Compiling Esterel into Sequential Code

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

Abstract

Embedded real-time software systems often need fine-grained parallelism and precise control over time, things typical real-time operating systems do not provide. The Esterel language has both, but Existing compilers produce slow code for large programs.

This paper presents the first Esterel compiler able to produce small, fast code for large programs. It can produce code half the size and up to a hundred times faster than code from existing compilers. Esterel's semantics allow the compiler to statically schedule concurrency and synthesize code that efficiently and predictably simulates context switches. The main contribution is an algorithm that synthesizes an efficient sequential program from the concurrent control-flow graph used as an intermediate representation. These techniques could be applied to any language with fine-grained parallelism.

1 Introduction

I propose a new compiler for the Esterel language [4] called EC. 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 restarts, reads its inputs, and computes its reaction.

Figure 1 shows a simple Esterel program with two concurrent threads. The first thread waits for the START signal and emits REQUEST. If it receives GRANT in the same cycle, it emits the GOT signal. In alternating cycles, the other thread emits GRANT in response to REQUEST. The threads restart when the RESET signal appears because of the `abort-when` RESET construct inside the outer `loop`.

For this or any other Esterel program, an Esterel compiler will generate a single C function that is called each cycle to simulate the system. Such a function saves its state in variables between calls.

Simulating concurrency is the main challenge. Pairs of threads, such as those in Figure 1, may communicate bidirectionally in the same cycle, so their execution must be interleaved. Large programs can have thousands of threads, so the cost of switching between threads must be kept low.

```
module EXAMPLE:

input RESET, START; output GOT;

signal REQUEST, GRANT in
  loop abort                        % RESET restarts the loop
    await START;
    emit REQUEST;
    present GRANT then emit GOT end
  ||                                % run concurrently
    loop
      present REQUEST then emit GRANT end;
      pause;                        % wait for the next cycle
      pause
    end
  when RESET end
end.
```

Figure 1: A simple Esterel program consisting of a single module. Figure 6 shows some of the code EC generated for this.

Automata-based compilers (Berry et al.'s v3 [4] and the Polis group's [8]) exhaustively simulate the program to generate a branching program for each state that produces side effects while deciding the next state. Here is code for a state in Figure 1:

```
if (RESET) state = 2;
else if (START) { GOT = 1; state = 5; }
else state = 2;
```

Automata code is fast but often exponentially larger than the source. The Polis compiler shares code between states after identifying it with a binary decision diagram, but this remains exponential.

The gate-based compilers (Berry et al.'s v4 and v5 [1, 2]) translate an Esterel program into logic gates and generate code from the gates. This approach handles large programs, but the code is slow because idle sections still take time. For example, here is a fragment of the code v5 generated from Figure 1 (the E variables hold intermediate results, the R variables hold state between cycles):

```
E4 = E3 && START; E5 = (E1 && RESET) || R0;
E6 = (R3 && E2) || E5; E7 = E6 && E4;
E8 = E4 && E7; if (E8) GOT = 1;
```

EC generates small, fast code for a large Esterel programs by employing a natural intermediate representation (essentially C with fork and join statements) and using a good algorithm to sequentialize it. The sequentializing algorithm, described in Section 3.2, uses

DAC 2000, Los Angeles, California

the natural idea of saving and restoring program counters to simulate concurrency. A thread is suspended by storing a constant in a program counter variable and resumed with a multi-way branch. This approach avoids the capacity problems of the automata compilers and most of the overhead of the gate-based compilers. The result can be a hundred times faster than gate code.

Currently EC can only handle programs where the instructions can run in the same order in all states. Most programs satisfy this, but Esterel's semantics permit data- or state-dependent orders. Different orders do not present a problem for the automata compilers since they treat each state separately. A data-dependent order appears as a cycle in the netlist generated by V5. The compiler removes such cycles by exhaustively simulating the netlist and resynthesizing the cycles [14]. This technique may also be applicable to EC.

