View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Columbia University Academic Commons

Efficient Code Generation from SHIM Models

Stephen A. Edwards and Olivier Tardieu
Columbia University, Department of Computer Science
{sedwards,tardieu}@cs.columbia.edu

Abstract

Programming concurrent systems is substantially more difficult than programming sequential systems, yet most embedded systems need concurrency. We believe this should be addressed through higher-level models of concurrency that eliminate many of the usual challenges, such as nondeterminism arising from races.

The SHIM model of computation provides deterministic concurrency, and there already exist ways of implementing it in hardware and software. In this work, we describe how to produce more efficient C code from SHIM systems.

We propose two techniques: a largely mechanical one that produces tail-recursive code for simulating concurrency, and a more clever one that statically analyzes the communication pattern of multiple processes to produce code with far less overhead. Experimentally, we find our tail-recursive technique produces code that runs roughly twice as fast as a baseline; our statically-scheduled code can run up to twelve times faster.

Categories and Subject Descriptors
D.3.4 [*Programming Languages–Processors*]: Compilers

General Terms Algorithms, Performance

Keywords Concurrency, Embedded Systems, Computed gotos, Code Synthesis, The SHIM model

1. Introduction

To improve embedded system programmer productivity, we need higher-level, less error-prone programming models. Concurrency, a common part of most embedded systems, is a particularly pernicious challenge to most programmers, yet even modern programming languages such as Java remain stuck in 1960's-era shared memory, locks, and threads.

We believe the way forward involves more domain-specific models of computation tailored for embedded systems. Our SHIM (Software/Hardware Integration Medium) language and model of computation [5], which provides deterministic (i.e., race-free) concurrency, is an example of this.

In this work, we address the challenge of producing efficient sequential C code from a concurrent SHIM model. While at first it may seem odd to use a concurrent programming language to be executed on a single processor, there are a number of reasons to consider doing so.

Our technique assists design-space exploration, where it can generate code for single or multiple processors. Balancing the load on multiple processors is necessary for the highest performance, meaning that it may be necessary to redistribute the processing load across them. To do this efficiently, we advocate describing a system with the finest concurrency possible to make it easy to move small processes from one processor to another to load-balance. Our static scheduling technique can remove most of the overhead due to simulating concurrency on a single processor. Fine-grained concurrency also helps if the number of processors is increased: the system can easily be split into more pieces.

Simulating an eventual hardware implementation is another application of our technique. The SHIM semantics are amenable to hardware synthesis (we describe a basic technique elsewhere [5]), and we plan an automatic hardware/software synthesis mechanism. Simulating a system is almost always part of the development process, and our techniques can be used for this.

We present two things: a technique for producing code that simulates SHIM-style concurrency without operating system support (we synthesize purely ANSI C), and a static scheduling procedure that can compile together arbitrary groups of processes for efficiency. We produce C for portability and to take advantage of the many low-level optimizations provided by good C compilers.

After reviewing related work, we describe the SHIM model of computation and the small language we have devised for it (Section 3), present our basic code generation technique that produces tail-recursive ANSI C code that simulates the concurrency in SHIM (Section 4), then present our static scheduling technique that is able to combine a group of concurrently-running processes into a single one that generally runs faster (Sections 5 and 6). We conclude with experimental results that show our static scheduling technique can produce code that runs as much as twelve times faster than a baseline translation of SHIM (Section 7).

2. Related Work

Ours is not the first compilation technique that generates statically-scheduled code from a concurrent formalism. Table 1 lists related techniques and how they compare to ours. A key difference is the model of computation being compiled. SHIM is an asynchronous model with rendezvous-style communication. Others use a synchronous model (Esterel, Lustre), FIFOs (Sgroi et al.), and shared memory (Phantom). See Edwards's survey [4] for more detail.

Another distinguishing feature of our work is its ability to statically compile arbitrary sets of processes in a system (the *Partial* column in Table 1), a very useful attribute because our compiler can be applied to very large systems once they are partitioned into pieces small enough to be compiled statically, and our technique does not place restrictions on which partitions are legal; they may be chosen to optimize code size or speed.

Another difference is our choice of how to abstract the state of the system during compilation. Like others, our technique trades

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

LCTES'06 June 14–16, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-362-X/06/0006. . . $5.00.

state tracked at compile time, which reduces execution time at the expense of potentially exponential code size, and state tracked at runtime, which is exactly the opposite. We consider our approach superior because empirically it appears to produce small automata that nevertheless lead to faster code.

Our code-generation procedure (i.e., that does not perform static scheduling) resembles Zhu and Lin's [14], which generates code for SHIM-like process in isolation and then runs them using a simple round-robin scheduler. Ours has a more efficient scheduler that uses a stack of pending function and tail-recursive calls.

Lin [11] proposes a static scheduling technique for generating code from a SHIM-like model. He translates his language into a Petri net, unrolls it, breaks it into loop iteration fragments, and generates code for each (concurrent) fragment. This can produce exponentially-large code for a single fragment and an exponential number of fragments. We only model communication and produce sequential fragments.

Cardelli and Pike's Squeak [3] takes a similar approach with a similar formalism: they use a CSP-like language and consider all possible control paths, generating a large automaton as a result. Their model does consider communication from the environment, but is not able to compile arbitrary portions of the system.

