Compiling Esterel into Sequential Code

Stephen A. Edwards
Synopsys, Inc.
700 East Middlefield Road
Mountain View, California 94043–4033
sedwards@synopsys.com

Abstract

This paper presents a novel compiler for Esterel, a concurrent synchronous imperative language. It generates fast, small object code by compiling away concurrency, producing a single C function requiring no operating system support for threads.

It translates an Esterel program into an acyclic concurrent control-flow graph from which code is synthesized that runs instructions in an order respecting inter-thread communication. Exceptions and preemption constructs become conditional branches. Variables save control state; conditional branches restore it.

Although designed for Esterel, this approach could be applied to compiling other synchronous concurrent languages.

1 INTRODUCTION

I propose a new way to compile Berry's imperative synchronous language Esterel [4], intended for specifying reactive real-time systems. Esterel has the control constructs of an imperative language like C, but includes concurrency, preemption, and a synchronous model of time like that used in synchronous digital circuits. In each clock cycle, the program is restarted, reads its inputs, and computes its reaction in bounded time.

Unlike other languages, Esterel allows bidirectional communication between concurrent threads within the same cycle; how my compiler handles the interleaving is a key contribution. Using operating system-supported threads would be prohibitively expensive (a small program might have a hundred threads), and other compilers have traded off size for speed solving this problem.

Automata-based compilers (Berry's V3 [4], the Polis group's [6]) weave together concurrently-running threads to produce fast code with zero overhead. The generated code comes from exhaustively simulating the program, and it can grow exponentially. The Polis compiler attempts to reduce code size by sharing code between states. It uses a binary-decision diagram to merge common subtrees, but the worst case remains exponential.

Berry's V5 compiler [2, 3] goes to the other extreme. In effect, it transforms every statement into its own thread; the generated code is a series of unnested "if statement is running then statement" instructions. This can be slow because in each cycle, some code for every source instruction must be executed, even those not running.

Proceedings of the 7th International Workshop on Hardware/Software Codesign (CODES '99), Rome, Italy, May 3–5, 1999.

My compiler adopts both techniques' strengths. It generates a fixed amount of code per source instruction (plus some overhead) and tries to generate straight-line code where possible. When a context switch is necessary, the generated code saves the first thread's control state in a variable and resumes the second with a multi-way branch. The result is nearly as fast as the automata-based compilers' (only about 50% slower; V5 can be ten times slower) and nearly as small as the V5 compiler's (perhaps 50% larger; automata-based code can be a hundred times larger).

Currently, this compiler can only handle "statically causal" programs, that is, there must be an order to all instructions that respects data dependencies. Many programs satisfy this, but a valid program may have a data- or state-dependent order. Automata-based compilers handle this by generating code ordered for each state. The V5 compiler uses exhaustive symbolic simulation followed by boolean resynthesis [8] to remove static dependency cycles. I have not yet tried to apply this technique to my compiler.

The flow of my compiler (a concurrent representation is generated, scheduled, and from it sequential code is produced) follows both Berry's V5 compiler and Lin's [7]. Instead of Esterel, Lin compiles a concurrent imperative language with rendezvous-style communication. Lin uses Petri nets as an intermediate representation, but they are awkward for modeling Esterel's synchronous communication, so I use a different formalism. Lin uses an automaton approach for generating sequential code, so it generates fast code, but a bad schedule can turn linearly-sized code into exponentially-sized code. This is hard to avoid since scheduling is NP-complete, so I have devised another code generator.

Bertin et al.'s recent Esterel compiler [5] resembles a compiled event-driven simulator. For each segment of code between pauses, it generates a small C function dispatched in response to incoming signals. Their code size/speed results are encouraging, but the compiler currently does not handle programs with interleaved threads.

This paper follows the compilation process after describing the Esterel language and the intermediate format generated by Berry's compilers. The compiler translates the intermediate format into a cyclic concurrent control-flow graph (CCFG), unrolls it to produce an acyclic CCFG, schedules this, and uses it to produce a sequential control-flow graph that is translated into C. The last section presents some preliminary results.

