Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2007, Article ID 52651, 31 pages
doi:10.1155/2007/52651

Research Article

Code Generation in the Columbia Esterel Compiler

Stephen A. Edwards and Jia Zeng

Department of Computer Science, Columbia University, New York, NY 10027, USA

Received 1 June 2006; Revised 21 November 2006; Accepted 18 December 2006

Recommended by Alain Girault

The synchronous language Esterel provides deterministic concurrency by adopting a semantics in which threads march in step with a global clock and communicate in a very disciplined way. Its expressive power comes at a cost, however: it is a difficult language to compile into machine code for standard von Neumann processors. The open-source Columbia Esterel Compiler is a research vehicle for experimenting with new code generation techniques for the language. Providing a front-end and a fairly generic concurrent intermediate representation, a variety of back-ends have been developed. We present three of the most mature ones, which are based on program dependence graphs, dynamic lists, and a virtual machine. After describing the very different algorithms used in each of these techniques, we present experimental results that compares twenty-four benchmarks generated by eight different compilation techniques running on seven different processors.

1. INTRODUCTION

Embedded software is often conveniently described as collections of concurrently running processes and implemented using a real-time operating system (RTOS).While the functionality provided by an RTOS is very flexible, the overhead incurred by such a general-purpose mechanism can be substantial. Furthermore, the interprocess communication mechanisms provided by most RTOSes can easily become unwieldy and easily lead to unpredictable behavior that is difficult to reproduce and hence debug. The behavior and performance of concurrent software implemented this way are difficult to guarantee.

The synchronous languages [1], which include Esterel [2], signal [3], and lustre [4], provide an alternative by providing deterministic, timing-predictable concurrency through the notion of a global clock. Concurrently running threads within a synchronous program execute in lockstep, synchronized to a global, often periodic, clock. Communication between modules is implicitly synchronized to this clock. Provided the processes execute fast enough, processes can precisely control the time (i.e., the clock cycle) at which something happens.

The model of time used within the synchronous languages happens to be identical to that used in synchronous digital logic, making the synchronous languages perfect for modeling digital hardware. Hence, executing synchronous languages efficiently also facilitates the simulation of hardware systems.

Unfortunately, implementing such languages efficiently is not straightforward since the detailed, instruction-level synchronization is difficult to implement efficiently with an RTOS. Instead, successful techniques "compile away" the concurrency through a variety of mechanisms ranging from building automata to statically interleaving code [5].

In this paper, we discuss three code generation techniques for the Esterel language, which we have implemented in the open source Columbia Esterel compiler. Such automatic translation of Esterel into efficient executable code finds at least two common applications in a typical design flow. Although Esterel is well suited to formal verification, simulation is still of great importance 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; generating automatically this from the specification can be a great help in reducing implementation mistakes.

1.1. The CEC code generators

CEC has three software code generators that take very different approaches to generate code. That three such different techniques are possible is a testament to the semantic distance between Esterel and typical processors. Unlike, say, a C compiler, where the choices are usually microscopic, our three techniques generate radically different styles of code.

Esterel's semantics require any implementation to deal with three issues: the concurrent execution of sequential threads of control within a cycle, scheduling constraints among these threads from communication dependencies, and how (control) state is updated between cycles. The techniques presented here solve these problems in very different ways.

Our techniques are Esterel-specific because its semantics are fairly unique. Dataflow languages such as lustre [4], for example, have no notion of the flow of control, preemption, or exceptions, so they have no notion of threads and thus no need to consider interleaving them, the source of most of the complexity in Esterel. Nácul's and Givargis's phantom compiler [6] handles concurrent programs with threads, but they do not use Esterel's synchronous communication semantics, so their challenges are also very different.

The first technique we discuss (Section 4) transforms an Esterel program into a program dependence graph—a graphical representation for concurrent programs developed in the optimizing compiler community [7]. This fractures a concurrent program into atomic operations, such as expression evaluations, then reassembles it based on the barest minimum of control and data dependencies. The approach allows the compiler to perform aggressive instruction reordering that reduces the context-switching overhead—the main source of overhead in executing Esterel programs.

The second technique (Section 5) takes a very different approach to schedule the behavior of concurrent threads. One of the challenges in Esterel is dealing with how a decision at one point in a program's execution can affect the control flow much later in its execution because another thread may have to be executed in the meantime. This is very different from most imperative languages where the effect, say, of an *if* statement always affects the flow of control immediately.

The second technique generates code that manages a collection of linked lists that track which pieces of code are to be executed in the future. While these lists are dynamic, their length is bounded at compile time so no dynamic memory management is necessary.

Unlike the PDG and list-based techniques, reduced code size, not performance, is the goal of the third technique (Section 6), which relies on a virtual machine. By closely matching the semantics of the virtual machine to those of Esterel, the virtual machine code for a particular program is more concise than the equivalent assembly. Of course, speed is the usual penalty for using such a virtual machine-based approach, and ours is no exception: experimentally, the penalty is usually between a factor of five and a factor of ten. Custom hardware for Esterel, which other researchers have proposed [8, 9], might be a solution, but we have not explored it.

Before describing the three techniques, we provide a short introduction to the Esterel language [2] (Section 2), then describe the GRC intermediate representation due to Potop-Butucaru [10] that is the starting point for each of the code generation algorithms (Section 3). After describing our code generation schemes, we conclude with experimental results that compare these techniques (Section 7) and a discussion of related work (Section 8).

2. ESTEREL

Berry's Esterel [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 one, the program computes its output and next state based on its input and the previous state by doing a bounded amount of work; no intracycle loops are allowed.

Esterel programs (e.g., Figure 1(a)) may contain multiple threads of control. Unlike most multithreaded software, however, Esterel's threads execute in lockstep: each sees the same cycle boundaries and communicates with other threads using a disciplined broadcast mechanism instead of shared memory and locks. Specifically, Esterel's threads communicate through signals that behave like wires in digital logic circuits. In each cycle, each signal takes a single Boolean value (*present* or *absent*) that does not persist between cycles. Interthread communication is simple: within a cycle, any thread that reads a signal must wait for any other threads that set its value.

Signals in Esterel may be pure or valued. Both kinds are either present or absent in a cycle, but a valued signal also has a value associated with it that persists between cycles. Valued signals, therefore, are more like shared variables. However, updates to values are synchronized like pure signals so interthread value communication is deterministic.

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.

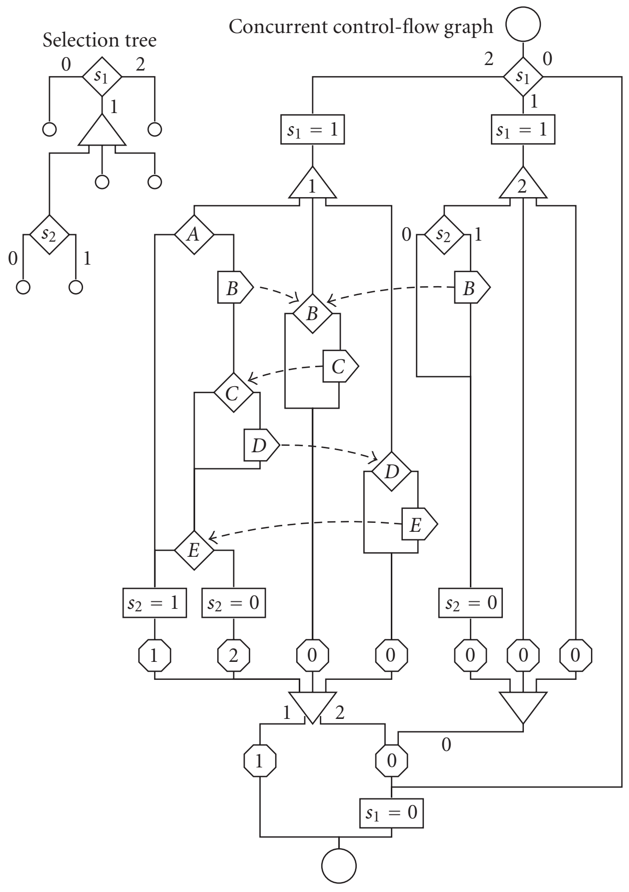
Recently, Berry has made substantial changes (mostly additions) to the Esterel language, which are currently embodied only in the commercial V7 compiler. The Columbia Esterel compiler only supports the older (V5) version of the language, although the compilation techniques presented here would be fairly easy to adapt to the extended language.

3. THE GRC REPRESENTATION

As in any compiler, we chose the intermediate representation in the Columbia Esterel compiler carefully because it affects how we write algorithms. We chose a variant of Potop-Butucaru's [10] graph code (GRC) because it is the result of an evolution that started with the IC code due to Gontier and Berry (see Edwards [11] for a description of IC), and it has proven itself as an elegant way to represent Esterel programs.

```
module grcbal3:
input A;
output B, C, D, E;
trap T in
  present A then
    emit B;
    present C then
      emit D
    end present;
    present E then
      exit T
    end present
  end present;
  pause;
  emit B
||
  present B then
    emit C
  end present
||
  present D then
    emit E
  end
end trap
end module
```

(a)



(b)

FIGURE 1: An example of (a) a simple Esterel module (b) the GRC graph.

Shown in Figure 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. In CEC, the GRC is produced through a syntax-directed translation followed by some optimizations to remove dead and redundant code. The control-flow portion of GRC was inspired by the concurrent control-flow graph described by Edwards [11] and is also semantically close to Boolean logic gates (Potop's version is even closer to logic gates—it includes a "merge" node that models when control joins after an if-else statement).

3.1. The selection tree

The selection tree (upper left corner of Figure 1(b)) represents the state structure of the program and is the simpler half of the GRC representation. The tree consists of three types of nodes: leaves (circles) that represent atomic states, for example, *pause* statements; exclusive nodes (diamonds) that represent choice, that is, if an exclusive node is active, exactly one of its subtrees is active; and fork nodes (triangles) that represent concurrency, that is, if a fork node is active, most or all of its subtrees are active.

Although the selection tree is used by CEC for optimization, for the purposes of code generation, it is just a way to enumerate the variables needed to hold the control state of an Esterel program between cycles. Specifically, each exclusive node becomes an integer-valued variable that stores which of its children may be active in the next cycle. In Figure 1(b), these variables are labeled s_1 and s_2. We encode these variables in the obvious way: 0 represents the first child, 1 represents the second, and so forth.

3.2. The control-flow graph

The control-flow graph (right side of Figure 1(b)) is a much richer object and the main focus of the code generation procedure. It is a directed, acyclic graph consisting of actions (rectangles and pointed rectangles, indicating signal emission), decisions (diamonds), forks (triangles), joins (inverted triangles), and terminates (octagons).

The control-flow graph is executed once from entry to exit in each cycle. The nodes in the graph test and set the state, represented by which outgoing arc of each exclusive node is active, test and set signal presence information, and perform operations such as arithmetic.

Fork, join, and terminate work together to provide Esterel's concurrency and exceptions, which are closely intertwined since to maintain determinism, concurrently thrown exceptions are resolved by the outermost one always taking priority.

When control reaches a fork node, control is passed to all of the node's successors. Such separate threads of control then wait at the corresponding join node until all of their sibling threads have arrived. Meanwhile, the GRC construction guarantees that all the predecessors of a join are terminate nodes that indicate what exception, if any, has been thrown. When control reaches a join, it follows the successor labeled with the highest numbered exception that was thrown, which corresponds to the outermost one.

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 (although data may), and control always reaches the join of an inner fork before reaching a join of an outer fork.

Together, join nodes—the inverted triangles in Figure 1(b)— and their predecessors, terminate nodes[1]—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 the 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 [12] first described this clever encoding.

The control-flow graph also includes data dependencies among nodes that set and test the presence of a particular signal. Drawn with dashed lines in Figure 1(b), there are dependency arcs from the emissions of B to the test of B, and between emissions and tests of C, D, and E.

Consider the small, rather contrived Esterel module (program) in Figure 1(a). It consists of three parallel threads enclosed in a *trap* exception handling block. Parallel operators (||) separate the three threads.

The first thread observes the A signal from the environment. If it is present, it emits B in response, then tests C and emits D in response, then tests E and throws the T trap (exception) if E was present. Throwing the trap causes the thread to terminate in this cycle, passing control beyond the *emit B* statement at the end of this thread. Otherwise, if A was absent, control passes to the *pause* statement, which causes the thread to wait for a cycle before emitting B and terminating.

Meanwhile, the second thread looks for B and emits C in response, and the third thread looks for D and emits E.

Together, therefore, if A is present, the first thread tells the second (through B), which communicates back to the first thread (through C), which tells the third thread (through D), which communicates back to the first through E. Esterel's semantics say that all this communication takes place in this precise order within a single cycle.

This example illustrates two challenging aspects of compiling Esterel. The main challenge is that data dependencies between *emit* and *present* statements (and all others that set and test signal presence) may require precise context switching among threads within a cycle. The other challenge is dealing with exceptions in a concurrent setting.

4. CODE GENERATION FORM PROGRAM DEPENDENCE GRAPHS

Broadly, all three of our code generation techniques divide an Esterel program into little sequential segments that can be executed atomically and then add code that passes control among them. Code for the blocks themselves differs little across the three techniques; the interblock code is where the important differences are found.

Beyond correctness, the main trick is to reduce the interblock (scheduling) code since it does not perform any useful calculation. The first code generator takes a somewhat counter-intuitive approach by first exposing more concurrency in the source program. This might seem to make for higher scheduling overhead since it fractures the code into smaller pieces, but in fact this analysis exposes more scheduling choices that enable a scheduler to form larger and hence fewer atomic blocks that are less expensive to schedule.

This first technique is a substantial departure from those developed for generating code from GRC developed by Potop-Butucaru [10]. In particular, in our technique, most control dependencies in GRC become control dependencies in C code, whereas other techniques based on netlist-style code generation transform control dependencies into data dependencies.

Practically, our first code generator starts with a GRC graph (e.g., Figure 1(b)) and converts the control-flow portion of it into the well-known program dependence graph (PDG) representation [7] (Figure 2(a)) using a slight modification of the algorithm due to Cytron et al. [13] to handle Esterel's concurrent constructs. Next, the procedure inserts assignments and tests of guard variables to represent context switches (Figure 2(b)), and finally generates very efficient, compact sequential code from the resulting graph (Figure 2(c)).

