Efficient, Deterministic, and Deadlock-Free Concurrency

Thesis proposal

Nalini Vasudevan
Department of Computer Science
Columbia University
naliniv@cs.columbia.edu

April 15, 2010

Abstract

The advent of multicore processors has made concurrent programming models mandatory. However, most concurrent programming models come with two major pitfalls: nondeterminism 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 the improper or the out of order use of these constructs leads to problems like deadlocks.

In this proposal, I argue for both determinism and deadlock-freedom. I propose tools that aid determinism - language constructs for writing deterministic programs, compilers that generate deterministic code and libraries that provide deterministic constructs. Additionally, I discuss techniques to check for features like deadlocks that result from the use of these deterministic constructs. The goal is to provide deadlock-free determinism with minimal loss of performance.

Contents

1 Introduction

Non-deterministic functional behavior arising from timing variability is one of the biggest problems of concurrent programming. The program in Figure 1 is non-deterministic. It uses Cilk [3]-like syntax. It creates two tasks f and g in parallel using the *spawn* construct. Clearly, x is getting modified concurrently by both the tasks, so the value printed by this program is either 3 or 5 depending on the schedule.

```
1  void f(int &a) {
2    a = 3;
3  }
4
5  void g(int &b) {
6    b = 5;
7  }
8
9  main() {
10   int x = 1;
11   spawn f(x)
12   spawn g(x);
13   sync; /* Wait for f and g to finish */
14   print x;
15 }
```

Figure 1: A non-deterministic parallel program

Such non-determinism makes debugging all but impossible because unwanted behavior is rarely reproducible. Re-running a non-deterministic program on the same input usually does not produce the same behavior. Deterministic replay systems [8, 1] facilitate debugging of concurrent programs but they incur a very high runtime overhead, and they are input dependent.

By contrast, all sequential programming languages (e.g., C) are deterministic: they produce the same output given the same input. Inputs include usual things such as files and command-line arguments, but for reproducibility and portability, things such as the processor architecture, the compiler, and even the operating system are not considered inputs. This helps programmers by making it simpler to reason about a program and it also simplifies verification because if a program produces the desired result for an input during testing, it will do so reliably.

Sequential consistent concurrent models are stricter than general concurrent models but they allow races. Models based on atomic transactions and locks are race free but are not deterministic. For instance, protecting x by a lock in Figure 1 will still produce non-deterministic output. Concurrent software languages are generally based on these models and use traditional shared memory, locks, and condition variables model (e.g., pthreads or Java). They are non-deterministic because the output of a program may depend on such things as the operating system's scheduling policy, the relative execution rates of parallel processors, and other things outside the application programmer's control. Not only does this demand a programmer consider the effects of these things when designing the program, it also means testing can only say a program *may* behave correctly on certain inputs, not that it will.

I agree with Bocchino et al. [15] that the programming environment should ensure input-output determinism. By determinism, we mean that the program's output should only depend on the input, and not on the environment (operating system schedule, processor, cache etc.). A few concurrent programming languages provide determinism through their language semantics. SHIM [11, 22],

Section(s)	Categories	The question to be answered
3.1	Determinism, Efficiency	Are deterministic languages efficient?
3.2	Determinism, Deadlock-freedom	How easy is it to statically detect deadlocks in deterministic programs?
3.3, 3.5	Determinism, Efficiency	Is optimization easy when there is determinism?
3.4	Determinism, Efficiency	Can we obtain determinism through a library?
4.1.1	Determinism, Deadlock-Freedom	Is it possible to resolve deadlocks during runtime in deterministic programs?
4.1.2	Determinism, Deadlock-freedom, Efficiency	Can we design deterministic, deadlock-free constructs for concurrency?

Table 1: Summary of my research

for example is an asynchronous concurrent language that is scheduling-independent: its input/output behavior is not affected by any non-deterministic scheduling choices taken by its runtime environment due to processor speed, the operating system, scheduling policy, etc. A SHIM program is composed of sequential tasks that synchronize whenever they want to communicate. The language is a subset of Kahn networks [16] (to ensure determinism) that employs the rendezvous of Hoare's CSP [13] for communication to keep its behavior tractable. SHIM is a strict subset of Kahn networks, it inherits Kahn's scheduling independence: the sequence of data values passed across each communication channel is guaranteed to be the same for all correct executions of the program (and potentially input-dependent). The central hypothesis of SHIM is that deterministic concurrent languages are desirable and practical. They relieve the programmer from considering different execution orders.

While deterministic concurrent models are interesting, they give rise to a number of problems. For example, if the tasks do not synchronize in the right order defined by the synchronization protocol, we obtain a deadlock. A deadlock is a situation when two or more tasks are indefinitely waiting for each other to finish. Deadlocks are frustrating and generally hard to manually detect during run-time.