I patterned EC after Lin's compiler [12]. Lin translates programs written in a C variant into Petri nets that represent sequencing, concurrency, and communication. These work well for the rendezvous-style communication in Lin's language, but are awkward for Esterel's synchronous style. Lin schedules and then simulates these Petri nets to generate very fast automata-style code. Unfortunately a bad schedule can cause an exponential explosion in code size. Even worse, finding a good schedule is NP-complete.

Sgroi et al. [13] also address scheduling concurrent software, but their formalism is also built on Petri nets and the behavior of their systems is not synchronous.

Bertin et al.'s 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 they cannot handle programs that require interleaved threads such as Figure 1.

2 Translating Esterel to Concurrent C

EC translates a complete Esterel program into a single C function called once per cycle. This function has no loops and restarts itself using state variables. EC generates this function by translating an Esterel program into concurrent C and sequentializing it using the algorithm described in Section 3.

EC represents concurrent C code as a concurrent control-flow graph (CCFG). A CCFG contains plain, conditional, fork, and join nodes, each with an expression. When control reaches a node, the node's expression is evaluated and control flows along one or more arcs leaving the node. A plain node has a single outgoing arc and its expression is usually an assignment. Control leaves a conditional node along the arc whose integer label matches the value of the expression (these become `if` or `switch` statements).

Fork and join nodes start and collect groups of parallel threads. Control flows out all arcs leaving a fork, starting a group of threads that will wait at a matching join node before continuing. Fork and join nodes may nest, but control may not pass between threads. Specifically, all paths from a particular fork meet for the first time at a unique join.

2.1 Resuming Threads and State Encoding

Figure 2 illustrates how EC translates preemption and multi-cycle behavior in Esterel (Figure 2a) into C (Figure 2b).

When control reaches an Esterel `pause` statement, the thread containing the `pause` stops and resumes there the next cycle. In C, a `pause` writes to the thread's state variable and branches to the end of the thread (e.g., line 4). In the next cycle, a `switch` statement (lines 3–18) branches to the statement immediately following the `pause` (e.g., line 5).

In the first cycle it runs, the `abort-when` construct simply runs its body. This is done by the `goto` statement in line 6.

```
                    1  Start: goto L1;
                    2  Resume:
                    3     switch (s & 0x3) {
pause;              4     L1:     s=1; goto Join;
pause;              5     case 1: s=2; goto Join;
abort               6     case 2: goto L2;
                    7     case 3: if (!A)
                    8       switch (s>>2 & 0x7) {
  pause;            9       L2:     s=3 | 0<<2; goto Join;
  pause;           10       case 0: s=3 | 1<<2; goto Join;
  pause;           11       case 1: s=3 | 2<<2; goto Join;
  pause;           12       case 2: s=3 | 3<<2; goto Join;
  pause            13       case 3: s=3 | 4<<2; goto Join;
                   14       case 4: ;
when A             15       }
                   16       s = 0; goto Join;
                   17     case 0: ;
                   18     }
                   19  Join:
```

(a) (b)

Figure 2: (a) An Esterel thread fragment. (b) Its translation into C (code for exception handling not shown).

In later cycles, the `abort` statement checks its condition (`A`) before resuming its body. Lines 7 and 8 do this. In the first cycle control reached the `abort`, line 9 set the lower two bits of `s` to 3, so in the next cycle the `switch` in line 3 jumps to line 7. Line 7 checks `A`. If `A` is absent, the body has not been preempted and line 8 runs. The `switch` on line 8 checks the third through fifth bits of `s` and branches to just after one of the `pause` statements within the body of the `abort`.

2.2 Translation Details

Although EC uses a CCFG as an intermediate representation, I describe the translation by showing how Esterel statements translate into concurrent C statements. The mapping to the CCFG is trivial.

For brevity, I only discuss Esterel constructs with interesting semantics. EC supports the full language.

