Compiling Esterel into Static Discrete-Event Code

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

Vimal Kapadia and Michael Halas
IBM
Poughkeepsie
New York, USA
vimal@kapadia.us michael@halas.us

Abstract

Executing concurrent specifications on sequential hardware is important for both simulation of systems that are eventually implemented on concurrent hardware and for those most conveniently described as a set of concurrent processes. As with most forms of simulation, this is easy to do correctly but difficult to do efficiently. Solutions such as preemptive operating systems and discrete-event simulators present significant overhead.

In this paper, we present a technique for compiling the concurrent language Esterel into very efficient C code. Our technique minimizes runtime overhead by making most scheduling decisions at compile time and using a very simple linked-list-based event queue at runtime.

While these techniques work particularly well for Esterel with its high-level concurrent semantics, the same technique could also be applied to efficiently execute other concurrent specifications.

1 Introduction

Synchronous languages such as Esterel [5] and Lustre [6] hold promise as formal design languages because of well-defined semantics and powerful primitives. Already in substantial industrial use [2], the synchronous languages stand poised to grow in importance and popularity.

Automatic translation of a synchronous language specification into efficient executable code—the subject of this paper—finds at least two common applications in a typical design flow. Although synchronous languages are well-suited to formal verification, simulation is still of great importance in a typical

⋆ Edwards and his group are supported by an NSF CAREER award, a grant from Intel corporation, an award from the SRC, and from New York State's NYSTAR program.

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

design flow and as is always the case with simulation, faster is always better. Furthermore, the final implementation may also involve single-threaded code running on a microcontroller; automatically generating this from the specification can be a great help in reducing implementation mistakes.

Unfortunately, the concurrent semantics of the synchronous languages make their implementation on uniprocessors problematic since the concurrency must somehow be simulated. The strength of any compilation technique for a synchronous language lies in its approach to this problem. See Edwards [11] for a survey of such approaches.

In this paper, we address the problem of generating efficient single-threaded code (specifically, a C program) from the synchronous language Esterel. New techniques for compiling Esterel are needed because of its growing popularity and because its combination of fine-grained concurrency with inexpensive, broadcast communication makes its compilation challenging.

Many techniques to compile Esterel have been proposed during the twenty years the language has existed. The first Esterel compiler [5] derived a single automaton for an entire concurrent program. This technique produces very efficient code at the cost of exponentially larger object code, making it impractical for all but the smallest programs. For example, only the three smallest examples in our experiments can be compiled using this technique.

The next generation of Esterel compilers [3] translated Esterel programs into circuits, topologically sorted the gates, then generated simple code for each gate. While this technique scales much better than the automaton compilers, it does so at a great loss in speed. The fundamental problem is that the program must execute code for every statement in every cycles, even for statements that are not currently active.

Our technique most closely resembles the SAXO-RT compiler developed at France Telecom [8]. Both techniques divide an Esterel program into "basic blocks" that are then sorted topologically and executed selectively based on run-time scheduling decisions. Our technique differs in two main ways. First, we only schedule blocks within the current cycle, which makes it unnecessary to ever *unschedule* a node (Esterel's preemption constructs can prevent the execution of statements that would otherwise run). Second, because of this, instead of a scoreboard-based scheduler, we are able to use one based on linked lists that eliminates conditional tests.

Maurer's Event-Driven Conditional-Free paradigm [12] also inspired us, although his implementation is geared to logic network simulation and does not appear applicable to Esterel. Interestingly, he writes his examples using a C-like notation that resembles the gcc computed *goto* extension we use, but apparently he uses inline assembly instead of the gcc extension.

Our technique also resembles that in Edwards' [9] EC compiler, but ours uses a more dynamic scheduler. During a cycle, the Synopsys compiler maintains a set of state variables, one per running thread. At each context switch point, the compiler generates code that performs a multi-way branch on one

of these variables. While this structure is easier for the compiler to analyze, it is not able to quickly skip over as much code as the technique presented here.

Potop-Butucaru's compiler [13,14] currently generates the fastest code for most large examples (we beat it on certain ones). His technique has the advantage of actually optimizing many of the state decisions in a program, something we have not yet applied to our compiler, and uses a technique much like the Synopsys compiler to generate code. Our results suggest that our technique can be superior for highly-concurrent programs.

2 Esterel

Berry's Esterel language [5] is an imperative concurrent language whose model of time resembles that in a synchronous digital logic circuit. The execution of the program progresses a cycle at a time and in each cycle, the program computes its output and next state based on its input and the previous state by doing a bounded amount of work; no intra-cycle loops are allowed.

Esterel is a concurrent language in that its programs may contain multiple threads of control. Unlike typical multi-threaded software systems, however, Esterel's threads execute in lockstep: each sees the same cycle boundaries and communicates with other threads using a disciplined broadcast mechanism.