Sgroi et al. [13] synthesize embedded software from a richer model than SHIM that allows unbounded buffers. While this is more convenient for certain specifications, it makes the scheduling problem much harder and raises the question of whether a system can be executed at all (SHIM systems never need unbounded buffers), and if so, what buffer sizes are necessary.

The synchronous languages community [1] has developed techniques for generating statically-scheduled code from concurrent formalisms with synchronous communication. Like our technique, the first Esterel compilers by Berry and Gonthier [2] compile a concurrent language into a single, often large automaton. They also abstract much of the state of a program to simplify the generated automaton, but they track control state, not communication as we do. Furthermore, unlike SHIM, the synchronous model of Esterel has difficulties with composition; no one has yet developed a technique for partial static scheduling of Esterel.

Halbwachs et al. [9] generate automata from Lustre [8] using a technique like that of early Esterel compilers. However, they only track the values of Boolean variables instead of communication or control state (Lustre, unlike Esterel, has no explicit control state). Like Esterel, Lustre's model of computation has difficulties with composition so it is not clear how to compile it in pieces.

French et al. [7] generates code from the more-complicated Verilog discrete-event language. Verilog semantics employ an ordered event queue, which could easily produce complex automata if tracked precisely. Instead, they abstract virtually all of the state of a Verilog program, always building an automaton consisting of a single loop. This works because most Verilog programs model synchronous logic with a periodic clock, but it also greatly limits the amount of computation that can be performed statically.

Nácul and Givargis's Phantom [12] compiles standard C programs written to a subset of the POSIX API that uses semaphores and shared memory, a nondeterministic asynchronous formalism.

Table 1. A comparison of other automata-generating compilers

Compiler	MoC	Partial	State Tracked
Ours	Async. Rendez.	yes	Communication
Zhu and Lin [14]	Async. Rendez.	yes	none
Lin [11]	Async. Rendez.	no	Comm. & Control
Squeak [3]	Async. Rendez.	no	Comm. & Control
Sgroi et al. [13]	Async. FIFOs	no	Comm. & Control
Esterel V3 [2]	Synchronous	no	Control
Lustre [9]	Synchronous	no	Boolean vars.
Verilog [7]	Async. Events	no	Communication
Phantom [12]	Async. mutex	no	Control

3. The SHIM Model and Language

In the SHIM model [5], a system consists of concurrently-running sequential processes that communicate exclusively through fixed, point-to-point communication channels with rendezvous. We have devised an imperative language with C-like syntax for describing SHIM systems. Figure 3(a) shows a simple example. Each process has local variables; there are no global variables. All processes execute concurrently.

Inter-process communication is synchronous: both sending and receiving processes must agree on when data is transferred; one always waits for the other. A process's formal arguments are input and output channels. Each appearance of a channel's name becomes a write operation if it appears on the left of an assignment and a read otherwise.

The topology of communication channels and the number of processes is fixed and each communication channel connects one writing process to one reader. The communication structure of a system is therefore a directed graph whose nodes are processes and whose arcs are channels. The graph may contain cycles.

Below, we describe the small imperative language we developed that embodies the SHIM model. It is to this language that we apply the compilation algorithms described in this paper.

3.1 Syntax

A SHIM program consists of a sequence of three kinds of top-level declarations: *struct*, *process*, and *network*.

Structs. Struct declarations are C-like type declarations. Current variable types in SHIM are Booleans, fixed-size signed and unsigned integers, structs, and arrays. Booleans are no different from 1-bit unsigned integers. There are no pointer types in SHIM.

```
struct s {
  bool b;          Boolean
  int32 i;         signed 32-bit integer (including sign bit)
  uint16 t[24];    array of 24 unsigned 16-bit integers
};
```

Processes. A process declaration looks like a C function declaration and contains imperative code that runs sequentially. It is introduced by the *process* keyword followed by the name of the process, the formal arguments of the process between parentheses, and the body of the process delimited with curly braces. E.g.,

```
process xor(int8 I, int8 J, int8 &O) { O = I^J; }
```

The formal arguments of a process are its ports. Mimicking the syntax of C++ pass-by-reference parameters, in SHIM, the & indicates an output port (i.e., a channel that may only be written; input ports may only be read). In this example, therefore, `I` and `J` are input ports and `O` is an output port.

The body of a process consists of regular C code. Currently, we support a subset of C: *if-else* and *switch-case-default* conditionals, *while* and *for* loops (including *break* and *continue*), *label* and *goto* statements, expressions (including assignments), block statements and local variable declarations.

Expressions may mix local variable names and port names freely. However, output ports cannot be read and input ports cannot be written (i.e., appear in an l-value position). In particular, no port can be incremented or decremented. Atomic assignments between structs or arrays are supported.

Networks. A network declaration, which instantiates a set of processes or subnetworks, is introduced by the *network* keyword followed by the name, the formal arguments, and the body of the network. The formal arguments are the ports of the network. The body of a network consists in a list of local channel declarations followed by a list of process and network instances.

```
network xor2(int8 I, int8 J, int8 K, int8 &O) {
  int8 X;
  xor(X/O);
  xor(X/I, K/J);
}
```

Local channels connect one process (or subnetwork) in a network to another process (or subnetwork) in the same network. The types of ports connected through a channel must match, i.e., an output port of type t may only be connected to an input port of type t. Local channel declarations resemble local variable declarations.

The ports of the network come from ports of processes (or subnetworks) of the network that have no matching reading or writing process within the network. Port declarations for networks are no different from port declarations for processes. Port and local channel declarations in networks may be omitted as they are inferred by the compiler.