Esterel statements that do not affect time have trivial translations:

Esterel	C
p ; *q*	*p*; *q*;
emit *S*	*S* = 1;
loop *p* end	for(;;) *p*;
present *S* then *p* else *q* end	if (*S*) *p*; else *q*;

The `exit` statement throws an exception that can be caught by a surrounding `trap T in ... handle T do`. This only happens after all threads in the same group are done for the cycle. To handle this, each thread sets an exit level when it stops at the end of a cycle. This level indicates termination (level 0), pausing (level 1), or an exception (levels 2 and higher). Exceptions take precedence over `pause`s, so a group of threads responds only to the highest level.

A `pause` statement resumes in the next cycle. The code for it sets its threads state to *k*, making the `switch` statement surrounding the thread send control to the `case` label next cycle. Raising the exit level to 1 indicates this thread has paused. The branch to *Join* stops the thread for the cycle.

```
pause                 state = k;
                      if (level < 1) level = 1;
                      goto Join;
                    case k:
```

The `await` statement is similar to `pause`, but it also pauses in later cycles until its signal is present.

```
                        goto Entry;
await S               case k:
                        if (!S) {
                        Entry:
                          state = k;
                          if (level < 1) level = 1;
                          goto Join;
                        }
```

Preemption statements introduce nested `switch` statements. In the first cycle, `abort` just runs its body. It restarts its body in later cycles only if the aborting signal is absent.

```
                        goto Entry;
                      case k:
abort                   if (!S)
                          switch (state) {
  body                      Entry: body;
when S                    }
```

`Suspend` runs its body in the first cycle and pauses in later cycles when the suspending signal is absent, leaving its thread's state unchanged.

```
                        goto Entry;
                      case k:
suspend                 if (S) {
                          if (level < 1) level = 1;
                          goto Join;
                        }
                        switch (state) {
  body                    Entry: body;
when S                  }
```

The `signal` statement creates a new, absent copy of its signal.

```
signal S in             S = 0;
                        goto Entry;
                      case k:
                        S = 0;
                        switch (state) {
  body                    Entry: body;
end                     }
```

The `exit` statement raises its process's exit level to two or more depending on the exception. Since this terminates the thread and its process, there is no need to set the thread's state.

```
exit T;                 if (level < 2) level = 2;
                        goto Join;
```

Parallel and `trap` statements are intertwined. An implicit `trap` surrounds each group of parallel threads, and the body of a trap is considered a separate thread. The `trap`/parallel combination resets the exit level for the enclosed process, runs the threads within, and handles the exit level they return. The process terminates if the level is zero (the `switch` falls through), pauses at level one, and handles exceptions at levels two and higher.

The threads have two entry points: one taken in the first cycle, the other taken in later cycles that uses `switch` statements to restart the threads. The `fork` statement passes control to each of its labels. The `join` waits until all the threads branch to it before continuing. A terminated thread sets its state to zero so control will go to the `case 0:` labels when other threads in the process continue to run.

```
trap T in               innerLevel = 0;
                        fork StartA, StartB;
                      case k:
                        innerLevel = 0;
                        fork ResumeA, ResumeB;

                      ResumeA:
                        switch (statep) {
  bodyA                 StartA: bodyA;
                        case 0: ;
                        }
                        goto InnerJoin;
||
                      ResumeB:
                        switch (stateq) {
  bodyB                 StartB: bodyB;
                        case 0: ;
                        }
                        goto InnerJoin;

                      InnerJoin:
                        join;
handle T do             switch (innerLevel) {
                        case 1:         /* paused */
                          state = k;
                          if (level < 1) level = 1;
                          goto OuterJoin;
                        case 2:         /* exited */
  handler                 handler;
end                     break;
                        }
```

2.3 Removing Loops by Unrolling

The algorithm to remove concurrency (Section 3) cannot handle loops in the CCFG, so EC duplicates code to remove them. Esterel prohibits infinite loops, but `loop`s, `trap`s, and parallel threads can conspire to create apparent infinite loops. The problem arises when an instruction can run twice or more in a cycle.

EC makes a separate copy of each reincarnation using an algorithm developed by Berry [2] to remove cycles in the netlists generated by the v5 compiler. Figure 3a shows a fragment that can run an instruction twice in a cycle; Figure 3b is the unrolled result.

Unrolling may appear costly but is reasonable in practice. Figure 4 shows even the largest examples do not double in size.