While techniques for generating sequential code from PDGs have been known for a while, they are not directly applicable to Esterel because they assume that the PDG started as sequential code, which is not the case for Esterel. Thus, our main contribution in the PDG code generator is an additional restructuring phase that turns a PDG generated from

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

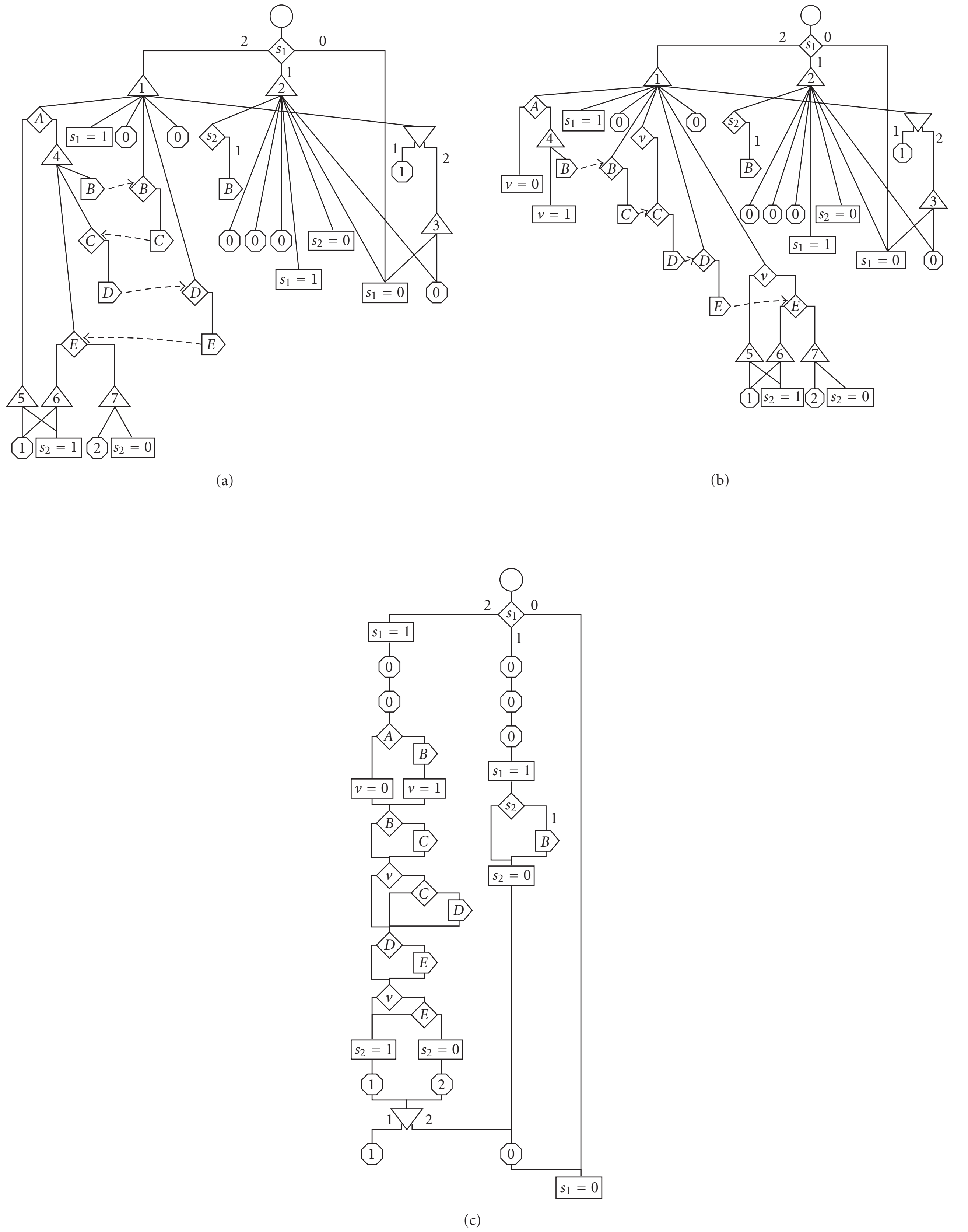


Figure 2: Applying the PDG code generator on the program in Figure 1 produces (a) a PDG, (b) restructures it (b), and (c) makes it sequential.

```
procedure Main
    Priority DFS (root node)                  Assign priorities
    Schedule DFS (root node)                  Schedule with respect
                                              to priorities
    Restructure()                             Insert guards
    Fuse guard variables
    Generate sequential code from the restructured graph
```

Algorithm 1: The main PDG procedure.

Esterel into a form suitable for the existing sequential code generators for PDGs.

The restructuring problem can be solved either by duplicating code, a potentially costly operation that may produce an exponential increase in code size, or by inserting additional guard variables and predicates. We take the second approach, using heuristics to choose where to cut the PDG and introduce predicates, and produce a semantically equivalent PDG that does have a simple sequential representation. Then we use a modified version of Simons' and Ferrante's algorithm [14] to produce a sequential control-flow graph from this restructured PDG and finally generate sequential C code from it.

Our algorithm works in three phases (see Algorithm 1). First, we compute a schedule—a total order of all the nodes in the PDG (Section 4.2). This procedure is exact in the sense that it always produces a correct result, but heuristic in the sense that it may not produce an optimal result. Second, we use this schedule to guide a procedure for restructuring the PDG that slices away parts of the PDG, moves them elsewhere, and inserts assignments and tests of guard variables to preserve the semantics of the PDG (Section 4.3). Finally, we use a slightly enhanced version of the sequentializing algorithm due to Simons and Ferrante to produce a control-flow graph (Section 4.4). Unlike Simons' and Ferrante's algorithm, our sequentializing algorithm always "finishes its job" (the other algorithm may return an error; ours never does) because of the restructuring phase.

4.1. Program dependence graphs

We use a variant of Ferrante et al.'s [7] program dependence graph. The PDG for an Esterel program is a directed graph whose nodes represent statements and whose arcs represent the partial ordering among statements that must be followed to preserve the program's semantics. In some sense, the PDG removes the maximum number of control dependencies among statements without changing the program's meaning. The motivation for the PDG representation is to perform statement reordering: by removing unnecessary dependencies, we give ourselves more freedom to change the order of statements and ultimately avoid much context-switching overhead.

There is an asymmetry between control dependence and data dependence in the PDG because they play different roles in the semantics of a program. A data dependence is "optional" in the sense that a particular execution of the program may not actually communicate through it (i.e., because the source or target nodes happen not to execute); a control dependence, by contrast, implies causality: a control dependence from one node to another means that the execution of the first node can cause the execution of the second.

A PDG is a rooted, directed acyclic graph $G = (S, P, F, r, c, D)$. S, P, and F are disjoint sets of statement, predicate, and fork nodes. Together, these form the set of all vertices in the graph, $V = S \cup P \cup F$. $r \in V$ is the distinguished root node. $c : V \to V^*$ is a function that returns the vector of control successors for each node (i.e., they are ordered). Each vertex may have a different number of successors. $D \subset V \times V$ is a set of data edges. If $c(v_1) = (v_2, v_3, v_4)$, then node v_1 can pass control to v_2, v_3, and v_4. The set of control edges can be defined as $C = \{(m, n) : c(m) = (\dots, n, \dots)\}$, that is, (m, n) is a control edge if n is some element of the vector $c(m)$. If a data edge $(m, n) \in D$, then m can pass data to node n.

The semantics of the graph rely mostly on the vertex types. A statement node $s \in S$ is the simplest: it represents a computation with a side-effect (e.g., assigning a value to a variable) and has no outgoing control arcs. A predicate node $p \in P$ also represents a computation but has outgoing control arcs. When executed, a predicate arc passes control to exactly one of its control successors depending on the outcome of the computation it represents. A fork node $f \in F$ does not represent computation; instead it merely passes control to all of its control successors. We call them fork nodes to emphasize that they represent concurrency; other authors call them "region nodes."

In addition to being rooted and acyclic, the structure of the directed graph (V, C) satisfies two important constraints.

The first rule arises because we want unambiguous semantics for the PDG, that is, we want knowledge of the state of each predicate to provide a crisp definition of what nodes execute and the ordering among them. Specifically, the predicate least common ancestor rule (PLCA) requires that for any node $n \in V$ with two different control paths to it from the root, the least common ancestor (LCA) of any pair of distinct predecessors of n is a predicate node (Figure 3(b)). PLCA ensures that there is at most one active path to any node. If the LCA node was a fork (Figure 3(a)), control could conceivably follow two paths to n, perhaps implying multiple executions of the same node, or at the very least leading to confusion over the relative ordering of the node.

The second rule arises from assuming that the PDG has eliminated all unnecessary control dependencies. Specifically, if n is a descendant of a node m, then there is some path from m to some statement node that does not include n (Figure 3(d)). Otherwise, m and n would have been placed under a common fork (Figure 3(c)). We call this the no post-dominance rule.

4.2. Scheduling

Building a sequential control-flow graph from a program dependence graph requires ordering the concurrently running

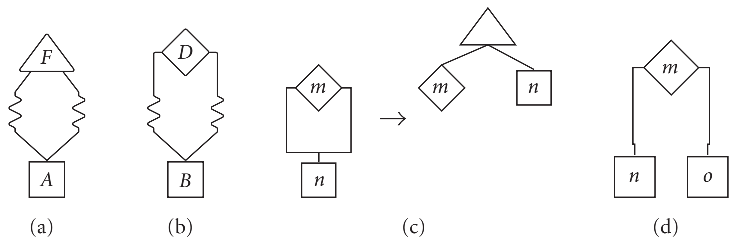


FIGURE 3: Motivation for the PLCA rule: (a) if there are two paths that do not cross from a fork F to an action A, should A be executed twice? (b) Having a decision D as the least common ancestor avoids the problem. Motivation for the no postdominance rule (c) if all paths from m pass through n, m and n are control-equivalent and should be under the same fork. (d) However, if there is some path from m that does not pass through n, n should be a descendant.

nodes in the PDG. In particular, the children of each fork node are semantically concurrent but must be executed in some sequential order. The main challenge is dealing with cases where data dependencies among children of a fork force their execution to be interleaved.

The PDG in Figure 2(a) illustrates the challenge. In this graph, data dependencies require the emissions of B, D, and E to happen before they are tested. This implies that the children under the fork node labeled 1 cannot be executed in any one sequence: the subtree rooted at the test for A must be executed partially, then the subtrees that test B and D may be executed, and finally the remainder of the subtree rooted at the test for A may be executed. This example is fairly straightforward, but such interleaving can become very complicated in large graphs with lots of data dependencies and reconverging control flow.

Duplicating certain nodes in the PDG of Figure 2(a) could produce a semantically equivalent graph with no interleaving but it also could cause an exponential increase in graph size. Instead, we restructure the graph and add predicates that test guard variables (Figure 2(b)). Unlike node duplication, this introduces extra runtime overhead, but it can produce much more compact code.

Our approach inserts guard variable assignments and tests based on cuts implied by a topological ordering of the nodes in a PDG. A cut represents a switch from an incompletely scheduled child of a fork to another child of the same fork. It divides the nodes under a branch of a fork into two or more subgraphs.

To minimize the runtime overhead introduced by this technique, we try to add few guard variables by making as few cuts as possible. Ferrante et al. [15] showed the minimum cut problem to be NP-complete, so we attempt to solve it cheaply with heuristics.

We first compute a schedule for the PDG then follow this schedule to find cuts where interleavings occur. We use a heuristic to choose a good schedule, that is, one implying few cuts, that tries to choose a good order in which to visit each node's successors. We identify the cuts while restructuring the graph.

```
procedure PriorityDFS($n$)
  if $n$ has not been visited, then
    add $n$ to the visited set
    for each control successor $s$ of $n$ do
      PriorityDFS($s$)
      $A[n] = A[n] \cup A[s]$
    for each control successor $s$ of $n$ do
      ComputeSuccPriority($n$, $s$)
    if $n$ has any incoming or outgoing data arcs, then
      add $n$ to $A[n]$

procedure ComputeSuccPriority($n$, $s$)
  $(a, b, c) = (0, 0, 0)$                    initialize
priorities
  if $s$ has neither incoming nor outgoing data arcs,
then
    $a$ = minimum priority number
    return
  for each $j \in A[s]$ do
    $x = 0$, $y = 0$
    for each data predecessor $p$ of $j$ do
      if there is a path from $n \rightsquigarrow p$, then
        increase $a$ by 1
        if there is not a path $s \rightsquigarrow p$, then
          increase $x$ by 1
      increase $c$ by 1
    for each data successor $i$ of $j$ do
      if there is a path $n \rightsquigarrow i$, then
        decrease $a$ by 1
      decrease $c$ by 1
    if $x \neq 0$, then
      for each $k \in A[j]$ do
        for each data successor $m$ of $k$ do
          if $n \rightsquigarrow m$ but not $s \rightsquigarrow m$, then
            increase $y$ by 1
    decrease $b$ by $x \cdot y$
set the priority vector of $s$ under $n$ to $(a, b, c)$
```

ALGORITHM 2: Successor priority assignment.

4.2.1. Ordering node successors

To improve the quality of the generated cuts, we use the heuristic algorithm in Algorithm 2 to influence the scheduling algorithm. It computes an order for successors of each node that the DFS-based scheduling procedure in Algorithm 3 uses to visit the successors.

```
procedure ScheduleDFS($n$)
  if $n$ has not been visited, then
    add $n$ to the visited set
    for each ctrl. succ. $i$ of $n$ in descending priority do
      ScheduleDFS($i$)
    for each data successor $i$ of $n$ do
      ScheduleDFS($i$)
    insert $n$ at the beginning of the schedule
```

Algorithm 3: The scheduling procedure.

```
(1) procedure Restructure
(2)   Clear the currently active branch of each fork
(3)   Clear master-copy ($n$) and latest-copy ($n$) for each
       node $n$
(4)   for each $n$ in scheduled order starting at the root
do
(5)     $D$ = DuplicationSet($n$)
(6)     for each node $d$ in $D$ do
(7)       DuplicateNode($d$)
(8)     for each node $d$ in $D$ do
(9)       ConnectPredecessors($d$)
```

Algorithm 4: The restructure procedure.

We assign each successor a priority vector of three integers (p_1, p_2, p_3) computed using the procedure described below, and later visit the successors in descending priority order while constructing the schedule. We totally order priority vectors $(p_1, p_2, p_3) > (q_1, q_2, q_3)$ if $p_1 > q_1$, or $p_1 = q_1$ and $p_2 > q_2$, or if $p_1 = q_1$, $p_2 = q_2$, and $p_3 > q_3$. For each node n, the A array holds the set of nodes at or below n that have any incoming or outgoing data arcs.

The first priority number of s_i, the ith subgraph under a node n, counts the number of incoming data dependencies. Specifically, it is the number of incoming data arcs from any other subgraphs also under node n to s_i minus the number of outgoing data arcs to other subgraphs under n.

The second priority number counts the number of elements that "pass through" the subgraph s_i. Specifically, it decreases by one for each incoming data arcs from a subgraph s_j to a node in s_i with a node m that is a descendant of s_i that has an outgoing data arc to another subgraph s_k ($j \neq i$ and $k \neq i$, but k may equal j).

The third priority counts incoming and outgoing data arcs connected to any nodes in sibling subgraphs. It is the total number of incoming data arcs minus the number of outgoing data arcs.

Finally, a node without any data arc entering or leaving its descendants is assigned a minimum first priority number.

4.2.2. Constructing the schedule

The scheduling algorithm (Algorithm 3) uses a depth-first search to topologically sort the nodes in the PDG. The control successors of each node are visited in order from highest to lowest priority (assigned by Algorithm 2). Ties are broken arbitrarily, and data successors are visited in an arbitrary order.

4.3. Restructuring the PDG

The scheduling algorithm presented in the previous section totally orders all the nodes in the PDG. Data dependencies often force the execution of subgraphs under fork nodes to be interleaved (control dependencies cannot directly induce interleaving because of the PLCA rule). The algorithm described in this section restructures the PDG by inserting guard variables (specifically, assignments to and tests of guard variables) according to the schedule to produce a PDG where the subgraphs under fork nodes are never interleaved.

The restructuring algorithm does two things: it identifies when the schedule which implies a subgraph must be cut away from an existing subgraph and reattaches the cut subgraphs to nodes that test guard variables to ensure that the behavior of the PDG is preserved.

4.3.1. The restructure procedure

The restructure procedure (Algorithm 4) steps through the nodes in scheduled order, adding a minimal number of nodes to the graph under construction that ensures that each node in the schedule can be executed without interleaving the execution of subgraphs under any fork. It does this in three phases for each node. First, it calls DuplicationSet (Algorithm 5, called from line (5) in Algorithm 4) to establish which nodes must be duplicated in order to reconstruct the control flow to the node n. The boundary between the set D and the existing graph can be thought of as a cut. Second, it calls DuplicateNode (Algorithm 6, called from line (7) of Algorithm 4) on each of these nodes to create new predicate nodes that reconstruct control using a previously cached result of the predicate test. Finally, it calls ConnectPredecessors (Algorithm 7, called from line (9) of Algorithm 4) to connect the predecessors of each of the nodes in the duplication set, which incidentally includes n, the node being synthesized.

The main loop in restructure (lines (4)–(9)) maintains two invariants. First, each fork maintains its currently active branch, that is, the successor in whose subgraph a node was most recently added. This information, tested in line (10) of Algorithm 5 and modified in line (7) of Algorithm 7, is used to determine whether a node can be added to an existing part of the new graph or whether the paths leading to it must be partially reconstructed to avoid introducing interleaving.