An instance resembles a function call. It consists of the name of a process or network followed by a list of actual arguments and a semicolon. The syntax for arguments associates formal and actual ports by name instead of position. For example, "`xor(X/I, K/J);`" instantiates process `xor` with actual ports "(`int8 X, int8 K, int8 &O`)", i.e., port name `X` is substituted for name `I`, name `K` for name `J`, whereas name `O` is left unchanged.

3.2 Semantics

A SHIM system is a hierarchical network of communicating sequential processes running concurrently.

Sequential behavior. Apart from communications, a process runs like its equivalent C function, although we adopt Java's semantics for expression evaluation in SHIM to achieve determinism.

If channel names appear in an expression, communications take place during the evaluation of that expression. Accesses to local variables are compiled into loads and stores; accesses to channels are compiled into reads and writes.

E.g., the assignment "`O = I ^ J;`" involving ports `int8 I`, `int8 J`, and `int8 &O`, consists of receiving one value on input channel `I`, receiving one value on `J`, combining the two values, and sending the result on output channel `O` in exactly that order.

Concurrency. Processes run concurrently and asynchronously unless they engage in communication. Reads and writes are blocking, that is, they behave as a synchronization barrier between reading and writing processes.

If a process reaches a write (respectively read) operation on a channel, it will block until the process at the other end of the channel reaches a read (respectively write) operation for the channel. Then one value gets transmitted from the writing process to the reading process and each process resumes executing. Note that a process may wait forever on a channel if the process on the other end never again attempts to communicate on the channel.

Scheduling and determinism. We have shown [5] that the behavior of a SHIM system does not depend on the scheduling algorithm if preemptive and fair in Kahn's sense [10]: the sequence of values transmitted on each channel, whether finite or infinite, is deterministic. This property allows us to choose scheduling policies based on efficiency rather than correctness criteria.

4. Tail-Recursive Code Generation

In this section, we present a code generation technique able to translate systems written in our SHIM language into reasonably efficient ANSI C that simulates concurrency using tail-recursive calls to function pointers.

Minimizing the time spent deciding which tasks to invoke and invoking them (scheduling overhead) is the main challenge in efficiently simulating concurrency on a single-threaded processor. A traditional multi-tasking operating system typically sets the context-switching frequency to a relatively low 100 times per second. Such an approach is unsuitable for SHIM systems with processes that must synchronize much more frequently.

Our approach uses an extremely simple scheduler—a stack of function pointers—that invokes fragments of concurrently-running processes using tail-recursion. At first glance, the danger of exhausting stack space appears to make recursion impractical. However, optimizing C compilers, such as recent versions of *gcc* running with `-O2`, can perform tail-recursion optimization that changes tail-recursive calls to jumps that do not consume stack space. The result is a practical, efficient way to jump among functions.

In some sense we are using function pointers and tail recursion as a way to perform computed *goto*s in ANSI C. While certain dialects of C (notably gcc's) provide true computed *goto*s, we wanted to make our code standards-compliant. *Switch* statements are another alternative, which we used previously to generate code from SHIM [6]. However, experimental results suggest tail-recursion is typically twice as fast as using *switch* (Table 2).

We translate the code for each process into a collection of functions. 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 somewhat wasteful to keep a function pointer per channel to remember where a process is to resume. In Section 5, however, we will need to relax the block-on-single-channel restriction to accommodate code that mimicks 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 1 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.

```
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;
}
```

```
void source_0(void);
void source_1(void);

void source_0() {
  source.tmp1 = 42;
  C_value = source.tmp1;        /* write to channel buffer */
  if (sink.blocked &&                 /* if reader blocked */
      C.reader) {
    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() {
  (*(--sp))(); return;
}
```

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

```
void sink_0(void);
void sink_1(void);

void sink_0() {
  if (source.blocked &&               /* if writer blocked */
      C.writer) {
    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 */
}

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;
}
```

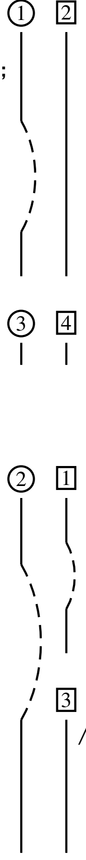


Figure 1. Synthesized code for two processes (in the boxes) that communicate and the `main()` function that schedules them.

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 others; there is no central scheduler code.

SHIM uses blocking rendezvous-style communication through point-to-point channels. This means that 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 1 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, records that when it is finally unblocked that control should continue at the *source_1* function (this is the purpose of writing to *C.writer*), and finally pops the next waiting process function from the stack and calls it.

