Compiling SHIM

Stephen A. Edwards Nalini Vasudevan

Columbia University, New York, NY, USA
{sedwards,naliniv}@cs.columbia.edu

Abstract. Embedded systems demand concurrency for supporting simultaneous actions in their environment and parallel hardware. Although most concurrent programming formalisms are prone to races and non-determinism, some, such as our SHIM (software/hardware integration medium) language, avoid them by design. In particular, the behavior of SHIM programs is scheduling-independent, meaning the I/O behavior of a program is independent of scheduling policies, including the relative execution rates of concurrent processes.
The SHIM project demonstrates how a scheduling-independent language simplifies the design, optimization, and verification of concurrent systems. Through examples and discussion, we describe the SHIM language and code generation techniques for both shared-memory and message-passing architectures, along with some verification algorithms.

1 Introduction

Embedded systems differ from traditional computing systems in their need for concurrent descriptions to handle simultaneous activities in their environment or to exploit parallel hardware. While it would be nice to program such systems in purely sequential languages, this greatly hinders both expressing and exploiting parallelism. Instead, we propose a fundamentally concurrent language that, by construction, avoids many of the usual pitfalls of parallel programming, specifically data races and non-determinism.

Most sequential programming languages (e.g., C) are deterministic: they produce the same output for 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.

By contrast, concurrent software languages based on the traditional shared memory, locks, and condition variables model (e.g., pthreads or Java) are not deterministic by this definition 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.

That deterministic concurrent languages are desirable and practical is the central hypothesis of the SHIM project. That they relieve the programmer from considering different execution orders is clear; whether they impose too many constraints is not something we attempt to answer here.

In this chapter, we demonstrate that determinism also benefits code synthesis, optimization, and verification by making it easier for an automated tool to understand a program's behavior. The advantage is particularly helpful for formal verification algorithms, which can ignore different execution interleavings of SHIM programs. Although statements in concurrently running SHIM processes may execute in different orders, 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 [1], one of whose main purposes is to check different execution interleavings for consistency. SHIM has no need for SPIN.

SHIM is an asynchronous concurrent language whose programs consist of independently running threads coded in an imperative C-like style that communicate exclusively through rendezvous channels. It is a restriction of Kahn networks [2] that replaces Kahn's unbounded buffers with the rendezvous of Hoare's CSP [3]. Kahn's unbounded buffers would make the language Turing-complete [7], and are difficult to schedule [8], so the restriction to rendezvous makes the language easy to analyze. Furthermore, since 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).

We started the SHIM (Software/Hardware Integration Medium) project after observing students having difficulty making systems that communicated across the hardware/software boundary [4]. Our first attempt [5] focused exclusively on this by providing variables that could be accessed by either hardware processes or software functions (both written in C dialect).

We found the inherent nondeterminism of this approach a key drawback. The speed at which software runs on processors is rarely known, let alone controlled, and since software and hardware run in parallel and communicate using shared variables, the resulting system was nondeterministic, making it difficult to test.

Table 1. The SHIM Wish List

Trait	**Motivation**
Concurrent	Hardware/software systems fundamentally parallel
Mixes synchronous and asynchronous styles	Software slower and less predictable than hardware; need something like multirate dataflow
Only requires bounded resources	Fundamental restriction on hardware
Formal semantics	No arguments about meaning or behavior
Scheduling-independent	I/O should not depend on program implementation

```
process sink(uint32 D) {
  int v;
  for (;;)  v = D; /* Read from D */
}

process receiver(uint32 C, uint32 &D) {
  int a, b, r, v;
  a = b = 0;
  for (;;)  {
    r = 1;
    while (r) {
      r = C;     /* Read from C */
      if (r != 0) {
        v = C;  /* Read from C */
        a = a + v;
      }
    }
    b = b + 1;
    D = b;        /* Write to D */
  }
}
```

```
process sender(uint32 &C) {
  int d, e;
  d = 0;
  while (d < 4) {
    e = d;
    while (e > 0) {
      C = 1;    /* Write to C */
      C = e;    /* Write to C */
      e = e − 1;
    }
    C = 0;      /* Write to C */
    d = d + 1;
  }
}

network main() {
  sender();
  receiver ();
  sink ();
}
```

Fig. 1. The dialect of SHIM on which the tail-recursive (Section 3) and static (Section 4) code generators worked. A process contains imperative code; its parameters are channels. A network runs processes or other networks in parallel.

After our first attempt, we started again from the wish list of Table 1. Our goal was to design a concurrent, deterministic (i.e., scheduling-independent) model of computation and started looking around. The synchronous model [6] embodied in languages like Lustre or Esterel assumes a single or harmonically related clocks and thus would not work well for software. The Signal language is based on a richer model whose clocks' rates can be controlled by data, but many find its syntax and semantics confusing. Furthermore, Signal does not guarantee determinism; establishing it requires sometimes-costly program-specific analysis.

In the rest of this chapter, we describe a series of code-generation techniques suitable for both sequential and parallel processors. Each actually works on a slightly different dialect of the SHIM language, although all use the Kahn-with-rendezvous communication scheme. The reason for this diversity is historical; we added features to the SHIM model as we discovered the need for them.

2 Shim with Processes and Networks

Programs in our first Kahn-with-rendezvous dialect of SHIM consisted of sequential processes and hierarchical network blocks (Figure 1). This dichotomy (which we later removed—see Section 5) came from mimicking a similar division in hardware description languages like Verilog.

The body of a network block consisted of instantiations (written in a function-call style) of processes or other networks (recursion was not permitted), which all ran in parallel. For succinctness, the compiler inferred the names of communication channels from process and network arguments, although this could be overridden.

Processes consisted of C-like code without pointers. Process arguments were input or output channels. Following C++ syntax, outputs were marked with ampersands (&), suggesting they were passed by reference. References to arguments would be treated as blocking write operations if they appeared on the left of an assignment statement and blocking reads otherwise.

The overall structure, then, of a SHIM program in this dialect was a collection of sequential processes running in parallel and communicating through point-to-point channels. The compiler rejected programs in which a channel was an output on more than one process.

3 Tail-Recursive Code Generation

Our first code-generation technique produces single-threaded C code for a uniprocessor [9]. The central challenge is efficiently simulating concurrency without (costly, non-portable) operating system support (we present a parallel code generator in Section 6). Our technique uses an extremely simple scheduler—a stack of function pointers—that invokes fragments of concurrently-running processes using tail-recursion.

In tail-recursive code generation, we translate the code for each process into a collection of C functions. In this dialect of SHIM, every program amounted to a group of processes running in parallel (Section 2). The boundaries of these functions are places where the process may communicate and have to block, so each such process function begins with code just after a read or a write and terminates at a read, a write, or when the process itself terminates.

At any time, a process may be running, runnable, blocked on a channel, or terminated. These states are distinguished by the contents of the stack, channel meta-data *struct*s, and the program counter of the process. When a process is runnable, a pointer to one of its functions is on the stack and its *blocked* field (defined in its local variable *struct*) is 0. A running process has control of the processor and there is no pointer to any of its functions on the stack. When a process is blocked, its *blocked* field is 1 and the *reader* or *writer* function pointer of at least one channel has a function pointer to one of the process's functions. When a process has terminated, no pointers to it appear on the stack and its blocked field is 0.

Normal SHIM processes may only block on a single channel at once, so it would seem wasteful to keep a function pointer per channel to remember where a process is to resume. In Section 4, we relax the block-on-single-channel restriction to accommodate code that mimics groups of concurrently-running processes.