The second invariant is that for each node that appears earlier in the schedule, the latest-copy array holds the most recent copy of that node. The node n can use these latest-copy nodes if they do not come from forks whose active branch does not lead to n.

```
(1) function DuplicationSet($n$)
(2)   $D = \{n\}$
(3)   Clear the visited set
(4)   DuplicationVisit($n$)
(5)   return $D$

(6) function DuplicationVisit($n$)
(7)   if $n$ has not been visited, then
(8)     Mark $n$ as visited
(9)     for each predecessor $p$ of $n$ do
(10)      if $p$ is a fork and $p \to n$ is not currently active, then
(11)        Include $n$ in $D$
(12)      if latest-copy($p$) is undefined, then
(13)        Include $n$ in $D$
(14)      if DuplicationVisit($p$), then
(15)        Include $n$ in $D$
(16) return true if $n \in D$
```

Algorithm 5: The DuplicationSet function. A node is in the duplication set if it is along a path from a fork node that leads to n but whose active branch does not.

```
(1) procedure DuplicateNode($n$)
(2)   if $n$ is a fork or a statement, then
(3)     Create a new copy $n'$ of $n$
(4)   else                                        n is a predicate
(5)   if master-copy($n$) is undefined, then      making first copy
(6)       Create a new copy $n'$ of $n$
(7)       master-copy($n$) = $n'$
(8)     else                                      making second or later copy
(9)       Create a new node $n'$ that tests $v_n$
(10)      if master-copy($n$) = latest-copy($n$), then    second copy
(11)        for $i = 0$ to (the number of successors of $n$) $-1$ do
(12)          Create a new statement node $a'$ assigning $v_n = i$
(13)          Attach $a'$ to the $i$th successor of master-copy($n$)
(14)      for each successor $f'$ of master-copy($n$) do
(15)        Find $a'$, the assignment to $v_n$ under $f'$
(16)        Add a data-dependence arc from $a'$ to $n'$
(17)    Attach a new fork node under each successor of $n'$
(18) for each successor $s$ of $n$ do
(19)   if $s$ is not in $D$, then
(20)     Set latest-copy($s$) to undefined
(21) latest-copy($n$) = $n'$
```

Algorithm 6: The DuplicateNode procedure. This makes either an exact copy of a node or tests cached control-flow information to create a node matching n.

4.3.2. *The DuplicationSet function*

The DuplicationSet function (Algorithm 5) determines the subgraph of nodes whose control flow must be reconstructed to execute the node n. It is a depth-first search that starts at the node n and works backward to the root. Since the PDG is rooted, all nodes in the PDG have a path to the root node and therefore DuplicationVisit traverses all nodes that are along any path from the root to n.

A node n becomes part of the duplication set D under three circumstances. The first case, tested in line (10), is when the immediate predecessor p of n is a fork but n is not the currently active branch of the fork. This indicates that executing n would require interleaving because the PLCA rule tells us that there cannot be a path to n from p through the currently active branch under p.

The second case, tested in line (12), occurs when the latest copy of a node is undefined. This occurs when a node is duplicated but its successor is not. The latest-copy array is cleared in lines (18)–(20) of Algorithm 6 when a node is copied but its successors are not.

The final case, line (14), occurs when any of n's predecessors are also in the duplication set.

As a result, every node in the duplication set D is along some path that leads from a fork node f to n that goes through a nonactive branch of f, or leads from a node that has not been copied "recently." These are exactly the nodes that must be duplicated to reconstruct all paths to n.

```
(1) procedure ConnectPredecessors($n$)
(2)   Let $n'$ = latest-copy($n$)
(3)   for each predecessor $p$ of $n$ do
(4)     Let $p'$ = latest-copy($p$)
(5)     if $p$ is a fork, then
(6)       Add a new successor $p' \to n'$
(7)       Mark $p \to n$ as the active branch of $p$∘
(8)     else                                   p is a predicate
(9)       for each arc of the form $p \to n$ do
(10)        Let $f'$ be the corresponding fork under $p'$
(11)        Add a successor $f' \to n'$
```

Algorithm 7: The ConnectPredecessors procedure. This connects every predecessor of n appropriately, possibly using nodes that were just duplicated. As a side effect, it remembers the active branch of each fork.

4.3.3. *The DuplicateNode procedure*

Once the DuplicationSet function has determined which nodes must be duplicated to reconstruct the control paths to node n, the DuplicateNode procedure (Algorithm 6) actually makes the copies. Duplicating statement or fork nodes is trivial (line (3)): the node is copied directly and the latest-copy array is updated (line (21)) to reflect the fact that this new copy is the most recent version of n, something that is later used in ConnectPredecessors. Note that statement nodes are only ever duplicated once, when they appear in the schedule. Fork nodes may be duplicated multiple times.

The main complexity in DuplicateNode comes when n is a predicate (lines (5)–(17)). The first time a predicate is duplicated (i.e., the first time it appears in the schedule), the master-copy array entry for it is undefined (it was cleared at the beginning of Restructure—line (3) of Algorithm 4), the node is copied directly, and this copy is recorded in the master-copy array (lines (6)-(7)).

After the first time a predicate is duplicated, its duplicate is actually a predicate node that tests v_n, a variable that stores the decision made at the predicate n (line (9)). There is just one special case: the second time a predicate is copied (and only the second time—we do not want to add these assignments more than once), assignment nodes are added under the first copy (i.e., the master-copy of n in the new graph) that saves the result of the predicate in the v_n variable. This is done in lines (11)–(13).

An invariant of the DuplicateNode procedure is that every time a predicate node is duplicated, the duplicate version of it has a new fork node placed under each of its successors (line (17)). While these are often redundant and can be removed, they are useful as anchor points for the nodes that cache the results of the predicate and in the uncommon (but not impossible) case that the successor of a predicate is part of the duplicate set but that the predicate is not.

4.3.4. The ConnectPredecessors procedure

Once DuplicateNode runs, all nodes needed to run n are in place but unconnected. The ConnectPredecessors procedure (Algorithm 7) connects these duplicated nodes to the appropriate nodes.

For each node n, ConnectPredecessors adds arcs from its predecessors, that is, the most recent copies of each. The only minor trick occurs when the predecessor is a predicate (lines (9)–(11)). First, DuplicateNode guarantees (line (17) of Algorithm 6) that every successor of a predicate is a fork node, so ConnectPredecessors actually connects the node to this fork, not to the predicate itself. Second, it can occur that a single node can have a particular predicate node appearing two or more times among its predecessors. The *foreach* loop in lines (9)–(11) connects all of these explicitly.

4.3.5. Examples

Running this procedure on Figure 4(a) produces the graph in Figure 4(b). The procedure copies nodes n1–n5. At this point, n0→n3 is the active branch under n0, which is not on the path to n6, so a cut is necessary. DuplicationSet returns {n1, n6}, so n1 will be duplicated. This causes DuplicateNode to create the two assignments to v1 under n1 and the test of v1. ConnectPredecessors then connects the new test of v1 to n0 and n6 to the test of v1. Finally, the algorithm just copies nodes n7–n13 into the new graph.

Figure 5 illustrates the operation of the procedure on a more complicated example. The PDG in (a) has some bizarre control dependencies that force the nodes to be executed in the order shown. The large number of forced interleavings generates a fairly complex final result, shown in Figure 5(e).

The algorithm behaves simply for nodes n0–n8. The state after n8 has been added as shown in Figure 5(b).

Adding n9, however, is challenging. DuplicationSet returns {n9, n6, n5} because n8 is the active node under n4, so DuplicateNode copies n9, makes a second copy of n6 (labeled n6′), creates a new test of v5, and adds the assignments to v5 under n5 (the fork under the “0” branch from n5 has been omitted for clarity). Adding n9’s predecessors is easy: it is just the new copy of n6, but adding n6’s predecessors is more complicated. In the original graph, n6 is connected to n3 and n5, but only n5 was duplicated, so n6′ is connected to v5 and to a fork of the copy of n3.

Figure 5(d) adds n10, which is simple because although n3 was the active branch under n1, n10 only has it as a predecessor.

Finally, Figure 5(e) shows the addition of n11, completing the graph. DuplicationSet returns {n11, n6, n3}, so n3 is duplicated and assignment nodes to v3 are added. Again, n6 is duplicated to become n6″, but this time n3 was duplicated.

4.3.6. Fusing guard variables

An unfortunate choice of schedule clearly illustrates the need for guard variable fusion. Consider the correct but nonoptimal schedule n0, n1, n2, n6, n9, n3, n4, n5, n7, n8, n10, n11, n12, n13 for the PDG in Figure 4(a). Figure 4(c) depicts the effect of so many cuts. The main waste is the cascade of conditionals along the right side of the graph (predicates on v1, v6, and v9). For efficiency, we replace such predicate cascades with single multiway conditionals.

Figure 4(d) illustrates the effect of fusing guard variables. The predicate cascade has been replaced by a single multiway branch that tests the fused guard variable v169 (formed by fusing predicates v1, v6, and v9). Similarly, group assignments to these variables are fused, resulting in three single assignments to v169 instead of three group concurrent assignments to v1, v6, and v9.

4.4. Generating sequential code

After the restructuring procedure described above, the PDG is structured such that the subgraphs under each fork node can be executed in a particular order. This order is nonobvious when there is reconvergence in the graph, and appears to be costly to compute. Fortunately, Simons and Ferrante [14] developed the external edge condition (EEC) as an efficient way to compute this ordering. Basically, the nodes in eec(n) are executed whenever any node in the subgraph under n is executed.

In what follows, $X < Y$ indicates that $G(X)$ must be scheduled before $G(Y)$; $X > Y$ indicates that $G(X)$ must be scheduled after $G(Y)$; $Y \sim X$ indicates that any order is acceptable; and $Y \neq X$ indicates that no order is acceptable. Here, $G(n)$ represents n and all its control descendants.

We reconstruct the graph by ordering fork successors. Given the EEC information, we use the rules in Steensgaard’s decision table [16] to order pairs of fork successors. When the table says any order is acceptable, we order the successors

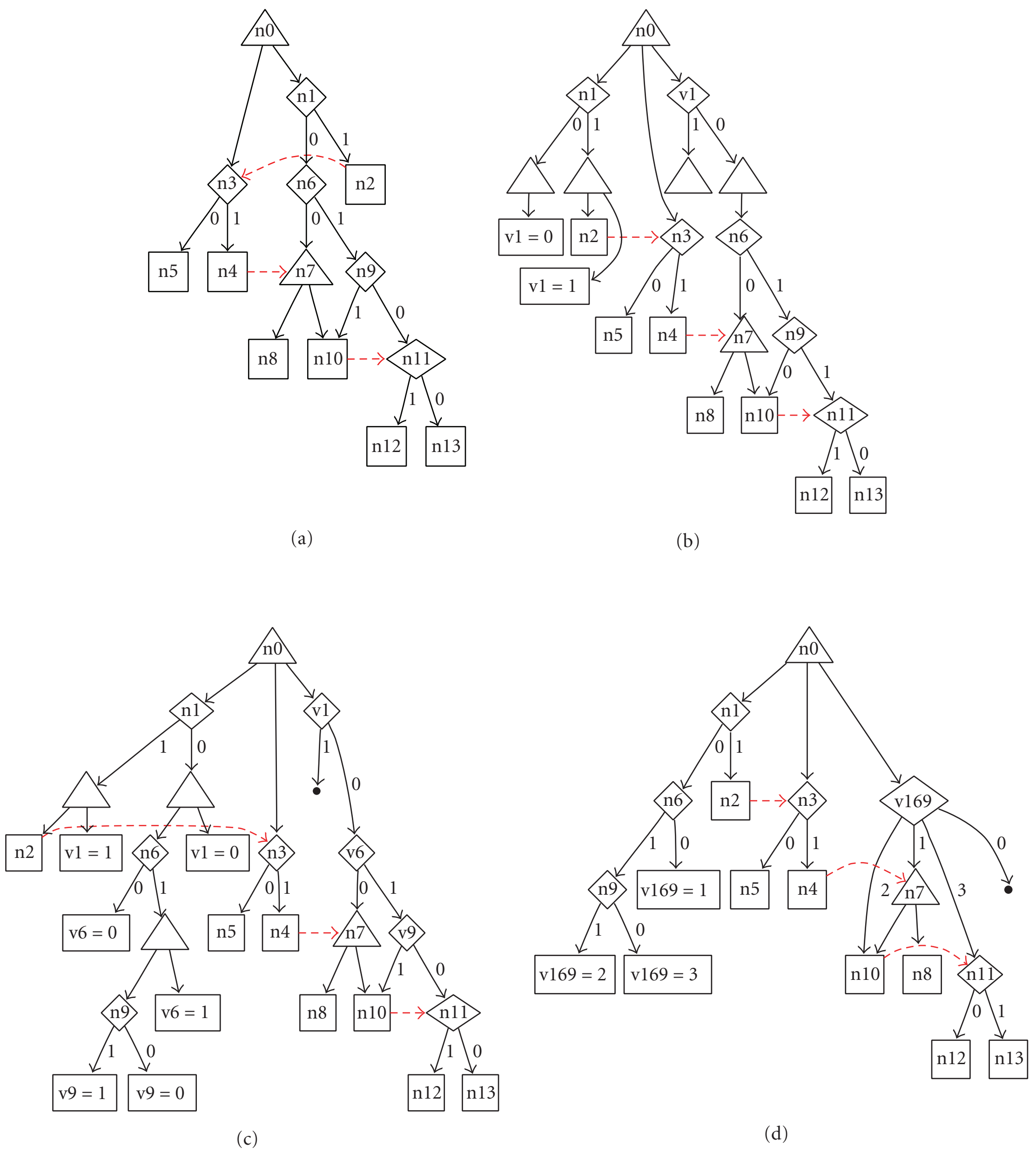


FIGURE 4: (a) A PDG requiring interleaving. (b) After restructuring. Only a single guard variable has been introduced. (c) After restructuring with a different schedule. (d) After guard variable fusion on (c).

based on data dependencies. However, if, say, the EEC table says $G(X)$ must be scheduled before $G(Y)$, yet the data dependencies indicate the opposite order, the data dependencies win and two additional nodes are inserted, one that sets a guard variable and the other that tests it. Algorithm 8 illustrates the procedure.

In Figure 4(b), data dependency forces n11 > n10, but the external edge condition could require n10 > n11 if there was a control edge from a descendant of n11 to a descendant of n10 (i.e., if there were more nodes under n10). In this case, n10 ≠ n11, so our algorithm will cut the graph at n11 and add a guard there.

This produces a sequential control-flow graph for the concurrent program. We generate structured C code from it using the algorithm described in Edwards [11].