Esterel's threads communicate through signals, which behave like wires in digital logic circuits. In each cycle, each signal takes a single Boolean value (*present* or *absent*) that does not automatically persist between cycles. Inter-thread communication is simple: within a cycle, any thread that reads the value of a signal must wait for any other threads that set that signal's value.

Statements in Esterel either execute within a cycle (e.g., *emit* makes a given signal present in the current cycle, *present* tests a signal) or take one or more cycles to complete (e.g., *pause* delays a cycle before continuing, *await* waits for a cycle in which a particular signal is present). Strong preemption statements check a condition in every cycle before deciding whether to allow their bodies to execute. For example, the *every* statement performs a reset-like action by restarting its body in any cycle in which its predicate is true.

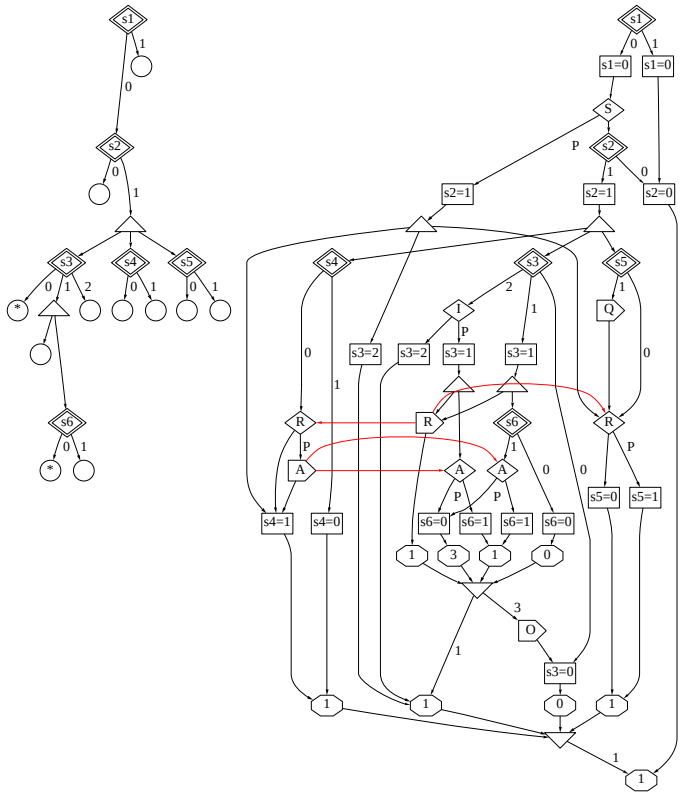
3 Executing Esterel

Esterel's semantics require any implementation to deal with three issues: the concurrent execution of sequential threads of control within a cycle, the scheduling constraints among these threads due to communication dependencies, and how (control) state is updated between cycles.

Our compiler generates C code that executes concurrently-running threads by dispatching small groups of instructions that can run without a context switch. These blocks are dispatched by a scheduler that uses linked lists of pointers to code that will be executed in the current cycle. The scheduling constraints are analyzed completely by the compiler before the program runs

```
module Example:
input I, S;
output O, Q;
signal R, A in
  every S do
      await I;
      weak abort
        sustain R
      when immediate A;
      emit O
    ||
      loop
        pause; pause;
        present R then
          emit A
        end present
      end loop
    ||
      loop
        present R then
          pause; emit Q
        else
          pause
        end present
      end loop
  end every
end signal
end module
```

(a)



(b)

Fig. 1. An Example. (a) A simple Esterel module modeling a shared resource. (b) The (simplified) GRC graph, consisting of a selection tree and a control-flow graph.

and affects both how the Esterel programs are divided into blocks and the order in which the blocks may execute. Control state is held between cycles in a collection of variables encoded with small integers.

4 The GRC Representation

We will illustrate the operation of our compiler on the small Esterel program in Fig. 1(a). It models a shared resource using three groups of concurrently-running statements. The first group (*await I* through *emit O*) takes a request from the environment on signal I and passes it to the second group of statements (*loop* through *end loop*) on signal R. The second group responds to requests on R with the signal A in alternate cycles. The third group simply makes Q a delayed version of R.

This simple example illustrates many challenging aspects of compiling Esterel. For example, the first thread communicates with and responds to the

second thread in the same cycle, i.e., the presence of R is instantaneously broadcast to the second thread, which, if the *present* statement is running, observes R and immediately emits A in response. In the same cycle, emitting A causes the *weak abort* statement to terminate and send control to *emit O*.

As is often the case, the inter-thread communication in this example means that it is impossible to execute the statements in the first thread without interruption: those in the second thread may have to execute partway through. Ensuring the code in the two threads executes in the correct, interleaved order at runtime is the main compilation challenge.