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

4.1 Generating Code

The main challenge in generating the code described above is identifying the process function boundaries. We use a variant of extended basic blocks: 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 2 depicts the code generation process for a simple process with some interesting control-flow. The process (Figure 2a) consists of two nested loops. We translate the SHIM code into a

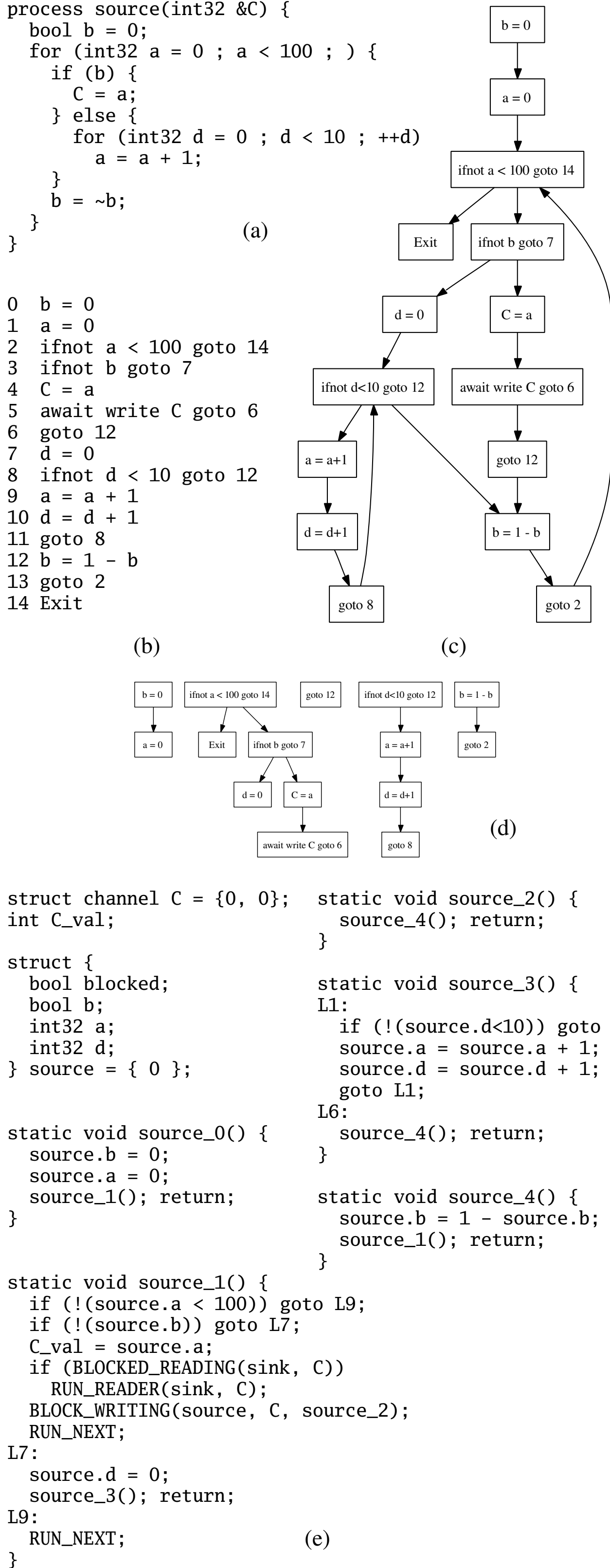


Figure 2. Generating tail-recursive code for a single process. Our compiler translates a process (a) into an intermediate representation (b). This is translated into a CFG (c), split into extended basic blocks (d), and each block becomes a function (e).

fairly standard linear IR (Figure 2b). 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 simple SHIM processes) only ever blocks on a single channel at a time, our static scheduling procedure (Section 5) 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.

5. Statically Scheduling SHIM Networks

In Section 4, we showed how to translate a network of concurrently-running SHIM processes into C code that simulates the concurrency using tail-recursive calls to function pointers. In this section, we describe how to compile together groups of concurrently-running processes into a single imperative process that we then substitute for the group of processes. We synthesize the entire system using the technique described in the previous section.

The advantage of compiling together a group of processs is run-time efficiency: 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 [4], 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 tradeoff.

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 complete system states, it would be impractically large for all but the simplest systems. Instead we track an abstract version of the system state in the automaton. While this does defer many computations to when the 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 produces impressive speed-ups (as much as 12×; see Section 7).

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 the set of runnable processes and the state of each channel, either 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, we do track what program counter values are possible in each state to streamline the generated code and reduces 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 state but with additional program counter possibilities, 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.

5.1 Example: A Closed Subsystem

Statically scheduling a group of processes that do not communicate with the outside world is the simpler case. Figure 3 shows such a simple system being transformed into an automaton. The system's three processes (Figure 3(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 3(b)) that are simulated to produce an automaton (Figure 3(c)). Our compiler then generates code for each state in the automaton and connects them with *gotos*, producing the IR in Figure 3(d). This IR is then passed to the normal code generation procedure described in Section 4 to produce executable C.

The structure of Figure 3(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 the process reaching a deadlock because the source has terminated.

Each state in Figure 3(c) is labeled with its name; the set of runnable processes (marked "+" when runnable, "-" otherwise) when the state begins; the state of each channel ("-" for clear, "R" when a reader is blocked on it, and "W" when a 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, 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 instruction 0, 2, 4, 6, or 8.