5. DYNAMIC LIST CODE GENERATION

The PDG technique we described in the previous section has one main drawback: it must assign a variable and later test it for threads that are not running. This can be inefficient, so our second code generation approach takes a different approach: it tries to do absolutely no work for parts of the program that do not run. The results are mixed: the generated

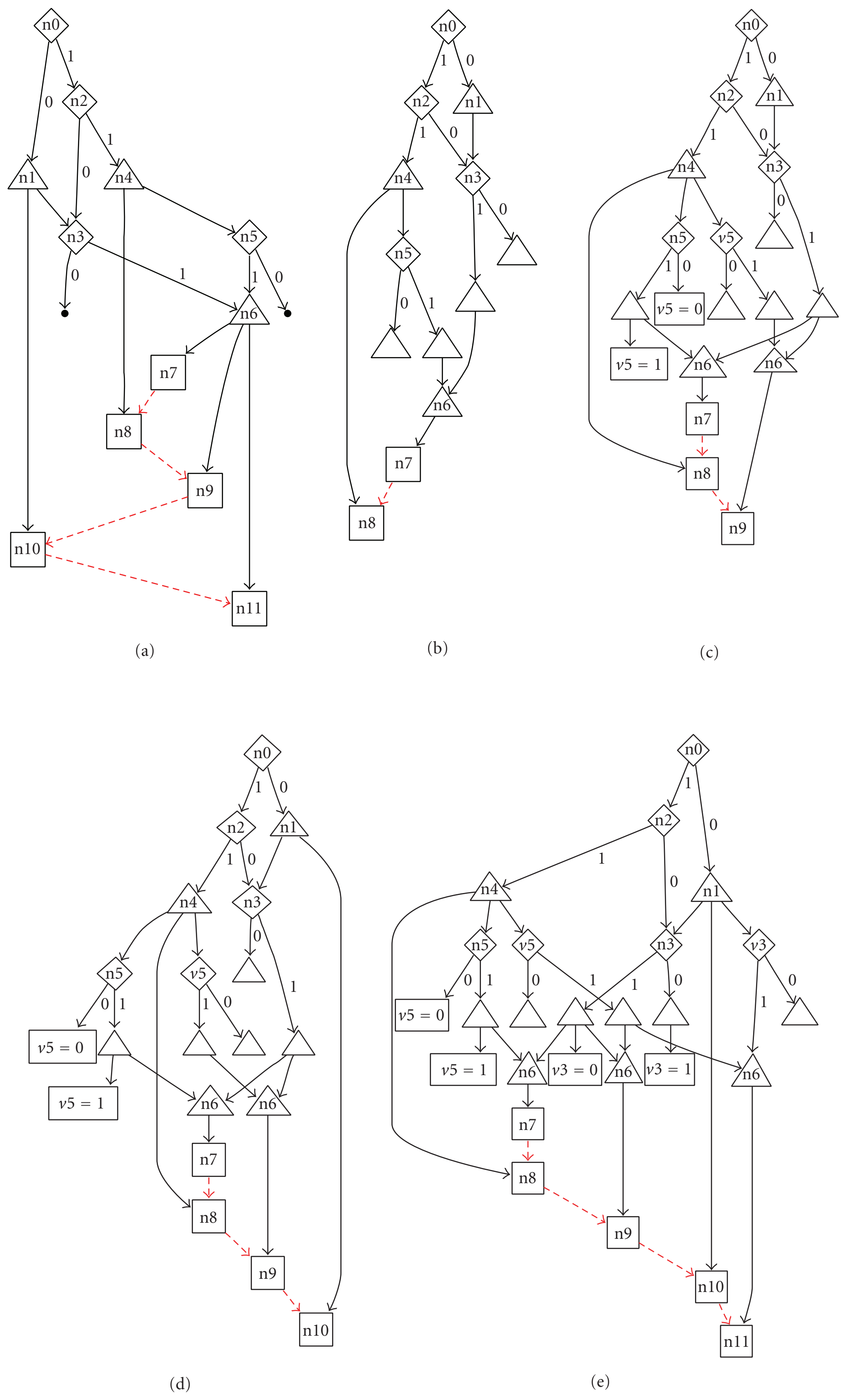


Figure 5: (a) A complex example. (b) After adding nodes n0–n8. (c) After adding n9, (d) n10, and (e) n11.

```
procedure OrderSuccessors($G$)
  for each node $n$ do
    if $n$ is a fork node, then
      original-successors = control successors of $n$
      clear the control successors of $n$
      for each $X$ in original-successors do
        for each control successor $Y$ of $n$ do
          if $X \sim Y$, then
            if $\exists(m, n) \in D$, $m \in G(X)$, $n \in G(Y)$, then
              insert $X$ before $Y$ in $n$'s successors
          else if $Y < X$, then
            if $\exists(m, n) \in D$, $m \in G(Y)$, $n \in G(X)$, then
              Cut $Y$
              insert $X$ before $Y$ in $n$'s successors
          else if $Y > X$, then
            if $\exists(m, n) \in D$, $m \in G(X)$, $n \in G(Y)$, then
              Cut $X$
            else
              insert $X$ before $Y$ in $n$'s successors
          else if $Y \neq X$, then
            if $\exists(m, n) \in D$, $m \in G(X)$, $n \in G(Y)$, then
              Cut $Y$
              insert $X$ before $Y$ in $n$'s successors
            else
              Cut $X$
        if $X$ was not inserted, then
          append $X$ to the end of $n$'s successors
```

Algorithm 8: The successor ordering procedure.

code is faster than the PDG technique only for certain examples, probably because the overhead for the code that runs is higher for this technique.

We based the second code generation technique on that in the SAXO-RT compiler [17, 18]. It produces 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 blocks that will be executed in the current cycle. The scheduling constraints are analyzed completely by the compiler before the program runs 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.

5.1. *Sequential code generation*

This code generation technique relies on the following observations: while arbitrary groups of nodes in the control-flow graph cannot be executed without interruption, many large groups 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 an inexpensive linked list to track which clusters must be executed in each level. By contrast, the scheduling of most discrete event simulators [19] demands a more costly data structure such as a priority queue.

The overhead in our scheme approaches a constant amount per cluster executed. By contrast, the overhead of the SAXO-RT compiler [20] 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.

The 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.

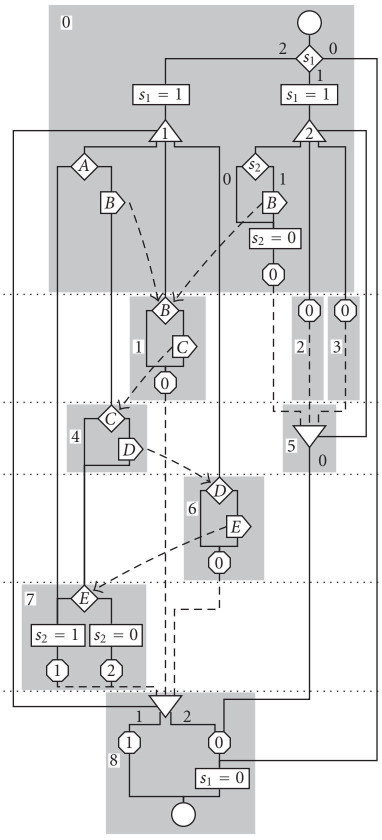
Figure 5 shows the effect of running our clustering algorithm (Algorithm 9, explained below) on the control-flow graph in Figure 1(b). The algorithm identified nine clusters. The algorithm does not always return the optimum (i.e., it may produce more clusters than necessary), but this is not surprising since the optimum scheduling problem is NP-complete (see Edwards [11]).

After nine clusters were identified, our levelizing algorithm, which uses a simple relaxation technique, grouped them into the six levels delineated by dotted lines in Figure 5. It observed that clusters 1, 2, and 3 have no dependencies, and thus can be executed in any order. As a result, it placed them together in the second level. Similarly, clusters 4 and 5 have no dependencies between them. The other clusters are all interdependent and must be executed in the order identified by the levelizing algorithm.

The main trick in our code generation technique is its synthesized scheduler, which maintains a sequence of linked lists. The generated code maintains a linked list of entry points for each level. In Figure 6(b), each `head` variable points to the head of the linked list of each level; each `next` variable points to the successor of each cluster.

The code in Figure 6(b) uses GCC's computed *goto* extension. This makes it possible to take the address of a label, store it in a void pointer (e.g., `void *head1 = &&C1`), and later branch to it (e.g., `goto *head1`) provided this does not cross a function boundary. We also provide a compiler flag that generates more standard C code by changing the generated code to use *switch* statements embedded in loops instead of *gotos*. However, using the computed-*goto* extension noticeably reduces scheduling overhead since a typical switch statement requires either a cascade of conditionals or at least two-bound checks plus a jump table.

Figure 7 illustrates the behavior of these linked lists. Figure 7(a) shows the condition at the beginning of every cycle: every level's list is empty—the *head* pointer for each level points to an end-of-level block that runs the next level. If no



(a)

```
#define sched1  next1 = head1, head1 = &&C1
#define sched2  next2 = head1, head1 = &&C2
#define sched3  next3 = head1, head1 = &&C3
#define sched4  next4 = head2, head2 = &&C4
#define sched5  next5 = head2, head2 = &&C5
#define sched6  next6 = head3, head3 = &&C6
#define sched7a next7 = head4, head4 = &&C7a
#define sched7b next7 = head4, head4 = &&C7b
#define sched8a next8 = head5, head5 = &&C8a
#define sched8b next8 = head5, head5 = &&C8b
#define sched8c next8 = head5, head5 = &&C8c

int example()
{
  /* successor of each block */
  void *next1, *next2, *next3, *next4;
  void *next5, *next6, *next7, *next8;
  /* head of each level's linked list */
  void *head1 = &&L1, *head2 = &&L2;
  void *head3 = &&L3, *head4 = &&L4;
  void *head5 = &&L5;

  switch (s1) {
  case 0: sched8c; break;
  case 1:
    s1 = 1;
    sched5; sched3; sched2;
    if (s2) B = 1;
    s2 = 0;
    break;
  case 2:
    s1 = 1;
    j1 = ~0;
    sched8a; sched6; sched1;
    if (A) {
      B = 1; sched4;
    } else sched7b;
    break;
  }
  goto *head1;
```

```
C1: if (B) C = 1;
    goto *next1;
C2: goto *next2;
C3: goto *next3;
L1: goto *head2;
C4: if (C) D = 1;
    sched7a; goto *next4;
C5: sched8b; goto *next5;
L2: goto *head3;

C6: if (D) E = 1;
    goto *next6;
L3: goto *head4;

C7a: if (E) {
       s2 = 0;
       j1 &= -(1 << 2);
     } else {
     C7b: s2 = 1;
       j1 &= -(1 << 1);
     }
     goto *next7;
L4:  goto *head5;

C8a:  switch (~j1) {
       case 1: break;
       case 3: C8b: C8c:
         s1 = 0; break;
       }
       goto *next8;
L5:

  if (B) {example_0_B(); B = 0;}
  if (C) {example_0_C(); C = 0;}
  if (D) {example_0_D(); D = 0;}
  if (E) {example_0_E(); E = 0;}
  A = 0;
  return s1 != 0;
}
```

(b)

Figure 6: (a) The control-flow graph from Figure 1(b) divided into clusters. Each row of clusters is a level. To guarantee each cluster has at most one active incoming control arc, control arcs reaching join nodes have been replaced with (dashed) data dependencies, and to replace these lost control dependencies, one arc has been added from each fork to its join. (b) The code our compiler generates for this graph (reformatted for clarity).

blocks where scheduled, the program would execute the code for cluster 0 only.

Figure 7(b) shows the pointers after executing *sched3*, *sched1*, and *sched4* (note that this particular combination never occurs in this program). Invoking the *sched3* macro (see Figure 6(b)) inserts cluster 3 into the first level's linked list by setting next3 to the old value of head1—L1—and setting head1 to point to C3. Invoking *sched1* is similar: it sets next1 to the new value of head1—C3—and sets head1 to C1. Finally, invoking *sched4* inserts cluster 4 into the linked list for the second level by setting next4 to the old value of head2—L2—and setting head2 to C4. This series of scheduling steps produces the arrangement of pointers shown in Figure 7(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. The level of each cluster is hardwired since this information is known at compile time.

A powerful invariant arises from the structure of the control-flow graph: 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.

As is often the case, clusters 7 and 8 have multiple entry points. This is easily handled by using a different value for the pointer to the entry point to the cluster but using the same "next" pointer. See the rules for `sched7a` and `sched7b` in Figure 6(b).

We use the dominator-based code structuring algorithm described in Edwards [11] to generate structured code for each cluster. This occasionally inserts a *goto* to avoid duplicating code.

5.2. *The clustering algorithm*

Algorithm 9 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 *fork* to its corresponding *join*. This operation guarantees that no node

```
(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)
```

Algorithm 9: 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.

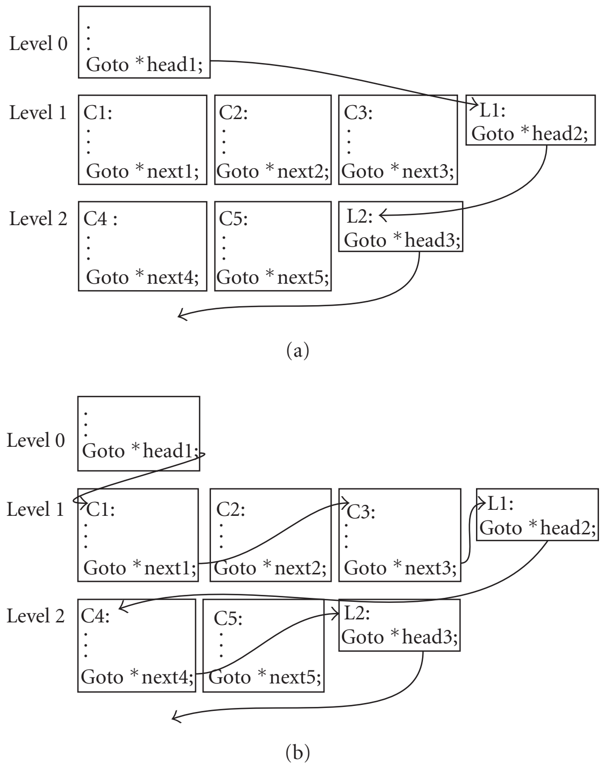


Figure 7: Cluster code and the linked list pointers: (a) at the beginning of a cycle; (b) after executing sched3, sched1, and sched4.

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). Figure 5 reflects this restructuring (dashed control-flow lines denote the arcs that were removed). 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, that is, 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).

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, that is, 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.

6. VIRTUAL MACHINE CODE GENERATION

Because embedded systems often have limited resources such as power, size, computation speed, and memory, we designed our third code generator to produce small executables rather than striving for the highest performance.

To reduce code size, we employ a virtual machine that provides an instruction set closely tailored to Esterel. Lightweight concurrency support is its main novelty (a context switch is coded in two bytes), although it also provides support for Esterel's exceptions and signals.

Along with the virtual machine, we developed a code-generation approach, which, like the algorithm devised by Edwards for the Synopsys Esterel compiler [11], translates a concurrent control-flow graph (i.e., GRC) into a sequential program with explicit context switches. We describe this in Section 6.6.

6.1. *The BAL virtual machine*

We designed our virtual machine (BAL: bytecode assembly language) 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 obvious to implement a virtual machine whose instruction set was customized to Esterel semantics. We devised the instruction set in Table 1 for this purpose.

Instructions in our virtual machine are one or more bytes long. The first byte is the opcode; any second or later bytes are operands, which may be both eight and sixteen bits.

Our virtual machine has separate "registers" for signal presence information, thread completion codes (exception), interthread state, and thread program counters in addition to a set of integervalued general-purpose registers and a small stack for evaluating expressions. To limit their addresses to a single byte, there may be no more than 256 of each type (except for the stack). The actual number of each is a compile-time constant and may be adjusted for a particular implementation environment.

6.2. *General-purpose and signal registers*

The general-purpose registers are the most standard. The value of such a register, an *int* in C, may be pushed onto the stack with *LDR* and set by popping a value from the stack with *STR*. Both have a register number as an operand. The value of registers persist between cycles, so they are used to hold the values of (integer-) valued signals, Esterel variables, and values of implicit counters. Conditional decrement—*CDEC*—is the other instruction that touches these registers. An oddball designed to support Esterel's counters, it replaces the top of the stack with 1 if the given register contains zero and 0 otherwise. Moreover, if the previous top-of-stack was nonzero, the given register is first decremented before being tested.

Signal-presence registers are Boolean-valued, are set and reset by the *EMIT* and *RST* instructions, can be read onto the stack by the *LDS* instruction, and begin each cycle in the absent state.

6.3. *Completion code registers*

Thread completion code registers are used for concurrent exception handling and hold small nonnegative integers (our C implementation uses *unsigned char*s). Like signals, their values are reset (to zero) at the beginning of each cycle. The value of a completion code register is set with the *TRM* instruction, which sets a given register to a given value provided the existing value in the register is less than the new one. Thus, the *TRM* instruction implicitly performs a *max* operation when multiple numbers are assigned to the same completion code register. Completion codes we use are the standard ones for Esterel [12], that is, 0 for termination, 1 for pause, and 2 and higher for traps.

The only instruction that tests a completion code register is *BCC*, which is one of two multiway branch statements in our instruction set. This reads a value *i* from the given completion code register and branches to the *i*th address following the instruction. This behavior mimics the C *switch* statement in that it uses a jump table to perform a multiway branch, but does not perform any range checking on the value. The *BCC* instruction is used for *join* nodes in the GRC, which has exactly these semantics.

Table 1: The virtual machine instruction set.

