An Esterel Virtual Machine for Embedded Systems

Becky Plummer [1]

Department of Computer Science
Columbia University
New York, USA

Mukul Khajanchi [2]

Department of Computer Science
Columbia University
New York, USA

Stephen A. Edwards [3]

Department of Computer Science
Columbia University
New York, USA

Abstract

Embedded systems often suffer from severe resource constraints such as limited memory for programs and data. In this work, we address the problem of compiling the Esterel synchronous language for processors with such constraints.

We introduce a virtual machine that executes a compact bytecode designed specifically for executing Esterel and present a compiler for it. Our technique generates code that is roughly half the size of optimized C code compiled using existing techniques.

We demonstrate the utility of our approach on the Lego RCX controller for the Mindstorms system. While we are not the first to execute Esterel on the RCX, our technique will allow larger programs than were previously possible.

[1] Email: rp2176@columbia.edu
[2] Email: mk2603@columbia.edu
[3] Email: sedwards@cs.columbia.edu Edwards and his group at Columbia 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. http://www.cs.columbia.edu/~sedwards

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

Embedded systems usually have limited resources such as power, size, computation speed, and memory. A key challenge, then, in implementing embedded systems is meeting requirements within these limits.

In this work, we address the problem of running reactive, embedded programs, specifically programs written in the Esterel synchronous language [2], in a constrained-memory environment. We propose an Esterel virtual machine whose instruction set has direct support for Esterel constructs—specifically concurrency—that otherwise require a fair amount of code on a normal, sequential processor. We simultaneously developed a compiler for the virtual machine that produces byte code that is roughly one-half the size of an equivalent optimized native executable.

We implemented our virtual machine on the Hitachi H8-based RCX microcontroller that is part of the Lego Mindstorms system.

Code compression for embedded system is a well-studied topic that has led to industrial solutions such as ARM's Thumb instruction set. This replaces the standard 32-bit ARM instruction set with a 16-bit variant that omits many instructions and register combinations. It generally provides a 20–30% reduction in code size. While using such a compact instruction set on compiled Esterel code would certainly work, the virtual-machine-based approach we propose achieves significantly higher compression ratios.

Running Esterel on the RCX microcontroller is also not novel, having been achieved before by Christophe Mauras and Martin Richard [4], with some help from Xavier Fornari. Their approach, however, is more traditional: like us, they use the BrickOS environment as their low-level interface to the hardware, but use a standard Esterel compiler that generates C that is cross-compiled onto the H8 microcontroller; their contribution is mostly in providing an API.

Roop et al. [7] have proposed an Esterel-specific instruction set, but their focus was on efficiency, not code size, and their approach appears to be limited to Esterel programs with no concurrency. For these reasons, we did not attempt to follow their work in designing our virtual machine.

Our compilation technique, built on the Columbia Esterel Compiler [4] translates the GRC-like intermediate representation [6,5] used within CEC into a bytecode of our own devising. We describe the intermediate representation in Section 2 and the bytecode in Section 3.

Our two contributions are the virtual machine and the compilation algorithm, which, like the algorithm devised by Edwards for the Synopsys Esterel compiler [3], translates a concurrent control-flow graph (i.e., GRC) into a sequential program with explicit context switches. We describe this in Section 4.

Finally, we present experimental results on our Lego RCX implementation in Section 6.

[4] http://www.emn.fr/x-info/lego/

2 Esterel and the GRC representation [5]

Berry's Esterel language [2] 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.