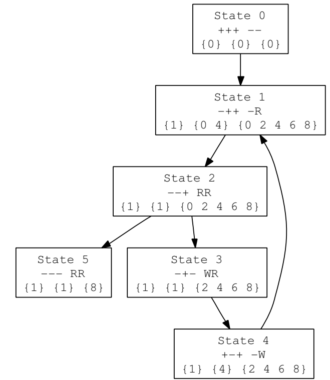
A theorem says that a SHIM system runs consistently under any reasonable scheduling policy [5]. Ours selects the lowest-numbered runnable process. The automaton we generate, therefore, depends on process labeling (currently from positions in the source file), but it is guaranteed to produce the same overall behavior. A better scheduling policy could improve the generated code.

```
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 IR generated from the automaton

Figure 3. 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 4 to be translated into C.

The automaton generation procedure starts with all processes runnable and all program counters at 0—State 0 in Figure 3(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 3(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. This shows how our choice of signature reduces the size of the automaton.

State 2 in Figure 3(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 3(d) is the IR generated from the automaton in Figure 3(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 [6]. The mechanism is analogous to the tail-recursive calls to function pointers described in Section 4, but keeps the code together.

5.2 Example: An Open Subsystem

It is only slightly more difficult to statically schedule a group of processes that communicate with their environment. Figure 4 shows how to perform static scheduling on a pair of buffers that only communicate with their environment, not each other. This illustrates the need for process generated from these two processes to be able to block on two channels simultaneously since the pair of buffers running concurrently has this ability. The tail-recursive code generation technique we described in Section 4 has this ability, but it goes unused when translating SHIM processes not constructed by static scheduling.

As in Figure 3, the SHIM code in Figure 4(a) is translated into the linear IR in Figure 4(b). This is simulated to produce the automaton in Figure 4(c), and finally the automaton is translated into the linear IR in Figure 4(d). Here, however, the automaton includes states where it (the two buffer processes) are blocked waiting to communicate with the environment (States 2, 4, 6, and 8).

As shown in Figure 4(d), the code for such states is simple: an *await* statement that blocks on multiple channels. The C code generated for such a state simply makes multiple *BLOCK_READING* and *BLOCK_WRITING* calls to indicate that the process is waiting on multiple channels simultaneously. One process in the environment will communicate first with this process and thus send control to the state in the automaton that assumes that particular communication takes place. The environment is free to choose which channel it communicates on first, but the SHIM semantics guarantee that the overall system behavior is consistent regardless of the choice.

The C code for State 4—the two-channel *await*—is

```
if (BLOCKED_READING(sink, C1))
  RUN_READER(sink, C1);
BLOCK_WRITING(twobuffers, C1, twobuffers_1);
if (BLOCKED_WRITING(source, C2)) {
  twobuffers_13(); return;
}
BLOCK_READING(twobuffers, C2, twobuffers_13);
RUN_NEXT;
```

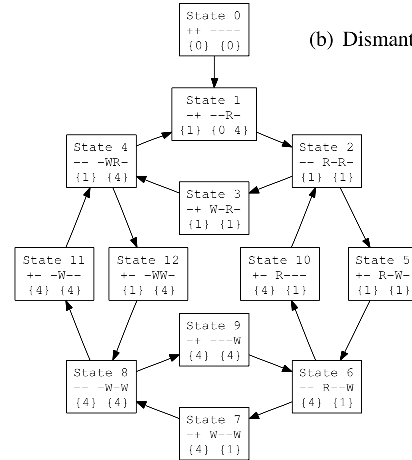
"Twobuffers" is the process generated from the automaton, "sink" reads channel 1, and "source" writes to channel 2.

```
process buffer(int32 &O,
               int32 I) {
  for (;;) O = I;
}

network twobuffers () {
  buffer(C2/I, C3/O);
  buffer(C0/I, C1/O);
}
```

(a) SHIM code

```
Process 0
0 PreRead 2
1 PostRead 2 tmp4
2 tmp3 = tmp4
3 Write 3 tmp3
4 goto 0

Process 1
0 PreRead 0
1 PostRead 0 tmp2
2 tmp1 = tmp2
3 Write 1 tmp1
4 goto 0
```

(b) Dismantled



(c) The automaton

```
0 /* State 0 (0) */
1 state0 = 1
2 goto 3

3 /* State 1 (1) */
4 switch state1
   case 0 goto 8
   case 4 goto 7
5 state1 = 1
6 goto 9
7 goto 5
8 goto 5

9 /* State 2 (blocked) */
10 await
    read 2 goto 20
    read 0 goto 11

11 /* State 3 (1) */
12 tmp2 = channel_0
13 received 0 in tmp2
14 tmp1 = tmp2
15 channel_1 = tmp1
16 state1 = 4
17 goto 18

18 /* State 4 (blocked) */
19 await
    read 2 goto 47
    write 1 goto 3

20 /* State 5 (0) */
21 tmp4 = channel_2
22 received 2 in tmp4
23 tmp3 = tmp4
24 channel_3 = tmp3
25 state0 = 4
26 goto 27

27 /* State 6 (blocked) */
28 await
     read 0 goto 29
     write 3 goto 41

29 /* State 7 (1) */
30 tmp2 = channel_0
31 received 0 in tmp2
32 tmp1 = tmp2
33 channel_1 = tmp1
34 state1 = 4
35 goto 36

36 /* State 8 (blocked) */
37 await
    write 3 goto 44
    write 1 goto 38

38 /* State 9 (1) */
39 state1 = 1
40 goto 27

41 /* State 10 (0) */
42 state0 = 1
43 goto 9

44 /* State 11 (0) */
45 state0 = 1
46 goto 18

47 /* State 12 (0) */
48 tmp4 = channel_2
49 received 2 in tmp4
50 tmp3 = tmp4
51 channel_3 = tmp3
52 state0 = 4
53 goto 36
```

(d) The IR generated from the automaton

Figure 4. Statically scheduling a pair of buffer processes.