Processes communicate through channels that consist of two things: a *struct channel* that contains function pointers to the reading or writing process that is blocked on the channel, and a buffer that can hold a single object being passed through the channel. A non-null function pointer points to the process function that should be invoked when the process becomes runnable again.

Figure 2 shows the implementation of a system consisting of a source process that writes 42 to channel C and a sink process that reads it. The synthesized C code consists of data structures that maintain a set of functions whose execution is pending, a buffer and state for each communication channel, *struct*s that hold the local variables of each process, a collection of functions that hold the code of the processes broken into pieces, a placeholder function called *termination_process* that is called when the system terminates or deadlocks, and finally a *main* function that initializes the stack of pending function pointers and starts the system.

Processes are scheduled by pushing the address of a function on the stack and performing a tail-recursive call to a function popped off the top of the stack. The C code for this is as follows.

```
void func1() {
  ...
 *(sp++) = func2;      /* schedule func2() */
  ...
 (*(--sp))(); return; /* run a pending function */
}

void func2() { ... }
```

Under this scheme, each process is responsible for running the next; there is no central scheduler code.

Because this code generator compiles a SHIM dialect that uses only point-to-point channels for blocking rendezvous-style communication (see Section 2), the first process that attempts to read or write on a channel blocks until the process at the other end of the channel attempts the complementary operation. Communication is the only cause of blocking behavior in SHIM systems (i.e., the scheduler is non-preemptive), so processes control their peers' execution at communication events.

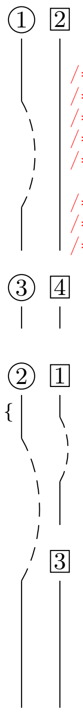
The sequence of actions at *read* and *write* events in the process is fairly complicated but still fairly efficient. Broadly, when a process attempts to read or write, it attempts to unblock its peer, if its peer is waiting, otherwise it blocks on the channel. Annotations in Figure 2 illustrate the behavior of the code. There are two possibilities: when the source runs first (①), it immediately writes the value to be communicated into the buffer for the channel (*C_Value* because the code maintains the invariant that a reader only unblocks a writer after it has read data from the channel buffer) and checks to see if the reader (the sink process) is already blocked on the channel.

Since we assumed the source runs first, the sink is not blocked so the source blocks on the channel. Next, the source records that control should continue at the *source_1* function (the purpose of setting *C.writer*) when the sink resumes it. Finally, it pops and calls the next waiting process function from the stack.

Later, (②) the sink checks if the source is blocked on C. In this source-before-sink scenario, the source is blocked so *sink_0* immediately jumps to

```
void (*stack[3])(void);  /* runnable process stack */
void (**sp)(void);       /* stack pointer */

struct channel {
  void (*reader)(void); /* process blocked reading, if any */
  void (*writer)(void); /* process blocked writing, if any */
};

struct channel C = { 0, 0 };
int C_value;

struct {                /* local state of source process */
  char blocked;         /* 1 = blocked on a channel */
  int tmp1;
} source = { 0 };

struct {                /* local state of sink process */
  char blocked;         /* 1 = blocked on a channel */
  int v;
  int tmp2;
} sink = { 0 };
```

```
process source(int32 &C) {
  C = 42; /* send on C */
}
```



```
void source_0() {                           ① 2
  source.tmp1 = 42;
  C_value = source.tmp1;                    /* write to channel buffer */
  if (sink.blocked && C.reader) {           /* if reader blocked */
    sink.blocked = 0;                       /* mark reader unblocked */
    *(sp++) = C.reader;                     /* schedule the reader */
    C.reader = 0;                           /* clear the channel */
  }
  source.blocked = 1;                       /* block us, the writer */
  C.writer = source_1;                      /* to continue at source_1 */
  (*(--sp))(); return;                      /* run next process */
}

void source_1() {                           ③ 4
  (*(--sp))(); return;
}

void sink_0() {                             ② 1
  if (source.blocked && C.writer) {         /* if writer blocked */
    sink_1(); return;                       /* go directly to sink_1 */
  }
  sink.blocked = 1;                         /* block us, the reader */
  C.reader = sink_1;                        /* to continue at sink_1 */
  (*(--sp))(); return;                      /* run next process */
}
                                              3
void sink_1() {
  sink.tmp2 = C_value;                      /* read from channel buffer */
  source.blocked = 0;                       /* unblock the writer */
  *(sp++) = C.writer;                       /* schedule the writer */
  C.writer = 0;                             /* clear the channel */
  sink.v = sink.tmp2;
  (*(--sp))(); return;                      /* run next process */
}

void termination_process() {}

int main() {
  sp = &stack[0];
  *(sp++) = termination_process;
  *(sp++) = source_0;
  *(sp++) = sink_0;
  (*(--sp))();
  return 0;
}
```

```
process sink(int32 C) {
  int v = C; /* receive */
}
```

Fig. 2. Synthesized code for two processes (in the boxes) that communicate and the `main()` function that schedules them.

sink_1, which fetches the data from the channel buffer, unblocks and schedules the writer, and clears the channel before calling the next process function, *source_1* (③).

When the sink runs first ($\boxed{1}$), it finds the source is not blocked and then blocks. Later, the source runs ($\boxed{2}$), writes into the buffer, discovers the waiting *sink* process, and unblocks and schedules *sink* before blocking itself. Later, *sink_1* runs ($\boxed{3}$), which reads data from the channel buffer, unblocks and schedules the writer, which eventually sends control back to *source_1* ($\boxed{4}$).

The main challenge in generating the code described above is identifying the process function boundaries. We use a variant of extended basic blocks (see Figure 3(c)): a new function starts at the beginning of the process, at a read or write operation, and at any statement with more than one predecessor. This divides the process into single-entry, multiple-exit subtrees, which is finer than it needs to be, but is sufficient and fast. The algorithm is simple: after building the control-flow graph of a process, a DFS is performed starting from each read, write, or multiple-fanin node that goes until it hits such a node. The spanning tree built by each DFS becomes the control-flow graph for the process function, and code is generated mechanically from there.

Figure 3 illustrates the code generation process for a simple process with some interesting control-flow. The process (Figure 3(a)) consists of two nested loops. We translate the SHIM code into a fairly standard linear IR (Figure 3(b)). Its main novelty is *await*, a statement that represents blocking on one or more channels. E.g., *await write C goto 6* indicates the process wants to communicate with its environment on channel C and will branch to statement 6 once this has occurred. Note that the instruction itself only controls synchronization; the actual data transfer takes place in an earlier assignment statement. Although this example (and in fact all SHIM processes) only ever blocks on a single channel at a time, our static scheduling procedure (Section 4) uses the ability to block on multiple channels simultaneously.

