Ensuring Deterministic Concurrency through Compilation

Nalini Vasudevan
Department of Computer Science
Columbia University
New York, NY
naliniv@cs.columbia.edu

Stephen A. Edwards
Department of Computer Science
Columbia University
New York, NY
sedwards@cs.columbia.edu

***Abstract*—Multicore shared-memory architectures are becoming prevalent but bring many programming challenges. Among the biggest is non-determinism: the output of the program does not depend merely on the input, but also on scheduling choices taken by the operating system.**

In this paper, we discuss and propose additional tools that provide determinism guarantees—compilers that generate deterministic code, libraries that provide deterministic constructs, and analyzers that check for determinism. Additionally, we discuss techniques to check for problems like deadlock that can result from the use of these deterministic constructs.

***Keywords*-Determinism, Concurrency, SHIM, Deadlocks**

I. Introduction

Non-deterministic functional behavior arising from timing variability—a data race—is among the nastiest thing a programmer may confront. It makes debugging all but impossible because the unwanted behavior is rarely reproducible. Inevitably, re-running a non-deterministic program on the same input will make the bug appear to go away.

We believe any parallel programming environment should ensure input-output determinism [7]. Sequential programming languages such as assembly or C have always guaranteed determinism, but most parallel environments do not.

A few concurrent programming languages provide determinism through their semantics. SHIM [2], [12], for example is designed to guarantee scheduling independence. SHIM adopts an asynchronous model but uses CSP-like [4] rendezvous communication so the input/output function of a SHIM program does not depend on any scheduling choices taken by the operating system.

Our main contribution has revolved around the SHIM programming language. We wish to demonstrate that determinism has advantages for code synthesis, optimization, and verification because it makes it easier for an automated tool to understand a program's behavior. The advantage is particularly helpful for formal verification algorithms, which can safely ignore alternative execution interleavings of SHIM programs.

We begin by describing the language, our contributions to the language and the field of deterministic concurrency, and related work. We conclude by exposing some of the open problems in this area that we plan to address in the future.

II. The SHIM Language

The SHIM model guarantees functional determinism by restricting inter-thread communication to a multi-way rendezvous mechanism. 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 share data, SHIM tasks must rendezvous.
- *send* and *recv* are blocking communication operators on channels.

```
f(chan int a) {
  // a is a copy of c
  a = 3; recv a; // a gets c's value
  // a = 5
}

g(chan int &b) {
  // b is an alias for c
  b = 5; send b; // synchronize with f
  // b = 5
}

main() {
  chan int c = 0;
  f(c); par g(c);
}
```

This 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 *b*. 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 tasks sharing the channel to synchronize with either a *send* or *recv*. At most one task may send data; the language allows multiple receivers. All receivers participating in the rendezvous receive the same value.

III. Generating Code from SHIM

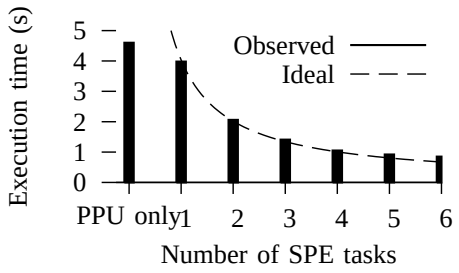
We developed a backend [3] for SHIM that generates C code that made calls to the POSIX thread (pthread) library to ask for parallelism. Each communication action acquires the lock on a channel and checks whether every process connected to it also had blocked (i.e., whether the rendezvous could occur). Table I shows statistics for a JPEG decoder [14] compiled 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.

Table I
BEHAVIOR OF A JPEG DECODER ON A QUAD-CORE MACHINE

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

Run on a 20 MB 21600 × 10800 image that expands to 668 MB.

We also developed a backend for IBM's CELL processor [17]. 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 1 shows execution times for an FFT on the cell engine. We observed a near-ideal speedup for the FFT on six SPUs.



Run on a 20 MB audio file, 1024-point FFTs

Figure 1. Behavior of FFT on the Cell Processor

IV. Deadlock Detection in SHIM

SHIM is not immune to deadlocks. Perhaps the simplest example is { recv a; recv b; } par { send b; send a; }. Here *recv a* of the first task waits for *send a* and *send b* of the second task waits for *recv b*. The two tasks therefore wait for each other infinitely.

SHIM does not need to be analyzed under an interleaved model of concurrency since most properties, including deadlock, are preserved across schedules. We [15] therefore used a synchronous model checker NuSMV [1] to detect deadlocks in SHIM—a surprising choice since SHIM's concurrency model is fundamentally asynchronous. We later took a compositional approach [11] in which we build an automaton for a complete system piece by piece. The result: our explicit model-checker outperforms the implicit NuSMV on these problems.

V. Buffer Optimization in SHIM

We also applied model checking to search for situations where buffer memory can be shared [16]. In general, each communication channel needs its own space to store any data being communicated over it. However, in certain cases, it is possible to prove that two channels can never be active simultaneously and thus share buffer memory.

```
void main()
{
 chan int a, b, c;
 { // Task 1
   send a = 6; // Send a (synchronize with task 2)
 } par { // Task 2
   recv a; // Receive a (synchronize with task 1)
   send b = a + 1; // Send 7 on b (synchronize with task 3)
 } par { // Task 3
   recv b; // Receive b (synchronize with task 2)
   send c = b + 1; // Send 8 on c (synchronize with task 4)
 } par { // Task 4
   recv c; // Receive c (synchronize with task 3}
   // c = 8 here
 }
}
```

Here, 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 on *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%.