```
1:  procedure visit(state s, PC sets I[1], ..., I[n])
2:    if state s is not part of the automaton then
3:      add state s to the automaton
4:      let s_p be the scheduled process, if any, for state s
5:      let s_I[p] = { ∅     if p = s_p
                     { I[p]  for all other processes
6:    let new PCs N[p] = I[i] − s_I[p] for all processes p
7:    let known PCs s_I[p] = s_I[p] ∪ N[p] for all processes p
8:    if there is a scheduled process s_p then
9:      clear R, the set of newly-reached state/PC pairs
10:     for each new PC i ∈ N[s_p], the scheduled process do
11:       R = R ∪ simulate(s_p, s, i)
12:     for each (s', i') ∈ R do
13:       add s' to the set of successors of s
14:     for each (s', i') ∈ R do
15:       visit(s', s_I[1], ..., s_I[s_p − 1], {i'}, s_I[s_p + 1], ...)
16:   else
17:     R' = simulate-blocked(s)
18:     add each successor in R' to the set of successors of s
19:     for each s' ∈ R' do
20:       visit(s', s_I[1], ..., s_I[n])
21:   if N[p] ≠ ∅ for a non-scheduled process p ≠ s_p then
22:     for each successor state s' of s do
23:       visit(s', N[1], ..., N[s_p − 1], ∅, N[s_p + 1], ..., N[n])
```

Figure 5. The DFS-based automaton construction algorithm.

The regularity of the automaton in figreftwobuffer-automaton is no accident since it arises from two identical, non-interacting processes running in parallel. The main part of the machine (i.e., excluding the initial state at the top) is arranged such that horizontal transitions correspond to communication with the second buffer and vertical transitions correspond to the first. The four corner states (2, 4, 6, and 8) correspond to the four cases in which both processes are waiting for input from the environment (i.e., both buffers waiting to read; first waiting to read, second waiting to write; first waiting to write, second waiting to read; and both waiting to write). In each of these states, the environment is free to communicate with either process, corresponding to the two outgoing arcs for each. Half of the states between these correspond to each buffer copying its input to its output (3, 5, 7, and 12); the other half correspond to each buffer finishing a write and preparing to read (1, 5, 9, and 11).

As this example suggests, statically scheduling groups of non-interactive processes can cause an exponential explosion in the number of states (our scheduling procedure effectively computes the cross-product of the processes). However, interesting subnetworks are connected and generate far fewer states.

6. The Automaton Construction Algorithm

Here, we describe in detail the automaton construction algorithm (Figure 5) that is at the core of our static scheduling procedure. It uses a variant of depth-first search that visits a state, computes its successors, and visits them until no new states are found.

Unlike traditional depth-first search, this may visit each state many times (at least once for every newly-reached program counter for the process). Since the successors depend on what code the scheduled process can run, every newly-discovered entry point may add successors. To limit this procedure, the algorithm only simulates newly-discovered entry points.

This algorithm makes a subtle but obvious choice to avoid introducing erroneous deadlocks. The generated automata must be "as responsive" as the original subnetwork, meaning that after some sequence of allowed communications, the generated automaton must be willing to engage in as many communications as the original subnetwork. If the automaton were lazy in the sense that it did not always make forward progress (i.e., run internal processes) where it could, the overall system might deadlock waiting for the automaton to do something that the original system would.

The solution is simple: the automaton runs as many processes as it can and only blocks when all of its processes have blocked.

Visit (Figure 5) takes a state s and a set of program counters $I[k]$ for each process. It first checks whether the state is new (line 2) and if so, adds it (line 3), determines which process it will run (i.e., makes the scheduling choice, line 4), and initializes the set of known program counter values for each process to be those passed to the procedure for all but the running process, which is initialized to the empty set (line 5).

By line 6, state s is part of the automaton, so the visit procedure calculates which PC values are new for this state and adds these to this state's set of PCs (line 7).

Next, there are two possibilities. Either there is some scheduled process s_p that can run (handled by lines 9–15), or every process in the subnetwork is blocked or terminated (handled by lines 17–20).

If there is some scheduled process, we simulate it for each new PC value (lines 9–11) using Figure 6. This returns a set of PC/state pairs that are added to the set of successors of s (line 13).

The procedure then recurses on each new state/PC pair, passing all known PCs for each process except the scheduled one, for which it passes the newly-discovered PC (lines 14–15).

Conversely, if there is no process that can run, we instead call simulate-blocked (Figure 7) to determine the successors of s (line 17), then visit each of these successors (lines 19–20).

Finally, if new PC values were found for any non-scheduled processes, they are passed to every successor state (lines 21–23). This is because the PCs for unscheduled processes do not change when the scheduled process runs.

6.1 Symbolic Simulation of a Process

Automaton construction calls the symbolic simulation procedure (Figure 6) to compute state successors. Given a process, an initial state, and an initial program counter for the process, *simulate* returns a set of state/PC pairs that are the context switching points the process can reach.

It computes successor states with a depth-first exploration of a process's control-flow graph. The *visit-instruction* procedure (lines 2–28 in Figure 6) performs the search.

Visit-instruction first checks whether the state/PC pair has been visited (line 3). Just checking whether a particular instruction has been visited is not enough because different paths through the process may produce different states at the same instruction.

After marking the state/PC visited, it either terminates or recurses depending on the type of instruction. Non-communication instructions (lines 6–13) are simple: a straight-line instruction recurses on its successor, and a branch instructions recurses on all of its successors. When control reaches the end of a process, it marks the process not running and adds a terminal state.

There are two cases for the first half of a read (lines 14–19). If the writer is blocked on the channel, the data is available and the read operation can proceed—the procedure recurses (line 16). Otherwise, data is not available so the process blocks on the channel and adds its successor as a final state (line 19).

Handling the second half of a read operation (lines 20–22) is easier. Here, the writer must have blocked on the channel, so we mark it as runnable and recurse on the next instruction.

The final case, writing (lines 23–27), always blocks, so it returns its successor as a final state, but if the reader is blocked on the channel, the write makes it runnable (line 25).