3 Removing Concurrency

EC compiles away concurrency by inserting code that suspends and resumes threads. Code generated this way runs faster than v5's gate code because it is a better fit to a processor's natural control behavior. The code from v3 runs faster by avoiding this overhead, but at the expense of extensive code duplication.

The procedure in this section produces a sequential control-flow graph (SCFG)—a CCFG without fork or join nodes. The procedure first adds data dependency arcs to the CCFG. Next, it schedules the nodes in this graph to identify where to suspend one thread and resume another. Finally, it builds the SCFG by copying the nodes in the CCFG in scheduled order, suspending and resuming threads as necessary. Suspending a thread adds nodes that save the thread's state in a variable. Resuming a thread adds a conditional node that branches on the saved state of the thread.

```
loop                 if (inLaterCycles) {
  trap T in
    loop               if (A) then B = 1;
      present A        /* pause (level 1) */
      then           ||
        emit B         /* exit T (level 2) */
      end;
      pause          }
    end
  ||                   if (A) then B = 1;
    pause;             /* pause (level 1) */
    exit T           ||
  end                  /* pause (level 1) */
end
                     inLaterCycles = 1;
```

(a) (b)

Figure 3: (a) A fragment containing code that can run twice in a cycle. `present A` runs every cycle. When `exit T` runs, it terminates the `trap`, causing the outer `loop` to immediately restart the threads. This causes `present A` to run again. (b) Pseudocode showing the effects of unrolling: the `present` statement has been duplicated.

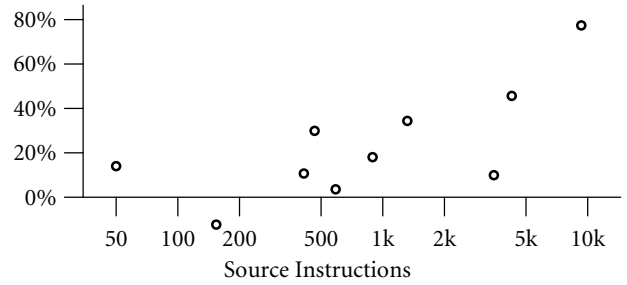


Figure 4: How unrolling increases program size. (Dead code removal caused the one decrease.)

3.1 Scheduling

A schedule orders all the instructions in the program so that none runs before its data is ready. It also identifies context switches—points where one thread must be suspended and another resumed.

A CCFG usually has many correct schedules, but some require more context switches than others. Minimizing context switches is desirable since each adds code, but doing this optimally appears to be NP-complete (it is as hard as the minimum feedback vertex set problem). Fortunately, a bad schedule generates code only quadratically larger and slower than a good schedule, and context switching overhead grows grows slowly in practice (that Figure 5 looks like a scaled version of Figure 4 bears this out).

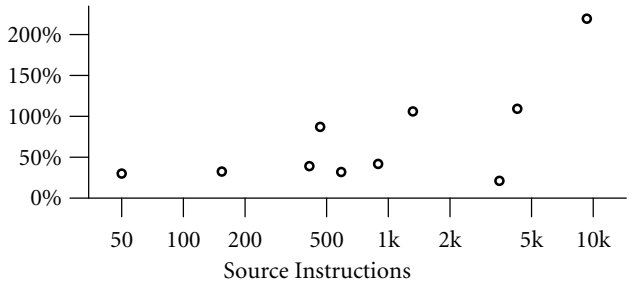


Figure 5: How unrolling and inserting context switching code increases program size.

To ensure the schedule obeys Esterel communication semantics, EC adds data dependence arcs from every node that emits a signal to every node that tests it. This is conservative since certain pairs of nodes may never execute in the same cycle.

Any topological order is a valid schedule. The compiler currently uses a depth-first search, but many better heuristics are possible.