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

2.1 The GRC Representation

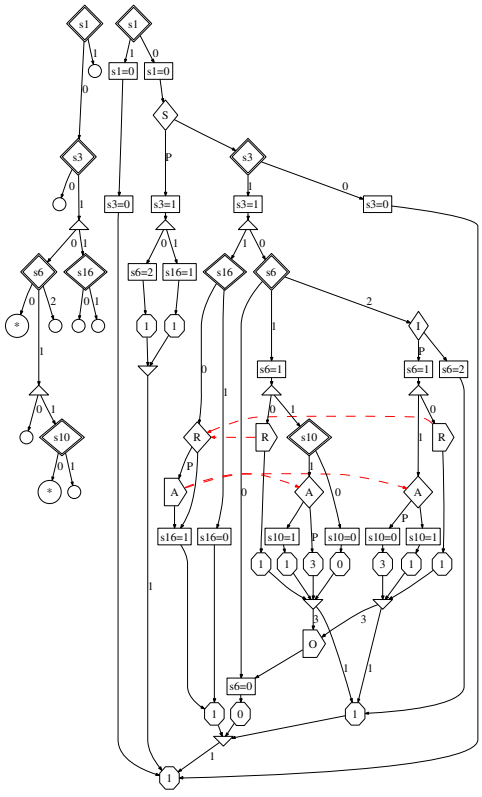
Consider the small Esterel program in Fig. 1(a). It models a shared resource using three groups of concurrently-running statements. The first group (*await I* through *emit O*) takes a request from the environment on signal I and passes it to the second group of statements (*loop* through *end loop*)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

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*.

[5] Much of this section was taken from Edwards, Kapadia, and Halas [4].

```
module Example:
input I, S;
output O;
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
  end every
end signal
end module
```

(a)



(b)

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

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.

The Columbia Esterel compiler translates Esterel into a variant of Potop-Butucaru's [6] 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 [3].

2.2 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.

2.3 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.

3 The BAL Virtual Machine

The design of our virtual machine arose from a desire to execute Esterel programs in as little memory as possible. Since Esterel programs, with their concurrency, preemption, and signals, behave very differently than, say, C programs, it seemed an obvious choice to implement a virtual machine whose instruction set was customized to Esterel semantics. We devised a compact eight-bit instruction set with just thirteen instructions, listed in Table 1.

Instructions in our virtual machine consist of one, two, or more bytes. The five low-order bits of the first byte encode the instruction type (we only use four currently; the other is for future expansion); higher-order bits in this byte sometimes encode additional information. For example, TWB uses the three higher order bits to distinguish whether it is testing state registers, signals, or termination codes.

Our virtual machine has four types of registers: thread program counters, signals, states, and completion codes. There may be up to 256 of each type, since each is indexed by a single byte; the exact number is a compile-time constant. Program counters hold the location where a thread will resume and are only accessed indirectly through the two *switch* instructions SWC and SWCU. Signal registers hold the presence/absence state of each signal, are set and cleared by SSIG and EMT, and are tested by the TWB and MWB instructions. State registers hold the state of threads *between* instants (cf. program counters, which hold the state of threads *within* an instant) and are set and tested by SSTT and TWB/MWB instructions. Completion code

Table 1
Summary of the Instruction Set

Opcode	**Description**	**Encoding(Hex)**
Signal, State and Thread Instructions		
SSIG	Set Signal	2A RR
	Clear Signal	0A RR
SSTT	Set State	0B RR VV
EMT	Emit a Signal	04 RR
STHR	Set Thread	07 TT HH LL
Control Flow Instructions		
END	Program End	03
EXIT	Terminate the Program	02
JMP	Jump	06 HH LL
NOP	No Operation	01
Branch, Switch and Terminate Instructions		
MWB	Multiway Branch On State	2D NL RR HH2 LL2 HH3 LL3 ...
	Multiway Branch On Completion Code	4D NL RR HH2 LL2 HH3 LL3...
TWB	Two Way Branch on State	29 RR HH LL
	Two Way Branch on Signal	49 RR HH LL
	Two Way Branch on Completion Code	69 RR HH LL
SWC	Switch Thread	05 TT
SWCU	Switch Unknown	0C
TRM	Set Completion Code for a Join	08 RR VV

RR = Register Number
VV = 8-bit Value
HH = High-order address byte
LL = Low-order address byte
NL = Number of labels
TT = Thread Number

registers store the exit level (i.e., 0 for terminate, 1 for pause, and 2 and higher for traps) for groups of concurrent threads following the usual Esterel numbering convention [1]. They are set by the TRM instruction and tested by TWB/MWB instructions.

The central challenge in the virtual machine was implementing Esterel's concurrency. So our virtual machine maintains a separate program counter for each thread and has context-switching instructions: SWC and SWCU.

The switch thread instruction, SWC, stores the PC for the current thread, takes an eight-bit thread number as an argument, and loads the PC from that thread number. A specialized version of this (SWCU) is used to switch back to the thread that called the current thread without having to pass the thread number as an argument.