Our generated C code uses the following macros:

```
#define BLOCKED_READING(r, ch) r.blocked && ch.reader
#define RUN_READER(r, ch) \
  r.blocked = 0, *(sp++) = ch.reader, ch.reader = 0
#define BLOCK_WRITING(w, ch, succ) w.blocked = 1, ch.writer = succ
#define BLOCKED_WRITING(w, ch) w.blocked && ch.writer
#define RUN_WRITER(w, ch) \
  w.blocked = 0, *(sp++) = ch.writer, ch.writer = 0
#define BLOCK_READING(r, ch, succ) r.blocked = 1, ch.reader = succ
#define RUN_NEXT (*(--sp))(); return
```

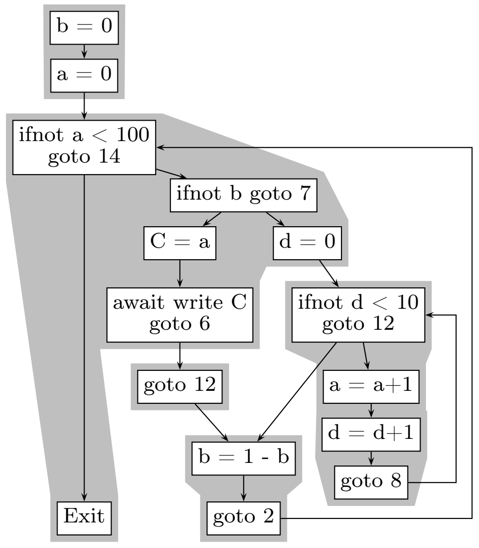
BLOCKED_READING is true if the given process is blocked on the given channel. RUN_READER marks the given process that is blocked on the given channel as runnable. BLOCK_WRITING marks the given process—the currently running one—as blocked writing on the given channel. The *succ* parameter specifies the process function to be executed when the process next becomes runnable. Finally, RUN_NEXT runs the next runnable process.

```
process source(int32 &C) {
  bool b = 0;
  for (int32 a = 0 ; a < 100 ; ) {
    if (b) {
      C = a;
    } else {
      for (int32 d = 0 ; d < 10 ; ++d)
        a = a + 1;
    }
    b = ~b;
  }
}
```

(a)

```
0  b = 0
1  a = 0
2  ifnot a < 100 goto 14
3  ifnot b goto 7
4  C = a
5  await write C goto 6
6  goto 12
7  d = 0
8  ifnot d < 10 goto 12
9  a = a + 1
10 d = d + 1
11 goto 8
12 b = 1 - b
13 goto 2
14 Exit
```

(b)



(c)

```
struct channel C = {0, 0};
int C_val;
struct {
  bool blocked;
  bool b;
  int32 a;
  int32 d;
} source = { 0 };

static void source_0() {
  source.b = 0;
  source.a = 0;
  source_1(); return;
}

static void source_1() {
  if (!(source.a < 100)) goto L9;
  if (!(source.b)) goto L7;
  C_val = source.a;
  if (BLOCKED_READING(sink, C))
    RUN_READER(sink, C);
  BLOCK_WRITING(source, C,
               source_2);
  RUN_NEXT;
L7:
  source.d = 0;
  source_3(); return;
L9:
  RUN_NEXT;
}

static void source_2() {
  source_4(); return;
}

static void source_3() {
L1:
  if (!(source.d<10)) goto L6;
  source.a = source.a + 1;
  source.d = source.d + 1;
  goto L1;
L6:
  source_4(); return;
}

static void source_4() {
  source.b = 1 − source.b;
  source_1(); return;
}
```

(d)

Fig. 3. Generating tail-recursive code for a single process. Our compiler translates a process (a) into an intermediate representation (b). This is translated into a CFG, split into extended basic blocks (c), finally each block becomes a function (d).

4 Code Generation from Static Schedules

The tail-recursive code generator we presented above uses a clever technique to reduce run-time scheduling to little more than popping an address off a stack and jumping to it, but even this amount of overhead can be high for an extremely simple process such as an adder.

In this section, we describe how to eliminate even this low scheduling overhead by compiling together groups of concurrently-running processes into a single imperative process that can be substituted for the group of processes [9]. Efficiency is the advantage of this approach: by analyzing the behavior of a group at compile time, we are able to eliminate most scheduling overhead. Our procedure is therefore similar to many known techniques for sequential code generation, but makes different trade-offs. It generates an automaton for a group of SHIM processes using exhaustive simulation that resembles the subset construction algorithm for generating deterministic finite automata from nondeterministic ones.

The disadvantage of this approach is a potential explosion in code size. Since it builds a product machine from concurrently-running processes, there is a danger of an exponential state explosion. We do not consider this a serious problem for two reasons: our abstraction of processes often leads to small machines for large systems, and it is always possible to synthesize smaller subsets of a system and run them dynamically. Our technique therefore provides a controllable time/space trade-off.

The complete state of a SHIM system comprises the program counter of each process and the value of each process-local variable. While we could build an automaton whose states exactly represent this, it would be impractically large for all but the simplest programs. Instead we track an abstract version of the system state in the automaton. While this does defer many computations to when the generated code is running, it greatly reduces the size of the automata and hence the generated code. Experimentally, we find this a good trade-off.

Because SHIM systems tend to have periodic communication patterns, it turns out we can compile away most of the scheduling overhead and still have small automata. Unfortunately, while compiling away context-switching overhead would also be nice, it would demand tracking combinations of reachable program counter states, something that easily grows exponential. We find our current solution a good trade-off that can produce impressive speed-ups.

Each state in our generated automaton represents the execution of one process between context-switch points or a point where the subnetwork is waiting for its environment. Each transition corresponds to as many as two separate communication events, so the automaton represents the system's communication pattern. For each state, we copy code from the state's process and replace context-switching points with *goto*s to code for the state's successors.

Each state's signature—the system state we insist be unique for each automaton state—is a flag for each process indicating whether it is runnable plus a flag for each channel that indicates whether the channel is clear, blocked on a reader, or blocked on a writer. We deliberately ignore program counters and local variables in the signature—our abstraction to produce compact automata.

Although we do not consider it part of a state's signature for matching purposes, we do track what program counter values are possible to streamline the generated code and reduce the size of the automaton by limiting both the amount of code generated for each state (unreachable code is omitted) and the number of successor states. Practically, when we reach a state with the same signature as an existing one but with new program counter values, we consider the two states identical and form the union of the program counter sets. Our simulation procedure thus combines a depth-first search and a relaxation procedure that finds a fixed point.

Figure 4 shows a simple program being transformed into an automaton. The program's three processes (Figure 4(a)) are a sink that always reads, a buffer that reads and then writes, and a source that sends four numbers and terminates. Our compiler dismantles processes into statement lists (Figure 4(b)) that are simulated to produce an automaton (Figure 4(c)). Our compiler then generates code for each state in the automaton and connects them with *goto*s, producing the IR in Figure 4(d). This IR is then passed to the normal code generation procedure described in Section 3 to produce executable C.

The structure of Figure 4(c) is typical of systems with periodic behavior that terminate: the first state initializes the system to bring it to where periodic behavior begins. The loop represents the periodic behavior, and the state just outside the loop represents a deadlock because the source has terminated.

Each state in Figure 4(c) is labeled with its name; the state of each process, either runnable (⊠) or blocked on a channel (□) when the state begins; the state of each channel ("`-`" for clear, "`R`" when its reader is blocked, and "`W`" when the writer is blocked); and a set of program counter values that each process may be in at the beginning of the state. Thus, in State 1, process 0 is blocked, processes 1 and 2 are runnable, no process is blocked the first channel (A), and the reader (the sink process) is blocked on the second channel (B). Moreover, the first process (sink) must be at instruction 1, the second process (buffer), may be at instruction 0 or 4, and the third process may be at 0, 2, 4, 6, or 8.

A SHIM system runs consistently under any reasonable scheduling policy [10]. We adopt a scheduling policy that selects the lowest-numbered runnable process. The automaton we generate, therefore, depends on process labeling (currently from positions in the source file), but it is guaranteed to produce the same overall behavior. A better scheduling policy could improve the generated code.

The automaton generation procedure starts with all processes runnable and all program counters at 0—State 0 in Figure 4(c). Our scheduling policy then runs the first process—the sink—which executes instruction 0 and blocks on channel 1 (B), so State 1 has channel 1 blocked on sink. The first runnable process, 1 (the buffer) starts at instruction 0 in State 1, tries to read from channel 0 (A), and blocks. This gives State 2, in which the first two processes (sink and buffer) are blocked and channels 0 and 1 are blocked on them.