```
1: function simulate(process p, state s, PC i)
2:   procedure visit-instruction(s, i)
3:     if (s, i) has not been visited then
4:       mark (s, i) as visited
5:       case type of instruction at PC i in process p of
6:       normal instruction :
7:         visit-instruction(s, i+1)
8:       branch instruction :
9:         for each successor i′ of the branch do
10:          visit-instruction(s, i′)
11:      end-of-process :
12:        mark p as not runnable
13:        F = F ∪ (s, i)
14:      read on channel c :
15:        if the writer is blocked on channel c then
16:          visit-instruction(s, i+1)
17:        else
18:          mark reader p as blocked on channel c
19:          F = F ∪ (s, i+1)
20:      just after a read on channel c :
21:        mark as runnable the writer of channel c
22:        visit-instruction(s, i+1)
23:      write on channel c :
24:        if the reader is waiting on channel c then
25:          mark the reader process as runnable
26:        mark writer p as blocked on channel c
27:        F = F ∪ (s, i+1)
28:  end procedure
29:  clear the set of visited state/pc pairs
30:  clear F, the set of final state/pc pairs
31:  visit-instruction(s, i)
32:  return F
```

Figure 6. The symbolic simulation procedure.

```
1: function simulate-blocked(state s)
2:   clear S, the set of successors for s
3:   for each external channel c do
4:     let s′ = s
5:     if c is blocked reading in s then
6:       set the reader of c to be runnable in s′
7:       set c to be blocked writing in s′
8:       add s′ to S
9:     else if c is blocked writing in s then
10:      set the writer of c to be runnable in s′
11:      set c to be blocked reading in s′
12:      add s′ to S
13:  return S
```

Figure 7. Computing the successors of a blocked state. This assumes the environment communicates on every blocked channel.

6.2 Handling Blocked States

The function for handling blocked states (Figure 7) is simpler. By definition, no process in the subnetwork is runnable in such a state, so the only possible way to make progress is for the environment to communicate with one of the blocked processes.

Simulate-blocked assumes the environment can and does communicate on every channel on which the processes are blocked. As explained in Section 5.2, the code generated for such a state is simply an *await* statement that blocks on every external channel on which processes in the subnetwork are blocked. When the environment communicates on one of these channels, the *await* sends to the state that came from assuming the communication took place.

7. Experimental Results

To evaluate our code generators, we generated code for some examples three different ways and compared their sizes and execution speeds. Table 2 shows our experimental results for generated C code running on a single Pentium 4 processor (a 2.5 GHz desktop running Linux). Each of these are small programs with simple processes—highly parallel with substantial scheduling overhead. E.g., the FIR examples are finite-impulse-response filters with a separate process for each fork, sum, coefficient, and delay.

To measure performance, we ran a million samples through each example and timed its execution using the Unix *clock* function. We chose the number of iterations to get a total running time around a second to minimize measurement error.

Our baseline is the code generation technique we developed earlier [6] that generates a single function per process and uses *switch* statements to resume them. The scheduler dispatches these functions from a linked list of runnable processes. One big difference is that the switch-based code generator does not support a process blocking on multiple channels simultaneously and therefore has slightly less overhead per communication operation.

The "Switch" column lists the number of bytes in the text segment of the executable from our baseline code generator after optimization with gcc -O2 on a Pentium 4 (compiled for a 386).

The two columns labeled *Tail-Recursive* list the executable sizes and execution time speedups for tail-recursive code generated by the procedure presented in Section 4. For most of the examples, this code is about 50% larger but twice as fast. We suspect the size overhead is due to the additional overhead of creating so many little functions (nearly 600 for the FIR19 example). The two FIR examples are exceptional, however. Each is about 10% slower. We attribute this to the switch-based code generator's slightly lower communication overhead (these examples are almost purely communication) that in the other examples is overcome by the more efficient tail-recursion-based context switching.

The *Static (partial)* columns list results for statically-scheduling most, but not all, of the processes in each example. For these, we statically scheduled the "core" of each example, but left the source and sink processes independent. This is a realistic application of static scheduling since it is easy to imagine a full system composed of multiple cores of this form.

Most of these examples are a bit larger than their purely tail-recursive counterparts, and only a bit faster (the speedup column lists times relative to the baseline code generation technique). The exceptions, again, are the FIR filters, which have substantially more processes than the other examples. This is not surprising—the smaller examples have much simpler communication patterns and therefore not as much scheduling overhead. The FIR filters, by contrast, are almost pure scheduling, and are therefore much more amenable to speed-up through static scheduling.

The sizes of the automata we generate are modest. We attribute this to our communication-based abstraction, which only models the (usually periodic) communication behavior of the system, not the complete system state. E.g., chains of buffer processes have a state for when the buffers are empty, when the buffer contains one data value, when it contains two, and so forth. Fortunately, this is only linear in buffer length.

