes in the cycle.

All revived tasks (including the task that signalled) now complete their communication by not waiting for their counter operations. A revived *recv* operation receives the last value seen on the channel. A revived *send* value puts the new value on the channel by performing a dummy write.

Figure 5 is a deadlocking SHIM program which we will use to illustrate our deadlock breaking technique. It consists of four simultaneously running tasks. Task *f*'s *send a* waits for *g*'s *recv a*, task *g*'s *send b* waits for *h*'s *recv b*, and task *h*'s *send c* waits for *f*'s *recv c*. In the absence of a deadlock breaker, these three tasks will wait forever.

If we break the deadlocks in the program, the program will run to termination. Suppose *f* calls *send a* first. As shown in Figure 5(a), it will realize that *g* is not ready to receive *a* and therefore add an edge from *f* to *g* in the dependency graph. *f* then checks if there is a cycle. Since there is not yet a cycle, *f* suspends itself. Next, if *h* calls *send c*, it finds that *f* is not ready to receive *c* and therefore *h* adds an edge from vertex *h* to vertex *f*, sees that there is no cycle and suspends itself—Figure 5(b). Next, if *i* calls *recv d*—Figure 5(c)—*i* realizes that *h* is not yet ready to *send d*. Therefore *i* adds an edge from vertex *i* to vertex *h*, sees that there is no cycle and suspends itself. Next, *g* calls *send b* and adds an edge from vertex *g* to *h*—Figure 5(d).

After *g* adds an edge from itself to *h*, it detects a cycle. It now removes the edges in the cycle, revives all the tasks in the deadlock—Figure 5(e). Now the revived tasks can proceed: *f* writes 1 to channel a, *g* writes 2 to channel b and *h* writes 3 to channel c. This modifies *main*'s copy of *a*, *b*, and *c*. The three tasks then advance to their next statements.

Next, the tasks *f*, *g*, and *h* deadlock again on their *recv*'s, forming a cycle: Figures 5(f), 5(g), and 5(h). The deadlock is broken by task *f*: Figure 5(i). *f* receives whatever was last put on the channel *c*, which is 3. Similarly, *g* receives 1 and *h* receives 2. Then tasks *f* and *g* terminate. Now task *h* calls *send d*—Figure 5(j)—and finds that *i* is ready to receive on channel *d*. The two tasks *h* and *i* rendezvous to communicate and terminate.

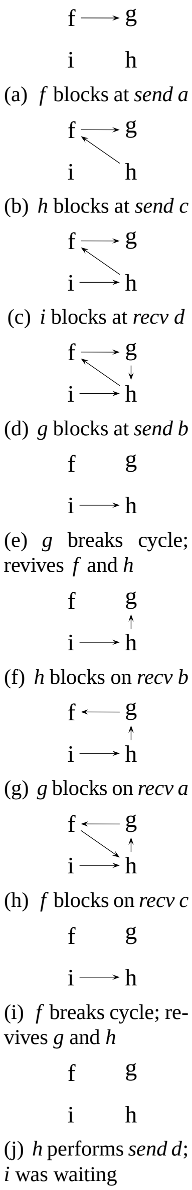
```
void f(out a, in c)
{
    send a = 1; // Deadlock point 1
    // Writes 1 to a after first deadlock
    recv c; // Deadlock point 2
    // Receives 3 after second deadlock
}

void g(out b, in a)
{
    send b = 2; // Deadlock point 1
    // Writes 2 to b after first deadlock
    recv a; // Deadlock point 2
    // Receives 1 after second deadlock
}

void h (out c, in b, out d)
{
    send c = 3; // Deadlock point 1
    // Writes 3 to c after first deadlock
    recv b; // Deadlock point 2
    // Receives 2 after second deadlock
    send d = 4;
}

void i (in d)
{
    recv d; // Receives 4
}

main() {
    // Create channels; initialize with 0
    chan int a = 0, b = 0, c = 0, d = 0;
    // Run f, g, h, and i in parallel
        f(a, c)
    par g(b, a)
    par h(c, b, d)
    par i(d);
    // Here: a = 1, b = 2, c = 3, d = 4
}
```