Esterel §2
↓
IC §3
↓ §4
Cyclic CCFG
↓ §4.3
Acyclic CCFG
↓ §5
CFG
↓ §6
C source

This work

2 BERRY'S ESTEREL LANGUAGE

For brevity, I only describe a subset of the Esterel language. The compiler supports the full language through more node types and data dependencies; the additions are straightforward.

Esterel [4] is a textual concurrent imperative language that treats time as does a synchronous digital circuit. The clock awakens mul-

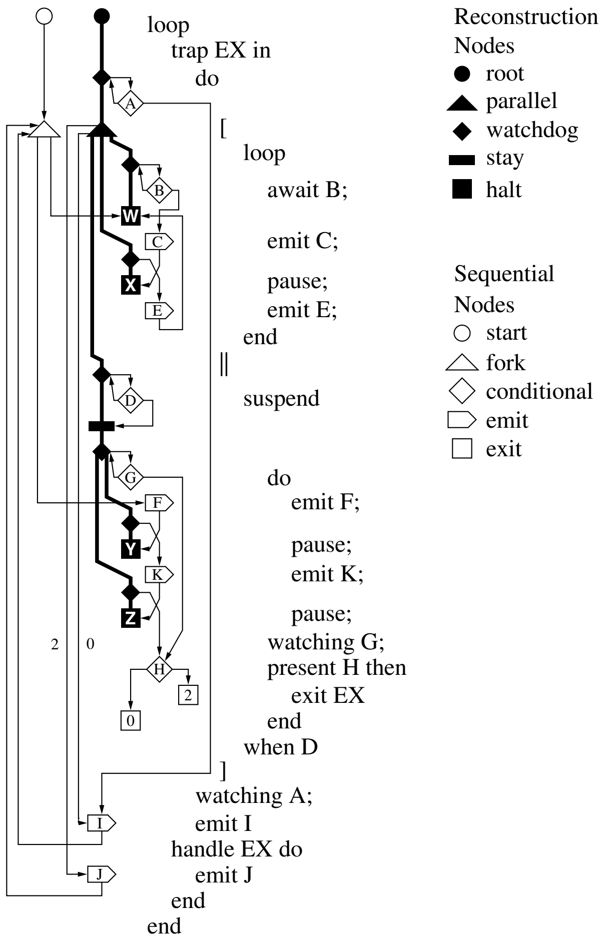


Figure 1: A small Esterel program (text down middle) with its IC representation. The darker lines and nodes make up the reconstruction tree. The IC node types are shown at right.

tiple threads of control and they run until they hit halt points (e.g., the *pause* statement) and sleep until the next cycle.

Composite statements can be built sequentially P ; Q, or concurrently [P || Q]. In [P || Q] ; R, R runs only after P and Q have terminated. An infinite loop is written *loop P end*.

For communication, Esterel uses signals that behave like wires in a combinational circuit. By default, each signal is absent each cycle, but the environment or an *emit* statement can make it present. Any statement (*present S then P else Q end*) that tests a signal waits for it to "stabilize" before proceeding.

Signals are global (part of a module's interface) or local to a block. The *signal S in P* statement introduces the signal S to the scope of statement P. When control reaches a *signal* statement, it creates a new, absent copy of signal S.

An Esterel program can wait for the next cycle (*pause*) or for a single signal (*await S*), but one of Esterel's strengths is its ability to nest preemption, suspension, exception, and concurrency. For example, the multi-cycle statement P in *do P watching S* runs and terminates normally except when S is present. In that cycle, P is terminated and does not run. The *suspend P when S* construct runs P except in cycles where S is present.

Esterel's exception facility complements the strong preemption of *do-watching* and *suspend*. When P in *trap T in P handle T do Q end* executes *exit T*, P runs until it pauses, then the exception handler Q runs. Thus P can terminate itself.

3 BERRY AND GONTHIER'S IC

This section describes the compiler's starting point, the intermediate code (IC) produced by the front end of Berry's compilers. The IC format represents an Esterel program as a directed graph of sequential statements (e.g., emits and conditionals) hanging from a "reconstruction tree." Figure 1 shows a small Esterel program and its IC translation. The dark nodes and arcs form the reconstruction tree; sequential statements and arcs are lighter.