One difficulty here was to handle the first invocation of any thread. Our compiler generates code at the beginning of the program that uses STHR instructions to initialize the current PC for each thread to a dedicated "not running" block that consists of a single SWCU instruction.

4 Sequential Code Generation

Our sequential code generation technique generates compact bytecode from CEC's GRC representation of the concurrent Esterel program. One of its main goals is to take advantage of the context-switching machinery in our VM, which we specifically designed to be easy. Generated C code requires a fair amount of overhead for each context switch, typically a *switch* statement; our VM allows us to encode a context switch in two bytes.

After our VM, our sequentializing algorithm is our main contribution. It adds context switches based on the schedule of nodes in the GRC representation of the Esterel program described in Section 2. Our addition to CEC is code that schedules the nodes, assigns a thread number to each node, sequentializes the graph, defines the path of execution, and finally outputs the BAL representation (Section 3). Fig. 3 shows the steps in generating BAL from GRC on a small example (a subset of Fig. 1(b)).

The first step in our algorithm is to use a simple topological sort to schedule the execution of the nodes in the program. Both control and data dependencies are considered in this phase (data dependencies are drawn as dashed lines in Fig. 1(b)), and we assume the graph is cycle-free.

Next, we assign a thread number to each node. This is straightforward—the topmost thread is numbered 0 and the threads under a fork are numbered sequentially starting with the next available thread number. Our one trick is to give the first child thread under each *fork* the same number as its parent. This is safe since the parent does not to run until all its children have terminated.

The next step is sequentialization, which we describe in detail in Section 5. Our algorithm introduces two new nodes to the GRC: *switch nodes* and *active points* (see the key in Fig. 2). An active point node represents the living