(a) *f* blocks at *send a*

(b) *h* blocks at *send c*

(c) *i* blocks at *recv d*

(d) *g* blocks at *send b*

(e) *g* breaks cycle; revives *f* and *h*

(f) *h* blocks on *recv b*

(g) *g* blocks on *recv a*

(h) *f* blocks on *recv c*

(i) *f* breaks cycle; revives *g* and *h*

(j) *h* performs *send d*; *i* was waiting

Figure 5: A deadlocking SHIM program and the effect of our deadlock breaking policy

Our method has the advantage of being able to run deadlock detection concurrently in linear time. However, two or more tasks may detect a cycle simultaneously; therefore we need only one of the tasks to take the responsibility of reviving other tasks. We therefore require some sort of synchronization when breaking the deadlock, but not to detect it.

6 Related Work

6.1 Determinizing Tools

A number of tools provide determinism. For example, in the absence of data races, Kendo [10] ensures a deterministic order of all lock acquisitions for a given program input. However, if we have the sequence *lock(A); lock (B)* in one thread and *lock(B); lock(A)* in another thread, a deterministic ordering of locks may still deadlock.

DMP [5] uses a deterministic token that is passed among all threads. A thread to modify a shared variable must first wait for the token and for all threads to block on that token. Although, deadlocks may be avoided, we believe this setting is non-distributed because it forces all threads to synchronize and therefore leads to a considerable performance penalty. In SHIM setting, only threads that share a particular channel must synchronize on that channel; other threads can run independently.

Burmin and Sen [3] provide a framework for checking determinism for multithreaded programs. Their tool does not introduce deadlocks, but their tool does not guarantee determinism because it is merely a testing tool that checks the execution trace with previously executed traces to see if the values match.

6.2 Programming Models

Other programming models guarantee determinism. StreamIt [12], for example, is deterministic dataflow language. It has simple static verification techniques for deadlock and buffer overflow. However, StreamIt is a strict subset of SHIM and StreamIt's design limits it to a smaller class of streaming applications.

Synchronous programming languages like Esterel [1] are deterministic. An Esterel program executes in clock steps and the outputs are synchronous with its inputs. Although an Esterel program is susceptible to causality problems, this form of deadlock can be detected at compile time. Unfortunately, synchronous models require constant, global synchronization and force designers to explicitly schedule virtually every operation. Although standard in hardware designs, global synchronization is costly in software. Furthermore, the presence of a single global clock effectively forces entire systems to operate at the same rate. Frustration with this restriction was one of the original motivations for SHIM.

6.3 Type Systems

Type and effect systems like DPJ [2] have been designed for deterministic parallel programming. These systems do not themselves introduce deadlocks, but type systems generally require programmer annotations. SHIM does not require annotations; it provides restrictions through its constructs. One may argue against learning a new programming paradigm or language like SHIM, but SHIM can be implemented as a library [14] and the deadlock detector could be incorporated into it.

6.4 Deadlock Detection

Deadlock detection algorithms take exponential time on general graphs. SHIM's constraint of never waiting on two channels sidesteps this exponential problem, rendering the cycle-finding algorithm linear time.

There are a number of run-time distributed deadlock detecting algorithms. Chandy, Misra, and Haas [4] is among the best known. According to their technique, whenever a process, say *i*, is waiting on a process, say j, *i* sends a probe message to j. j sends the same message to all the processes it is waiting on and so on. If the probe message comes back to *i*, then *i* reports a deadlock.

Like others, Chandy et al. concentrate on the multiple-path problem where multiple edges may leave a single vertex. Probe messages must be duplicated at these nodes. We can apply the same algorithm to our setting, but since we have at most one outgoing edge per vertex, we do not have to duplicate messages.

We [11, 13] have developed static deadlock mechanisms for finding deadlocks at compile time; it is the programmer's job to break them. Also, these techniques neither scale well with a large number of tasks nor do they support dynamic creation of tasks.