My work has mainly been with the SHIM programming language. With SHIM, I demonstrate that determinism has advantages for optimization, and verification because it makes it easier for an automated tool to understand a program's behavior. The advantage is particularly helpful for verification, which for SHIM can ignore differently interleaved executions. Although statements in concurrently running SHIM processes may execute in different orders, SHIM's determinism guarantees this will not affect the result of the program and hence any property being checked.

In this proposal, I begin by describing the SHIM language in Section 2. I then discuss my contributions to the language. In Section 3.1, I demonstrate the efficiency of the language on shared-memory and heterogeneous processors through compilers that generate code for these architectures. Then, in Section 3.2, I discuss static verification techniques to find deadlocks in SHIM.

In Section 3.3, I apply similar verification techniques to find opportunities for memory optimization in SHIM. Then in Section 3.4, we discuss ways of implementing SHIM as a library. I have also extended these ideas to other languages like X10 in Section 3.5. Finally, I conclude by exposing some of the open problems in this area that I wish to address in the future. Table 1 summarizes my work.

2 The SHIM Programming Language

The SHIM model guarantees functional determinism by restricting inter-thread communication to a multi-way rendezvous mechanism. There are no shared variables in SHIM; if two tasks have to communicate, they do it through explicit rendezvous communication. The SHIM language, which embodies the model, is a C-like language with additional constructs for concurrency. Specifically,

- *p par q* runs the statements *p* and *q* concurrently, waiting for both statements to finish before proceeding. There are no global variables. To access shared data, SHIM tasks communicate through multi-way rendezvous.
- *send* and *recv* are blocking communication operators on channels.

```
1  f(chan int a) {
2  // a is a copy of c
3    a = 3; recv a; // a gets c's value
4    // a = 5
5  }
6
7  g(chan int &b) {
8    // b is an alias for c
9    b = 5; send b; // synchronize with f
10   // b = 5
11 }
12
13 main() {
14   chan int c = 0;
15   f(c); par g(c);
16   // c = 5;
17 }
```

The program creates two tasks, f 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 reference; any modification of b is therefore reflected in main's c. f takes c by value, and hence maintains a local copy of it. Suppose f wants to receive the updated value, then it explicitly calls *recv* on a. This statement synchronizes with *send b* of g to exchange values.

The language prohibits any variable from being passed by reference 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 a reference variable can act as a sender; pass-by-value channels are always receivers. The compiler rejects programs that do not follow this rule.

To make communication deterministic, a send or receive forces all the task sharing the channel to synchronize on either *send* or *recv*, with at most one task acting as a sender, but allowed to have multiple receivers. All receivers participating in the rendezvous receive the same value.

In the presence of a fair scheduler, any SHIM program is guaranteed to give the same output for a given input.

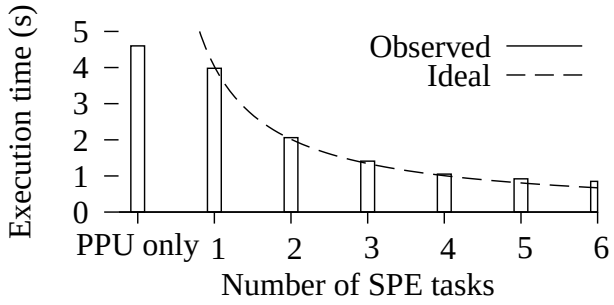


Figure 2: Behavior of FFT on the Cell Processor (Run on a 20 MB audio file, 1024-point FFTs)

.

3 Contributions

My work has been to prove that simple, deterministic semantic helps both programming and automated program analysis. I have been able to devise effective mechanisms for code generation and analysis (e.g., deadlock detection, memory reduction) in SHIM that can gain deep insight into the behavior of programs. In this section, I discuss my contributions in brief.

3.1 Generating Code from SHIM

Deterministic concurrency generally adds additional synchronization to programs and therefore results in performance degradation. To measure the performance, Edwards, Tardieu and I developed a backend [12] for SHIM that generates C code that made calls to the POSIX thread (pthread) library to ask for parallelism. Each communication action acquires the lock on a channel and checks whether every process connected to it had also blocked (i.e., whether the rendezvous could occur). Table 2 shows statistics for a JPEG Decoder [24] with our pthread backend on an Intel Quad Core Machine. The parallel version achieves a 3.05× speedup: 76% of an ideal 4× speedup on four cores.

Cores	Tasks	Time	Speedup
1	Sequential	25s	1.0
4	3	16	1.6
4	4	9.3	2.7
4	5	8.7	2.9
4	6	8.2	3.05
4	7	8.6	2.9

Table 2: Behavior of a JPEG decoder on a quad-core machine (Run on a 20 MB 21600 × 10800 image that expands to 668 MB).