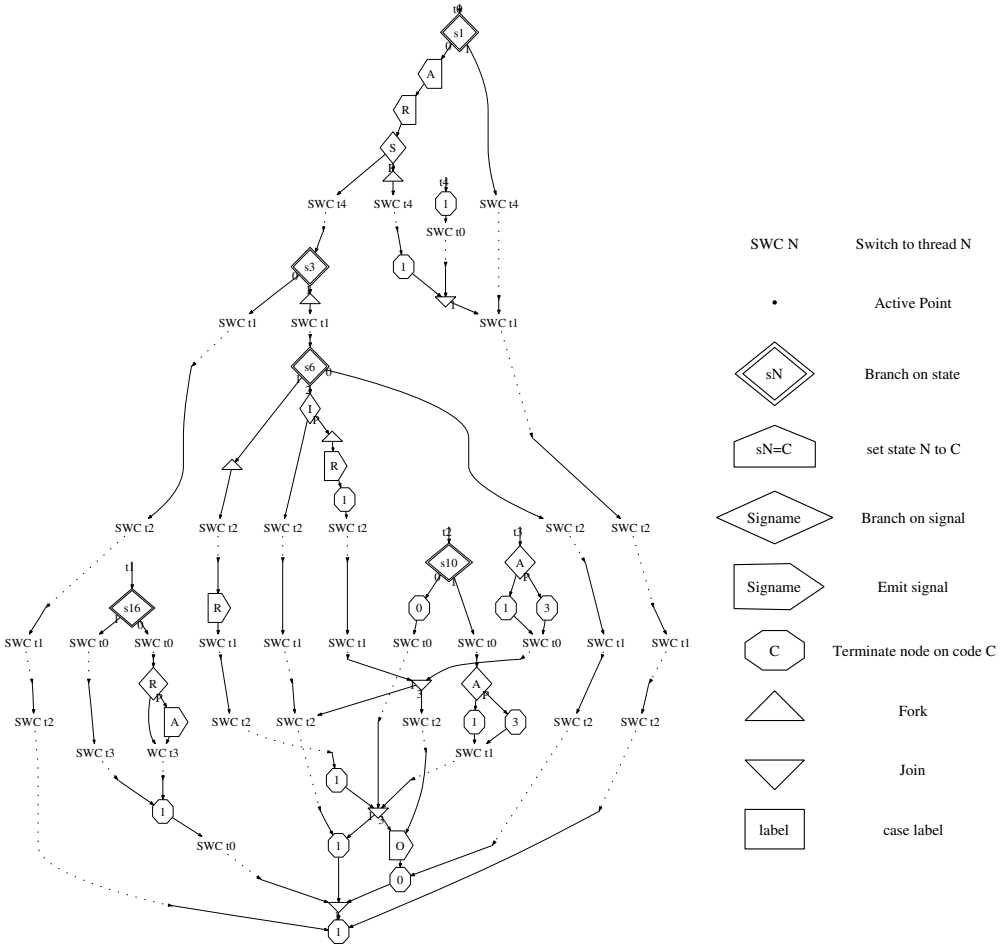
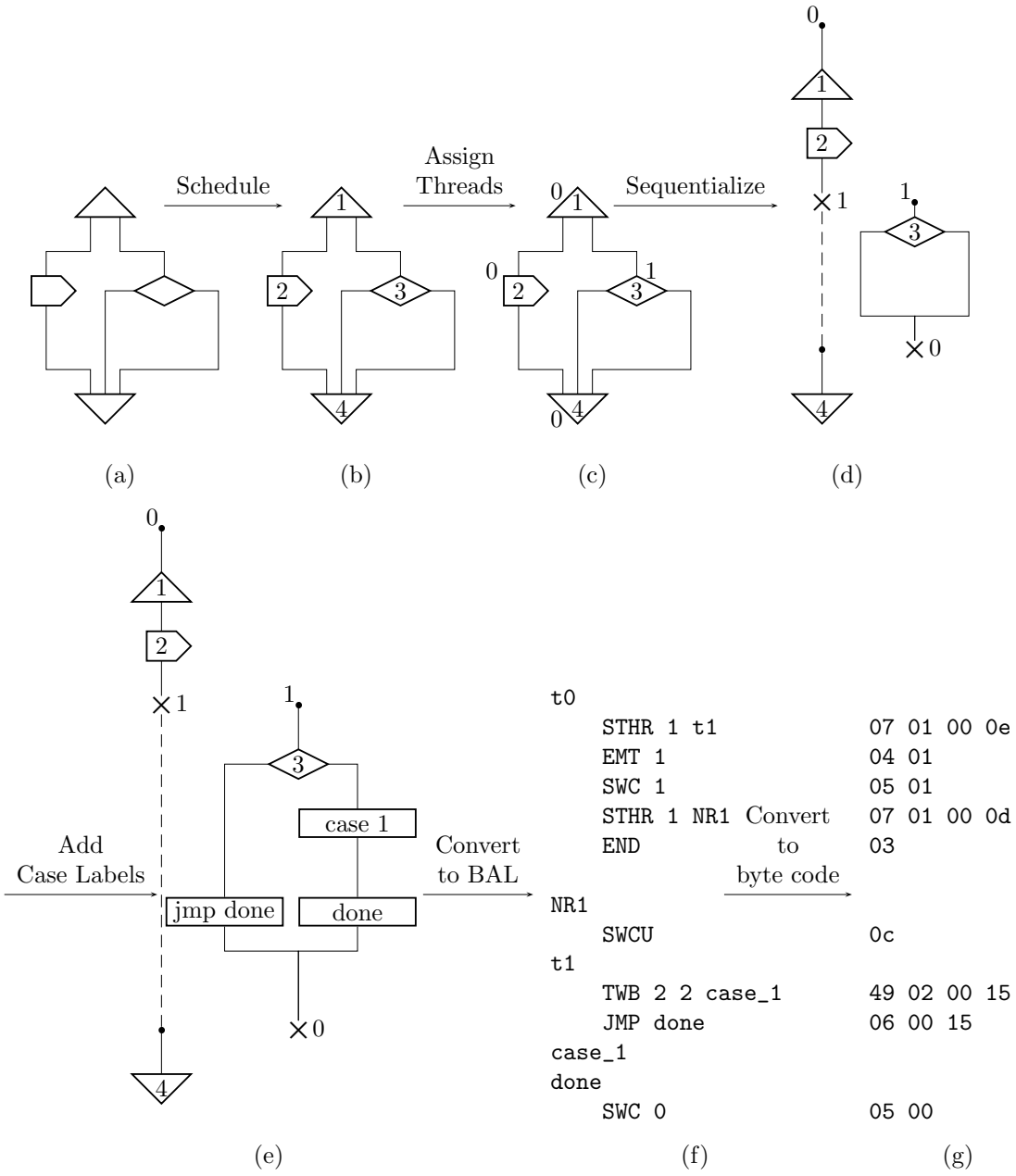