Our compiler translates Esterel into a variant of Potop-Butucaru's [14] graph code (GRC). Shown in Fig. 1(b), GRC consists of a selection tree that represents the state structure of the program and an acyclic concurrent control-flow graph that represents the behavior of the program in each cycle. A straightforward syntax-directed translation produces this GRC from the program's abstract syntax tree. The control-flow portion of GRC is equivalent to the concurrent control-flow graph described in Edwards [10].

4.1 The Selection Tree

The selection tree (left of Fig. 1(b)) is the simpler half of the GRC representation. The tree consists of three types of nodes: leaves (circles) that represent atomic states, e.g., *pause* statements; exclusive nodes (double diamonds) that represent choice, i.e., if an exclusive node is active, exactly one of its subtrees is active; and fork nodes (triangles) that represent concurrency, i.e., if a fork node is active, all of its subtrees are active.

Although the selection tree is used during the optimization phase of our compiler, for the purposes of code generation it is just a complicated way to enumerate the variables needed to hold the control state of an Esterel program between cycles. Specifically, each exclusive node represents an integer-valued variable that stores which of its children may be active in the next cycle. In Fig. 1(b), these are labeled s1 through s6. We encode these variables in the obvious way: 0 represents the first child, 1 represents the second, and so forth.

4.2 The Control-Flow Graph

The control-flow graph (right of Fig. 1(b)) is a much richer object and the main focus of the code-generation procedure. It is a traditional flowchart consisting of actions (rectangles and pointed rectangles, indicating signal emission) and decisions (diamonds) augmented with fork (triangles), join (inverted triangles), and terminate (octagons) nodes.

The control-flow graph is executed once from entry to exit for each cycle of the Esterel program. The nodes in the graph test and set the state variables represented by the exclusive nodes in the selection tree and test and set Boolean variables that represent the presence/absence of signals.

The fork, join, and terminate nodes are responsible for Esterel's concurrency and exception constructs. When control reaches a fork node, it is passed to all of the node's successors. Such separate threads of control then wait at the corresponding join node until all the incoming threads have arrived.

Esterel's structure induces properly nested forks and joins. Specifically, each fork has exactly one matching join, control does not pass among threads before the join, and control always reaches the join of an inner fork before reaching a join of an outer fork. In Fig. 1(b), each join node has two corresponding forks, and the topmost two forks are owned by the lowest join.

Together, join nodes—the inverted triangles in Fig. 1(b)—and their predecessors, terminate nodes[2]—the octagons—implement two aspects of Esterel's semantics: the "wait for all threads to terminate" behavior of concurrent statements and the "winner-take-all" behavior of simultaneously-thrown exceptions. Each terminate node is labeled with a small nonnegative integer completion code that represents a thread terminating (code 0), pausing (code 1), and throwing an exception (codes 2 and higher). Once every thread in a group started by a fork has reached the corresponding join, control passes from the join along its outgoing arc labeled with the highest completion code of all the threads. That the highest code takes precedence means that a group of threads terminates only when all of them have terminated (the maximum is zero) and that the highest-numbered exception—the outermost enclosing one—takes precedence when it is thrown simultaneously with a lower-numbered one. Berry [4] first described this clever encoding.

4.3 An Example

Consider the behavior of the program in Fig. 1(a) represented by the control-flow graph on the right of Fig. 1(b). The topmost node tests state variable s1, which is initially set to 1 to indicate the program has not yet started. The test of S immediately below the nodes that assign 0 to s1 implements the *every S* statement by restarting the two threads when S is present (indicated by the label P on the arc from the test of S). The test of s2 just below S encodes whether the body of the *every* has started and should be allowed to proceed.

The fork just below the rightmost s2=1 action resumes the three concurrent statements by sending control to the tests of state variables s3, s4, and s5. Variable s3 indicates whether the first thread is at the *await I* (=2), sustaining R while checking for A (=1), or has terminated (=0). Variable s6 could actually be removed. It is a side effect arising from how our compiler translates the *weak abort* statement into two concurrent statements, one of which tests A. The variable s6 indicates whether the statement that tests A has terminated, something that can never happen.

[2] Instead of terminate and join nodes, Potop-Butucaru's GRC uses a single type of node, *sync*, with distinct input ports for each completion code. Our representation is semantically equivalent and easier to represent.

When s3 is 1 or s3 is 2 and I is present, these two threads emit R and test A. If A is present, control passes through the terminate 3 node to the inner join. Because this is the highest exit level (the other thread, which emits R, always terminates at level 1), this causes control to pass from the join along the arc labeled 3 to the node for *emit O* and to the action s3=0, which effectively terminates this thread.