Edwards and I also developed a backend for IBM's CELL processor [27]. Being a direct offshoot of the pthread backend, it allows the user to assign computationally intensive tasks to the CELL's synergistic processing units (SPUs); remaining tasks run on the CELL's POWERPC core (PPU). Figure 2 shows execution times for an FFT on the cell engine. We observed a near-ideal speedup for the FFT on six SPEs.

Example	Lines	Channels	Tasks	Result	Runtime	Memory	States
Source-Sink	35	2	11	No Deadlock	0.2 s	3.9 MB	97
Pipeline	30	7	13	No Deadlock	0.1	2.0	95
Prime Number Sieve	35	51	45	No Deadlock	1.7	25.4	3.2×10^9
Berkeley	40	3	11	No Deadlock	0.2	7.2	139
FIR Filter	100	28	28	No Deadlock	0.4	13.4	4134
Bitonic Sort	130	65	167	No Deadlock	8.5	63.8	25
Framebuffer	220	11	12	No Deadlock	1.7	11.6	9593
JPEG Decoder	1020	7	15	May Deadlock	0.9	85.6	571
JPEG Decoder Modified	1025	7	15	No Deadlock	0.9	85.6	303

Table 3: Results of applying our deadlock detector technique on SHIM programs

3.2 Deadlock Detection for SHIM

The language design of SHIM makes verification easy. One of the properties I tried to verify is deadlock-freedom. SHIM is not immune to deadlocks (e.g., { recv a; recv b; } par { send b; send a; } is about the simplest example); *recv a* of the first task waits for *send a*, and *send b* of the second task waits for *recv b*, and therefore the two tasks wait for each other infinitely. Deadlocks in SHIM cannot occur because of race conditions. For example, because SHIM does not have races, there are no race-induced deadlocks, such as the "grab locks in opposite order" deadlock race present in many other languages. In other words, if a SHIM program deadlocks, it deadlocks under all schedules for a given input.

In general, SHIM does not need to be analyzed under an interleaved model of concurrency since most properties, including deadlock, are the same under any schedule. So all the clever partial order tricks used by model checkers such as SPIN [14], are not necessary for SHIM.

Edwards and I [25] first considered using the symbolic synchronous model checker NuSMV [9] to detect deadlocks in SHIM—an interesting choice since SHIM's concurrency model is fundamentally asynchronous. Our approach was to abstract away data operations and choose a specific schedule in which each communication event takes a single cycle. This reduced the SHIM program to a set of communicating state machines suitable for the NuSMV model checker.

We ran our deadlock detector on various SHIM programs on a 3 GHz Pentium 4 machine with 1 GB RAM. Table 3 lists our results. The Lines column shows for each example the number of lines of code including comments. The Channels and Tasks columns list the number of channels and concurrently running tasks we find after expanding the tasks into a tree and removing non-trivial tasks. Runtimes include the time taken for compilation, abstraction, generating the NuSMV model, and running the NuSMV model checker. As expected, the model checking time dominates on the larger examples. The Memory column reports the total resident set size used by the verifier. The States column reports the number of reachable states NuSMV found. Our tool is able to quickly check programs (in seconds) while using a reasonable amount of memory. While larger programs will be harder to verify, our technique is clearly practical for modestly sized programs.

Recently, Shao, Edwards and I [20] continued our work on deadlock detection in SHIM. This time, we took a compositional approach where we build an automaton for a complete system piece by piece. Our insight is that we can usually abstract away internal channels and simplify the automaton without introducing or avoiding deadlocks. The result is that even though we are doing explicit model-checking, we can often do it much faster than a state-of-the art symbolic model checker such as NuSMV.

Example	Lines	Channels	Tasks	Bytes Saved	Buffer Reduction	Runtime	States
Source-Sink	35	2	11	4	50 %	0.1 s	394
Pipeline	35	5	9	16388	25	0.1	68
Bitonic Sort	35	5	13	12	60	0.1	135
Prime Number Sieve	40	5	16	12	60	0.5	122
Berkeley	40	3	11	4	33.33	0.6	285
FIR Filter	110	28	28	52	46.43	13.8	74646
Framebuffer	185	11	16	28	0.002	1.3	15761
FFT	230	14	15	344068	50	0.6	3750
JPEG Decoder	1020	7	15	772	50.13	1.8	517