The reconstruction tree coordinates exceptions, preemption, and concurrency by dictating how the program restarts at the beginning of a cycle. In the second and later cycles, control starts at the root of the tree and heads toward the halts reached in the last cycle. Along the way, watchdog nodes check preemption conditions, stay nodes entered via a sequential arc prevent control from reaching pauses beneath, and parallel nodes both split the flow of control and handle exceptions through their outgoing sequential arcs. Parallel nodes are always paired with a fork node that starts their threads.

Esterel's exception mechanism is unique because it interacts with concurrency. Throwing an exception terminates both the local flow of control and all concurrently-running threads. In a cycle, each concurrently-running thread of control runs until it stops in one of three ways, distinguished by an integer termination level: 0 for normal termination, 1 for pause, and 2, 3, or higher for an exception. The termination code for a group of threads is the maximum of the threads' codes, and each parallel has outgoing sequential arcs for levels 0, 2, 3, etc. as necessary.

Consider the program in Figure 1. In the first cycle, control begins at the start node (the unfilled circle), splits at the fork and flows to both W and through *emit F* to Y.

In the second and later cycles, control starts at the reconstruction tree's root and heads toward the "active" halts—those reached in the previous cycle. Control first goes to watchdog A and to the conditional that tests A. If A is present, control flows to *emit I* and back around to the fork, restarting the loop. Otherwise if A is absent, control continues down the reconstruction tree to the parallel.

Control splits two ways at the parallel: toward W or X and toward Y or Z. If W is active, control goes to watchdog B, which, if B is present, sends control to *emit C* and on to X. Otherwise, if B is absent, control returns to the reconstruction tree and flows to W.

If X is active, control flows to the watchdog just above it, which immediately sends control to *emit E* and back around the loop to W.

If Y or Z is active, control flows from the parallel down to watchdog D. If D is present, control flows to the stay and stops—Y or Z stays active for the next cycle. Otherwise, if D is absent, control flows to watchdog G. If G is present, control flows to H. If H is present, control flows to the exit at level 2 (raising the exception), otherwise the thread exits at level 0.

Finally, if Y or Z is active and both G and D are absent, control flows down one of the branches from watchdog G. If Y is active, control flows toward it and through the watchdog that sends control to *emit K* and on to Z. If Z is active, the watchdog just above it sends control directly to H.

4 TRANSLATING IC TO A CCFG

This section begins the description of the compiler, showing how it translates Esterel programs in the complex IC format into semantically simpler concurrent control-flow graphs (CCFGs). All synchrony, preemption, suspension, and exception constructs are converted to conditionals, producing a control-flow graph whose concurrency will be removed by the procedure in Section 5.

A CCFG is a hierarchical directed graph containing entry nodes with no predecessors, an exit node with no successors, statement nodes with one successor, conditional nodes with multiple successors, and

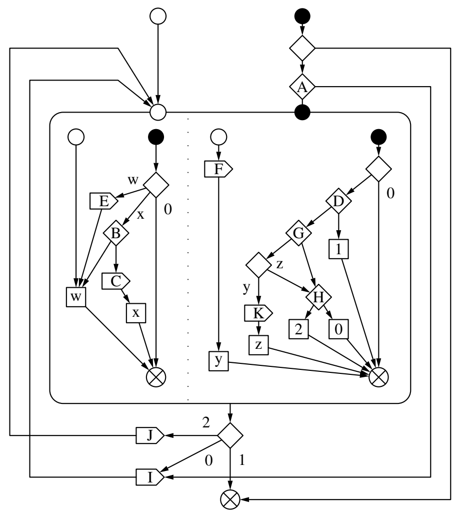


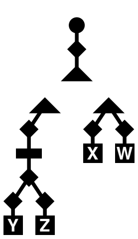
Figure 2: The small IC program translated into a Concurrent Control-Flow Graph (CCFG).

hierarchical nodes containing one or more CCFGs. A CCFG behaves like a normal control flow graph—control follows a path from entry to exit—but when control reaches a hierarchical node, it starts each contained graph, called a thread, at one of its entry nodes depending on how control reached the hierarchical node. Control leaves a hierarchical node when all its threads have exited.

Figure 2 shows the example in Figure 1 translated into a CCFG.