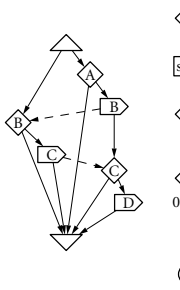
Unfortunately, EC rejects some valid programs because they do not have static schedules. Esterel permits instruction order to be data-dependent. The automata compilers handle this by allowing each state to have a different instruction order. In the netlists generated by the V5 compiler, data-dependent orders appear as cycles. The compiler removes these cycles by resynthesizing the cyclic portions of the netlist [14]. Whether this sometimes costly solution can also be used in EC is an open problem.

3.2 Building the Graph

This section describes the most important algorithm in the compiler. It builds a sequential control-flow graph (SCFG) that runs the CCFG nodes in scheduled order. For blocks of code in the same thread it simply copies the graph. Switching between threads is more challenging.

```
present A then
 emit B;
 present C then
  emit D
 end
end
||
present B then
 emit C
end
```

(a)



(b) (c)

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

(d)

The Esterel fragment (a) has two threads that communicate through signals `B` and `C`. Communication dependencies (- - ▸) have been added to the CCFG (b) showing that the left thread must interrupt the right. An SCFG that achieves this is shown in (c). The SCFG begins with the first two nodes from the right thread, ⟨A⟩ and ⟨B⟩ followed by two nodes [s=0] and [s=1] that suspend the right thread by writing its state to a variable. The left thread's nodes ⟨B⟩ and ⟨C⟩ follow these, followed by ⟨s⟩, a two-way branch that resumes the right thread. Finally, the last two nodes in the right thread, ⟨C⟩ and ⟨D⟩, run. (d) shows the corresponding C code.

The algorithm synthesizes an SCFG by copying each CCFG node in scheduled order and attaching its incoming arcs. These arcs begin at a set of SCFG nodes called the *potentials* of the CCFG node. As it copies CCFG nodes, the algorithm maintains each node's potential set and a tree of running threads to identify context switches.

Three things happen when a CCFG node is copied. First, if the node's thread is not running, any running thread from the same fork is suspended and the node's thread is resumed. Second, arcs are added from each potential node. Finally, the new SCFG node becomes a potential of each of its immediate successors in the CCFG.

When a thread is suspended, it records what instruction it will execute next so it can be resumed. Any CCFG node in the thread with at least one potential could run next, so the algorithm creates a new SCFG node that writes to the thread's state variable for each of these (e.g., [s=0] and [s=1]). Arcs are added from each potential of the CCFG node to this new SCFG node. The potential sets of the now-suspended CCFG nodes are emptied and the suspended nodes recorded.

To resume a thread, the algorithm adds an SCFG node that tests the thread's state variable (e.g., ⟨s⟩) and runs the node that was

```
if (state6_0 & 1) {     /* state of first thread */
  if (START) {
    REQUEST = 1;
    tmpPC_2_0 = 31;     /* suspend first thread */
  } else {
    exit6 < 1 ? (exit6 = 1) : 0, state6_0 = 1;
    goto L20;
  }
} else {
L20:
  tmpPC_2_0 = 26;       /* suspend first thread */
}
if (tmpPC_2_1 == 38) {  /* resume second thread */
  if (REQUEST) {
    GRANT = 1;
  }
  exit6 < 1 ? (exit6 = 1) : 0, state6_1 = 1;
}
if (tmpPC_2_0 == 31) {
  if (GRANT) {          /* resume first thread */
    GOT = 1;
    state6_0 = 0;
  } else {
    state6_0 = 0;
  }
}
```

Figure 6: Part of the code EC generated for Figure 1.

about to run before the thread was suspended. This new resume node (a multi-way branch) becomes a potential of each formerly suspended node. Arcs are added from the potentials of this thread's fork (created when the last thread suspended) to this new node.

4 Synthesizing C

The compiler generates C code with nested `if` and `switch` statements for conditional SCFG nodes; other nodes become expression statements, usually assignments.

EC ends each conditional block at the immediate postdominator of the conditional node that started it. This is the earliest node that all paths from a node must pass through on the way to the sink, which EC computes using Lengauer and Tarjan's algorithm [11].

A recursive procedure builds an abstract syntax tree from the end of the graph backwards to make all `goto` statements branch forward. Since the SCFG is generally not in nested form, some `goto` statements are inevitable.