The loop in Figure 4(c) (States 1, 2, 3, and 4) is periodic behavior: the buffer blocks trying to read, the source emits a token, the buffer reads it, the source reads it, and the loop repeats.

The simulation traces the loop four times because the source can be at four control points waiting to write on A, but this does not create new states because each has the same signature; here our choice of signature shrinks the automaton.

State 2 in Figure 4(c) has two successors: the loop (State 3) and State 5. This is a choice between the three PC values (2, 4, and 6) that lead to a write on the A channel and a fourth (8) that brings it to termination. State 5 corresponds to the state in which no process is runnable; the buffer is waiting to read from the source and the sink is waiting to read from the buffer.

Figure 4(d) is the IR generated from the automaton in Figure 4(c). Each state produces a code fragment, some of which begin with a *switch* that sends control to where the process suspended. The code for each state ends by assigning a constant to the process's state variable that indicates where it should resume. We describe the generation of such *switch* statement code elsewhere [11]. The mechanism is analogous to the tail-recursive calls to function pointers described in Section 3, but keeps the code together.

5 SHIM with Functions, Recursion, and Exceptions

The processes-and-networks dialect of SHIM (Section 2) worked, but we quickly discovered we missed function calls. We also found that we wanted the ability to run lightweight blocks of code in parallel rather than require new processes be declared. Finally, we decided to add exceptions, which turned out to be interesting but technically challenging. The result was, by design, a much more C-like language, which simplified the task of porting existing C programs into SHIM.

We began [12] by removing the process/network dichotomy by introducing the *par* construct, which starts two or more code blocks in parallel and waits for them to terminate. To uphold the SHIM model and its goal of scheduling-independence, our compiler actually split each code block into a separate functions and carefully determined which variables to pass into and out of it.

A key trick was to infer a pass-by-reference parameter for any variable modified by the code block and a pass-by-value parameter for any variable only read by the code block. Furthermore, we prohibited any variable from being passed by reference more than once at a call site, thus prohibiting any more than one alias for each variable at any given time.

For example,

```
  x = y;
par
  y = x;
```

swaps the values of x and y. Internally, the compiler expands this into functions:

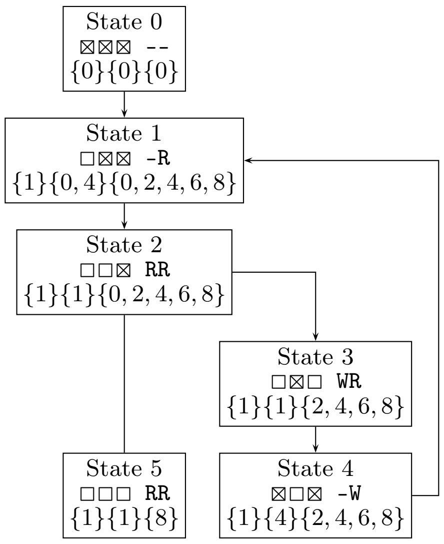
```
void block_1(int &x, int y) { x = y; }
void block_2(int &y, int x) { y = x; }

  block_1(x, y);
par
  block_2(y, x);
```

```
process sink(int32 B) {
  for (;;)  B;
}

process buffer(int32 &B,
               int32 A) {
  for (;;)  B = A;
}

process source(int32 &A) {
  A = 17;
  A = 42;
  A = 157;
  A = 8;
}

network main() {
  sink();
  buffer();
  source();
}
```

(a) Shim code

```
sink
0 PreRead 1
1 PostRead 1 tmp3
2 goto 0

buffer
0 PreRead 0
1 PostRead 0 tmp2
2 tmp1 := tmp2
3 Write 1 tmp1
4 goto 0

source
0 tmp4 := 17
1 Write 0 tmp4
2 tmp5 := 42
3 Write 0 tmp5
4 tmp6 := 157
5 Write 0 tmp7
6 tmp8 := 8
7 Write 0 tmp8
8 Exit
```

(b) Dismantled



(c) The automaton

```
0  /* State 0 (sink) */
1  sink_state = 1
2  goto 3

3  /* State 1 (buffer) */
4  switch buffer_state
     case 0: goto 8
     case 4: goto 7
5  buffer_state = 1
6  goto 9
7  goto 5
8  goto 5

9  /* State 2 (source) */
10 switch source_state
     case 0: goto 29
     case 2: goto 25
     case 4: goto 21
     case 6: goto 17
     case 8: goto 15
11 value__V0 = 17
12 A__V0 = value__V0
13 source_state = 2
14 goto 30
15 source_state = 8
16 goto 42
17 value__V3 = 8
18 A__V0 = value__V3
19 source_state = 8
20 goto 30
21 value__V2 = 157
22 A__V0 = value__V2
23 source_state = 6
24 goto 30
25 value__V1 = 42
26 A__V0 = value__V1
27 source_state = 4
28 goto 30
29 goto 11

30 /* State 3 (buffer) */
31 value__V5 = A__V0
32 received 0 in value__V5
33 value__V4 = value__V5
34 B__V1 = value__V4
35 buffer_state = 4
36 goto 37

37 /* State 4 (sink) */
38 value__V6 = B__V1
39 received 1 in value__V6
40 sink_state = 1
41 goto 3

42 /* State 5 (blocked) */
43 exit
```

(d) The generated IR

Fig. 4. Synthesizing the automaton for three concurrently-running processes. The shim code (a) is first translated into a linear ir (b) that splits read operations into two halves. Simulating these processes produces an automaton (c), from which a different type of ir is generated (d). This is passed to the code generation algorithm in Section 3 to be translated into C.

```
void sink(chan uint32 D) {
  for (;;)  recv D;
}

void receiver(chan uint32 C,
              chan uint32 &D) {
  int a;
  a = D = 0;
  for (;;)  {
    while (next C)
      a = a + next C;
    D = D + 1;
    send D;
    next D = a;
  }
}
```

```
void sender(chan uint32 &C) throws Done {
  int d, e;
  for ( d = 0 ; d < 4 ; d = d + 1) {
    for ( e = d ; e > 0 ; e = e − 1 ) {
      next C = 1;
      next C = e;
    }
    next C = 0;
  }
  throw Done;
}

void main() {
  chan uint32 C, D;
  try
    sender(C); par receiver(C, D); par sink(D);
  catch (Done) {}
}
```

Fig. 5. The program of Figure 1 coded in the latest SHIM dialect. We added *par*, *chan*, *send*, *recv*, *next*, *try*, *catch*, and *throw* and removed the distinction between processes and networks: both are now functions.

Here, the two *block* functions run in parallel. The first is passed a reference to x and a copy of y, the second a reference to y and a copy of x. Thus, the assignments can happen in either order and produce the same result.

We also made the syntax for communication on channels more explicit by adding *send*, *recv*, and *next* keywords, rather than make it simply a side-effect of referencing a channel. Although clarity was the main motivation for adding *send* and *recv* (the previous policy confused many users), was also found we were often reading a value from a channel and storing it locally so we could refer to it multiple times. We attempted to retain the communication-in-an-expression syntax by introducing the *next* keyword, which sends on a channel if it appears on the left side of an assignment and receives when it appears elsewhere. While it can make for very succinct code (*next* b = *next* a is a succinct way of writing a buffer), users continue to find it confusing and prefer the *send* and *recv* syntax. Figure 5 shows the program of Figure 1 coded in this new dialect.