The second thread, topped by the test of s4, either checks R and emits A in response, or simply sets s4 to 0 so it will be checked in the next cycle.

The third thread, which starts at the test of s5, initially emits Q if s5 is 1, then sets s5 to 1 if R is present.

Although the behavior of the state assignments, tests, and completion codes is fairly complicated, it is easy to translate into imperative code. Unfortunately, concurrency complicates things: because two of the threads cannot be executed atomically since the presence of signals R and A must be exchanged during their execution within a cycle. Generating sequential code that implements this concurrency is our main contribution.

5 Sequential Code Generation

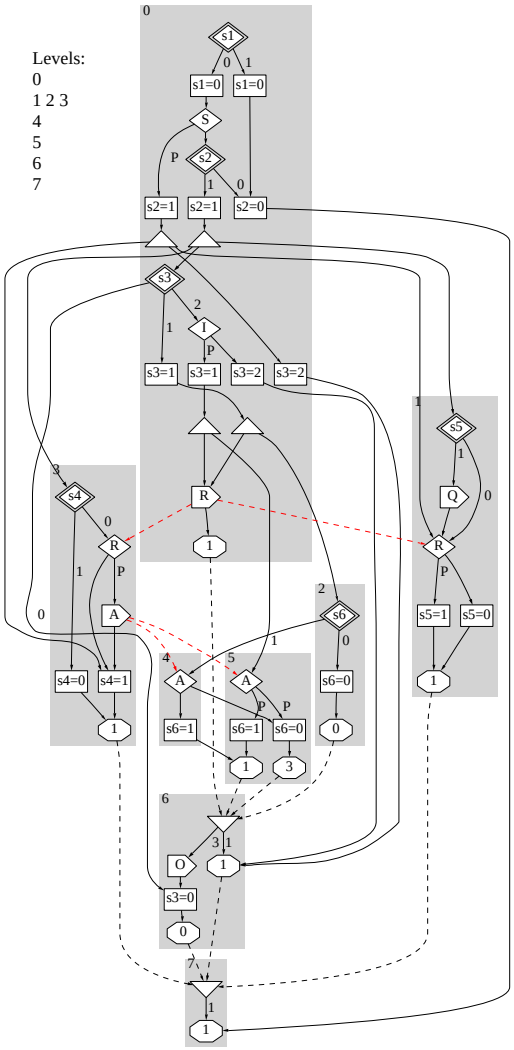
Our code generation technique relies on the following observations: while arbitrary clusters of nodes in the control-flow graph cannot be executed without interruption, many large clusters often can be; these clusters can be chosen so that each is invoked by at most one of its incoming control arcs; because of concurrency, a cluster's successors may have to run after some intervening clusters have run; and groups of clusters without any mutual data or control dependency can be invoked in any order (i.e., clusters are partially ordered).

Our key contribution comes from this last observation: because the clusters within a level can be invoked in any order, we can use a very inexpensive singly-linked list to track which clusters must be executed in each level. By contrast, most discrete-event simulators [1] are forced to use a more costly data structure such as a priority queue for scheduling.

The overhead in our scheme approaches a constant amount per cluster *executed*. By contrast, the overhead of the SAXO-RT compiler [8] is proportional to the total number of clusters in the program, regardless of how many actually execute in each cycle, and the overhead in the netlist compilers is even higher: proportional to the number of statements in the program.

Our compiler divides a concurrent control-flow graph into clusters of nodes that can execute atomically and orders these clusters into levels that can be executed in any order. The generated code contains a linked list for each level that stores which clusters need to be executed in the current cycle. The code for each cluster usually includes code for scheduling a cluster in a later level: a simple insertion into a singly-linked list.

Fig. 2(a) shows the effect of running our clustering algorithm on the control-flow graph of Fig. 1(b). The algorithm identified eight clusters, but