Table 4: Results with our static buffer optimization technique

3.3 Buffer Sharing in SHIM

SHIM's elegant design also facilitates memory optimization. Edwards and I applied a model-checking approach to search for situations where buffer memory can be shared [26]. In general, each communication channel needs its own space to store any data being communicated over it, but in certain cases, it is possible to prove that two channels can never be active simultaneously.

```
1  void main()
2  {
3   chan int a, b, c;
4   { // Task 1
5     send a = 6; // Send a (synchronize with task 2)
6   } par { // Task 2
7     recv a; // Receive a (synchronize with task 1)
8     send b = a + 1; // Send 7 on b (synchronize with task 3)
9   } par { // Task 3
10    recv b; // Receive b (synchronize with task 2)
11    send c = b + 1; // Send 8 on c (synchronize with task 4)
12  } par { // Task 4
13    recv c; // Receive c (synchronize with task 3}
14    // c = 8 here
15  }
16 }
```

In the above program, the main task starts four tasks in parallel. Tasks 1 and 2 communicate on *a*. Then, tasks 2 and 3 communicate on *b* and finally tasks 3 and 4 on *c*. The value of *c* received by task 4 is 8. Communication on *a* cannot occur simultaneously with that of *b* because task 2 forces them to occur sequentially. Similarly communications on *b* and *c* are forced to be sequential by task 3. Communications on *a* and *c* cannot occur together because they are forced to be sequential by the communication on *b*. Our tool understands this and reports that *a*, *b*, and *c* can share buffers because their communications never overlap, thereby reducing the program's buffer requirements by 66%.

Our technique produces a static abstraction of a SHIM program's dynamic behavior, which we then analyze to find buffers that can share memory. Experimentally, we find our technique runs quickly on modest-sized programs and can sometimes reduce memory requirements by half; Table 4 lists the results of running the experiments on a 3 GHz Pentium 4 Linux machine with 1 GB RAM. The columns list the number of lines of code in the program, the total number of channels it uses, the number of tasks that take part in communication (i.e., excluding any functions that perform no communication), the number of bytes of buffer memory saved by applying our algorithm, what percentage this is of overall buffer memory, the time taken for analysis (in-

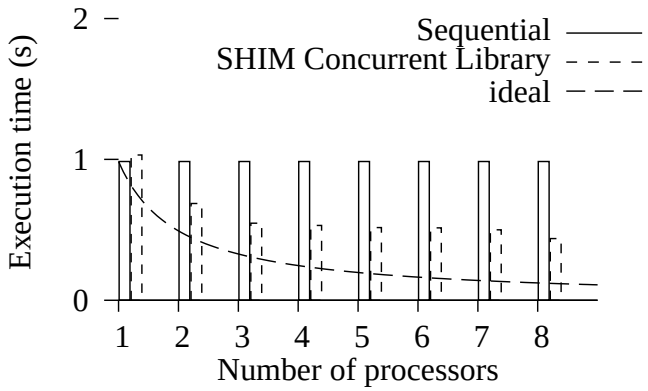


Figure 3: Performance of a Systolic Filter with our library

cluding compilation, abstraction, verification, and grouping buffers), and the number of states our algorithm explored.

3.4 A SHIM-like Library in Haskell

Next, I wanted to explore the feasibility of a model like SHIM as a library. Singh, Edwards and I [29] developed a deterministic concurrent communication library for an existing multi-threaded language. We implemented the SHIM model in the Haskell functional language, which supports transactional memory. Figure 3 compares the execution times of a concurrent Systolic 1-D filter running on 50 000 samples that uses our library, with the sequential version. The experiments were run on an 8-core Intel Machine.

Our experiences do provide insight for the library vs. language debate. While the library approach has the advantage of leveraging features of the host language, we encountered a number of infelicitous that made the library difficult to implement and use.

3.5 Clock Optimization in X10

I have extended my ideas to other concurrent languages. Tardieu, Dolby, Edwards and I developed a tool [30] that mitigates the overhead of general-purpose clocks in IBM's X10 language by analyzing how programs use the clocks and then by choosing optimized implementations when available. These clocks are similar to SHIM's communication constructs.

Clocks in X10 are a mechanism for providing synchronization barriers in concurrent programming languages. They are usually implemented using primitive communication mechanisms and thus spare the programmer from reasoning about low-level implementation details such as remote procedure calls and error conditions. Clocks provide flexibility, but programs often use them in specific ways that do not require their full implementation. We developed a tool that mitigated the overhead of general-purpose clocks by statically analyzing how programs use them and choosing optimized implementations when available.

Our technique models an X10 program as a finite automaton; we ignore data but consider the possibility of clocks being aliased. We pass this automaton to the NuSMV model checker [9],

which reports erroneous usage of a clock and whether a particular clock follows certain idioms. If the clocks are used properly, we use the idiom information to restructure the program to use a more efficient implementation of each clock. The result is a faster program that behaves like one that uses the general-purpose library.