Fig. 2. The sequentialized graph of the concurrent Esterel program in Fig. 1(a).

sections of the threads where we have not yet processed the upcoming node. A switch node represents a context switch demanded by the schedule, i.e., when the next node in scheduled sequence belongs to a differently-numbered thread. Pairs of switch nodes are inserted at such context-switch points. Fig. 2 shows the effect of running our sequentializing algorithm on the Esterel example shown in Fig. 1(a).

After sequentialization, our compiler adds *case label*, *jump label*, and *done label* nodes to the graph to define the path of execution for the program (Fig. 3(e)). The locations of these are determined by computing the reverse immediate dominators of the nodes in the graph, which tells us where control reconverges after each switch and fork node (see Edwards [3] for details).

Finally, the BAL is generated by performing a depth-first search on the graph and generating a BAL instruction for each node. The depth-first search stops when it encounters a *jump label* node since we are guaranteed the code at the destination label will be generated at some other point. The BAL representation (Fig. 3(f)) is then assembled to get the byte code for the virtual machine, (Fig. 3(g)).




```
t0
    STHR 1 t1              07 01 00 0e
    EMT 1                  04 01
    SWC 1                  05 01
    STHR 1 NR1             07 01 00 0d
    END                    03

NR1
    SWCU                   0c
t1
    TWB 2 2 case_1         49 02 00 15
    JMP done               06 00 15
case_1
done
    SWC 0                  05 00
```

Fig. 3. Translating GRC to bytecode. Starting with a fragment of the concurrent GRC graph (a), we schedule the nodes in the graph (b) and assign thread numbers (c). Next, the graph is sequentialized as described in Section 5. After sequentialization (d), the execution path is set by adding case labels (e). It is then converted to BAL (f) and assembled to produce bytecode (g).