(a)

```
#define sched1a next1 = head1, head1 = &&C1a
#define sched1b next1 = head1, head1 = &&C1b
#define sched2  next2 = head1, head1 = &&C2
#define sched3a next3 = head1, head1 = &&C3a
#define sched3b next3 = head1, head1 = &&C3b
#define sched4  next4 = head2, head2 = &&C4
#define sched5a next5 = head3, head3 = &&C5a
#define sched5b next5 = head3, head3 = &&C5b
#define sched5c next5 = head3, head3 = &&C5c
#define sched6a next6 = head4, head4 = &&C6a
#define sched6b next6 = head4, head4 = &&C6b
#define sched6c next6 = head4, head4 = &&C6c
#define sched7a next7 = head5, head5 = &&C7a
#define sched7b next7 = head5, head5 = &&C7b

int cycle() {
  void *next1;
  void *next2;
  void *next3;
  /* other next pointers */

  void *head1 = &&END_LEVEL_1;
  void *head2 = &&END_LEVEL_2;
  /* other level pointers */

  if (s1) { s1 = 0; goto N26; }
  else {
    s1 = 0;
    if (S) {
      s2 = 1; code0 = -1;
      sched7a; sched1b; sched3b;
      s3 = 2; sched6b;
    } else {
      if (s2) {
        s2 = 1;
        code0 = -1;
        sched7a; sched1a; sched3a;
        switch (s3) {
        case 0: sched6c; break;
        case 1:
          s3 = 1; code1 = -1;
          sched6a; sched2; goto N38;
        case 2:
          if (I) {
            s3 = 1; code1 = -1;
            sched6a; sched5a;
       N38: R = 1; code1 &= -(1 << 1);
          } else { s3 = 2; sched6b; }
          break;
      } } else {
   N26: s2 = 0; sched7b;
  } } }
  goto *head1;

C1a: if (s5) Q = 1;
C1b: if (R) s5 = 1;
     else s5 = 0;
     code0 &= -(1 << 1);
     goto *next1;

C2:  if (s6) sched4;
     else s6 = 0;
     goto *next2;

C3a: if (s4) s4 = 0;
     else {
       if (R) A = 1;
C3b:   s4 = 1;
     }
     code0 &= -(1 << 1);
     goto *next3;

END_LEVEL1: goto *head2;
```

(b)

Fig. 2. (a) The control-flow graph from Fig. 1(b) divided into blocks. Control arcs reaching join nodes have been replaced with (dashed) data dependencies to guarantee each block has at most one active incoming control arc. (b) The code our compiler generates for the first two levels: clusters 0, 1, 2, and 3 (reformatted to fit space).

this is no ideal: a better algorithm would have combined clusters 4 and 5, but it is not surprising that our simple-minded algorithm misses the optimum since the optimum scheduling problem is NP-complete (see Edwards [10]).

After eight clusters were identified, our levelizing algorithm, which uses a simple relaxation technique, grouped them into the six levels listed at the top of Fig. 2(a). It observed that clusters 1, 2, and 3 have no interdependencies, can be executed in any order, and placed them together in the second level. The other clusters are all interdependent and must be executed in the order identified by the levelizing algorithm.

Our main contribution is our semi-dynamic scheduler based on a sequence of linked lists. The generated code maintains a linked list of entry points for each level. In Fig. 2(b), the `head1` variable points to the head of the linked list for the first level (the complete code has more such variables) and the `next1` through `next3` variables point to the successors of clusters 1 through 3.

The code in Fig. 2(b) takes advantage of gcc's computed *goto* extension to C. This makes it possible to take the address of a label, store it in a void pointer (e.g., `head1 = &&C1a`) and later branch to it (e.g., `goto *head1`) provided this does not cross a function boundary. While not strictly necessary (in fact, we include a compiler flag that changes the generated code to use *switch* statements embedded in loops instead of *gotos*), using this extension substantially reduces scheduling overhead since a typical switch statement requires at least two bounds checks plus either a jump table lookup or a cascade of conditionals.

Fig. 3 illustrates the behavior of these linked lists. Fig. 3(a) shows the condition at the beginning of every cycle: every level's list is empty—the *head* pointer for each level points to its *END_LEVEL* block. If no blocks where scheduled, the program would execute the code for cluster 0 only.

Fig. 3(b) shows the pointers after executing *sched3a*, *sched1b*, and *sched4* (note: this particular combination cannot occur in practice). Invoking the *sched3a* macro (see Fig. 2(b)) inserts cluster 3 into the first level's linked list by setting next3 to the old value of head1—END_LEVEL1—and setting head1 to point to C3a. Invoking *sched1b* is similar: it sets next1 to the new value of head1—C3a—and sets head1 to C1b. Finally, invoking *sched4* inserts cluster 4 into the linked list for the second level by setting next4 to the old value of head2—END_LEVEL2—and setting head2 to C4. This series of scheduling steps produces the arrangement of pointers shown in Fig. 3(b).

Because clusters in the same level may be executed in any order, clusters in the same level can be scheduled cheaply by inserting them at the beginning of the linked list. The `sched` macros do exactly this. Note that the level of each cluster is hardwired since this information is known at compile time.

A powerful invariant that arises from the structure of the control-flow graph is the guarantee that each cluster can be scheduled at most once during any cycle. This makes it unnecessary for the generated code to check that it never inserts a cluster in a particular level's list more than once.