Our analysis flow has been designed to be flexible and amenable to supporting a growing variety of patterns. In the sequel, we focus on inexpensive queries that can be answered by treating programs as sequential. While analysis time is negligible, speedup is considerable and varies across benchmarks from a few percent to a $3\times$ improvement in total execution time.

4 Research Plan

4.1 Future Work

I plan to extend my ideas in different directions. In this section, I discuss some of my short term goals.

4.1.1 Runtime Deadlock Removal for SHIM

SHIM is a deterministic concurrent programming language, but it is prone to deadlocks. The static deadlock detector for SHIM is not completely accurate since it may give false positives. Secondly, even if the deadlock is detected correctly, it is the programmer's job to rectify the code. I therefore propose a dynamic deadlock detection algorithm, that breaks deadlock cycles during program execution, but deterministically. Given a program with deadlock, the resultant behavior is a deadlock-free execution whose output is only dependent on the input.

Whenever a set of tasks deadlock in SHIM, we'll use a magic wand that wakes up the deadlocked tasks and tells them to go ahead with their executions without waiting for their counter operations from peer tasks. At this point, all the deadlocked tasks synchronize to break the deadlock and go ahead with 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 SHIM's principle. Therefore, we still maintain determinism even after introducing a run-time deadlock breaker to the basic SHIM model.

(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

```
1  void f(out a, in c)
2  {
3      send a = 1; /* Deadlocking action */
4      /* Writes 1 to channel 'a' after the deadlock is broken */
5      recv c; /* Another deadlocking action */
6      /* Receives 3 after the deadlock is broken */
7  }
8
9  void g(out b, in a)
10 {
11     send b = 2; /* Deadlocking action */
12     /* Writes 2 to channel 'b' after the deadlock is broken */
13     recv a; /* Another deadlocking action */
14     /* Receives 1 after the deadlock is broken */
15 }
16
17 void h (out c, in b, out d)
18 {
19     send c = 3; /* Deadlocking action */
20     /* Writes 3 to channel 'c' after the deadlock is broken */
21     recv b; /* Another deadlocking action */
22     /* Receives 2 after the deadlock is broken */
23     send d = 4;
24 }
25
26 void i (in d)
27 {
28     recv d; /* Receives 4 */
29 }
30
31 main() {
32     /* Create 4 channels and initializes them with 0 */
33     chan int a = 0, b = 0, c = 0, d = 0;
34     /* Run f, g, h and i in parallel */
35     f(a, c) par g(b, a) par h(c, b, d) par i(d);
36     /* Here: a = 1, b = 2, c = 3, d = 4 */
37 }
```

Figure 5: An example of a deadlock and the effect of my deadlock breaking algorithm

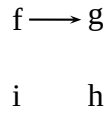
To remove deadlocks, we maintain a dependency graph during runtime. The vertices of the graph are task numbers. Whenever a task p calls *send* on a channel, it waits for a peer task q to do its counter operation *recv* on the same channel. If task q is also ready to communicate, then the two tasks rendezvous 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. This cycle finding algorithm is not expensive because of the following reason. By SHIM semantics, at any instant, a task can at most block on one channel. Therefore, there is at most one outgoing edge from any task p. See Figure 4. Consequently, the cycle finding algorithm takes linear time with 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 to the shared dependency graph can be done concurrently. This is because no two tasks are going to add the same edge (i.e., an edge with the same end vertices).

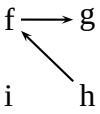
Two or more tasks can check for a cycle concurrently and at least one task in the deadlock will detect a cycle. This is because every task adds the edge first and then checks for a cycle. If

a cycle is found, then the first task to detect a cycle, clears the cycle by removing the edges in the dependency graph and signals all other blocked processes in the cycle to revive.

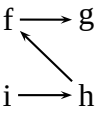
All revived tasks (including the task that signalled) now complete their communication by not waiting for their counter operations. A revived *recv* operation receives the last value seen on the channel. A revived *send* value puts the new value on the channel by performing a dummy write.