Instruction	Description	Encoding
	Signal, state, and thread instructions	
EMIT *sig*	Emit signal	00 GG
RST *sig*	Reset signal	01 GG
SET *st const*	Set state	02 SS VV
STRT *th a*	Start thread	03 TT HH LL
	Control-flow instructions	
TICK	End of instant	04
JMP *a*	Jump	05 HH LL
	Branch, switch, and terminate instructions	
BST *st* $a_1 \cdots a_n$	Branch on state variable	06 SS HH_1 LL_1 $\cdots$ HH_n LL_n
BCC *cc* $a_1 \cdots a_n$	Branch on completion code	07 CC HH_1 LL_1 $\cdots$ HH_n LL_n
BNZ *a*	Branch on nonzero	08 HH LL
SWC *th*	Switch to thread	09 TT
TRM *cc code*	Terminate thread	0B CC VV
	Load/store instructions	
LDI *const*	Load immediate	0C HH LL
LDS *sig*	Load signal status	0D GG
LDR *reg*	Load register	0E RR
STR *reg*	Store register	0F RR
	Arithmetic and logical instructions	
ADD	Add	10
SUB	Subtract	11
MUL	Multiply	12
DIV	Divide	13
MOD	Remainder	14
CMP	Compare	15
NEQ	Compare not equal	16
LT	Less than	17
LEQ	Less than or equal	18
GT	Greater than	19
GEQ	Greater than or equal	1A
AND	Logical AND	1B
OR	Logical OR	1C
NEG	Negate (unary)	1D
NOT	Logical NOT (unary)	1E
CDEC *reg*	Conditional decrement	1F RR

RR = register number, SS = state number, GG = signal number,
TT = thread number, CC = code number, VV = 8-bit value,
HH = high-order byte, LL = low-order byte.

6.4. Arithmetic and the stack

Our VM uses a stack for arithmetic operations. It includes the usual stack-based arithmetic operators, which each pops the top two instructions off the stack, performs the operation, and pushes the result back on the stack. The instructions include arithmetic, comparison, and logical operators. Most are binary, but there are also unary negation and logical NOT operators that only modify the top-of-stack.

The stack is only used for storing temporary results of expression evaluation. Esterel's concurrency semantics conveniently never demand a context switch take place during an expression evaluation. This makes it possible to have a single stack shared among all threads with no danger of collision since context switches only ever take place when the stack is empty. Similarly, the stack is always empty at the end of a cycle.

The *LDI*, *LDS*, and *LDR* instructions push an immediate value, a signal status, and a register value onto the stack respectively. The *STR* instruction pops the top of the stack into a register. The *BNZ* instruction tests the top of the stack and branches if it is nonzero. It is used to implement Esterel's *present* and *if* statements.

6.5. Concurrency

Concurrency support is the main novelty in our VM. It is fairly simple: there is a collection of program counters, one for each thread. The *STRT* instruction initializes a given program counter with a specific address, and the *SWC* instruction stores the running program counter before switching to the given thread.

Esterel's semantics avoid the need to store context beyond a program counter for each thread. The signal presence registers are by default visible to all threads (Esterel scoping rules limit which threads may access which signals, but our compiler enforces this, not our VM). By the same token, the general-purpose registers are global. The completion code registers are always accessed by multiple threads and are global. Thus the stack is the only possible candidate for being local, but as mentioned above Esterel's semantics allow the stack to be always empty when a context switch occurs, meaning it does not have to be stored as part of a thread's context.

The generated code for an Esterel program always starts with a sequence of *STRT* statements, one for each thread, that initializes each to a typically trivial code sequence that handles the case when the thread is never started. This is a subtle point: at first glance, it would seem possible to simply mark never-running threads as dead and pass control over them when an *SWC* instruction attempts to send control to them. However, the issue arises of where to send control *after* attempting to send it to a terminated thread. It turns out that our scheduling procedure requires fairly detailed, context-specific rules for this so the easiest way to handle the situation is to simply have special "dead" code for each thread that simply passes control to the next scheduled thread when control reaches it.

The initial *STRT* statements set the PC of each thread to its "dead" version, but a later *STRT* statement can override these values and point the PC for a thread to the start of actual code for it. Such statements are generated for *fork* nodes in the GRC.

6.6. Sequential code generation

Our sequential code generation technique generates 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 designed to match with this procedure. Generated C code requires a fair amount of overhead for each context switch, such as a *switch* statement; our VM encodes a complete context switch in two bytes.

The design of our VM and the matching code generator are the main contributions of this approach. The synthesis algorithm adds context switches based on the schedule of nodes in the GRC representation of the Esterel program described in Section 3. We added a module to CEC that schedules the nodes, assigns a thread number to each node, sequentializes the graph, and finally outputs the BAL representation (Section 6.1). Figure 8 shows the graph and generated code produced by sequentializing the program in Figure 1.

Our algorithm begins by topologically sorting the nodes in the control-flow portion of the GRC for the program. This produces a schedule that respects both control and data dependencies (data dependencies are drawn as dashed lines in Figure 1(b)), and we assume the graph is cycle-free.

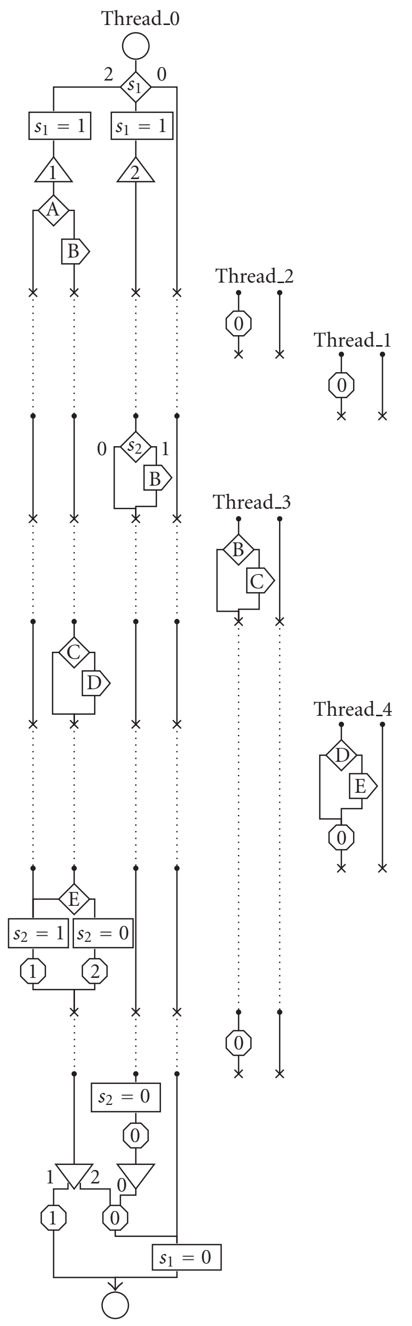
Next, we assign a thread number to each node. This is straightforward: the topmost thread is numbered 0 and the threads under a fork are numbered sequentially starting with the next available thread number. Our one trick is to give the first child thread under each *fork* the same number as its parent. This is safe since a parent never runs until all its children have terminated.

The next step is sequentialization, which we describe in detail in the next section. Our algorithm introduces two new types of control-flow graph nodes: a *switch* node, which becomes an *SWC* statement in the generated BAL and corresponds to a point where control transfers from one thread to another; and an *active* node, which corresponds to a point where control can transfer in from another thread via a context switch.

Finally, the BAL is generated by performing a depth-first search on the graph and generating one or more BAL instructions for each node. Needed jump instructions and labels are added at this point. The generated code could benefit from certain classical optimizations such as straightening [21]; we have not yet implemented these optimizations.

6.7. The sequentializing algorithm

Algorithm 10 shows our sequentializing algorithm. Before these runs, all the nodes are scheduled to respect control and data dependencies and each node is annotated with the thread in which it resides.



(a)

```
thread_0:
      STRT 1 dead_thread_1
      STRT 2 dead_thread_2
      STRT 3 dead_thread_3
      STRT 4 dead_thread_4
      BST  0 L2 L12 L30
L2:   SWC  2
      SWC  3
      SWC  4
      SWC  3
L9:   SET  0 0
done: TICK
L12:  SET  0 1
      STRT 1 thread_1
      STRT 2 thread_2
      SWC  2
      BST  1 L17 L29    thread_1:
L17:  SWC  3                  TRM 0 0
      SWC  4                  SWC 0
      SWC  3
      SET  1 0          dead_thread_1:
      TRM  0 0                SWC 0
      BCC  0 L26 - -
L26:  JMP  L9           thread_2:
L29:  EMIT 1  ; B             TRM 0 0
      JMP  L17                SWC 1
L30:  SET  0 1
      STRT 3 thread_3   dead_thread_2:
      STRT 4 thread_4         SWC 1
      LDS  0  ; A
      BNZ  L46          thread_3:
      SWC  2                  LDS 1 ; B
      SWC  3                  BNZ L79
      SWC  4            L75:  SWC 0
L38:  SET  1 1                TRM 1 0
      TRM  1 1                SWC 0
L41:  SWC  3            L79:  EMIT 2 ; C
      BCC  1 - L45 L26        JMP L75
L45:  JMP  done
L46:  EMIT 1  ; B       dead_thread_3:
      SWC  2                  SWC 0
      SWC  3                  SWC 0
      LDS  2  ; C
      BNZ  L57          thread_4:
L52:  SWC  4                  LDS 3 ; D
      LDS  4  ; E             BNZ L90
      BNZ  L55          L88:  TRM 1 0
      JMP  L38                SWC 0
L55:  SET  1 0          L90:  EMIT 4 ; E
      TRM  1 2                JMP L88
      JMP  L41
L57:  EMIT 3  ; D       dead_thread_4:
      JMP  L52                SWC 0
```

(b)

FIGURE 8: (a) The sequentialized control-flow graph for the example in Figure 1 and (b) the BAL code generated from it. X's represent context switch points; dotted lines indicate the successors of those points.

```
(1) Set $c$, the current thread, to be the first thread
(2) Create a new NOP node $n'$ for $n$, the first node in the schedule
(3) Set $e[n] = n'$                                  existing version of n
(4) Add $n'$ to $r[c]$                               ready-to-run nodes in c
(5) for each node $n$ in scheduled order do
(6)   Set $t$ to the thread of $n$
(7)   if $t \neq c$, then              new thread ≠ current: context switch
(8)     for each ready-to-run node $n' \in r[c]$ do      suspend thread c
(9)       Add an arc from $n'$ to a new switch-to-$t$ node
(10)      Replace $n'$ in $r[c]$ with the new switch
(11)      Set $e[n']$ to the new switch
(12)    for each ready-to-run node $n' \in r[t]$ do     activate thread t
(13)      Add an arc from $n'$ to a new active point node
(14)      Replace $n'$ in $r[t]$ with the new active point
(15)      Set $e[n']$ to the new active point
(16)    Set $c = t$                                  remember current thread
(17)  In the new graph, make a copy $n'$ of $n$               synthesize
(18)  Add an arc from $n$'s stub $e[n]$ to $n'$       attach predecessors
(19)  Remove $n'$ from $r[t]$                              just "ran" $n'$
(20)  for each successor $s$ of $n$ do
(21)    if $s$ is null, then
(22)      Add an arc from $n'$ to a new NOP node
(23)    else
(24)      Set $a$ = the thread of $s$
(25)      if $n$ is fork and $a \neq t$, then        starting a new thread
(26)        Create $l$ and $d$, live and dead active points for $a$
(27)        Start $d$ as $a$ at the beginning of the program
(28)        Add $l$ and $d$ to $r[a]$
(29)        Set $e[s] = l$
(30)      else if $s$ is a join and $a \neq t$, then     ending this thread
(31)        Create a NOP node $s'$ for $s$
(32)        Add an arc from $n'$ to $s'$
(33)        Add $s'$ to $r[t]$                stay in thread t to force C.S.
(34)      else                                         s is a normal node
(35)        if $e[s]$ does not exist, then
(36)          Create a NOP node $s'$ for $s$
(37)          Set $e[s] = s'$
(38)        Add an arc from $n'$ to $e[s]$
(39)        Add $e[s]$ to $r[t]$
```

Algorithm 10: The sequentializing algorithm.

The algorithm steps through the nodes in scheduled order, copying each to a new, sequential graph, and connecting predecessors and successors along the way. Context switch and active point nodes are added to the new graph based on when the schedule says a context switch occurs. Figure 9 illustrates the operation of the algorithm on a small example.

The algorithm maintains two associative arrays. For each node n in the original graph, $e[n]$ is the "stub" in the new graph for it, that is, a node to which the copy of n should be attached.

The other array is $r[t]$, which holds, for each thread t, the set of stub nodes in the new graph that correspond to ready-to-run nodes in the thread t, that is, those nodes that have a stub node that will activate them.

The algorithm consists of one main loop, lines (5)–(39), that steps through the nodes in scheduled order. This loop has a number of invariants: when the loop considers a node n, all its predecessors have been synthesized and there is a stub node for it (i.e., $e[n]$ has been defined).

The body of the main loop performs three main functions. First, if necessary, a context switch is performed from the currently running thread c (set initially in line (1) and updated in line (16)) to t, the thread of the current node n. This context switch is performed in lines (7)–(16). After the possible context switch, the algorithm copies the GRC node n to the new graph and connects it (lines (17)–(19)). Finally, the algorithm connects the successors of the newly copied node n' (lines (20)–(39)). There are special cases for null successors, fork nodes, and when control reaches a join.

6.8. *An example*

Figure 9 depicts the structure of the sequential graph being built for the GRC in Figure 9(a). The product, after removing the redundant NOP nodes that are built for convenience by the algorithm, is shown in Figure 9(b). A number on a node in these two subfigures indicates its position in the schedule.

In Figures 9(c)–9(i), we write a number on a node n to indicate that it is the stub for n, that is, $e[n]$, and enclose in a gray box all the nodes in the two $r[t]$ sets.

The algorithm begins by creating a NOP node that will be the stub for the first node in the schedule (line (2)), and making it the sole ready-to-run node for the outermost thread c (line (4)).

This first time through the loop, no context switch is needed, so the algorithm immediately copies the node (line (17)), makes it the successor of its stub—the NOP node created in line (2) (line (18))—and removes it from $r[c]$.

The successors of the first node are both "normal" and in the same thread, so they are handled by lines (34)–(39). For each successor, this creates a NOP node for it (a stub) if one did not exist already (lines (35)–(37)), adds an arc from the newly synthesized node n' to the stub (line (38)), and indicates that this node is now in the ready-to-run set (line (39)).

Figure 9(c) shows the state of the graph and the r and e arrays at the end of the first iteration of the loop.

The second node to be synthesized is a fork, which is more complicated. Again, a context switch is unnecessary, so the algorithm copies the fork node into the sequential graph, connects it to the stub for 2 created at the end of the last iteration, and removes this stub from the r set (lines (17)–(19)).

This time, however, the node is a fork so some of its successors are handled by lines (25)–(29). A fork has two types of successors: successors that represent new threads, and one that is taken as being part of the same thread as the fork itself. As mentioned earlier, our algorithm always "reuses" the parent thread for one child since we are guaranteed that the parent and child never run simultaneously. This simplifies the generated code and reduces the number of threads that needs to be considered.

In this example, we assume that node 3 is in the same thread as node 2, but that node 4 is in a separate thread. The new thread case is handled by lines (25)–(29), which creates two new nodes for the thread, "l," that becomes the entry point for the code in the thread, and "d," which will behave as the thread when it is not started. The code under d will be nothing but *SWC* statements, so it will always immediately pass control to the next thread in the schedule. This is done by adding d to the ready-to-run nodes for thread a (the one being activated by the fork, done in line (28)). Since it is ready, the context switching code in lines (7)–(16) will attach switch nodes and active points to it, but because it is never considered a stub for any node (only l is: line (29)), it will never have a node from the GRC attached to it.

Code that starts each node is placed at the beginning of the program in line (27). Such code initializes the program counters of all the threads to their dead state; the code generated at a fork node overwrites these values if the fork is executed.

The successor node 3, which is in the same thread as 2, is handled by the normal rules in lines (34)–(39): it creates the stub node for 3 and adds it to the r set for the main thread.

Figure 9(d) shows the state of the graph after the fork has been added. There are now two threads and thus two r sets. One of the ready-to-run stubs in the second thread is where node 4 (the first node in that thread) will be attached; the other is for the "dead" version of the thread.