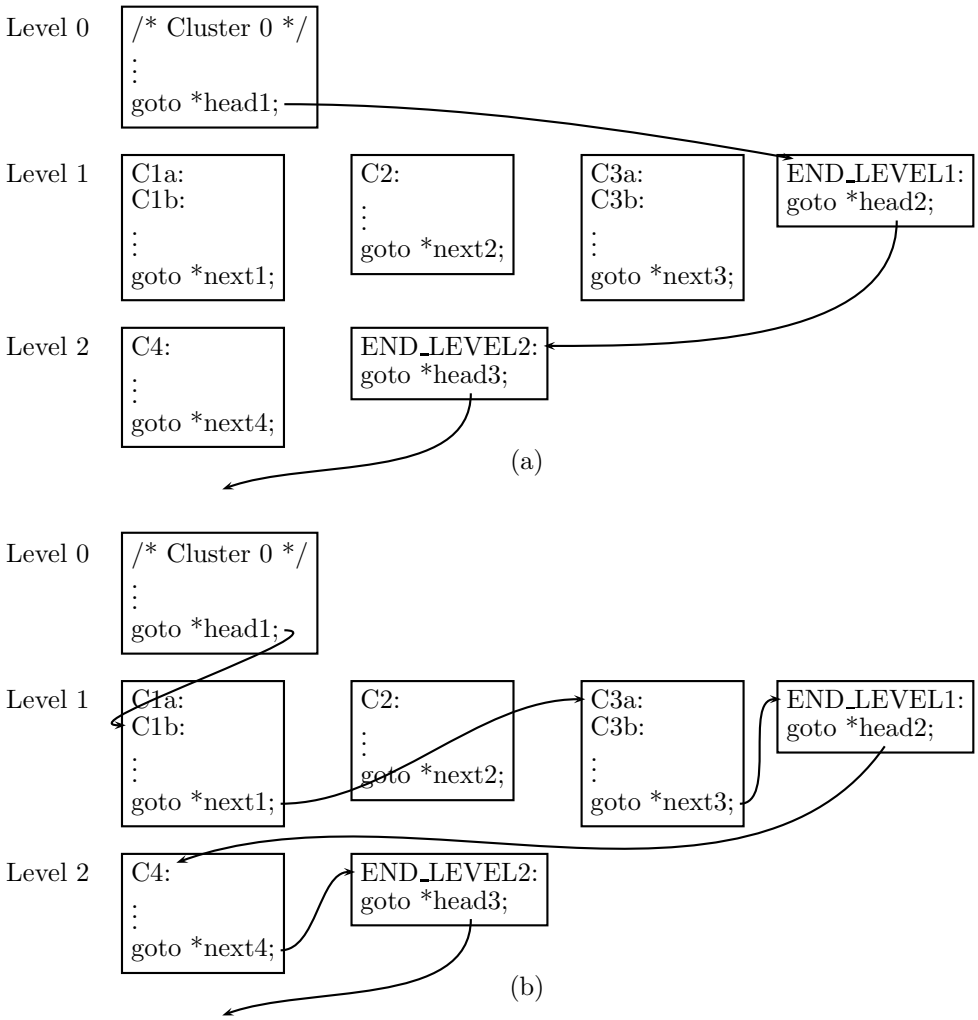


Fig. 3. Cluster code and the linked list pointers. (a) At the beginning of a cycle. (b) After executing sched3a, sched1b, and sched4.

As is often the case, both cluster 1 and 3 have multiple entry points. This is easy to support because the structure of the graph guarantees that at most one entry point for each cluster will be be scheduled each cycle.

We use the dominator-based code structuring algorithm described in Edwards [10] to generate structured code for each cluster. Some *goto*s are necessary to avoid duplicating code. Fig. 2(b) has two: `N26` and `N38`.

6 The Clustering Algorithm

Fig. 4 shows our clustering algorithm. It is heuristic and certainly could be improved, but is correct and produces reasonable results.

One important modification is made to the control-flow graph before our clustering algorithm runs: all control arcs leading to join nodes are removed and replaced with data dependency arcs, and a control arc is added from each

```
 1: add the topmost control-flow graph node to F, the frontier set
 2: while F is not empty do
 3:   randomly select and remove f from F
 4:   create a new, empty pending set P
 5:   add f to P
 6:   set C_i to the empty cluster
 7:   while P is not empty do
 8:     randomly select and remove p from P
 9:     if p is not clustered and all of p's predecessors are then
10:       add p to C_i (i.e., cluster p)
11:       if p is not a fork node then
12:         add all of p's control successors to P
13:       else
14:         add the first of p's control successors to P
15:       add all of p's successors to F
16:       remove p from F
17:   if C_i is not empty then
18:     i = i + 1 (move to the next cluster)
```

Fig. 4. The clustering algorithm. This takes a control-flow graph with information about control and data predecessors and successors and produces a set of clusters $\{C_i\}$, each of which is a set of nodes that can be executed without interruption.

fork to its corresponding *join*. This guarantees that no node ever has more than one active incoming control arc (before this change, each *join* had one active incoming arc for every thread it was synchronizing). Fig. 2(a) partially reflects this restructuring: the additional arcs off the forks have been omitted to simplify an already complex diagram. This transformation also simplifies the clustering algorithm, which would otherwise have to handle *join*s specially.