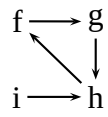
(a) *f* blocks at *send a*



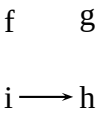
(b) next, *h* blocks at *send c*



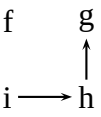
(c) Next, *i* blocks at *recv d*



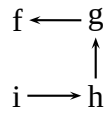
(d) Next, *g* blocks at *send b*



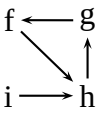
(e) *g* detects a cycle, revives *f* and *h*, and breaks the cycle



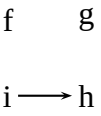
(f) *h* blocks on *recv b*



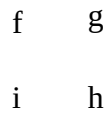
(g) *g* blocks on *recv a*



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



(i) *f* detects a cycle, revives g and h, and breaks the cycle



(j) *h* does *send d*, communication successful because i is already waiting

Figure 6: Building the Dependency Graph for Figure 5

We will now run this technique on a simple example shown in Figure 5. There are four tasks running simultaneously. Task *f*'s *send a* waits for *g*'s *recv a*. Task *g*'s *send b* waits for *h*'s *recv b*. Task *h*'s *send c* waits for *f*'s *recv c*. In the absence of a deadlock breaker, the three tasks wait for each other causing a deadlock.

If we break the deadlocks in the program, the program will terminate. The deadlock breaking technique for Figure 5 is shown in Figure 6. Suppose *f* calls *send a* first (Figure 6(a)), it realizes that *g* is not ready to receive *a* and therefore *f* adds an edge from vertex *f* (itself) to vertex *g* in the dependency graph. It then checks if there is a cycle. Since the cycle has not yet formed, *f* suspends itself. Next, if *h* calls *send c* (Figure 6(b)), it finds that *f* is not ready to receive *c* and therefore *h* adds an edge from vertex *h* (itself) to vertex *f*, sees that there is no cycle and suspends itself. Next, if *i* calls *recv d* (Figure 6(c)), it 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* (Figure 6(d)) and it adds an edge from vertex *g* (itself) to *h*.

After *g* adds an edge from itself to *h*, *g* runs the deadlock detection algorithm and detects a cycle. It (Figure 6(e)) now removes the edges in the cycle, revives all the tasks in the deadlock. Now the revived tasks go ahead with their operations (without waiting for the counterparts). *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 move to their next statements.

Next, the tasks *f*, *g* and *h* deadlock again on their *recv*'s forming a cycle (Figures 6(f), 6(g) and 6(h)). The deadlock is broken by one of the tasks (task *f* in Figure 6(i)). *f* receives whatever was last put on the channel *c* which is 3. Similarly, *g* receives 1, *h* receives 2. Then tasks *f* and *g* terminate. Now task *h* calls *send d* (Figure 6(j)) and finds that *i* is ready to receive on channel *d*. The two tasks *h* and *i* rendezvous to communicate, and they finally terminate.

The advantages of this method are that the deadlock detection technique can run concurrently and 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 to break the deadlock but not to detect a deadlock.

4.1.2 Deterministic, Deadlock-free X10

X10 [7, 19] is a parallel, distributed object-oriented language. To a Java-like sequential core, it adds constructs for concurrency and distribution through the concepts of *activities* and *places*. An activity is a unit of work, like a thread in Java; a place is a logical entity that contains both activities and data objects.

Activities that share a place share a common heap. This brings the possibility of data races and non-determinism in X10 programs. I wish to build new constructs that provide determinism in X10. In contrast to SHIM, I plan to design deadlock-free constructs for determinism.

The idea is as follows. We allow multiple tasks to write to a shared variable concurrently, but we define a commutative, associative reduction operator that will operate on these writes.

The program in Figure 7 creates three tasks in parallel *f*, *g* and *h*. *f* and *g* are modifying *x*. For simplicity, I have used Cilk[3]-like syntax. Even though *f* and *g* are modifying *x* concurrently, *f* sees the effect of *g* only when it executes *next*. Similarly *g* sees the effect of *f* only when it executes *next*. When a task executes *next*, it waits for all tasks that share variables with it to also execute *next*. The *next* statement is like a barrier. At this statement, the shared variables are reduced using the reduction operator. In the example in Figure 7, the reduction operator is $+$ because x is declared with a reduction operator $+$ in line 25. Therefore after the *next* statement, the value of *x* is 8 and it is reflected everywhere. Function *h* also rendezvous with *f* and *g* by executing *next* and thus it obtains the new value 8.

It is also possible to *not* define a reduction operator on a shared variable. Then, the first task among the spawned process (in program source order) overwrites the value. For example, in Figure 7, if the declaration of x in line 25 is *int x* rather than *int (+) x*, then x's value after *next* will be the value written by $f(x)$ which is 3. This is because $f(x)$ is the first concurrent process that is spawned.

```
1  void f(int &a) {
2   /* a is 1 */
3     a = 3;
4   /* a is 3 , x is 1 */
5     next; /* The reduction operator is applied */
6   /* a is now 8, x is 8 */
7  }
8
9  void g(int &b) {
10  /* b is 1 */
11    b = 5;
12  /* b is 5, x is 1 */
13    next; /* The reduction operator is applied */
14  /* b is now 8, x is 8 */
15 }
16
17 void h (int &c) {
18  /* c is 1 , x is 1 */
19    next;
20  /* c is now 8, x is 8 */
21
22 }
23
24 main() {
25    int (+) x = 1;
26    /* If there are multiple writers, reduce
27       using the + reduction operator */
28    spawn f(x);
29    spawn g(x);
30    spawn h(x);
31    sync;
32    /* x is 8 */
33 }
```

Figure 7: Allowing multiple writers

4.2 Plan for completion of the research

Table 5 shows my plan for completion of the research.