The following subsections describe the translation: break the IC program into threads, assign states within the threads, translate reconstruction tree nodes into conditionals to produce a cyclic graph with two-entry threads, and unroll this to produce an acyclic graph with single-entry threads. This CCFG will be run once per cycle.

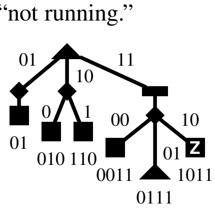
4.1 Threads



The first translation step divides the Esterel program into threads (sequential groups of statements) to identify concurrently-running pieces of code and where control may resume between cycles. Each thread corresponds to a subtree in the reconstruction tree rooted at a parallel or the return with halts and parallels for leaves. (Figure 1's threads are on the right.) At the end of each cycle, each thread is either at one of its leaves or is terminated.

Control generally forks at a parallel, but there may be more branches at a parallel node than threads beneath. For example, the parallel in Figure 1 has three children but only two threads beneath it.

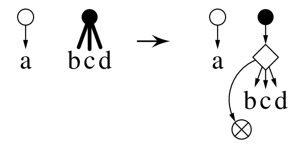
At the beginning of each cycle, the generated code walks down the reconstruction tree through each running thread. Within a thread, it walks toward the active leaf. The path is dictated by the thread's state variable, which is encoded by concatenating the branch numbers starting from the root of the thread's subtree. Less-significant bits are assigned to branches nearer the root, so each branch only checks a few bits in the state. State 0 means "not running."



Here is an example. Say this thread stopped at halt Z last cycle. This would set its state to 1011 because Z is along the 11 and 10 branches from the root. Control would start at the root, test the lower two bits of the state (11), branch to the stay, branch to the watchdog beneath it, test the third and fourth bits of the state (10), and branch to the halt.

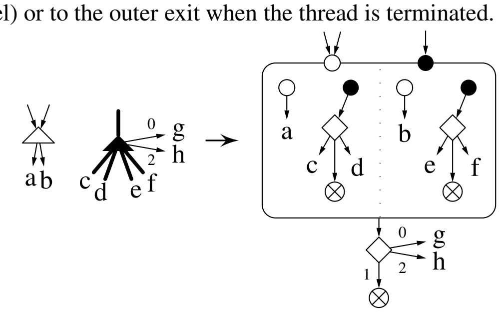
4.2 Building the CCFG

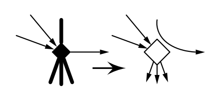
This section describes how each node in an IC graph becomes one or more nodes in the CCFG. The nesting of Esterel threads becomes the nesting of hierarchical nodes. Translating reconstruction tree nodes, especially fork/parallel pairs, is the main challenge.



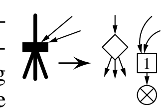
The "start" IC node and the root of the reconstruction tree become the "start" (open circle) and "restart" (filled circle) entry points of the outermost CCFG. A conditional under "restart" tests the outermost thread's control state and either branches down the reconstruction tree or to the outermost exit node. Execution in the first cycle begins at "start," and "restart" in later cycles.

Each fork/parallel pair becomes a hierarchical node containing one or more threads. Like the outer level, each thread has a "start" and a "restart" entry point (corresponding to the fork and parallel) and a single exit. The restart entry goes to a conditional that tests its thread's state and either branches down the reconstruction tree or jumps directly to the thread's exit node. A conditional node following the hierarchical node checks the threads' exit level and jumps to one of the exception handlers (the sequential arcs leaving the parallel) or to the outer exit when the thread is terminated.

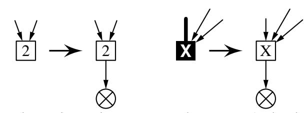




The incoming reconstruction arc on a watchdog is shunted to its outgoing sequential arc. Incoming sequential arcs are routed to a conditional that tests its thread's control state and branches to one of the outgoing reconstruction arcs.



The incoming reconstruction arc on a stay goes directly to a conditional that tests its thread's control state and branches to one of the outgoing reconstruction arcs. Incoming sequential arcs are shunted to an exit at level 1 (pause) leading to the thread's exit.