The algorithm manipulates two sets of CFG nodes. The frontier set F holds the set of nodes that might start a new cluster, i.e., those nodes with at least one clustered predecessor. F is initialized in line 1 with the first node that can run—the entry node for the control-flow graph—and is updated in line 15 when the node p is clustered. The pending set P, used by the inner loop in lines 7–16, contains nodes that could be added to the existing cluster. P is initialized in line 5 and updated in lines 12–14.

The algorithm consists of two nested loops. The outermost (lines 2–18) selects a node f at random from the frontier F (line 3) and tries to start a cluster around it by adding it to the pending set P (line 5). The innermost (lines 7–16) selects a node p at random from the pending set P (line 8) and tries to add it to the current cluster C_i.

The test of p's predecessors in line 9 is key. It ensures that when a node p is added to the current cluster, all its predecessors have already been clustered. This ensures that in the final program, all of p's predecessors will be executed before p. If this test succeeds, p is added to the cluster under construction in line 10.

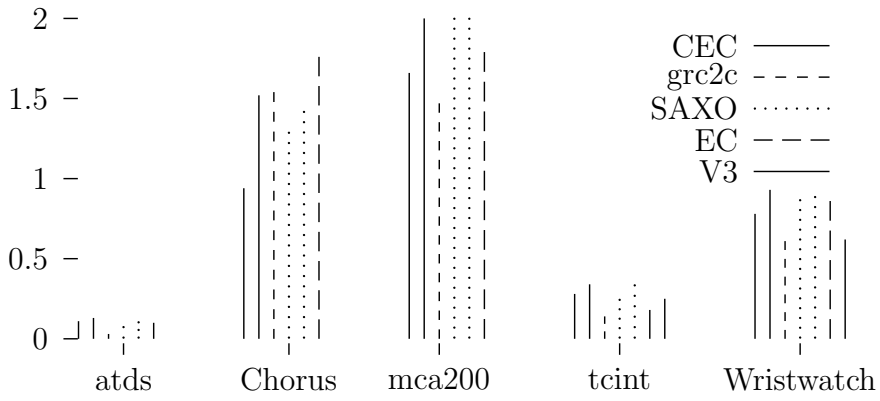


Fig. 5. Experimental results. The height of each line indicates the number of seconds taken to execute 1 000 000 reactions on a 1.7 GHz Pentium 4 (shorter is better). The two lines for CEC are for computed-goto and switch-based code. The two lines for the SAXO compiler for a "fast" and normal version of the code. grc2c is due to Potop-Butucaru [14], SAXO is due to Closse et al. [8], EC is the Synopsys compiler due to Edwards [10], and V3 is the automata-based compiler [5].

All of p's control successors are added to the pending set in line 12 if p is not a fork node, and only the first if p is a fork (line 14). This test partially breaks clusters at fork nodes, ensuring that all the nodes within a cluster are connected with sequential control flow, i.e., they do not run concurrently. Always choosing the first successor under a fork is arbitrary and may not be the best. In general, the optimum choice of which thread to execute depends on the entire structure of the threads. But even the simple-minded rule of always executing the first thread under a fork, as opposed to simply scheduling it, greatly reduces the number of clusters and significantly improves performance.

7 Experimental Results

Fig. 5 shows our experimental results. Due to the paucity of large public-domain Esterel programs, we have only tested the speed of our compiler on five medium-sized examples. Table 1 reproduces these results numerically and Table 2 provides some statistics for the examples.

Our results are mixed: Potop-Butucaru's grc2c beats us on four of the five examples, but we are substantially faster on the largest example, Chorus. Furthermore, we are faster than the SAXO compiler on the three largest examples (Chorus, mca200, and Wristwatch). This is expected: our technique should become faster than the SAXO compiler on larger examples since our (similar) technique has less overhead for unexecuted parts of the program.

The number of clusters and levels in Table 2 suggests why our technique is better on some programs and worse on others. A key contribution of our technique is the use of one linked list per level for scheduling. The more clusters there are per level (measured, e.g., by the C/L average in Table 2),

Example	CEC	(switch)	grc2c	SAXO	(fast)	EC	V3	V5
atds	0.11	0.13	0.03	0.11	0.08	0.10		66.0
Chorus	0.94	1.52	1.54	1.42	1.29	1.76		51.0
mca200	1.66	2.75	1.47	2.62	2.35	1.79		29.0
tcint	0.28	0.34	0.14	0.34	0.25	0.18	0.25	1.3
Wristwatch	0.78	0.93	0.61	0.89	0.87	0.86	0.62	2.1

Table 1

Numerical version of Fig. 5: time, in seconds, to run 1 000 000 iterations of the generated code. The (switch) column shows times for CEC generating *switch* statements instead of computed *goto*s. The (fast) column is for the fast version of the SAXO compiler. V3 represents automata-based code, which can only be generated for two of the examples. The V5 column lists times for code generated by the netlist-based V5 compiler.