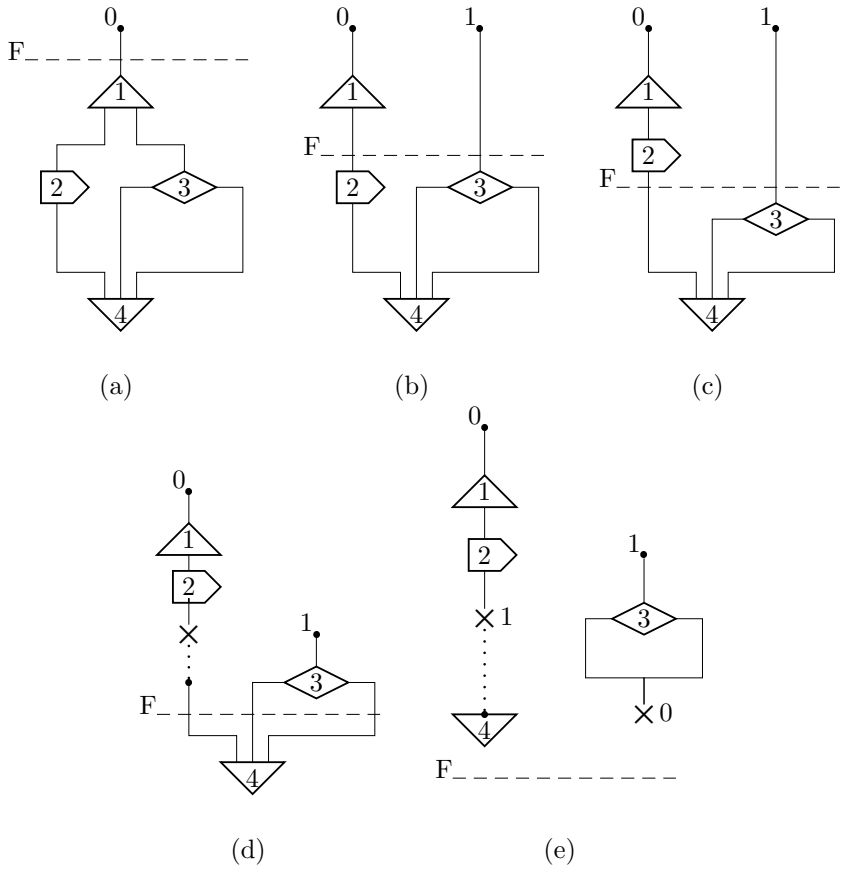


Fig. 4. The behavior of the sequentializing algorithm. The dotted line labeled F represents the frontier. The frontier starts at the top of the graph (a) and moves down a node at a time in scheduled order (b). When a node is in the same thread as the most recently moved one, it is simply moved above the frontier (c). However, when the next node is from a different thread, a switch is added to the previous thread and a active point is added to the new thread just above the just-moved node (d). The algorithm is complete when the frontier has swept across all nodes in scheduled order (e).

5 The Sequentializing Algorithm

Fig. 5 shows our sequentializing algorithm. Before the GRC representation of the Esterel program is sequentialized, node numbers and thread numbers are assigned to each node in the graph and the nodes are scheduled. After the nodes are scheduled, the sequentializing algorithm produces a version of the graph containing threads and context switches featured in Fig. 2.

The algorithm maintains two sets of CFG nodes, the parent set $P[t]$ and the active set $A[t]$, for each thread t. The parent set acts as the set of nodes for a thread that have already been visited and sequentialized. The active set is the set of nodes for each thread that have not yet been visited but are children of the nodes in the parent set. These two sets are separated by a *frontier*. Fig. 4 shows the frontier as a dashed line labeled F.

```
1: for each thread t in G do
2:    create new active point p
3:    copy first node n of t in G to n′ new node in G′
4:    connect p and n′
5:    add p to P[t]
6:    add n′ to A[t]
7: t′ = the first thread
8: for each node n in scheduled order do
9:    t is thread of n
10:   if t ≠ t′ then
11:      for each parent p in P[t′] do
12:         for each successor c of p in A[t′] do
13:            create switch node s from t′ to t
14:            connect s between p and c
15:      replace P[t′] with the set of new switch nodes
16:   move n to P[t] and remove it from A[t]
17:   for each unreached successor c of n do
18:      copy c to c′ new node in G′
19:      if n is a fork then
20:         add child to A[thread of c]
21:      else
22:         add child to A[t]
23:   t′ = t {remember the last thread}
```

Fig. 5. The sequentializing algorithm

When our algorithm considers the next node in the schedule, it checks whether the thread of this node is the same as the thread of the last node that was processed (line 10). If the threads are the same, then the node is simply moved from the active set into the parent set (line 16) and the successors of the node are added to the active set (lines 17–22). This is why node 2 simply moves above the frontier on Fig. 4(c). If the threads are not the same then switch nodes are added between the parents and their children in the active set for the last thread (lines 11–14). Then the current node is considered. Hence, in Fig. 4(d), when node 3 gets processed, a switch node gets inserted into thread 0 after node 2 but before node 4. Once the context switch and active point are added between the parent and child, the context switch moves into the processed section and becomes the new parent for the child node. Fig. 4(d) shows the state after the switch node and active point have been added to the parent set of thread 0.

The algorithm consists of several parts. The initialization (lines 1–6) of the parent set and active set involves creating an active point and connecting it to the first node in the thread (lines 2–4). The active point is added to the parent set and the first node to the active set for that thread (lines 5–6). In the next section, the algorithm processes each node in scheduled order by

moving it from the active set into the parent set and then adding each of its children to the active set (lines 8–23).

The testing of the thread (line 10) is the key part of the algorithm. We test if the thread number of the current node is the same as the last node that was processed. If they are not the same, then we need to execute a context switch. The context switch is created by considering each node in the parent set for the last thread. For each parent and active child pair a switch node is created and inserted between them (lines 11–14).

6 Experimental Results

We ran our virtual machine on a Pentium 4-class desktop machine and also ported it to the Hitachi H8-based RCX microcontroller used by the Lego Mindstorms. We used brickOS 0.2.6 on the RCX. Tables 2 and 3 show results.