We also added the facility for multiway rendezvous. Although a channel may be passed by reference only once, it may be passed by value an unlimited number of times. In this case, each function that receives a pass-by-value copy of the channel is required to participate in any rendezvous on the channel. A primary motivation of this was to facilitate debugging—it is easy now to add processes that monitor channels without affecting a system's behavior. In retrospect, this facility is sparsely used, difficult to implement, and slows down the more typical point-to-point communication unless carefully optimized away.

```
void buffer(int i, int &o) {
  for (;;) {
    recv i;
    o = i;
    send o;
  }
}
```

```
void fifo(int i, int &o, int n) {
  int c; int m = n − 1;
  if (m)
    buffer(i, c) par fifo(c, o, m);
  else
    buffer(i, o);
}
```

Fig. 6. An n-place FIFO specified using recursion, from Tardieu and Edwards [12]

5.1 Recursion

When we introduced function calls, we had to consider how to handle recursion. Our main goal was to make basic function calls work, allowing the usual re-use of code, but we also found that recursion, especially bounded recursion, was an interesting mechanism for specifying more complex structures.

Figure 6 illustrates this style. The recursive *fifo* procedure calls itself repeatedly in parallel, effectively instantiating *buffer* processes as it goes. This recursion runs only once, when the program starts, to set up a chain of single-place buffers.

We developed a technique for removing bounded recursion from SHIM programs [13]. One goal was to simplify SHIM's translation into hardware, where general recursion would require memory for a stack and choosing a size for it, but it has found many other uses. In particular, if all the recursion in a program is bounded, the program is finite-state, simplifying other analysis steps.

The basic idea of our work was to unroll recursive calls by exactly tracking the behavior of variables that control the recursion. Our insight was to observe that for a recursive function to terminate, the recursive call must be within the scope of a conditional. Therefore, we need to track the predicate of this conditional, see what can affect it, and so forth.

Figure 7 shows the transformation of a simple FIFO. Our procedure produces the static version in Figure 7(b) by observing that the n variable controls the predicate around *fifo*'s recursive call. Then it notices n is set first to 3 by *fifo3* and generates three specialized versions of *fifo*—one with $n = 3$, $n = 2$, and $n = 1$—simplifies each, then inlines each function, since each is only called once.

Of course, in the worst case our procedure could end up trying to track every variable in the program, which would be impractical, but in programs written with this idiom in mind, recursion control only involved a few variables, making it easy to resolve.

5.2 Exceptions

At this stage, we also added exceptions [14], without question the most technically difficult addition we have made. Inspired by the Esterel language [15], where exceptions are used not just for occasional error handling but as widely as, say, if-then-else, we wanted our exceptions to be widely applicable and be concurrent and scheduling-independent.

```
void fifo3(chan int i, chan int &o) {
  fifo(i, o, 3);
}

void fifo(chan int i, chan int &o, int n) {
  if (n > 1) {
    chan int c;
    buf(i, c); par fifo(c, o, n−1);
  } else buf(i, o);
}

void buf(chan int i, chan in &o) {
  int tmp;
  for (;;) {
    tmp = recv i;
    send o = tmp;
  }
}
```

(a)

```
void fifo3(chan int i, chan int &o) {
  chan int c1, c2, c3;
    buf(i, c1);
  par
    buf(c1, c2);
  par
    buf(c2, o);
}

void buf(chan int i, chan in &o) {
    int tmp;
    for (;;) {
      tmp = recv i;
      send o = tmp;
    }
}
```

(b)

Fig. 7. Removing bounded recursion, controlled by the n variable, from (a) gives (b). After Edwards and Zeng [13].

For sequential code, the desired exception semantics were clear: throwing an exception immediately sends control to the most-recently-entered handler for the given exception, terminating any functions that were called in between.

For concurrently running functions, the right behavior was less obvious. We wanted to terminate everything leading up to the handler, including any concurrently running relatives, but we insisted on maintaining SHIM's scheduling independence, meaning we had to carefully time when the effect of an exception was felt. Simply terminating siblings when one called an exception would be nondeterministic: the behavior would then depend on the relative execution rates of of the processes and thus not be scheduling independent.

Our solution was to piggyback the exception mechanism on the communication system. The idea was that a process would only learn of an exception when it attempted to communicate, since it is only at rendezvous points that two processes agree on what time it is. Thus, SHIM's exception mechanism is layered on the inter-process communication mechanism to preserve determinism while providing powerful sequential control.

To accommodate exceptions, we introduced a new "poisoned" state for a process that represents when it has been terminated by an exception and is waiting for its relatives to terminate. Any process that attempts to communicate with a poisoned process will itself become poisoned. In Figure 9, the first thread throws an exception; the second thread is poisoned when it attempts to rendezvous on i, and the third is poisoned by the second when it attempts to rendezvous on j.

The idea was simple enough, and the interface it presented to the programmer could certainly be used and explained without much difficulty, but implementing

```
void main() {
  int i;  i = 0;
  try {
    i = 1;
    throw T;
    i = i * 2; // not executed
  } catch(T) {
    i = i * 3; // i = 3
  }
}
```

(a)

```
void main() {
  int i;
  i = 0;
  try {           // thread 1
    throw T;
  } par {         // thread 2
    for (;;)      // never terminated
      i = i + 1;
  } catch(T) {}
}
```

(b)

Fig. 8. (a) Sequential exception semantics are classical. (b) Thread 2 never feels the effect of the exception because it never communicates. From Tardieu and Edwards [14].

```
void main() {
  chan int i = 0, j = 0;
  try {                                 // task 1
    while (i < 5) send i = i + 1;
    throw T;                            // poisons itself
  } par {                               // task 2
    for (;;)  send j = recv i + 1;    // poisoned by task 1
  } par {                               // task 3
    for (;;)  recv j;                   // poisoned by task 2
  } catch (T) {}
}
```

Fig. 9. Transitive poisoning: *throw T* poisons the first task, which poisons the second when the second attempts *recv i*. Finally the third is poisoned when it attempts *recv j* and the whole group terminates.

it turned out to be a huge challenge, despite there being fairly simple set of structural operational semantics rules for it.

The real complexity came from a combination of having to implement exception scope, which limits how far the poison propagates (it does not propagate outside the scope of the exception) and how that interacts with the scope of multiple, concurrently thrown exceptions.

6 Generating Threaded Code

To handle multiway rendezvous and exceptions on multiprocessors, we needed a new technique. Our next backend [16] generates C code that calls the POSIX thread library. Here, the challenge is minimizing overhead. Each communication action acquires the lock on a channel, checks whether every connected process had also blocked (whether the rendezvous could occur), and then checks if the channel is connected to a poisoned process (an exception had been thrown).

```
void h(chan int &A) {
  A = 4; send A;
  A = 2; send A;
}

void j(chan int A) throws Done {
  recv A;
  throw Done;
}

void f(chan int &A) throws Done {
  h(A); par j(A);
```

```
void g(chan int A) {
  recv A;
  recv A;
}

void main() {
  try {
    chan int A;
    f(A); par g(A);
  } catch (Done) {}
}
```

Fig. 10. A SHIM program with exceptions

6.1 An Example

We will use the example in Figure 10 to illustrate threaded code generation. There, the *main* function declares the integer channel *A* and passes it to tasks *f* and *g*, then *f* passes it to *h* and *j*. Tasks *f* and *h* send data with *send A* Tasks *g* and *j* receive it with *recv A*.

Task *h* sends the value four to tasks *g* and *j*. Task *h* blocks on the second *send A* because task *j* does not run a matching *recv A*.