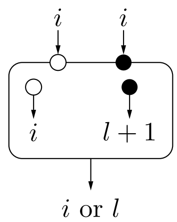
On exit and halt nodes, incoming arcs are unchanged, but an outgoing arc leading to the thread exit is added. An exit node sets the exit level of its hierarchical node; higher levels take precedence. A halt node both sets the state of its thread and sets the exit level to 1.

Conditional and emit nodes become CCFG nodes directly.

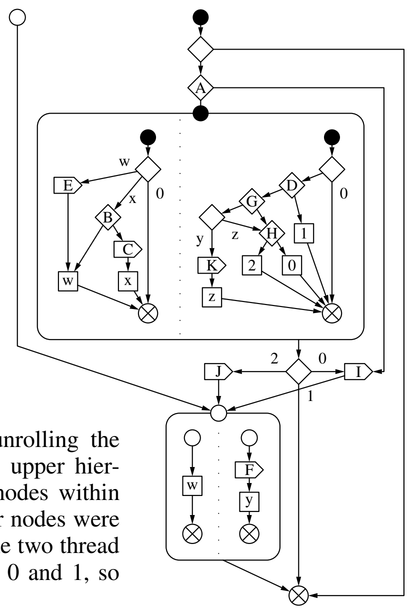
4.3 Removing False Cycles

Valid code often has unreachable control cycles that would confuse the code synthesis procedure. The compiler removes them with Berry's procedure for "curing schizophrenia" [3].

All false control cycles pass through a hierarchical node's start entry. Figure 2 has such a cycle passing through F, y, and J, but halt y exits at level 1 so the conditional will not run J. Only the code reachable from the start entry must be duplicated, and only once per nesting level. This may produce a quadratically-larger graph, but this rarely occurs in practice.



The unrolling procedure makes one copy per incarnation of each CCFG node. Each copy of a node is given an integer incarnation label from zero to the hierarchical nesting level l; the outermost level is 0. The successors of a non-hierarchical node labeled i are copied and labeled i. When a hierarchical node is labeled i through the "start" entry, the nodes within and its successor are also labeled i, and the conditional's unreachable branches are removed, breaking any cycles. Through "restart" the hierarchical node and the nodes within are labeled $l+1$ (the nesting level in the hierarchical node) and the successor is labeled l.



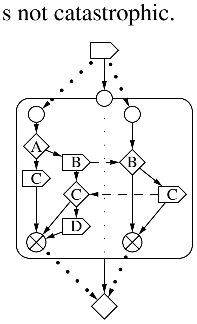
Here is the result of unrolling the CCFG in Figure 2. The upper hierarchical node and the nodes within were labeled 1; all other nodes were labeled 0. Halt w and the two thread exits were labeled both 0 and 1, so they were duplicated.

5 SYNTHESIZING A CFG

The next step synthesizes a sequential control-flow graph that runs the nodes in the CCFG in an order that respects communication. This order is determined by topologically sorting a graph of control and data dependencies, and it determines where to add "context switching" code that saves and restores the control state of concurrently-running threads. The result is a sequential control-flow graph from which it is easy to generate C.

5.1 Dependencies and Scheduling

A schedule imposes a global order on all nodes in a CCFG that respects all control and data dependencies. It is used to identify context switches—where control state must be saved and restored. Minimizing these is desirable because they usually require additional code, but the problem appears to be NP-complete. Fortunately, inserting code for context switches can only grow the code quadratically so using a sub-optimal schedule is not catastrophic.



Phantom control dependence arcs are added for scheduling. These (···►) lead to and from each thread's entry and exit node.

Data dependence arcs (– – ►) are added from every emission of a signal to every reachable node that tests the signal. Two nodes are mutually reachable if both are able to execute in the same cycle, or equivalently, if it is not necessary to take different branches leading from a conditional to run both. This can be tested with a depth-first search on a "linearized" CCFG formed by pasting together the entry and exit nodes in concurrently-running threads in an arbitrary order. Any valid set of executable instructions (but not their order) corresponds to a path through this graph. The compiler tests reachability with a quadratic algorithm.

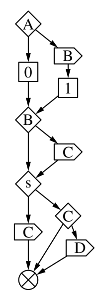
Any topological order of the dependence graph is a valid schedule. The compiler currently uses a depth-first search.