Synthesizing node 3 is easy because its thread is already running. Figure 9(e) shows the state, which has replaced the stub for 3 with that for 6 in the ready set for the main thread.

Synthesizing node 4 causes a context switch. First, lines (8)–(11) attache a switch node to each ready-to-run stub in the first thread and make these the new ready-to-run stubs for each of the nodes. This will cause control to switch to the other thread. Next, active nodes are added after every ready-to-run stub in the new thread t (lines (12)–(15)); these replace the ready-to-run stubs. Finally, c, the current thread is set to t (line (16)).

Now that the context switch has been performed, the algorithm synthesizes node 4 (attaching it to the newly created active point) and then attaches its successors. Node 4 has a single successor, the join node 6. Since this is in a different thread (the first one), it is handled by lines (30)–(33). This code has one subtle difference from the normal case: $e[s]$ is not set to the newly created NOP for this node. This is because the thread of the join itself (one of the children of the fork) will eventually synthesize the join as part of that thread, as it should. Instead, the NOP acts as a sort of placeholder to ensure that a final context switch that returns the thread of the join will be placed there (adding it to the ready-to-run set in line (33) ensures this).

Figure 9(f) shows the state after the context switch has occurred and node 4 has been synthesized. There are still two ready sets, one for each thread, but the set for the main thread consists of nothing but switch nodes (the crosses are labeled 5 and 6, since they are stubs for those nodes).

Synthesizing node 5 also causes a context switch. Switch nodes are added after the two ready nodes for the right thread by lines (8)–(11), active points for the two ready nodes in the left thread are added by lines (12)–(15), node 5 is synthesized and attached in lines (17)–(19), and the stub for node 7 is attached to it and made ready. Figure 9(g) depicts this state.

Next, node 6, the join, is synthesized. No context switch is necessary. Since an active point for 6 was created by the last context switch, the new node 6 is attached to it. The one different thing here is that there is already a stub for node 7 (created because node 5 also had it as a successor), so the test on line (35) fails and the arc from the synthesized version of 6 is connected to the old node. Figure 9(h) shows this case.

Finally, node 7 is synthesized in a straightforward way to produce the graph in Figure 9(i). This is the final graph modulo the many redundant NOPs that were inserted as

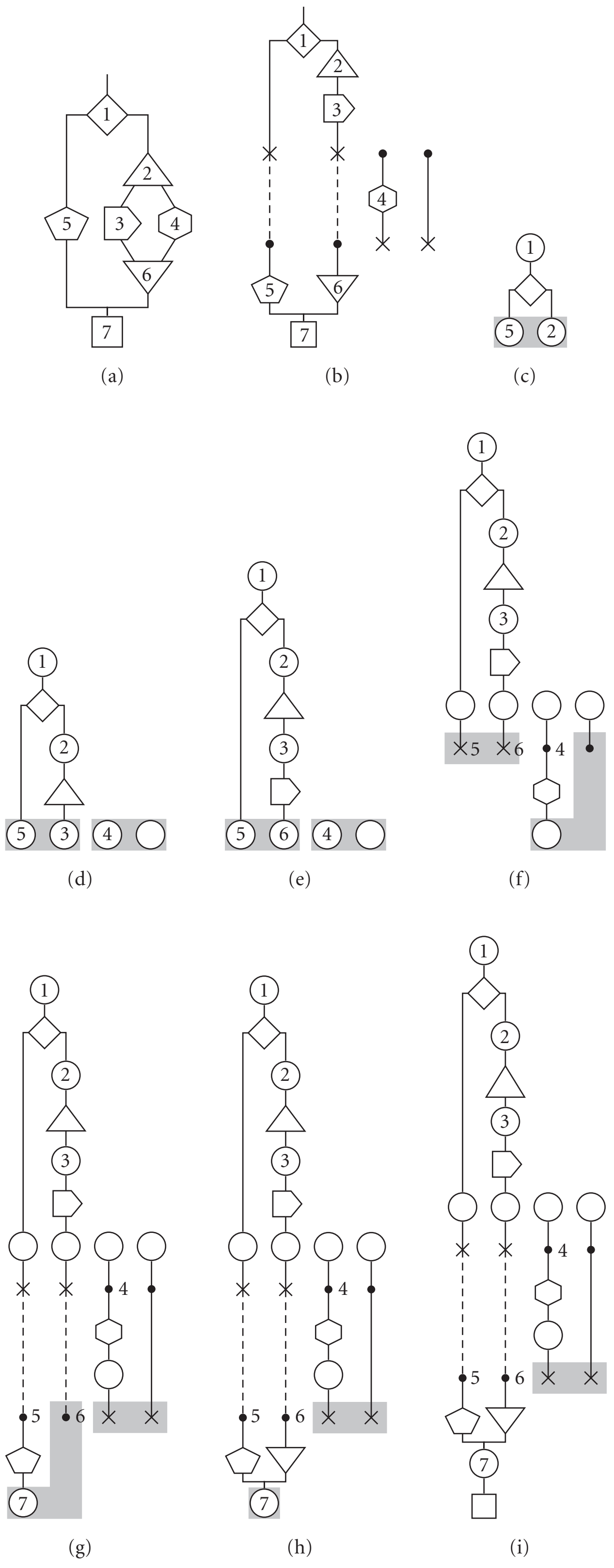


Figure 9: The operation of the sequentializing algorithm. Applied to the graph in (a), the algorithm produces the sequentialized graph in (b) after NOPs (round nodes) are removed. In steps (c)–(i), numbers indicate which nodes are "stubs." Gray boxes indicate the set of ready nodes for each thread. Synthesizing the decision is simple (c), the fork starts another thread (d), the emit is also easy since it is in the same thread as the fork (e), but synthesizing the hexagon requires a context switch (f), as synthesizing the pentagon does (g). Synthesizing the join is again easy because it is in the same thread as the pentagon (h), and finally the square is very easy (i).

placeholders. Figure 9(b) shows the graph after these NOPs have been removed. It is from graphs such as these that we generate bytecode.

7. EXPERIMENTAL RESULTS

We gathered a set of benchmarks (Table 2), generated code for them using CEC, and compared the size and speed of this code with that from the V3, V5, and V7 compilers on a variety of processors (Table 3). Most of the examples are fairly small and taken from the CEC regression test suite (grcbal3 is the example in Figure 1), but a few are "real" programs. The two warehouse examples, due to White and Lüttgen, were written to demonstrate the use of Esterel as a robot controller. The lego examples are from Martin Richard's Esterel-on-Lego website.[2] Abcd is a simple combinational lock due to Berry, tcint is a Turbochannel bus controller, atds-100 is another bus controller, and ww is the wristwatch due to Berry, all parts of the Estbench benchmark suite. Chorus is a task controller [18], mca-200 is a shock absorber controller [24], and counter16 is a 16-bit counter modeled in a discrete-event style due to Henning Zabel and Wolfgang Müller. This last benchmark illustrates the power of Esterel's *run* statement: the Esterel source file is only 800 lines long, including comments, but balloons to nearly 7000 when macros are expanded.

7.1. The benchmarks

The *Lines* column in Table 2 provides a rough measure of the complexity of the benchmark: it lists the number of lines of code generated from the CEC pretty printer after *run* statements have been expanded. Esterel does not have function calls per se, but does have this macro-like module facility. The *Lines* column takes this expansion into account and more closely reflects the complexity of the example.

The *Threads* column provides a measure of concurrency: it lists the number of threads identified by the VM code generation procedure, which always treats one child of each fork in the GRC as being part of the fork's thread. Thus, a number of the benchmarks (notably warehouse-rcx2) contain no concurrency, while others (notably chorus) spawn an enormous number of threads.

The *Signals* column lists the number of signals the program manipulates, including both I/O and internal signals: another rough complexity measure.

The *Clusters* and *Levels* columns provide a rough measure of the likely efficiency of the dynamic list code generator. Each cluster is an atomic block of code (the gray boxes in Figure 5) and each level is a maximal set of clusters that can be executed in any order (separated by dotted lines in Figure 5). The code generator is most efficient when there is only one cluster, which only occurs for purely sequential examples such as warehouse-rcx2. The next best is many clusters in few levels, which occur in highly parallel examples such as chorus. This is preferred since the overhead scales more with the number of levels, not the number of clusters. Least desirable is when there are roughly as many levels as clusters, indicating a lot of concurrency with many dependencies (e.g., multi9). For such programs, the dynamic list approach decays into the SAXO-RT approach.

The *Cuts* column gives a rough idea of the amount of overhead from the PDG technique. It lists the number of additional cut variables introduced to make the PDG sequentializable and corresponds directly to the number of extra assignments and conditionals needed just to manage context switching. For example, only one such extra variable is needed in Figure 2.

The *Switches* column gives a measure of context-switching overhead in the virtual machine technique: it lists the number of context-switching points required in the heuristically found schedule (e.g., equal to the number of rows of dots and X's in the graph in Figure 8(a)).

7.2. Execution time on a Pentium 4

Figure 10 shows the average per cycle execution time for each benchmark on a 2.5 GHz Pentium 4. Note that the time scale in this graph is logarithmic and the abscissa just indicates example number. To obtain these numbers, we fed a pseudorandom input sequence to each example for as many cycles as necessary to get the execution time into the one-second range, then divided the execution time (measured between two calls of the clock() C library call) by the number of iterations. The times, therefore, include input-generation overhead, but this is little more than a few function calls and constant assignments per cycle.

As expected, the virtual machine (the triangles) runs the slowest in all cases, as much as forty times slower than the PDG approach for the smallest examples, although the penalty is lower for the larger examples.

The V5 code is usually the next fastest, but the V3 (automata-based) compiler is sometimes slower. The V7 and two lists techniques compete for second place, but the PDG approach is consistently faster. The two variants of the list compiler, -goto and -switch, respectively, use GCC's computed goto extension and switch statements to transfer control between clusters. Not surprisingly, because they require less overhead (e.g., unlike a *switch*, they do not require code for a range check or a fetch from a lookup table), the computed gotos are consistently faster, although not dramatically so.

The behavior of the code from the V3 compiler shows the most variance, it is sometimes much slower (e.g., lego2, multi1), and sometimes the fastest (atds-100, ww). This is another of its disadvantages: its utility is much less predictable. The V3 compiler timed out (ran for more than an hour) on many of the large examples.

The *Clusters*, *Levels*, *Cuts*, and *Switches* columns in Table 2 give a strong hint about why the PDG technique is generally superior. The number of cuts—a measure of overhead in the code generated from the PDG—is usually much lower than the number of clusters and sometimes even

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

Table 2: The benchmarks.

Benchmark	Lines	Threads	Signals	Clusters	Levels	Cuts	Switches
Dacexample [22]	25	5	5	10	7	4	12
Lego3	25	1	6	1	1	0	0
Lego2	27	1	7	1	1	0	0
Grcbal3 (Figure 1)	30	5	6	9	6	1	9
Multi1	32	4	11	10	5	0	8
Multi2	38	5	10	7	4	0	6
Lego1	42	1	7	1	1	0	0
Multi9	43	6	10	11	6	0	10
Multi4a	46	6	12	12	8	0	10
Multi8	62	10	12	18	6	0	14
Multi3	99	20	12	35	19	5	40
Multi7	107	13	19	18	7	2	18
Multi6	113	27	19	47	15	1	44
Warehouse-rcx2 [23]	136	1	22	1	1	0	0
Graycounter	156	19	25	47	9	10	54
Abcd	176	7	32	21	7	13	30
Warehouse-rcx1 [23]	274	7	41	16	7	12	19
Tcint	357	54	63	103	18	24	100
Mejia2	358	60	39	127	19	43	136
Ww	360	49	48	96	13	30	130
Atds-100	622	73	83	156	17	35	155
Chorus [18]	3893	320	358	662	23	335	861
Mca200 [24]	5354	95	91	148	16	80	184
Counter16	6780	1311	723	2099	266	222	2582

Table 3: The platforms.

Processor	Speed	L1 cache	L2 cache	System	GCC	VM bytes
Pentium 4	2.5 GHz	8 K	512 K	Dell optiplex	4.1.0	868
Pentium 3	1 GHz	16 K	256 K	Dell dimension	4.0.2	891
XScale (ARM)	400 MHz	32 K	none	Sharp zaurus	2.95.2	868
PowerPC 750	350 MHz	64 K	1024 K	Macintosh G3	4.1.0	1152
MIPS R5000	180 MHz	32 K	1024 K	SGI O_2	3.4.6	1208
UltraSPARC-IIi	300 MHz	16 K	512 K	Sun ultra 10	3.4.5	1112
Renesas H8/300	16 MHz	none	none	Lego RCX	4.0.2	722

the number of levels. This suggests that the PDG has successfully restructured the code to avoid most costly context switches, that is, those in which a control state must be stored and retrieved. The *Switches* column brings this into even sharper relief: the number of cuts is often much lower than the number of context switches. The PDG technique was able to completely eliminate many context switches.

The lego1, lego2, lego3, and warehouse-rcx2 examples are unusual because they are purely sequential (i.e., consist of a single thread). As a result, they run much faster than their similarly sized counterparts because they require no runtime overhead for context switching. Furthermore, the advantage the PDG technique generally has over dynamic lists disappears for these examples because each generates roughly the same code in the absence of context switches.

7.3. Code size on a Pentium 4

Figure 11 shows the code sizes on the Pentium 4. Here, the V5 compiler usually generates larger code, but the V3 compiler sometimes generates exorbitantly large executables. The multi7 example is the most extreme—the V3 code is over 400× larger than that from the PDG. The lists-switch compiler is generally next, followed by lists-goto, PDG, and finally the VM for the larger examples. All numbers were collected with the *size* command and only report the size of the executable code, not the space required for data, state, signal presence information, and so forth, which is largely the same for the different techniques anyway.

The number for the VM includes both the bytecode and the size of the virtual machine itself. For the smaller

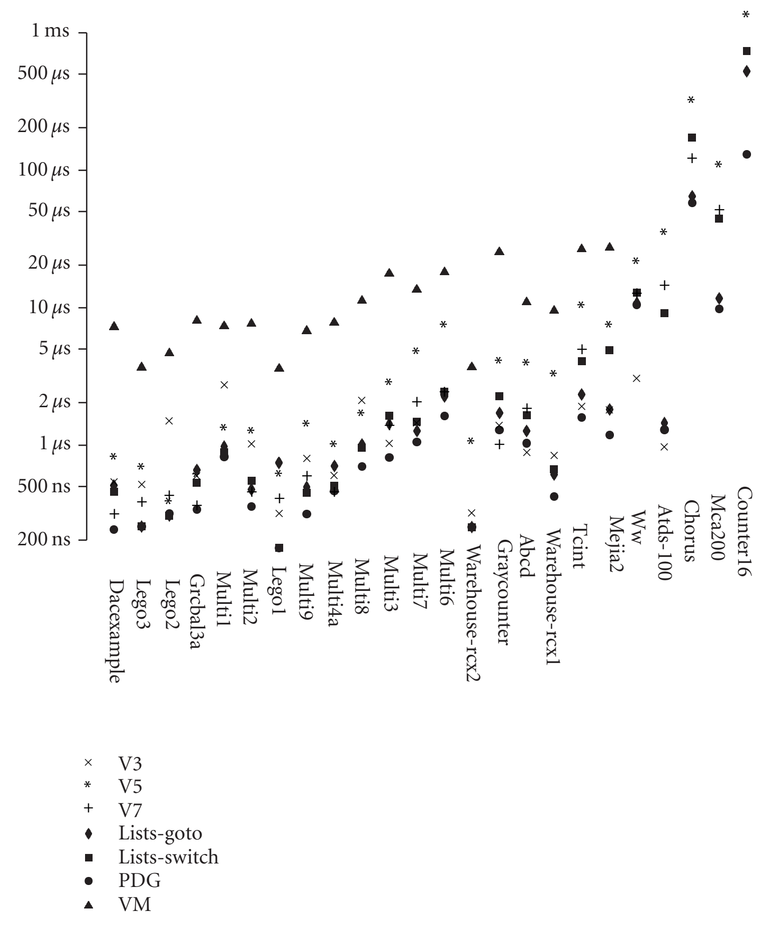


Figure 10: Times on the Pentium 4.

examples, the size of the virtual machine (868 bytes on the P4) dominates and leaves the VM solution both larger and slower. However, above about 2 K, the much smaller size of the bytecode tends to dominate. Figure 12 shows the trend: past a certain point (around 2 K on every platform), the VM tends to produce much smaller code, ranging from a few percent better to as much as 1/4 the size of the native object.