VI. A SHIM-like Library in Haskell

We developed a deterministic concurrent communication library [19] for an existing multi-threaded language. We implemented the SHIM model in the Haskell functional language, which supports transactional memory. Figure 2 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.

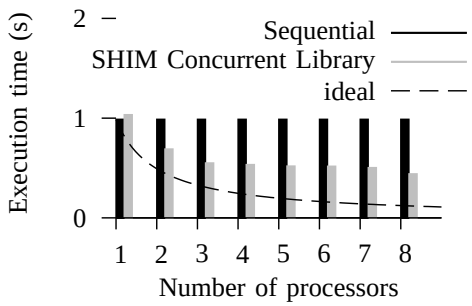


Figure 2. Performance of a Systolic Filter with our library

VII. Related Work

Like SHIM, the StreamIt language [13] is deterministic, but its dataflow model is a strict subset of SHIM's. Our work so far has been on a more flexible model. Regarding optimization, Sermulins et al. [10] present cache aware optimizations that exploit communication pattern in StreamIt programs. They aim to improve instruction and data locality at the cost of data buffer size. Instead, we tried to reduce buffer sizes in section V.

Statements in concurrently running SHIM processes may execute in different orders but SHIM's determinism guarantees this will not affect any result and hence most properties. This is in great contrast to the motivation for the SPIN model checker [5], one of whose main purposes is to check different execution interleavings for consistency. SHIM has no need for SPIN.

Our SHIM library in Haskell resembles that of Scholz [9], which also provides a concurrency model in Haskell; Scholz's library is not necessarily deterministic. Unlike Scholz, we implement our mechanisms atop the existing concurrency facilities in Haskell [6]. We therefore believe it is easy to implement SHIM as a library in any concurrent programming language.

There exist other tools like Kendo [8] that provide determinism, but these tools work mostly at run time, incurring extra overhead. Our focus has been in the language and compiler level.

We have extended our ideas to other concurrent languages. We developed a tool [20] 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.

VIII. Conclusions

The central hypothesis of the SHIM project is that its simple, deterministic semantics helps both programming and automated program analysis. We have been able to devise truly effective mechanisms for clever code generation and analysis (e.g., deadlock detection, buffer optimization). The bottom line: if a programming language does not have simple semantics, it is really hard to analyze its programs quickly or precisely.

We have also attempted to solve the two major problems of concurrent programming: non-determinism and deadlocks. The SHIM model is deterministic and we have provided static techniques to detect deadlocks in SHIM programs.

IX. Future Work

We plan to extend our ideas in various directions. Below, we discuss some of our short term goals.

A. A Deterministic, Deadlock-free Language

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.

We propose a dynamic deadlock detection algorithm that deterministically breaks deadlock cycles during program execution. As a result, the behavior of a program with deadlock can be made deadlock-free and still have the property that its output is only dependent on its input.

B. Automatic and Deterministic Buffer Sizing

Many concurrent programming models like SHIM use rendezvous communication: the sender and the receiver each wait for each other for the communication to succeed. There are two problems with this model. First, the sender cannot go ahead to do some other computation. Therefore, there is a performance bottleneck. Second, a program with these blocking constructs may be susceptible to deadlocks.

A remedy is to use bounded buffers instead of rendezvous communication. We propose to implement the rendezvous communication of SHIM, using bounded buffers and devise methods to automate it. By increasing the number of places in the buffer, we plan to maintain the determinism and the characteristics of the original program but we should be able to resolve deadlocks in certain cases.

C. A Determinizing Compiler

Our final goal is a determinizing compiler [18]: starting from any program, our compiler will insert just enough additional synchronization to guarantee deterministic behavior, even in the presence of nondeterministic scheduling choices. A brute-force solution would simply generate sequential code, but our compiler will strive to preserve parallelism to impose a minimal loss of performance.

Acknowledgment

We thank Julian Dolby, Baolin Shao, Satnam Singh, and Olivier Tardieu for contributing to sections of the work. This research was mainly supported by NSF (grant 0614799) and partly by IBM and Microsoft.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

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

[6] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Proceedings of Principles of Programming Languages (POPL)*, pages 295–308, St. Petersburg Beach, Florida, January 1996.

[7] 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.

[8] 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.

[9] Enno Scholz. Four concurrency primitives for Haskell. In *ACM/IFIP Haskell Workshop*, pages 1–12, La Jolla, California, June 1995. Yale Research Report YALE/DCS/RR–1075.

[10] Janis Sermulins, William Thies, Rodric Rabbah, and Saman Amarasinghe. Cache aware optimization of stream programs. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 115–126, New York, NY, USA, 2005. ACM.

[11] 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.

[12] 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.

[13] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the International Conference on Compiler Construction (CC)*, volume 2304 of *Lecture Notes in Computer Science*, pages 179–196, Grenoble, France, April 2002.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.