The minimum-code-size scheduling problem appears to be NP-complete because it contains the NP-complete minimum feedback vertex set problem. A proof sketch: given a directed graph, create a program with one thread per node, each thread having a single conditional invoking two threads beneath it. The first of these threads depends on signals corresponding to incoming arcs in the given graph, the second emits signals corresponding to outgoing arcs. A cyclic graph would force some thread pairs to be split, so their conditionals would be duplicated. Splitting a node corresponds to removing a vertex, and each split vertex increases code size by a constant. So asking whether there is a schedule that produces code of a certain size is equivalent to asking whether fewer than some number of vertices must be removed to make the graph acyclic.

A valid program may have (unsensitizable) cycles in its control/data dependence graph, which this compiler cannot currently handle. It may be possible to apply the solution used in Berry's V5 compiler, which performs state-machine-like analysis of the program but uses efficient symbolic techniques [8] to synthesize loop-free net lists from the offending section.

5.2 Synthesizing Context Switches

The next step synthesizes a CFG that executes the nodes in scheduled order. The algorithm steps through the schedule and attaches each node to the CFG being constructed, synthesizing code that saves and restores the control state of concurrently-running threads as needed. This is analogous to simulating all control paths through the CCFG and using the results to build a CFG. Running this procedure on last section's example produces the result at right.

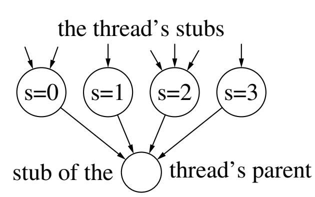


```
if (A) {
  B=1; s=1;
} else {
  s=0;
}
if (B) C=1;
if (s) {
  if (C) D=1;
} else {
  C=1;
}
```

During synthesis, the CFG contains statement nodes with one successor, conditional nodes with two or more successors, and stub nodes with no successors. When a scheduled node is added, it replaces a stub, stubs are added for its successors, and its outgoing arcs are connected from the node to the new stubs.

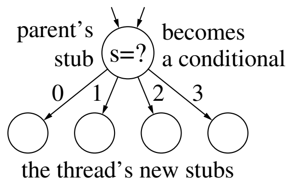
Two adjacent nodes in the schedule residing in different threads forces a context switch, which is complicated in a hierarchical setting. More precisely, when two adjacent nodes or their parents reside in different threads of the same hierarchical node, the thread containing the first node and all its children must be suspended and the thread containing the second node resumed.

The synthesis process suspends and resumes a thread of control by creating code that saves and restores its control state—the program counter. This is done by saving and regenerating stubs, which correspond to all active control states.

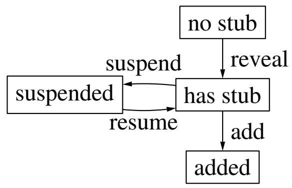


When a thread is suspended, each of its stubs is replaced with a node that assigns a unique value to a variable that saves the thread's control state. These nodes branch to a new stub that represents the thread's parent—the hierarchical node in which the thread resides. When a thread has only one stub, replacing it with an assignment is unnecessary.

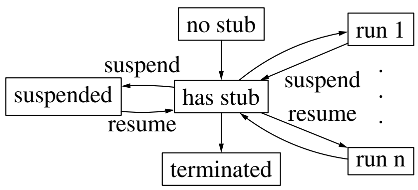
Resuming a thread reverses the suspension process. The stub for the thread's parent is replaced with a conditional branch that checks the saved control state and jumps to newly-added stubs that correspond to each saved control state. If the thread was suspended at only one stub, the conditional branch is unnecessary—the stub of the parent simply becomes the thread's single stub.



During the synthesis process, each non-hierarchical node may be in one of four states. It has no stub initially, but gains one when one of its predecessors is added. If the thread in which the node resides is suspended, the stub disappears and is recreated when the thread is resumed. Eventually, the node is added to the CFG and nothing more happens to it.



The "state" of a hierarchical node also evolves through the synthesis process. A hierarchical node has no stub initially and gains one when one of its predecessors is added. Suspending this node's thread removes the stub, but this may only happen after all threads within the node are suspended. Since only one of the threads within the node may be running (i.e., have stubs) at any time, one is suspended and another resumed to switch between them.