Figure 6 shows a fragment of code generated from Figure 1. Conditionals nest, `goto`s remain, and the code is readable.

5 Results

I ran experiments to compare the outputs of EC, the automata-based v3 compiler, the gate-based v5 compiler, and the output of the v5 compiler after logic optimization (v5A). I compiled the C code with Sun's optimizing compiler in `-O` mode and ran it on a 336 MHz UltraSPARC. To measure average cycle times I ran the programs for a second and counted the cycles. I generated inputs with a synthesized test bench. Measured times do not include running the testbench: I timed it separately and subtracted it out.

The example programs, listed in Table 1, are all real applications and range from the very small (fifty lines) to realistic industrial examples (ten thousand lines).

Example	Inst.	Lines	States	Threads
Berry's runner	50	55	7	7
Combination lock	154	101	33	11
Turbochannel bus	412	687	287	85
Wristwatch	465	1088	42	87
Comm. protocol [9]	589		>10000	40
Video generator	892	948	152	138
Task sequencer [3]	1318		>700	247
Shock absorber [7]	3485	2243		135
Avionics fuel	4260	4594		944
Avionics display [3]	9305			3330

Table 1: Examples used in experiments. "Inst." is the number of nodes in the CCFG before unrolling (the graphs' horizontal axes).

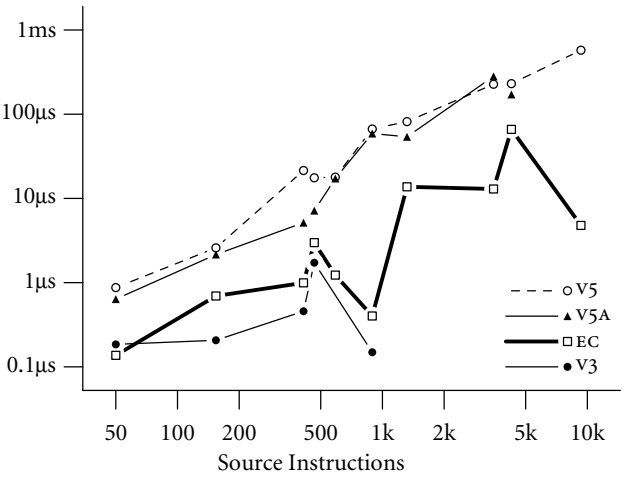


Figure 7: Average cycle times for random inputs

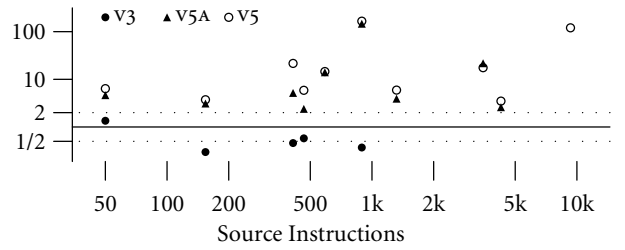


Figure 8: Relative average cycle times (EC = 1)

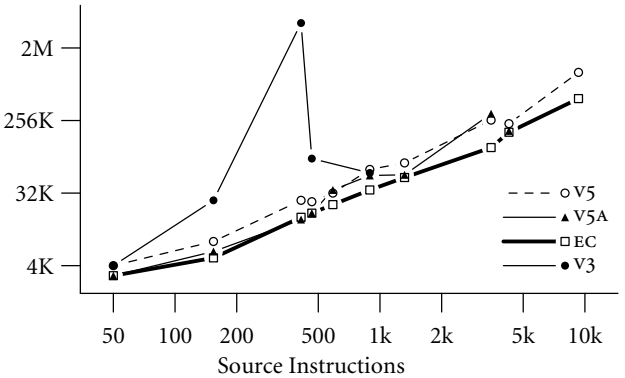


Figure 9: Object code sizes

The average cycle times plotted in Figure 7 show the code generated by EC is consistently faster than v5 with or without logic optimization, and can be over one hundred times faster. Figure 8 shows the same times divided by EC's time to show relative speeds.