As we described earlier, SHIM's exceptions enable a task to gracefully interrupt its concurrently running siblings. A sibling is "poisoned" by an exception only when it attempts to communicate with a task that raised an exception or with a poisoned task. For example, when *j* in Figure 10 throws *Done*, it interrupts *h*'s second *send A* and *g*'s seconds *recv A*, resulting in the death of *h* and *g*. An exception handler runs after all the tasks in its scope have terminated or been poisoned.

6.2 The Static Callgraph Assumption

For efficiency, our compiler assumes the communication and call graph of the program is known at compile time. We reject programs with unbounded recursion and can expand programs with bounded recursion [13], allowing us to transform the call graph into a call tree. This duplicates code to improve performance: fewer channel aspects are managed at run time.

We encode in a bit vector the subtree of functions connected to a channel. Since we know at compile time which functions can connect to each channel, we assign a unique bit to each function on a channel. We check these bits at run time with logical mask operations. In the code, something like A_f is a constant that holds the bit our compiler assigns to function f connected to channel A, such as 0x4.

```
lock(A.mutex);                         /* acquire lock for channel A */
A.blocked |= (A_h|A_f|A_main); /* block h and ancestors on A */
event_A();                             /* alert channel of the change */
while (A.blocked & A_h) {              /* while h remains blocked */
  if (A.poisoned & A_h) {              /* were we poisoned? */
    unlock(A.mutex);
    goto _poisoned;
  }
  wait(A.cond, A.mutex);               /* wait on channel A */
}
unlock(A.mutex);                       /* release lock for channel A */
```

Fig. 11. C code for *send A* in function *h()* of Figure 10

6.3 Implementing Rendezvous Communication

Implementing SHIM's multiway rendezvous communication with exceptions is the main code generation challenge.

The code at a send or receive is straightforward: it locks the channel, marks the function and its ancestors as blocked, calls the *event* function for the channel to attempt the communication, and blocks until communication has occurred. If it was poisoned, it branches to a handler. Figure 11 is the code for *send A* in *h* in Figure 10.

For each channel, our compiler generates an *event* function that manages communication. Our code calls an *event* function when the state of a channel changes, such as when a task blocks or connects to a channel.

Figure 12 shows the *event* function our compiler generates for channel *A* in Figure 10. While complex, the common case is quick: when the channel is not ready (one connected task is not blocked on the channel) and no task is poisoned, *A.connected != A.blocked* and *A.poisoned == 0* so the bodies of the two *if* statements are skipped.

If the channel is ready to communicate, *A.blocked == A.connected* so the body of the first *if* runs. This clears the channel *blocked = 0*) and *main*'s value for *A* (passed by reference to *f* and *h*) is copied to *g* or *j* if connected.

If at least one task connected to the channel has been poisoned, *A.poisoned != 0* so the body of the second *if* runs. This code comes from unrolling a recursive procedure at compile time, which is possible because we know the structure of the channel (i.e., which tasks connect to it). The speed of such code is a key advantage over a library.

This exception-propagation code attempts to determine which tasks, if any, connected to the channel should be poisoned. It does this by manipulating two bit vectors. A task *can_die* if and only if it is blocked on the channel and all its children connected to the channel (if any) also *can_die*. A *poisoned* task may *kill* its sibling tasks and their descendants. Finally, the code kills each task in the *kill* set that *can_die* and was not *poisoned* before by setting its *state* to POISON and updating the channel accordingly (*A.poisoned |= kill*).

```
void event_A() {
  unsigned int can_die = 0, kill = 0;
  if (A.connected == A.blocked) {          /* communicate */
    A.blocked = 0;
    if (A.connected & A_g) *A.g = *A.main;
    if (A.connected & A_j) *A.j = *A.main;
    broadcast(A.cond);
  } else if (A.poisoned) {                 /* propagate exceptions */
    can_die = blocked & (A_g|A_h|A_j); /* compute can_die set */
    if (can_die & (A_h|A_j) == A.connected & (A_h|A_j))
       can_die |= blocked & A_f;
    if (A.poisoned & (A_f|A_g)) {      /* compute kill set */
      kill |= A_g; if (can_die & A_f) kill |= (A_f|A_h|A_j);
    }
    if (A.poisoned & (A_h|A_j)) { kill |= A_h; kill |= A_j; }
    if ( kill &= can_die & ~A.poisoned) { /* poison some tasks? */
      unlock(A.mutex);
      if ( kill & A_g) { /* poison g if in kill set */
        lock(g.mutex);
        g.state = POISON;
        unlock(g.mutex);
      }
      /* also poison f, h, and j if in kill set ... */
      lock(A.mutex);
      A.poisoned |= kill ; broadcast(A.cond);
} } }
```

Fig. 12. C code for the *event* function for channel *A* of Figure 10

```
lock(main.mutex); main.state = POISON; unlock(main.mutex);
lock(f.mutex); f.state = POISON; unlock(f.mutex);
lock(j.mutex); j.state = POISON; unlock(j.mutex);
goto _poisoned;
```

Fig. 13. C code for *throw Done* in function *j()* of Figure 10

Code for throwing an exception (Figure 13) marks as POISON all its ancestors up to where it will be handled. Because the compiler knows the call tree, it knows how far to "unroll the stack," i.e., how many ancestors to poison.

6.4 Starting and Terminating Tasks

It is costly to create and destroy a POSIX thread because each has a separate stack, it requires interaction with the operating system's scheduler, and it usually requires a system call. To minimize this overhead, because we know the call graph of the program at compile time, our compiler generates code that creates at the beginning as many threads as the SHIM program will ever need. These threads are only destroyed when the SHIM program terminates; if a SHIM task terminates, its POSIX thread blocks until it is re-awakened.

```
lock(A.mutex); /* connect */
A.connected |= (A_f|A_g);
event_A();
unlock(A.mutex);

lock(main.mutex);
main.attached_children = 2;
unlock(main.mutex);

lock(f.mutex); /* pass args */
f.A = &A;
unlock(f.mutex);

/* A is dead on entry for g,
   so do not pass A to g */

lock(f.mutex); /* run f() */
f.state = RUN; broadcast(f.cond);
unlock(f.mutex);

lock(g.mutex); /* run g() */
g.state = RUN; broadcast(g.cond);
unlock(g.mutex);

lock(main.mutex); /* wait for children */
while (main.attached_children)
  wait(main.cond, main.mutex);
if (main.state == POISON) {
  unlock(main.mutex);
  goto _poisoned;
}
unlock(main.mutex);
```

Fig. 14. C code for calling *f()* and *g()* in *main()* of Figure 10

Figure 14 shows the code in *main* that runs *f* and *g* in parallel. It connects *f* and *g* to channel *A*, sets its number of live children to 2, passes function parameters, then starts *f* and *g*. The address for the pass-by-reference argument *A* is passed to *f*. Normally, a value for *A* would be passed to *g*, but our compiler found this value is not used so the copy is avoided (discussed below). After starting *f* and *g*, *main* waits for both children to return. Then *main* checks whether it was poisoned, and if so, branches to a handler.

Reciprocally, Figure 15 shows the code in *f* that controls its execution: an infinite loop that waits for *main*, its parent, to set its *state* field to running, at which point it copies its formal arguments into local variables and runs its body.

If a task terminates normally, it cleans up after itself. In Figure 15, task *f* disconnects from channel A, sets its *state* to STOP, and informs *main* it has one less running child.