For each of the examples shown, we built the byte code to be executed on the virtual machine using the sequentializing algorithm. We also built the C code for the Esterel program using an alternate path: the Columbia Esterel compiler, which generates "linked-list" code [4]. Finally, we built the object code for this C code for both x86 and H8 using gcc running with optimization (`-O2`).

For each example, the size of the byte code is at least 47% smaller than the compiled C code for both the x86 and H8 processors. The code for the virtual machine occupies 814 bytes on the H8, independent of the program it must run; this does not include space for the registers.

Table 3 shows execution times. We compared the speed of the code running on the VM on an x86 with that of the compiled, optimized C code running on the same machine (a Pentium 4 running at 2.5 GHz). The listed number are per-tick execution times, collected by running 1000000 cycles on random input data. The dacexample times are an outlier; it is unrealistically small.

7 Conclusions

We have presented a virtual-machine-based approach for implementing Esterel programs in memory-constrained environments. We presented a virtual machine designed with Esterel in mind (in particular, it supports instruction-level concurrency) and a novel compilation algorithm for it that statically schedules the concurrency to eliminate most dynamic run-time behavior.

Our virtual machine is deliberately very simple and closely paired with the Esterel language. It has signal status registers, completion code registers, per-thread program counters, and inter-instant state-holding registers. Most operations on these are classical, but two instructions explicitly implement concurrency by passing control to another thread.

Our compilation scheme statically schedules the concurrent behavior of the program and generates straight-line code for each thread that includes explicit

Table 2
Code sizes for various examples.

Example	**BAL**	**x86**		**H8**	
dacexample	369	917	60%	842	57%
abcd	870	2988	71%	2648	68%
greycounter	1289	3571	64%	2836	55%
tcint	5667	11486	51%	10074	51%
atds-100	10481	38165	73%	26334	60%

BAL: the size of our bytecode (in bytes)
x86: the size of optimized C code for an x86
H8: the size of optimized C code for an Hitachi H8
Percentages represent the size savings of using bytecode.

Table 3
Execution speeds for compiled versus virtual machine code.

Example	**x86**	**BAL**	
dacexample	0.06μs	1.1μs	18×
tcint	0.28μs	1.1μs	4×
atds-100	0.20μs	1.4μs	7×

instructions for context-switching between threads. As a result, the order in which threads are executed is known at compile time and therefore does not introduce overhead, but the details about what instructions are executed is determined at run-time.

Experimentally, we find that the bytecode for our virtual machine is roughly half the size of optimized native assembly code generated from C, and runs between 4 and 7 times slower than optimized C code. We validated this on both an x86-based desktop machine and a small microcontroller—an Hitachi H8 in the Lego Mindstorms RCX controller.

Our virtual machine currently only supports a pure subset of Esterel, i.e., it does not support arithmetic and calls to external functions. We plan to add arithmetic by adding stack-based arithmetic instructions to the VM. Fortunately, it is never necessary to context switch during the evaluation of an arithmetic expression, so it will only be necessary to maintain a single stack shared by all threads. Adding support for externally-called functions is another possibility, although it raises some tricky dynamic library issues. Work on these extensions is ongoing.

References

[1] Berry, G., *Preemption in concurrent systems*, in: *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science **761** (1993), pp. 72–93.

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

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

[4] Edwards, S. A., V. Kapadia and M. Halas, *Compiling Esterel into static discrete-event code*, in: *Proceedings of Synchronous Languages, Applications, and Programming (SLAP)*, Electronic Notes in Theoretical Computer Science (2004).

[5] Potop-Butucaru, D., "Optimizing for Faster Simulation of Esterel Programs," Ph.D. thesis, INRIA, Sophia-Antipolis, France (2002).

[6] Potop-Butucaru, D., *Optimizations for faster execution of Esterel programs*, in: *Proceedings of the 1st International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, Mont St. Michel, France, 2003, pp. 227–236.

[7] Roop, P. S., Z. Salcic and M. W. S. Dayaratne, *Towards direct execution of Esterel programs on reactive processors*, in: *Proceedings of the International Conference on Embedded Software (Emsoft)*, Pisa, Italy, 2004.