On small examples, the v3 compiler produces code about twice as fast as EC's, but it cannot compile the large examples. EC's code is slower because of the overhead from context switching. Figure 5 suggests context switching code increases the size of the program by about 50%. Since these statements are inserted uniformly, this should increase runtimes by the same amount. The other 50% may come from the internal communication v3 is able to compile away.

Figure 7 also shows how the average cycle time of code from the v5 compiler closely tracks the program size. This is an artifact of the gate-based approach, which runs some code in each cycle for every instruction in the program. The speed of the code from EC and v3 track each other and reflect the number of source instructions that actually need to be executed each cycle.

Figure 9 plots the size of the object code generated for the example programs. The output of v3 can be hundreds of times larger than the others. The code from v5 is between one and two times larger than that from EC. It may appear the code from v3 is growing smaller for larger programs, but this is an artifact of certain examples being deliberately designed to have few states.

The wristwatch (just under 500 instructions) shows the least performance variance because it calls more external functions than any other example. Naturally the quality of generated code matters little when other code is running.

EC and v5 without logic optimization can produce C code faster than it can be compiled. Both EC and v5 took about eleven seconds to compile the largest example, but Sun's C compiler worked ten minutes on EC's output and an hour on v5's.

Running v5 with logic optimization can be time-consuming. I ran it on the largest example for over 100 minutes before giving up.

Compilation times for v3 can range from very short to unacceptably long (hours). Unfortunately, patience in running v3 is not rewarded since long runs produce impractically large executables.

I did not run the Polis group's compiler [8] because it uses the output of v3, and so would only have been able to compile the same small examples as v3. I would expect its code to be about the same speed as v3's but smaller.

6 Conclusions

This paper has presented a new way to compile the synchronous language Esterel that preserves 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. Ultimately, it produces mostly structured C code that contains some gotos.

Experiments show EC produces code that can be a hundred times faster and half the size of code from other high-capacity compilers.

EC is currently used to generate simulation code in CoCentric™ System Studio (described under an earlier name by Buck and Vaidyanathan [6]), an environment that allows designers to specify systems using a mixture of dataflow graphs and hierarchical finite-state machines. To compile a simulation, System Studio translates control behavior into Esterel programs, EC compiles them into C, and the result is linked with dataflow code generated by System Studio.

Other applications are possible, in addition to compiling Esterel, EC could easily be adapted to compile other synchronous, concurrent languages, such an Lavagno and Sentovich's ECL [10].

References

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

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

[3] G. Berry, A. Bouali, X. Fornari, E. Ledinot, E. Nassor, and R. De Simone. Esterel: A formal method applied to avionic software development. *Science of Computer Programming*, 36(1):5–25, Jan. 2000.

[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 méthode 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] J. Buck and R. Vaidyanathan. Heterogeneous modeling and simulation of embedded systems in El Greco. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign (CODES)*, San Diego, California, May 2000.

[7] M. Chiodo, D. Engels, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, K. Suzuki, and A. Sangiovanni-Vincentelli. A case study in computer-aided co-design of embedded controllers. *Design Automation for Embedded Systems*, 1(1):51–67, Jan. 1996.

[8] 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, California, June 1995.

[9] F. Clouté, J.-N. Contensou, D. Esteve, P. Pampagnin, P. Pons, and Y. Favard. Hardware/software co-design of an avionics communication protocol interface system: an industrial case study. In *Proceedings of the Seventh International Workshop on Hardware/Software Codesign (CODES)*, pages 48–52, Rome, Italy, May 1999.

[10] L. Lavagno and E. Sentovich. ECL: A specification environment for system-level design. In *Proceedings of the 36th Design Automation Conference*, pages 511–516, New Orleans, Louisiana, June 1999.

[11] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

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

[13] M. Sgroi, L. Lavagno, Y. Watanabe, and A. 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] T. R. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *Proceedings of the European Design and Test Conference*, pages 328–333, Paris, France, Mar. 1996.

Berry et al.'s Esterel site: http://www.inria.fr/meije/esterel/