By contrast, if a task is poisoned, it may still have children running and it may also have to poison sibling tasks so it cannot entirely disappear yet. In Figure 15, task *f*, if poisoned, does not disconnect from *A* but updates its *poisoned* field. Then, task *f* waits for its children to return. At this time, *f* can disconnect its (potentially poisoned) children from channels, since they can no longer poison siblings. Finally, *f* informs *main* it has one less running child.

For IBM's CELL processor, we developed a backend [17] that is a direct offshoot of the pthreads backend but allowed the user to assign certain (computationally intensive) tasks directly to the CELL's eight synergistic processing units (SPUs); the rest of the tasks ran on the CELL's standard PowerPC core (PPU). We did this by replacing these offloaded functions with wrapper functions that communicated across the PPU-SPU boundary. Function calls across the boundary turned out to be fairly technical because of data alignment restrictions on function arguments, which we would have preferred to be stack-resident. This, and many

```
int *A; /* value of channel A */

_restart:
lock(f.mutex);
while (f.state != RUN)
  wait(f.cond, f.mutex);
A = f.A; /* copy arg. */
unlock(f.mutex);


/* body of the f task */

_terminated:
lock(A.mutex); /* disconnect f */
A.connected &= ~A_f;
event_A();
unlock(A.mutex);

lock(f.mutex); /* stop */
f.state = STOP;
unlock(f.mutex);
goto _detach;
```

```
_poisoned:
lock(A.mutex); /* poison A */
A.poisoned |= A_f;
A.blocked &= ~A_f; event_A();
unlock(A.mutex);

lock(f.mutex); /* wait for children */
while (f.attached_children)
  wait(f.cond, f.mutex);
unlock(f.mutex);

lock(A.mutex); /* disconnect j, h */
A.connected &= ~(A_h|A_j);
A.poisoned &= ~(A_h|A_j);
event_A();
unlock(A.mutex);

_detach: /* detach from parent */
lock(main.mutex);
--main.attached_children;
broadcast(main.cond);
unlock(main.mutex);
goto _restart;
```

Fig. 15. C code in function *f()* controlling its execution

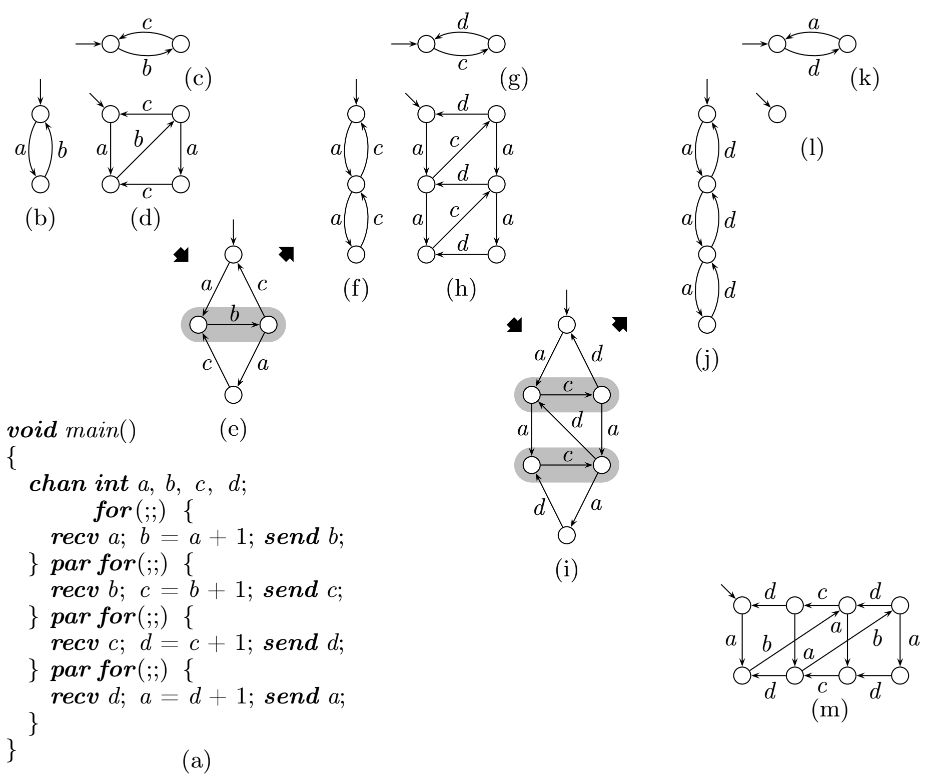
more fussy aspects of coding for the CELL, did convince us that language at a higher level than C is appropriate for such heterogeneous multicore processors.

7 Detecting Deadlocks

SHIM, although race free, is not immune to deadlocks. A simple example is { recv a; recv b; } par { send b; send a; }, which deadlocks because the first task is attempting to communicate on *a* first yet the second, which is also connected to *a*, excepts *b* to be first. Fortunately, SHIM's scheduling independence means that for a given input sequence, a SHIM program behaves the same for all executions of the program, and thus either always deadlocks or never deadlocks. In particular, SHIM does not need to be analyzed under an interleaved model of concurrency, rendering all the partial order reduction tricks of model checkers such as Holzmann's SPIN [1] unnecessary for SHIM.

Our first attempt at detecting deadlocks in SHIM [18] employs the symbolic synchronous model checker NuSMV [19]—an interesting choice since SHIM's concurrency model is fundamentally asynchronous. Our approach abstracts away data operations and chooses a specific schedule in which each communication event takes a single cycle. This reduces the SHIM program to a set of communicating state machines suitable for the NuSMV model checker.

We have since continued our work on deadlock detection in SHIM [20]. Here we take a compositional approach in which we build an automaton for a complete



```
void main()
{
  chan int a, b, c, d;
        for(;;) {
    recv a; b = a + 1; send b;
  } par for(;;) {
    recv b; c = b + 1; send c;
  } par for(;;) {
    recv c; d = c + 1; send d;
  } par for(;;) {
    recv d; a = d + 1; send a;
  }
}
```

Fig. 16. Analyzing a four-task SHIM program (a) for deadlock. Composing the automata for the first (b) and second (c) tasks gives a product automaton (d). Channel b only appears in the first two tasks, so we abstract away its effect by identifying (e) and merging (f) equivalent states. Next, we compose this simplified automaton with that for the third task (g) to produce another (h). Now, channel c will not appear again, so again we identify (i) and merge (j) states. Finally, we compose this with the automaton for the fourth task (k) to produce a single, deadlocked state (l) because the fourth task insists on communicating first on d but the other three communicate first on a. The direct composition of the first three tasks without removing channels (m) is larger—eight states.

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 brute-force symbolic model checker such as NuSMV.

Figure 16 shows our technique in action. Starting from the (contrived) program, we first abstract the behavior of the first two tasks into simple automata. The first task communicates on channel a, then on channel b, then repeats; the second task does the same on channels b and c. We compose these automata by allowing either to take a step on unshared channels but insisting on a rendezvous when a channel is shared. Then, since channel b is local to these two tasks, we abstract away its behavior by merging two states. This produces a simplified automaton that we then compose with the automaton for the third task. This time, channel c is local, so again we simplify the automaton and compose it with the automaton for the fourth task. The automaton we obtained for the first three tasks insists on communicating first on a then d; the fourth tasks communicates on d then a. This is a deadlock, which manifests itself as a state with no outgoing arcs.

For programs that follow such a pipeline pattern, the number of states grows exponentially with the number of pipeline stages (precisely, n stages produce 2^n states), yet our analysis only builds machines with $2n$ states before simplifying them to $n + 1$ states at each step. Although we still have to step through and analyze each of the n stages (leading to quadratic complexity), this is still a substantial improvement.