Thus, I plan to defend my thesis in May 2011.

5 Related Work

A number of groups are working on a similar problem. In this section, I review some of the related work and compare them with mine.

5.1 Determinizing Tools

There are a number of tools that provide determinism. For example, in the absence of data races, Kendo [18] ensures a deterministic order of all lock acquisitions for a given program input. However, if we have the sequence *lock(A); lock (B)* by one thread and *lock(B); lock(A)* by another thread, the deterministic ordering of locks could still lead to a deadlock.

The DMP [10] tool uses a deterministic token that is passed around all threads. A thread to modify a shared variable must first wait for the token and for all threads to block on that token.

Timeline	Work	Progress
Spring 2007	Compiling SHIM to Shared Memory Multiprocessors	DATE 2008 [12]
Summer 2007	A SHIM-like Library in Haskell	IPDPS 2008 [29]
Fall 2007	Static Deadlock Detection for SHIM	MEMOCODE 2008 [25]
Spring 2008	Compiling SHIM to Heterogeneous Multicores	SAC 2009 [27]
Summer 2008	Analysis and Specialization of Clocks in X10	CC 2009 [30]
Fall 2008	Buffer Sharing in SHIM Programs	MEMOCODE 2009 [26]
Spring 2009	Compositional Deadlock Detection	EMSOFT 2009 [20]
Fall 2009	Overview and Ideas for Thesis	IPDPS Workshop 2010 [17]
Spring 2010	Deterministic Concurrency in X10	In progress at IBM
Fall 2010	Run-time deadlock detection in SHIM	To do
Spring 2011	Thesis Writing and Defense	To do
After graduation	A Determinizing Compiler	PLDI WACI [28]

Table 5: Plan for completion of my research

Although deadlocks may be avoided, we believe this setting is strictly a non-distributed setting because it forces all threads to synchronize, and therefore there is a considerable performance penalty. In our SHIM setting, only threads that share a particular channel have to synchronize on the channel, while other threads can run independently.

Burmin and Sen [5] 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.

5.2 Programming Models

Apart from SHIM, there are a few programming models and languages that provide explicit determinism. StreamIt [23], for example is a synchronous dataflow language that provides determinism. 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 small class of streaming applications.

Synchronous programming languages like Esterel are completely deterministic. An Esterel program executes in clock steps and the outputs are conceptually synchronous with its inputs. It is a finite state language that is easy to verify formally. An Esterel program is susceptible to causalities. Causalities are similar to deadlocks, but can be easily detected at compile-time. The problem with synchronous models is that they do not perform well. To my knowledge, most Esterel compilers generate sequential code and there are hardly any compilers that generate concurrent code off Esterel.

5.3 Type Systems

Finally, type and effect systems like DPJ [4] have been designed for deterministic parallel programming. These systems do not introduce deadlocks by themselves. However, in general type systems require the programmer to manually annotate the program. SHIM does not require annotation but provides restrictions through its constructs. One may argue for the need to learn a new

programming paradigm or language like SHIM, but we believe that SHIM can be implemented as a library [29].

5.4 Software Transactional Memory

Software Transactional Memory (STM) [21] is an alternative to locks: a thread completes modifications to shared memory without regard for what other threads might be doing. At the end of the transaction, it validates and commits if the validation was successful, otherwise it rolls back and re-executes the transaction. STM mechanisms avoid races but do not solve the non-determinism problem.

Tools such as Grace [2] use principles similar to STM, and impose an order in which transactions have to be committed. Grace solves the determinism problem but it incurs a lot of run-time overhead.

5.5 Deadlock Detection

Most concurrent programming languages require all possible interleavings to be considered while checking for deadlocks. We take advantage of SHIM's determinism, and consider one of the many interleavings while verifying a program for deadlocks. This is in great contrast to the motivation for the SPIN model checker [14], one of whose main purposes is to check different execution interleavings for consistency. SHIM has no need for SPIN.

Run-time deadlock detection algorithms run in exponential time for general graphs. SHIM's design of waiting on at most one channel at a time prevents this exponential run-time problem and thus converts the cycle finding algorithm to linear time. There are a number of run-time distributed deadlock detecting algorithms, the well known being that of Chandy, Misra and Haas [6]. 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 detects a deadlock. Like other algorithms, their work concentrates on the multiple-paths problem where there exists multiple outgoing edges from a single vertex. Probe messages are duplicated at nodes that have more than one outgoing edge. 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.