In the worst case, the nesting depth of the nodes is proportional to the number of nodes, so the synthesized code may be quadratic in the size of the CCFG.

6 SYNTHESIZING C

The compiler currently produces C code with unnested if-then-else, switch-case, and goto statements after topologically ordering the CFG nodes to force all branches to be forward. Such code is correct but hard to read. Structuring the generated code, perhaps using Baker's work [1], remains to be done.

7 RESULTS

Table 1 shows preliminary code size/speed results.

I ran these experiments by generating code for each example, linking it with an automatically-generated testbench that quickly generates random inputs (times include the time to do this), and timing 100 000 cycles. The distribution of the inputs is suspect, but I do not believe it biases the results toward a particular compiler.

I did not run V5's logic synthesis post-optimization step. This might have halved running times and executable sizes.

The new compiler produces code whose size is comparable to the V5 compiler, but whose speed is closer to the V3-style automaton compilation technique. Its advantage may be less for larger examples, however; I suspect this is due to increasing context switching overhead (it grows faster than linearly), but more experiments are necessary.

8 CONCLUSIONS

This paper has presented a new way to compile the synchronous reactive language Esterel that tries to retain much of the program's original control structure for a code size and speed advantage. It translates Esterel's preemption and exception constructs into conditional branches and compiles away its concurrency by statically scheduling the instructions and inserting code that saves control state in variables and restores it with conditional branches. It produces a sequential control-flow graph that is translated directly to C.

In the future, I intend to devise scheduling heuristics, add some optimizations, improve the readability of the generated code, and explore the possibility of resynthesizing tightly-coupled portions of the program using automata techniques.

			V3		V5		new	
	nodes	states	size	time	size	time	size	time
runner	50	7	4.9K	0.11	5.6K	0.25	4.7K	0.11
abcd	154	33	27K	0.16	8.6K	0.43	11K	0.27
tcint	412	287	4.1M	0.55	26K	2.4	28K	0.66
ww	483	42	120K	0.36	28K	2.1	36K	0.67
atds	892	152	58K	0.29	59K	6.2	52K	0.39
gg	1318	>700			73K	8.0	110K	3.9

nodes	Number of IC nodes
states	Number of control states between cycles
V3	Berry's V5 compiler, automaton mode (-A -Lc:-inline)
V5	Berry's V5 compiler, sorted circuit mode
new	Compiler presented in this paper
size	of .o file produced by Sun's cc -O
time	in seconds to run 100 000 random inputs (336MHz Ultra SPARC II)

runner	Berry's example	ww	wristwatch
abcd	lock controller	atds	video pulse generator
tcint	bus controller	gg	user-interface

Table 1: Comparison of this compiler with Berry's automaton and boolean equation-based compilers.

REFERENCES

[1] B. S. Baker. An algorithm for structuring flowgraphs. *Journal of the ACM*, 24(1):98–120, Jan. 1977.

[2] G. Berry. Esterel on hardware. *Philosophical Transactions of the Royal Society of London. Series A*, 339:87–104, 1992.

[3] G. Berry. The constructive semantics of pure Esterel. Book in preparation, 1996.

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

[5] V. Bertin, M. Poize, and J. Pulou. Une nouvelle methode de compilation pour le language ESTEREL [A new method for compiling the Esterel language]. In *Proceedings of GRAISyHM-AAA*, Lille, France, Mar. 1999. In French.

[6] M. Chiodo, P. Giusto, A. Jurecska, L. Lavagno, H. Hsieh, K. Suzuki, A. Sangiovanni-Vincentelli, and E. Sentovich. Synthesis of software programs for embedded control applications. In *Proceedings of the 32nd Design Automation Conference*, pages 587–592, San Francisco, CA, June 1995.

[7] B. Lin. Efficient compilation of process-based concurrent programs without run-time scheduling. In *Proceedings of Design, Automation and Test in Europe*, pages 211–217, Paris, France, Feb. 1998.

[8] T. R. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *Proceedings of the European Design and Test Conference*, Mar. 1996.

Berry et al.'s Esterel work can be found at http://www.inria.fr/meije/esterel/.