Of course, our technique cannot always reduce an exponential state space to a polynomial one, but we find it often did on the example programs we tried.

8 Sharing Buffers

We also applied the model-checking approach from the previous section to search for situations where buffer memory can be shared [21]. 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.

In the program in Figure 17, 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 them. 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 pattern and reports that a, b, and c can share buffers because their communications never overlap, thereby reducing the total buffer requirements by 66% for this program.

9 Conclusions

The central hypothesis of the SHIM project is that its simple, deterministic semantics helps both programming and automated program analysis. That we have been able to devise truly effective mechanisms for clever code generation (e.g., static scheduling) and analysis (e.g., deadlock detection) that can gain deep insight into the behavior of programs vindicates this view. The bottom line: if a programming language does not have simple semantics, it is really hard to analyze its programs quickly or precisely.

```
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
 }
}
```

Fig. 17. A SHIM program that illustrates the possibility of buffer sharing. Channels *a*, *b*, and *c* are never active simultaneously and can therefore share buffer space.

Algorithms where there is a large number of little, variable-sized, but independent pieces of work to be done do not mesh well with SHIM's scheduling-independent philosophy as it currently stands. The obvious way to handle this is to maintain a bucket of tasks and assign each task to a processor once it has finished its last task. The order in which the tasks is performed, therefore, depends on their relative execution rates, but this does not matter if the tasks are independent. It would be possible to add scheduling-independent task distribution and scheduling to SHIM (i.e., provided the tasks are truly independent or, equivalently, confluent); exactly how is an open research question.

Exceptions have been even more painful than multiway rendezvous. They are extremely convenient from a programming standpoint (e.g., SHIM's rudimentary I/O library wraps each program in an exception to allow it to terminate gracefully; virtually every compiler test case includes at least a single exception), but extremely difficult to both implement and reason about.

An alternative is to turn exceptions into syntactic sugar layered on the exception-free SHIM model. We always had this in the back our minds: an exception would just put a process into an unusual state where it would communicate its poisoned state to any process that attempts to communicate with it. The problem is that the complexity tends to grow quickly when multiple, concurrent exceptions and scopes are considered. Again, exactly how to translate exceptions into a simpler SHIM model remains an open question.

That buffering is mandatory for high-performance parallel applications is hardly a revelation; we confirmed it anyway. The SHIM model has always been able to implement FIFO buffers (e.g., Figure 6), but we have realized that they are sufficiently fundamental to be a first-class type in the language. We are currently working on a variant of the language that replaces pure rendezvous communication with bounded, buffered communication. Because it will be part of the language, it will be easier to map to unusual environments, such as the DMA mechanism for inter-core communication on the CELL processor.

SHIM has already been an inspiration for aspects of some other languages. We ported its communication model into the Haskell functional language [22] and proposed a compiler that would impose its scheduling-independent view of the work on arbitrary programs [23]. Certain SHIM ideas, such as scheduling analysis [24], have also been used in IBM's X10 language.

Acknowledgments

Many have contributed to SHIM. Olivier Tardieu created the formal semantics, devised the exception mechanism, and instigated endless (constructive) arguments. Jia Zeng developed the static recursion removal algorithm. Baolin Shao designed the compositional deadlock detection algorithm. The NSF has supported the SHIM project under grant 0614799.

References

1. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering **23**(5) (May 1997) 279–294
2. Kahn, G.: The semantics of a simple language for parallel programming. In: Information Processing 74: Proceedings of IFIP Congress 74, Stockholm, Sweden, North-Holland (August 1974) 471–475
3. Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM **21**(8) (August 1978) 666–677
4. Edwards, S.A.: Experiences teaching an FPGA-based embedded systems class. In: Proceedings of the Workshop on Embedded Systems Education (WESE), Jersey City, New Jersey (September 2005) 52–58
5. Edwards, S.A.: SHIM: A language for hardware/software integration. In: Proceedings of Synchronous Languages, Applications, and Programming (SLAP). Electronic Notes in Theoretical Computer Science, Edinburgh, Scotland (April 2005)
6. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Guernic, P.L., de Simone, R.: The synchronous languages 12 years later. Proceedings of the IEEE **91**(1) (January 2003) 64–83
7. Buck, J.T.: Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model. PhD thesis, University of California, Berkeley (1993) Available as UCB/ERL M93/69.
8. Parks, T.M.: Bounded Scheduling of Process Networks. PhD thesis, University of California, Berkeley (1995) Available as UCB/ERL M95/105.
9. Edwards, S.A., Tardieu, O.: Efficient code generation from SHIM models. In: Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES), Ottawa, Canada (June 2006) 125–134

10. Edwards, S.A., Tardieu, O.: Shim: A deterministic model for heterogeneous embedded systems. In: Proceedings of the International Conference on Embedded Software (Emsoft), Jersey City, New Jersey (September 2005) 37–44
11. Edwards, S.A., Tardieu, O.: Shim: A deterministic model for heterogeneous embedded systems. ieee Transactions on Very Large Scale Integration (vlsi) Systems **14**(8) (August 2006) 854–867
12. Tardieu, O., Edwards, S.A.: R-shim: Deterministic concurrency with recursion and shared variables. In: Proceedings of the International Conference on Formal Methods and Models for Codesign (memocode), Napa, California (July 2006) 202
13. Edwards, S.A., Zeng, J.: Static elaboration of recursion for concurrent software. In: Proceedings of the Workshop on Partial Evaluation and Program Manipulation (pepm), San Francisco, California (January 2008) 71–80
14. Tardieu, O., Edwards, S.A.: Scheduling-independent threads and exceptions in shim. In: Proceedings of the International Conference on Embedded Software (Emsoft), Seoul, Korea (October 2006) 142–151
15. Berry, G., Gonthier, G.: The Esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming **19**(2) (November 1992) 87–152
16. Edwards, S.A., Vasudevan, N., Tardieu, O.: Programming shared memory multiprocessors with deterministic message-passing concurrency: Compiling shim to Pthreads. In: Proceedings of Design, Automation, and Test in Europe (date), Munich, Germany (March 2008) 1498–1503
17. Vasudevan, N., Edwards, S.A.: Celling shim: Compiling deterministic concurrency to a heterogeneous multicore. In: Proceedings of the Symposium on Applied Computing (sac). Volume III., Honolulu, Hawaii (March 2009) 1626–1631
18. Vasudevan, N., 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) 49–58
19. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: 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., Copenhagen, Denmark (July 2002) 359–364
20. Shao, B., Vasudevan, N., Edwards, S.A.: Compositional deadlock detection for rendezvous communication. In: Proceedings of the International Conference on Embedded Software (Emsoft), Grenoble, France (October 2009)
21. Vasudevan, N., Edwards, S.A.: Buffer sharing in csp-like programs. In: Proceedings of the International Conference on Formal Methods and Models for Codesign (memocode), Cambridge, Massachusetts (July 2009)
22. Vasudevan, N., Singh, S., Edwards, S.A.: A deterministic multi-way rendezvous library for Haskell. In: Proceedings of the International Parallel and Distributed Processing Symposium (ipdps), Miami, Florida (April 2008) 1–12
23. Vasudevan, N., Edwards, S.A.: A determinizing compiler. In: Proceedings of Program Language Design and Implementation (pldi), Dublin, Ireland (June 2009)
24. Vasudevan, N., Tardieu, O., Dolby, J., Edwards, S.A.: Compile-time analysis and specialization of clocks in concurrent programs. In: Proceedings of Compiler Construction (cc). Volume 5501 of Lecture Notes in Computer Science., York, United Kingdom (March 2009) 48–62