7 Conclusions

We have presented a deterministic, deadlock free model. Any program that uses our model will never deadlock and is guaranteed to be deterministic. The runtime deadlock detector is the contribution of this paper; a formal proof of our hypothesis is future work.

Based on our model and related work, our conclusions are as follows. Generally, if synchronization is highly centralized and non-distributed, then there are no deadlocks but there is a high performance penalty. On the other hand, distributed synchronization imposes a lower performance penalty at the danger of more deadlocks. In our model, the synchronization is distributed and we resolve deadlocks at runtime.

Summarily, our model efficiently addresses the two major pitfalls of concurrent programming: nondeterminism and deadlocks. We do not claim that our method is the best; our technique does limit the programmer's flexibility, but we believe that the discussion of this paper will provide insight on achieving both determinism and deadlock-freedom, and the ideas here can be used for other concurrent models and languages.

References

[1] BERRY, G., AND GONTHIER, G. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming 19*, 2 (Nov. 1992), 87–152.

[2] BOCCHINO, JR., R. L., ADVE, V. S., DIG, D., ADVE, S. V., HEUMANN, S., KOMURAVELLI, R., OVERBEY, J., SIMMONS, P., SUNG, H., AND VAKILIAN, M. A type and effect system for deterministic parallel java. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications* (New York, NY, USA, 2009), ACM, pp. 97–116.

[3] BURNIM, J., AND SEN, K. Asserting and checking determinism for multithreaded programs. In *ESEC/FSE '09: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium* (New York, NY, USA, 2009), ACM, pp. 3–12.

[4] CHANDY, K. M., MISRA, J., AND HAAS, L. M. Distributed deadlock detection. *ACM Trans. Comput. Syst. 1*, 2 (1983), 144–156.

[5] DEVIETTI, J., LUCIA, B., CEZE, L., AND OSKIN, M. Dmp: deterministic shared memory multiprocessing. In *ASPLOS* (2009), ACM, pp. 85–96.

[6] EDWARDS, S. A., AND TARDIEU, O. SHIM: A deterministic model for heterogeneous embedded systems. In *Proceedings of the International Conference on Embedded Software (Emsoft)* (Jersey City, New Jersey, Sept. 2005), pp. 37–44.

[7] HOARE, C. A. R. Communicating sequential processes. *Communications of the ACM 21*, 8 (Aug. 1978), 666–677.

[8] JR., R. L. B., ADVE, V. S., ADVE, S. V., AND SNIR, M. Parallel programming must be deterministic by default. In *HOTPAR '09: USENIX Workshop on Hot Topics in Parallelism* (Mar. 2009).

[9] KAHN, G. The semantics of a simple language for parallel programming. In *Information Processing 74: Proceedings of IFIP Congress 74* (Stockholm, Sweden, Aug. 1974), North-Holland, pp. 471–475.

[10] OLSZEWSKI, M., ANSEL, J., AND AMARASINGHE, S. Kendo: efficient deterministic multithreading in software. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (New York, NY, USA, 2009), ACM, pp. 97–108.

[11] SHAO, B., VASUDEVAN, N., AND EDWARDS, S. A. Compositional deadlock detection for rendezvous communication. In *Proceedings of the International Conference on Embedded Software (Emsoft)* (Grenoble, France, Oct. 2009), pp. 59–66.

[12] THIES, W., KARCZMAREK, M., GORDON, M., MAZE, D., WONG, J., HO, H., BROWN, M., AND AMARASINGHE, S. StreamIt: A compiler for streaming applications, Dec. 2001. MIT-LCS Technical Memo TM-622, Cambridge, MA.

[13] VASUDEVAN, N., AND EDWARDS, S. A. Static deadlock detection for the SHIM concurrent language. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE)* (Anaheim, California, June 2008), pp. 49–58.

[14] VASUDEVAN, N., SINGH, S., AND EDWARDS, S. A. A deterministic multi-way rendezvous library for Haskell. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)* (Miami, Florida, Apr. 2008), pp. 1–12.