7.4. Execution time on other platforms

The behavior on the Pentium 4 is representative, but we also compared the relative performance of our compiler with V3, V5, and V7 on the platforms in Table 3. Most are desktop workstations of various vintages, but our XScale processor (which runs an ARM-compatible instruction set) runs in a Sharp Zaurus SL-5600, a personal digital assistant running a small Linux environment, and the Renesas H8 is a very small processor running inside the Lego Mindstorms RCX controller. We used brickOS 0.2.6 on the RCX. In each case, we used a version of GCC running at optimization level `-O2` to generate the executable.

Figure 13 shows statistics for the average execution times on the different platforms, normalized to the average cycle time of the V5 compiler in the same example on the same platform. Each cluster of boxes represents a particular compilation technique, and each box represents a platform (the P4 on the left, the H8 on the right). Note that there is only time data for the H8 for the PDG and BAL+VM.

This is a standard box plot on a logarithmic scale: the ends of the lines represent the minimum and maximum times, the dot in the center represents the median, and the box spans the middle two quartiles. For example, the second-to-rightmost box in the PDG group indicates that roughly half of the benchmarks ran in less than one-fifth the time it took the V5 code to run on the SPARC and the fastest ran in nearly 1/100th the time.

Figure 13 better quantifies the speed penalty of the VM. The penalty compared to the V5 compiler was smallest on the SPARC, with over half the examples taking less than twice as long to run under the VM. The VM fared the worst on the P4 and H8, taking over five times as long on half the examples.

The PDG-based technique produces consistently faster code, although the performance of the V3 compiler is much more variable. Using computed gotos appears to produce a modest speed improvement over using switch statements in the dynamic list technique.

The wide variability of results in these graphs is due to the big differences in compilation techniques and in

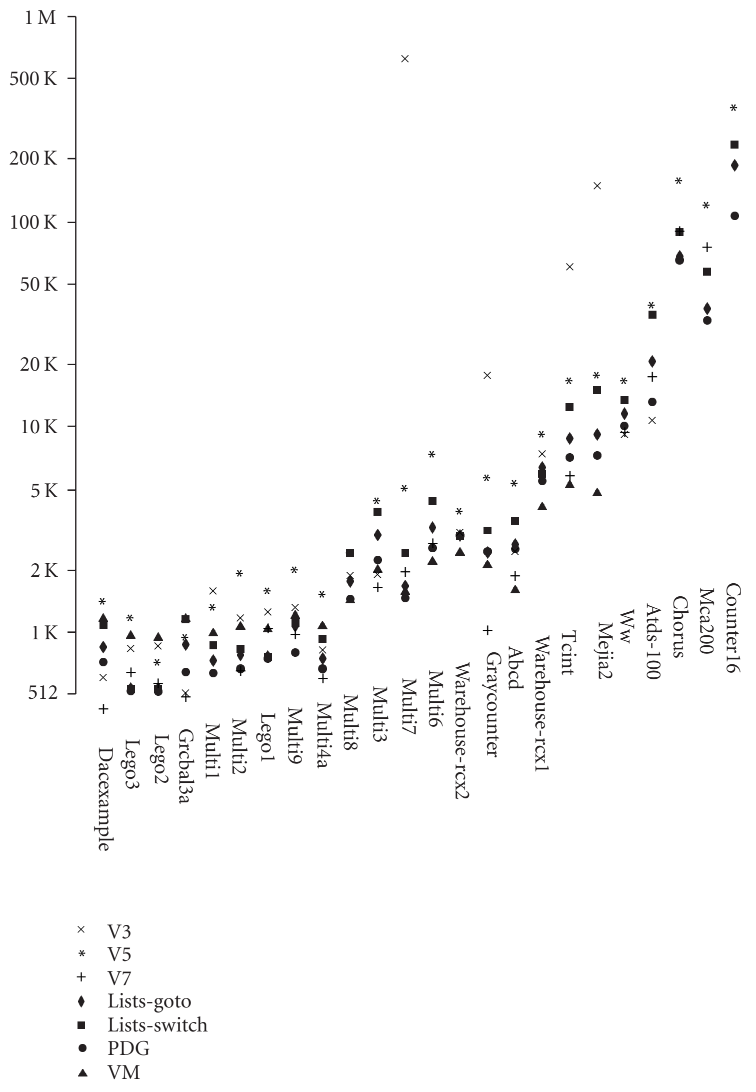


Figure 11: Sizes on the Pentium 4.

the character of the benchmark programs themselves. The biggest difference, say, between the V3 and V5 techniques is that the running time of code from V5 tends to directly follow the size of the source program, whereas the running time of code from the V3 compiler is more closely correlated with the "activity" of the program, that is, how many statements run in each cycle. Different benchmarks invariably exhibit different activities, thus leading to the wide range of relative execution rates.

7.5. Code size on other platforms

Figure 14 is a similar figure showing the distributions of executable sizes across the different platforms. The BAL group is special: these are sizes for the BAL bytecode only; they do not include the size of the VM and as such should not be compared directly. We included them to show how the bytecode is always much smaller than the equivalent executable.

The results are fairly consistent, except for the H8. GCC that appears to be able to generate very compact code for the H8 and as such, the improvements relative to V5 are less dramatic. In particular, the bytecode is usually about 1/3 the size of the equivalent executable on the H8, while the improvement is between 1/5 and 1/10 on the other platforms. We attribute the apparently larger variance in BAL+VM sizes to the constant VM overhead—notice that bytecode by itself varies about as much as the native executables.

The BAL column by itself gives a rough measure of the code density on the different processors: the H8 appears to have the best, followed by the P4, the ARM, the P3, PPC, SPARC, and MIPS.

Figures 15 and 16 plot the average cycle time as a function of executable size. Such correlation would be unexpected for arbitrary C programs (loops can affect the ratio arbitrarily), but is reasonable for code generated from Esterel programs since it is loop-free. The VM exhibits roughly the same trend,

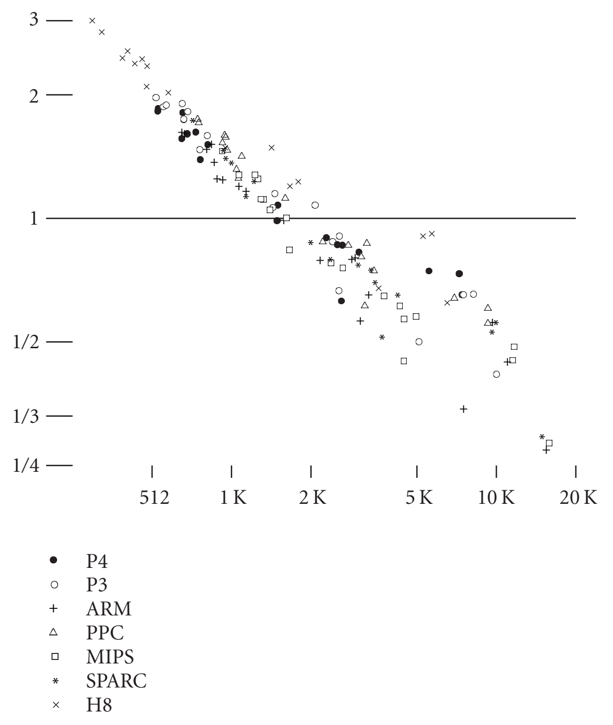


Figure 12: VM executable size versus PDG. The horizontal axis represents the size of the PDG executable; the vertical represents the ratio between the VM and the PDG. For larger examples, the VM executables are substantially smaller.

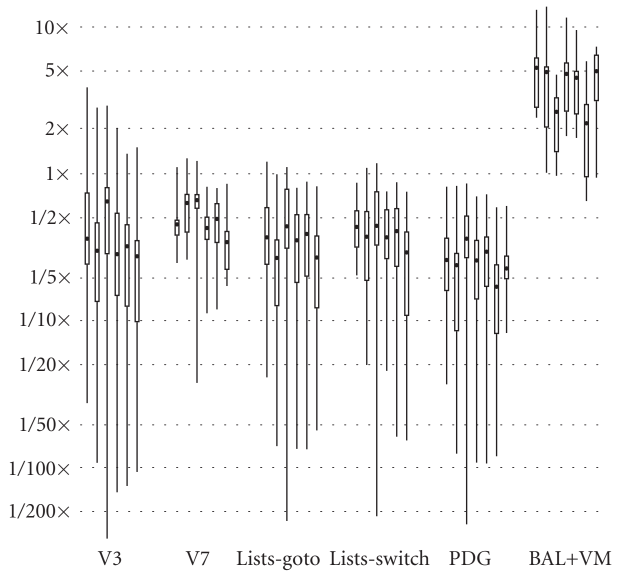


Figure 13: Average execution time statistics for P4, P3, ARM, PPC, MIPS, SPARC, and H8 (PDG and VM only). Baseline: code from the V5 compiler.

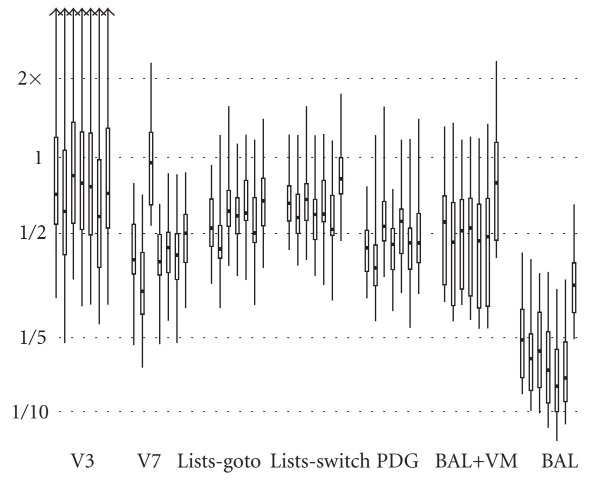


Figure 14: Executable size statistics for P4, P3, ARM, PPC, MIPS, SPARC, and H8. Baseline: code from the V5 compiler.

although is fundamentally much slower. Not surprisingly, the code from the V5 compiler shows the most correlation since the generated code contains very few long-distance jumps. On the other hand, the V3 compiler produces very different results, mostly because it occasionally produces very large executables that nonetheless run quickly.

The scatter plot for th e P4, which has only an 8 K L1 cache (see Table 3), but a generous 512 K L2 cache, shows a fairly consistent trend with no obvious discontinuities.

The plot for the ARM, which has only a single 32 K cache, does show a discontinuity around 32 K. While we have few benchmarks of that size, it does appear that the trend line grows substantially steeper at about that point, pointing out the importance of small code footprints even for systems with larger main memories.

7.6. Comparison with other compilers

We ran a few more experiments with other compilers, but were only able to run a few examples because these other compilers have effectively disappeared. Table 4 shows size and time statistics for these examples. V5+SIS is the V5 compiler followed by running the SIS logic optimizer [25], V7 is the latest commercial compiler from Esterel Technologies, SAXO and SAXO-fast are two outputs from the SAXO compiler due to Weil et al. [18], EC is the Synopsys Esterel compiler [11], and GRC2C is the compiler of Potop-Butucaru [10].

It is difficult to draw any strong conclusions from these data because there are so few examples, although it is clear that GRC2C and V3 generate good code for these examples. From these data, it may appear that the V3 compiler is practical, but this is an erroneous conclusion: it can only run on three of the five examples, and these were developed for the V3 compiler and were probably written very deliberately to make them practical.

Low compilation times were never a particular goal of CEC, and the experimental results of Table 5 bear this out. Here, we compare the time CEC takes to generate C code using the PDG technique with the V5 compiler running on a 2.5 GHz Pentium 4 machine under Linux and GCC 4.1.0.

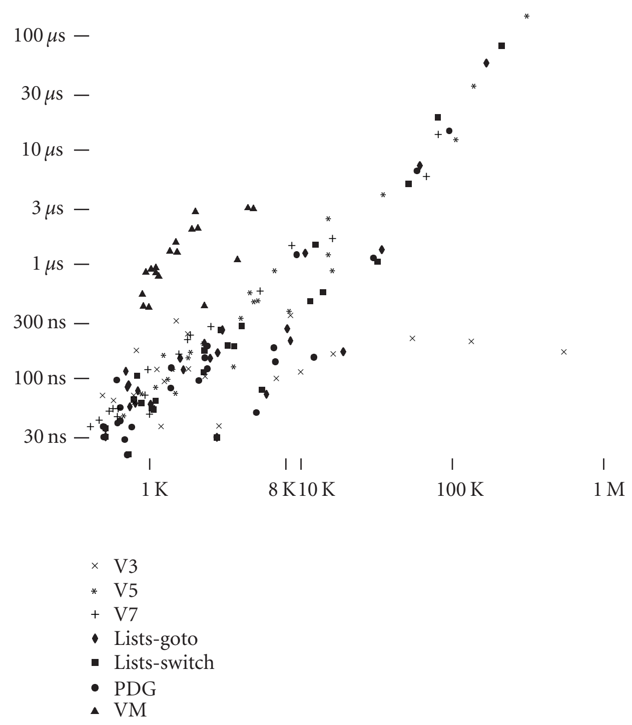


Figure 15: Times versus sizes on the Pentium 4.

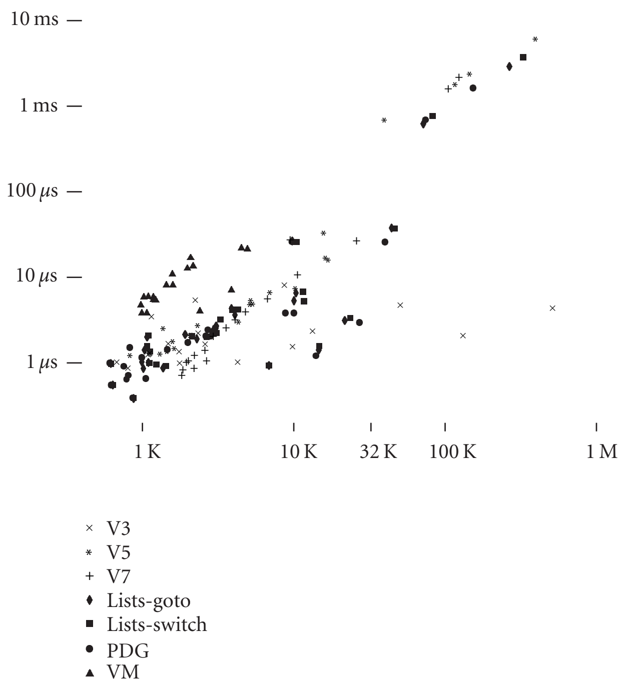


Figure 16: Times versus sizes on the ARM.

Table 4: Sizes and times on the P4 for selected benchmarks.

Compiler	Atds-100	Chorus	Mca200	Tcint	Ww
V3	10942 bytes	—	—	61091	9374
V5	39282	158233	120011	16921	16898
V5+SIS	17688	—	90841	5361	8470
V7	17786	90424	75355	5851	9535
SAXO	21285	71324	38885	9217	13409
SAXO-fast	19117	63876	38245	7949	11841
EC	16312	73788	34845	7472	11590
GRC2C	18403	62835	33604	7661	10832
Lists-goto	21069	68505	38174	8913	11765
Lists-switch	35651	90153	57681	12626	13708
PDG	13427	65493	33396	7236	10287
V3	97 ns	—	—	190	310
V5	3500	32000	11000	1000	2200
V5+SIS	1600	—	11000	430	1300
V7	1400	12000	5200	500	1300
SAXO	130	3600	1400	210	1100
SAXO-fast	130	3200	1300	180	1000
EC	120	7200	950	170	1000
GRC2C	130	2800	620	150	910
Lists-goto	150	6400	1200	230	1100
Lists-switch	910	17000	4500	410	1300
PDG	130	5800	980	160	1100

We report results on the larger benchmarks (>100 lines). For each, we report the average times required to produce C code (i.e., the runtime of the CEC and V5 compilers), for GCC to compile the generated code, and the sum of the two. The results show that CEC is substantially slower on most benchmarks. However, it generates C code that GCC finds substantially more palatable. We suspect this because V5 generates essentially straight line code (i.e., not containing any branches) while the code from the PDG technique is substantially more structured.