The *Static (full)* columns are results for statically scheduling all the processes in each network, i.e., considering it a closed system and analyzing it completely. The results, but size and speed, are substantially better than any of the other approaches because it eliminates unneeded flexibility. In the partial case, the generated automaton must consider all possible environmental communication events. By contrast, in the full case, there is no environment to be considered: the scheduler is free to choose a particular scheduling policy and ignore all other possible behaviors.

Table 2. Experimental Results

Example	Lines	Processes	Switch	Tail-Recursive		Static (partial)			Static (full)		
			size	size	speedup	size	speedup	states	size	speedup	states
Berkeley	36	3	860	1299	2.9	1033	2.6	5	551	7.8	6
Buffer2	25	4	832	1345	2.0	1407	2.4	10	403	11	8
Buffer3	26	5	996	1579	2.1	1771	2.6	20	443	10	10
Buffer10	33	12	2128	3249	1.7	5823	4.8	174	687	12	24
Esterel1	144	5	3640	5971	1.9	8371	2.9	49	5611	5.9	56
Esterel2	127	5	4620	7303	2.0	6871	2.5	24	2539	5.2	18
FIR5	78	19	4420	6863	0.92	6819	4.8	229	1663	7	79
FIR19	190	75	17052	25967	0.90	67823	5.9	2819	7287	7.1	372

Executables and times are from gcc -O2 running on a 2.5 GHz Pentium 4
Lines = lines of SHIM source code   Processes = total number of concurrent processes   size = number of bytes in text segment
speedup = relative to switch-based code generation   states = number of states in the automaton

Thus, the generated automaton has fewer states for most examples, especially the largest ones. This leads to smaller code size (full static scheduling actually produces the smallest code for the largest examples). Even more notable is the speed-up, which is consistently better, especially for the smaller examples. This is because most overhead has been scheduled away.

8. Conclusions

We presented two techniques for generating C code that runs the concurrent SHIM model in software: one that produces tail-recursive ANSI C code to quickly perform context switches among concurrently-running processes, and one that statically schedules groups of processes to combine them into a single imperative process that is equivalent to the group.

We implemented both of these techniques in a compiler that compiles our SHIM language, which we described in Section 3. We plan to release this compiler open-source, although we have not done so yet because it remains fairly immature. The compiler is currently about 4000 lines of OCAML.

Experimental results suggest our tail-recursive code generally operates twice as fast as code from our *switch*-statement-based code generator [6]. The statically-scheduled code, however, can be even faster; as much as twelve times on one example. Surprisingly, the increase in speed was often accompanied by a decrease in code size, making it superior in every respect.

Our compiler currently performs basic optimizations (e.g., dead code removal), but does not, say, merge or remove dead variables. Some of these basic optimizations are, of course, handled by the C compiler, but our compiler has greater insight into the code since it understands the concurrency and communication model. In the future, we plan to implement many basic optimizations in our SHIM compiler to further improve the quality of the generated code.

At least two other open questions remain in our work: how best to choose which processes should be scheduled together and how best to schedule processes statically. At the moment, we rely on the user to supply the groups of processes to schedule together, but we envision a heuristic algorithm might be able to provide hints, e.g., that tightly-coupled processes should be compiled together and non-communicating ones, such as the pair of unconnected buffers described in Section 5.2, should not be.

The scheduling policy employed in our static scheduler is essentially nondeterministic (it is affected by the order in which processes are written in the source program). While the SHIM semantics guarantee the system behaves consistently under all scheduling policies, we suspect the quality and efficiency of the generated code can be improved dramatically by more carefully choosing the static scheduling policy.

References

[1] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.

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

[3] Luca Cardelli and Rob Pike. Squeak: A language for communicating with mice. In *Proceedings of the Twelfth ACM Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 199–204, San Francisco, California, 1985.

[4] Stephen A. Edwards. Compiling concurrent languages for sequential processors. *ACM Transactions on Design Automation of Electronic Systems*, 8(2):141–187, April 2003.

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

[6] Stephen A. Edwards and Olivier Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. *IEEE Transactions on Very Large Scale Integrated (VLSI) Systems*, 2006. To appear.

[7] Robert S. French, Monica S. Lam, Jeremy R. Levitt, and Kunle Olukotun. A general method for compiling event-driven simulations. In *Proceedings of the 32nd Design Automation Conference*, pages 151–156, San Francisco, California, June 1995.

[8] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[9] Nicholas Halbwachs, Pascal Raymond, and Christophe Ratel. Generating efficient code from data-flow programs. In *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming (PLILP)*, volume 528 of *Lecture Notes in Computer Science*, Passau, Germany, August 1991. Springer-Verlag.

[10] 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.

[11] Bill Lin. Software synthesis of process-based concurrent programs. In *Proceedings of the 35th Design Automation Conference*, pages 502–505, San Francisco, California, June 1998.

[12] André Costi Nácul and Tony Givargis. Code partitioning for synthesis of embedded applications with Phantom. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 190–196, San Jose, California, November 2004.

[13] Marco Sgroi, Luciano Lavagno, Yosinori Watanabe, and Alberto Sangiovanni-Vincentelli. Synthesis of embedded software using free-choice Petri nets. In *Proceedings of the 36th Design Automation Conference*, pages 805–810, New Orleans, Louisiana, June 1999.

[14] Xiaohan Zhu and Bill Lin. Compositional software synthesis of communicating processes. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, pages 646–651, Austin, Texas, October 1999.