References

[1] Gautam Altekar and Ion Stoica. Odr: output-deterministic replay for multicore debugging. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 193–206, New York, NY, USA, 2009. ACM.

[2] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: safe multithreaded programming for c/c++. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 81–96, New York, NY, USA, 2009. ACM.

[3] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[4] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 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*, pages 97–116, New York, NY, USA, 2009. ACM.

[5] Jacob Burnim and Koushik Sen. 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*, pages 3–12, New York, NY, USA, 2009. ACM.

[6] K. Mani Chandy, Jayadev Misra, and Laura M. Haas. Distributed deadlock detection. *ACM Trans. Comput. Syst.*, 1(2):144–156, 1983.

[7] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, 2005.

[8] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of java multithreaded applications. In *SPDT '98: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 48–59, New York, NY, USA, 1998. ACM.

[9] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV version 2: An OpenSource tool for symbolic model checking. In *Proceedings of the International Conference on Computer-Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364, Copenhagen, Denmark, July 2002.

[10] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. Dmp: deterministic shared memory multiprocessing. In *ASPLOS*, pages 85–96. ACM, 2009.

[11] Stephen A. Edwards and Olivier Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, pages 37–44, Jersey City, New Jersey, September 2005.

[12] Stephen A. Edwards, Nalini Vasudevan, and Olivier Tardieu. Programming shared memory multiprocessors with deterministic message-passing concurrency: Compiling SHIM to Pthreads. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, pages 1498–1503, Munich, Germany, March 2008.

[13] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Upper Saddle River, New Jersey, 1985.

[14] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.

[15] Robert L. Bocchino Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel programming must be deterministic by default. In *HOTPAR '09: USENIX Workshop on Hot Topics in Parallelism*, March 2009.

[16] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74: Proceedings of IFIP Congress 74*, pages 471–475, Stockholm, Sweden, August 1974. North-Holland.

[17] Stephen A. Edwards Nalini Vasudevan. Ensuring deterministic concurrency through compilation. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops*.

[18] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 97–108, New York, NY, USA, 2009. ACM.

[19] Vijay A. Saraswat, Vivek Sarkar, and Christoph von Praun. X10: concurrent programming for modern architectures. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 271–271, New York, NY, USA, 2007. ACM.

[20] Baolin Shao, Nalini Vasudevan, and Stephen A. Edwards. Compositional deadlock detection for rendezvous communication. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, pages 59–66, Grenoble, France, October 2009.

[21] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[22] Olivier Tardieu and Stephen A. Edwards. Scheduling-independent threads and exceptions in SHIM. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, pages 142–151, Seoul, Korea, October 2006.

[23] W. Thies, M. Karczmarek, M. Gordon, D. Maze, J. Wong, H. Ho, M. Brown, and S. Amarasinghe. StreamIt: A compiler for streaming applications, December 2001. MIT-LCS Technical Memo TM-622, Cambridge, MA.

[24] Nalini Vasudevan and Stephen A. Edwards. A JPEG decoder in SHIM. Technical Report CUCS–048–06, Columbia University, Department of Computer Science, New York, New York, USA, December 2006.

[25] Nalini Vasudevan and Stephen A. Edwards. Static deadlock detection for the SHIM concurrent language. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 49–58, Anaheim, California, June 2008.

[26] Nalini Vasudevan and Stephen A. Edwards. Buffer sharing in CSP-like programs. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, Cambridge, Massachusetts, July 2009.

[27] Nalini Vasudevan and Stephen A. Edwards. Celling SHIM: Compiling deterministic concurrency to a heterogeneous multicore. In *Proceedings of the Symposium on Applied Computing (SAC)*, volume III, pages 1626–1631, Honolulu, Hawaii, March 2009.

[28] Nalini Vasudevan and Stephen A. Edwards. A determinizing compiler. In *Programming Languages Design and Implementation (PLDI) - Fun Ideas and Thoughts Session*, Dublin, Ireland, June 2009.

[29] Nalini Vasudevan, Satnam Singh, and Stephen A. Edwards. A deterministic multi-way rendezvous library for Haskell. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–12, Miami, Florida, April 2008.

[30] Nalini Vasudevan, Olivier Tardieu, Julian Dolby, and Stephen A. Edwards. Compile-time analysis and specialization of clocks in concurrent programs. In *Proceedings of Compiler Construction (CC)*, volume 5501 of *Lecture Notes in Computer Science*, pages 48–62, York, United Kingdom, March 2009.