Example	Description	Size	Clusters	Levels	C/L	Threads
atds	Video generator	622	156	16	9.8	138
Chorus	Task scheduler [15]	3893	662	22	30.1	563
mca200	Shock absorber [7]	5354	148	15	9.9	135
tcint	Bus controller	357	101	19	5.3	85
Wristwatch	Berry's example	360	87	13	6.7	87

Table 2

Statistics for the examples. Size is the number of Esterel source lines after *run* statement expansion and pretty-printing. Clusters is the number of clusters found by the algorithm in Fig. 4. Levels is the number of levels the clusters were compressed into. C/L is the ratio of clusters to levels. Threads is the number of concurrent threads as reported by the EC compiler [10].

the more our technique differentiates itself from the SAXO compiler. The results bear this out: Chorus, which has the largest number of clusters per level on average exhibits the largest improvement over the other techniques.

We also ran the netlist-based V5 compiler on these examples and found that the runtimes are uniformly much worse than any of the other compilers. The results for the Wristwatch show the least variance because it calls the most user-defined functions, something none of these compilers attempt to optimize.

These timing results were obtained by applying a random sequence of inputs to the code generated by each compiler and measuring the time it took to execute 1 000 000 reactions. We note that the ratio of our measured times differ noticeably from those reported by Potop-Butucaru [14]. This can be attributed to a variety of factors including a different processor (Potop-Butucaru used a P3, our results are on a P4) and perhaps different stimulus.

8 Conclusions

In this paper, we presented an improved way to generate code for Esterel that uses linked lists to track which blocks of code are to be executed. This results in improved running times over an existing compiler (SAXO-RT [8]) that uses a similar technique based on bit-mapped flags and in the largest example we tried, a substantial improvement over the previously fastest-known compiler (Potop-Butucaru's grc2c [14]).

While our technique is not an improvement of the same magnitude as the move from netlist-style compilers, it is competitive, generates readable code, and appears to work especially well on large concurrent programs.

Source and object code for the compiler described in this paper is freely available as part of the Columbia Esterel Compiler distribution available from `http://www.cs.columbia.edu/~sedwards/cec/`.

References

[1] J. Banks, J. S. Carson, B. L. Nelson, and D. M. Nicol. *Discrete-Event System Simulation*. Prentice Hall, Upper Saddle River, New Jersey, third edition, 2000.

[2] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, Jan. 2003.

[3] G. Berry. Esterel on hardware. *Philosophical Transactions of the Royal Society of London. Series A*, 339:87–103, Apr. 1992. Issue 1652, Mechanized Reasoning and Hardware Design.

[4] G. Berry. Preemption in concurrent systems. In *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *Lecture Notes in Computer Science*, pages 72–93, Bombay, India, Dec. 1993. Springer-Verlag.

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

[6] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *ACM Symposium on Principles of Programming Languages (POPL)*, Munich, Jan. 1987. Association for Computing Machinery.

[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] E. Closse, M. Poize, J. Pulou, P. Venier, and D. Weil. SAXO-RT: Interpreting Esterel semantic on a sequential execution structure. In *Proceedings of*

Synchronous Languages, Applications, and Programming (SLAP), volume 65.5 of *Electronic Notes in Theoretical Computer Science*, Grenoble, France, Apr. 2002. Elsevier Science.

[9] S. A. Edwards. Compiling Esterel into sequential code. In *Proceedings of the 37th Design Automation Conference*, pages 322–327, Los Angeles, California, June 2000. Association for Computing Machinery.

[10] S. A. Edwards. An Esterel compiler for large control-dominated systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(2):169–183, Feb. 2002.

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

[12] P. M. Maurer. Event driven simulation without loops or conditionals. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 23–26, San Jose, California, Nov. 2000.

[13] D. Potop-Butucaru. *Optimizing for Faster Simulation of Esterel Programs.* PhD thesis, INRIA, Sophia-Antipolis, France, Aug. 2002.

[14] D. Potop-Butucaru. Optimizations for faster execution of Esterel programs. In *Proceedings of Memocode*, pages 227–236, Mont St. Michel, France, June 2003.

[15] D. Weil, V. Bertin, E. Closse, M. Poize, P. Venier, and J. Pulou. Efficient compilation of Esterel for real-time embedded systems. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 2–8, San Jose, California, Nov. 2000.