Table 5: Comparison of compilation times (seconds on average).

Benchmark	CEC			V5		
	C code	GCC	Total	C code	GCC	Total
Multi7	0.019	0.037	0.056	0.022	0.063	0.085
Multi6	0.047	0.037	0.084	0.026	0.084	0.11
Warehouse-rcx2	0.036	0.063	0.099	0.028	0.053	0.081
Graycounter	0.033	0.040	0.073	0.027	0.072	0.099
Abcd	0.033	0.036	0.069	0.027	0.070	0.097
Warehouse-rcx1	0.11	0.12	0.22	0.048	0.12	0.17
Tcint	0.20	0.11	0.31	0.068	0.21	0.28
Mejia2	0.14	0.080	0.22	0.066	0.22	0.29
Ww	0.12	0.13	0.24	0.069	0.23	0.30
Atds-100	0.70	0.17	0.87	0.13	0.51	0.65
Chorus	69	0.75	70	0.51	2.0	2.5
Mca200	3.2	0.32	3.6	0.55	1.5	2.0
Counter16	30	0.79	31	1.1	3.1	4.1

Part of CEC's longer runtimes is due to simple sloppy programming. The CEC code, for example, makes extensive, perhaps even gratuitous use of the stock standard template library; we suspect that V5 does not. The biggest time consumer in the PDG flow appears to be a simulation algorithm used by the basic optimization procedure (e.g., dead code removal), not the PDG generation procedure. Regardless, the worst running time we observed (for the chorus example) was only slightly more than a minute, which we consider acceptable.

8. RELATED WORK

Many techniques to compile Esterel have been proposed during the language's twenty-year history. Berry and Cosserat [26] were the first. They translated each program into a flat automaton by directly interpreting the operational semantics of the language. This technique was fairly time consuming, but produced efficient code at the expense of size: the generated code may be exponentially larger than the source, making it impractical for all but the smallest programs.

Next, Gonthier, as part of his thesis [27], devised a more efficient way to generate the automaton by simulating a control-flow graph-like representation known as IC. This formed the basis for the successful V3 compiler [2], but did not mitigate the exponential code size problem.

The V5 generation of Esterel compilers [28] 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 cost in speed. The fundamental problem is that the program must execute code for every statement in every cycle, even for statements that are not currently active.

Further progress in code generation came in 1999, when Edwards [29] and a group at France Telecom [17] independently developed two techniques that produced much faster code that was roughly the same size as that from the circuit-based compilers. The techniques we describe here are direct descendants of these two approaches.

Potop-Butucaru [10], as part of his Ph.D. thesis [30], developed a much improved version of the circuit-based code generation technique, incorporating a number of very clever optimizations to improve the quality of the generated code.

Ferrante et al. [15] were the first to propose an algorithm for generating sequential code from an acyclic PDG, but their technique only works when no node duplication (or equivalently, the addition of predicates) is necessary.

Later, Simons and Ferrante [14] presented an efficient algorithm for generating sequential code from an acyclic PDG. Their major contribution is a technique for computing "external edge" information for each node and using this during the synthesis procedure. The input to their algorithm is limited to a graph with only control dependencies; they assume that data dependencies have somehow been incorporated into the control dependencies.

Building on Simons' and Ferrante's work, Steensgaard [16] removed the requirement that the control dependencies in the PDG be acyclic, thereby allowing loops in the generated code (earlier work assumed that loops had somehow been removed), but still assumed that the generated code did not require either node duplication or the insertion of additional predicates. We have not integrated Steensgaard's cyclic extensions because they were unnecessary in our application.

Our PDG technique extends Simons' and Ferrante's in two ways. First, we propose a cutting algorithm that restructures the PDG and inserts additional predicate nodes before it is passed to Simons' and Ferrante's basic algorithm, making it work for all valid acyclic PDGs. Second, we consider data dependencies to generate correct code for all valid PDGs.

Our dynamic list technique resembles the SAXO-RT compiler developed at France Telecom [20]. 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 eliminate conditional tests.

Maurer's event-driven conditional-free paradigm [31] also inspired this code generator, 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.

Potop-Butucaru's compiler [10, 30] generates very good code for most large examples, although 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 CEC, and uses a technique much like the Synopsys compiler to generate code.

Using a virtual machine to implement a high-level language has a long history that includes Pascal [32], Modula-2, Prolog, Smalltalk [33], Java, and C#. Portability and ease of implementation are typical motivations for the use of a virtual machine, but code size is another motivation [34].

Code compression for embedded systems 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 registers 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 not novel—Christophe Mauras and Martin Richard[3] did so with some help from Xavier Fornari. Their approach, however, is more traditional: like us, they use the BrickOS environment to 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.

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

Roop et al. [9] 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.

Li and von Hanxleden [8] also propose an Esterel-specific processor that has certain similarities to our virtual machine. One big difference is that it uses a priority-based approach to decide which thread executes next. We expect in software that managing a priority queue would be much less efficient than the explicit context switching our VM currently uses, but in hardware the difference is less clear. Implementing our Esterel VM in hardware to improve its speed is an obvious avenue for future research.

9. CONCLUSIONS

We have presented three different code generation techniques for Esterel, which we have implemented in the open-source Columbia Esterel compiler. The three take very different approaches to address the fundamental challenge of executing Esterel: simulating concurrency and handling synchronization among concurrently running threads. The first removes all unnecessary control dependencies to produce a control-dependence graph that is then cleverly reassembled to produce a sequential program with very few additional conditionals. It consists of a heuristic scheduler followed by an exact restructuring procedure.

The second technique generates code that maintains a linked list of process fragments that must execute in the future. Such an approach reduces the per-fragment overhead, but apparently not as much as the code motion enabled by the PDG technique. For examples with many concurrent threads, however, it works quite well.

The third technique strives for code size instead of performance by employing a virtual machine with semantics closely matching those of Esterel itself. In particular, our virtual machine provides direct support for multiple threads of control through a low-overhead context-switching instruction. It also has signal status registers, completion code registers, per-thread program counters, and interinstant state- and value-holding registers. Most operations are classical, and we perform arithmetic with a stack, one instruction explicitly implements concurrency by passing control to another thread.

Our compilation scheme for the virtual machine statically schedules the concurrent behavior of the program and generates straight line code for each thread that includes explicit 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 are determined at runtime.

Of the three, the PDG approach seems to be the best all-around (it is currently the default in CEC), but the lists approach can have advantages when the activity of the program is very low and the concurrency is high.

None of our three approaches attempt to handle dynamically causal (but statically cyclic) programs. This is pragmatic choice that reflects the current thinking in the Esterel community, which has largely decided that dynamically causal programs are of theoretical interest but do not appear to be terribly useful in practice. It might be possible to adapt the dynamic lists approach to handle dynamic programs since it provides the most flexibility in controlling the execution order of various blocks, but we have not done so. An alternative approach would be to first remove the cyclic structures in the program (which can be done by the V5 compiler; Lukoschus proposed an alternative [35]), then pass this to one of the acyclic code generators, but again we have not attempted this. See Berry [36] and Potop-Butucaru [30] for a discussion of dynamically causal programs.

Another limitation of our approaches is that they cannot be applied to compile Esterel programs in pieces. This is a failing of all techniques virtually for Esterel and stems from their semantics. There are serious technical issues with compiling Esterel programs in pieces. We made one failed attempt to begin to address the separate compilation and know of another failed attempt. Furthermore, the performance advantages obtained by our techniques are due mostly to being able to do complete program analysis, something probably no longer possible in a separate compilation setting. Clearly, this is an avenue for future research.

While it has become very difficult to improve on existing Esterel compilation techniques, hybrid approaches may further advantages. The automata style construction of the V3 compiler generates very efficient code when it is able, but capacity limitations make it of limited use by itself. A better approach would be to use the automata approach on tightly communicating parts of a program (i.e., ones that benefit the most from the automaton approach and are small enough so that it is still practical) and use the PDG approach to connect them. Edwards and Tardieu [37] successfully used such an approach to compile an asynchronous language.

The virtual machine approach is interesting, but suffers from the performance problems typical of virtual machines. Mating the technique with custom hardware is an obvious idea. The Esterel processor of Li and von Hanxleden [8] would be an obvious starting point.

Source code for the Columbia Esterel compiler along with all the benchmarks used in this paper are available from http://www.cs.columbia.edu/~sedwards/cec.

ACKNOWLEDGMENTS

Cristian Soviani helped to implement CEC. Michael Halas and Vimal Kapadia were largely responsible for the design of the dynamic list-based code generator. Becky Plummer and Mukul Khajanchi did much of the design and implementation of the Esterel virtual machine; Mukul was responsible for the H8 platform. Julien Boucaron helped to generate examples from the V7 compiler. Dumitru Potop-Butucaru supplied the GRC2C examples as well as many stimulating discussions. Olivier Tardieu and Luca Carloni also provided stimulating suggestions and arguments. But none of this would have happened without Gérard Berry. The authors are supported by an NSF Career Award, gifts from Intel and Altera, an Award from the SRC, and from New York State's NYSTAR Program.

REFERENCES

[1] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, 2003.

[2] G. Berry and G. Gonthier, "The Esterel synchronous programming language: design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992.

[3] A. Benveniste, P. Le Guernic, and C. Jacquemot, "The SIGNAL software environment for real-time system specification, design, and implementation," in *Proceedings of IEEE Control Systems Society Workshop on Computer-Aided Control System Design (CACSD '89)*, pp. 41–49, Tampa, Fla, USA, December 1989.

[4] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "LUSTRE: a declarative language for programming synchronous systems," in *Proceedings of the Symposium on Principles of Programming Languages (POPL '87)*, pp. 178–188, Munich, Germany, January 1987.

[5] S. A. Edwards, "Tutorial: compiling concurrent languages for sequential processors," *ACM Transactions on Design Automation of Electronic Systems*, vol. 8, no. 2, pp. 141–187, 2003.

[6] A. C. Nácul and T. Givargis, "Code partitioning for synthesis of embedded applications with phantom," in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD '04)*, pp. 190–196, San Jose, Calif, USA, November 2004.

[7] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, 1987.

[8] X. Li and R. von Hanxleden, "A concurrent reactive Esterel processor based on multi-threading," in *Proceedings of the 21st Annual ACM Symposium on Applied Computing (SAC '06)*, vol. 1, pp. 912–917, Dijon, France, April 2006.

[9] P. S. Roop, Z. Salcic, and M. W. Sajeewa Dayaratne, "Towards direct execution of Esterel programs on reactive processors," in *Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT '04)*, pp. 240–248, Pisa, Italy, September 2004.

[10] D. Potop-Butucaru, "Optimizations for faster execution of Esterel programs," in *Proceedings of the 1st International Conference on Formal Methods and Models for Codesign (MEMOCODE '03)*, pp. 227–236, Mont St. Michel, France, June 2003.

[11] S. A. Edwards, "An Esterel compiler for large control-dominated systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 2, pp. 169–183, 2002.

[12] G. Berry, "Preemption in concurrent systems," in *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, vol. 761 of *Lecture Notes in Computer Science*, pp. 72–93, Bombay, India, December 1993.

[13] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, 1991.

[14] B. Simons and J. Ferrante, "An effficient algorithm for constructing a control flow graph for parallel code," Tech. Rep. TR-03.465, IBM, Santa Teresa Laboratory, San Jose, Calif, USA, February 1993.

[15] J. Ferrante, M. Mace, and B. Simons, "Generating sequential code from parallel code," in *Proceedings of the 2nd International Conference on Supercomputing (ICS '88)*, pp. 582–592, Saint Malo, France, July 1988.

[16] B. Steensgaard, "Sequentializing program dependence graphs for irreducible programs," Tech. Rep. MSR-TR-93-14, Microsoft, Redmond, Wash, USA, October 1993.

[17] 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, March 1999.

[18] D. Weil, V. Bertin, E. Closse, M. Poize, P. Venier, and J. Pulou, "Efficient compilation of Esterel for realtime embedded systems," in *Proceedings of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '00)*, pp. 2–8, San Jose, Calif, USA, November 2000.

[19] J. Banks, J. S. Carson, B. L. Nelson, and D. M. Nicol, *Discrete-Event System Simulation*, Prentice-Hall, Upper Saddle River, NJ, USA, 3rd edition, 2000.

[20] E. Closse, M. Poize, J. Pulou, P. Venier, and D. Weil, "SAXO-RT: interpreting Esterel semantic on a sequential execution structure," *Electronic Notes in Theoretical Computer Science*, vol. 65, no. 5, pp. 864–878, 2002, in *Proceedings of Synchronous Languages, Applications, and Programming (SLAP '02)*.

[21] S. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, San Francisco, Calif, USA, 1997.

[22] S. A. Edwards, "Compiling Esterel into sequential code," in *Proceedings of the 37th Design Automation Conference (DAC '00)*, pp. 322–327, Los Angeles, Calif, USA, June 2000, Association for Computing Machinery.

[23] D. White and G. Lüttgen, "Accessing databases within Esterel," in *Synchronous Programming (SYNCHRON '04)*, S. A. Edwards, N. Halbwachs, R. V. Hanxleden, and T. Stauner, Eds., Dagstuhl Seminar Proceedings, no. 04491, Dagstuhl, Germany, 2005.

[24] M. Chiodo, D. Engels, P. Giusto, et al., "A case study in computer-aided co-design of embedded controllers," *Design Automation for Embedded Systems*, vol. 1, no. 1-2, pp. 51–67, 1996.

[25] E. M. Sentovich, K. J. Singh, L. Lavagno, et al., "SIS: a system for sequential circuit synthesis," Tech. Rep. UCB/ERL M92/41, University of California, Berkeley, Calif, USA, May 1992.

[26] G. Berry and L. Cosserat, "The Esterel synchronous programming language and its mathematical semantics," in *Seminar on Concurrency*, S. D. Brooks, A. W. Roscoe, and G. Winskel, Eds., vol. 197 of *Lecture Notes in Computer Science*, pp. 389–448, Heidelberg, Germany, 1984.

[27] G. Gonthier, "Sémantiques et modèles d'exécution des langages réactifs synchrones; application à Esterel. [Semantics and models of execution of the synchronous reactive languages: application to Esterel]," Thèse d'informatique, Université d'Orsay, Orsay Cedex, France, 1988.

[28] G. Berry, "Esterel on hardware," *Philosophical Transactions of the Royal Society of London. Series A*, vol. 339, pp. 87–103, 1992.

[29] S. A. Edwards, “Compiling Esterel into sequential code,” in *Proceedings of the 7th International Conference on Hardware/Software Codesign (CODES ’99)*, pp. 147–151, Rome, Italy, May 1999, Association for Computing Machinery.

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

[31] P. M. Maurer, “Event driven simulation without loops or conditionals,” in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD ’00)*, pp. 23–26, San Jose, Calif, USA, November 2000.

[32] D. W. Barron, Ed., *Pascal: The Language and Its Implementation*, John Wiley & Sons, New York, NY, USA, 1981.

[33] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, Mass, USA, 1983.

[34] J. E. Smith and R. Nair, *Virtual Machines*, Morgan Kaufmann, San Francisco, Calif, USA, 2005.

[35] J. Lukoschus, *Removing cycles in Esterel programs*, Ph.D. thesis, Department of Computer Science, Christian-Albrechts-Universität Kiel, Kiel, Germany, 2006.

[36] G. Berry, “The constructive semantics of pure Esterel,” draft book, 1999.

[37] S. A. Edwards and O. Tardieu, “Efficient code generation from SHIM models,” in *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES ’06)*, vol. 2006, pp. 125–134, Ottawa, Ontario, Canada, June 2006.