ESUIF: An Open Esterel Compiler

Stephen A. Edwards

Department of Computer Science

Columbia University

www.cs.columbia.edu/~sedwards

Not Another One. . .

My research agenda is to push Esterel compilation technology further.

We still don’t have a technique that builds fast code for large programs.

No decent Esterel compiler available in source form.

Brief History of Esterel Compilers

Automata-based

- V1, V2, V3 (INRIA/CMA) [Berry, Gonthier 1992]
- Still the best for small programs with few states
- Does not scale

Netlist-based

- V4, V5 (INRIA/CMA)
- Scales very nicely
- Produces code that runs hundreds of times slower for sequential programs

Only executables available (**`www.esterel.org`**)

Brief History of Esterel Compilers

Control-flow-graph based

- My work: EC [DAC 2000, TransCAD 2002]
- Produces very efficient code for acyclic programs only

Discrete-event based

- SAXO-RT [Weil et al. 2000]
- Produces very efficient code for acyclic programs only
- Being improved at Esterel Technologies?

Both proprietary; unlikely to be released.

Neither currently copes with statically cyclic programs.

ESUIF

New, open-source compiler being developed at Columbia

Based on SUIF 2 system from Stanford University

Much more modular: implemented as many little passes

Common database represents program throughout

SUIF 2 Database

Main component of the SUIF 2 system

User-customizable object-oriented database

Written in C++

Not highly efficient, but very flexible

SUIF 2 Database

Database schema written in their own “hoof” format

C++ implementation automatically generated

```
concrete MyClass {
  int x;
}
```

$\Rightarrow$

```
class MyClass : public SuifObject
{
public:
  int get_x();
  void set_x(int the_value);
  ~MyClass();
  void print(...);
  static const Lstring
    get_class_name();
}
```

Three Intermediate Representations

AST-like representation from front end

- Primitives: abort, emit, present, suspend, etc.

Lower-level “C-like” representation

- Primitives: if-then-else, try, resume, parallel, etc.

C code

- Primitives: if, goto, expressions
- SUIF 2 includes a complete C schema

My New Intermediate Representation

Intermediate Representation

```
var := expr
if (expr) { stmts } else { stmts }
Label:
goto Label

break n
continue
try { stmts } catch 2 { stmts } ...
resume { stmts } catch 1 { stmts } ...
parallel { resumes } catch 1 { stmts } ...

fork Label1, Label2, ...
join
```

Intermediate Representation

```
var := expr
if (expr) { stmts } else { stmts }
Label:
goto Label
```

Self-explanatory

Signals represented as variables.

Restrictions on where a goto may branch.

Intermediate Representation

```
break n
continue
try { stmts } catch 2 { stmts } ...
resume { stmts } catch 1 { stmts } ...
parallel { resumes } catch 1 { stmts } ...
```

Numerically-encoded "exceptions"

Based on Esterel's completion codes

0=terminate 1=pause 2,3,. . . =exit

Implementing Exceptions

```
trap T1 in        try {
  exit T1           break 2           goto Catch2;
                                      goto Catch0;
handle T1 do      } catch 2 {       Catch2:
  c := 1            c := 1            c = 1;
end               }                 Catch0:
```

`try` becomes a few labels.

`break` becomes a goto.

Resume/Continue

```
abort      resume {                      goto E
                                      C: switch (s) {
                                         case 0: goto St0;
                                         case 1: goto St1;
                                         }
 pause      break 1                   E: s = 0; goto Ca1; St0:
 pause      break 1                      s = 1; goto Ca1; St1:
                                         goto Ca0;
           } catch 1 {                Ca1:
             break 1                    so = 0; goto Ca1o; St0o:
when A       if (!A) continue           if (!A) goto C;
           }                          Ca0:
```

`resume` becomes a multi-way branch plus some labels.

`continue` sends control to the multi-way branch.

Resume/Continue

First cycle:

```
   goto E
C: switch (s) {
   case 0: goto St0;
   case 1: goto St1;
   }
E: s = 0; goto Ca1; St0:
   s = 1; goto Ca1; St1:
   goto Ca0;
Ca1:
  so = 0; goto Ca1o;

St0o: if (!A) goto C;
Ca0:
```

Second cycle:

```
   goto E
C: switch (s) {
   case 0: goto St0;
   case 1: goto St1;
   }
E: s = 0; goto Ca1; St0:
   s = 1; goto Ca1; St1:
   goto Ca0;
Ca1:
  so = 0; goto Ca1o;

St0o: if (!A) goto C;
Ca0:
```

Parallel and Exit

trap T1 in	`try {`
trap T2 in	`  try {`
	`    parallel {`
	`      resume {`
exit T1	`        break 3 }`
||	`      resume {`
exit T2	`        break 2 }`
	`    } catch 1 {`
	`       break 1; continue }`
handle T2 do emit B end	`  } catch 2 { B := 1 }`
handle T1 do emit A end	`} catch 3 { A := 1 }`

Parallel

pause;

pause

||

pause

```
parallel {
  resume {
    break 1
    break 1
  }
  resume {
    break 1
  }
} catch 1 {
  break 1
  continue
}
```

Parallel Behavior

```
parallel {
  resume {
    break 1
    break 1
  }
  resume {
    break 1
  }
} catch 1 {
  break 1
  continue
}
```

```
parallel {
  resume {
    break 1
    break 1
  }
  resume {
    break 1
  }
} catch 1 {
  break 1
  continue
}
```

A Minor Point on Completion Codes

Berry's encoding reduces the exit code if it is not handled.

```
try {
  break 5
} catch 2 { ... }
```

generates **`break 4`** in Berry's encoding.

I assign each trap its own completion code; they pass unchanged.

Simpler semantics vs. the danger of larger codes.

Irrelevant in HW, probably not a problem for SW.

Code Generation Ideas

Static Unrolling

Can always evaluate cyclic programs by computing least fixed point through iteration:

$$\mathrm{lfp}(F) = F^n(\bot)$$

Suggests three-valued evaluation is necessary. What does that mean with control-flow?

Theorem

Suggested by Berry:

If F is monotonic, has a unique least fixed point that is maximal (i.e., $\mathrm{lfp}(F) \sqsubseteq y$ implies $y = \mathrm{lfp}(F)$), and is defined on a finite CPO, then it can be computed using

$$\mathrm{lfp}(F) = F^n(x)$$

where x is two-valued and n is the height of the domain.

Proof: $\bot \sqsubseteq x$ (trivial), so $F(\bot) \sqsubseteq F(x)$, $F^2(\bot) \sqsubseteq F^2(x)$, $\ldots$, $F^n(\bot) \sqsubseteq F^n(x)$. However, since $F^n(\bot) = \mathrm{lfp}(F)$ is maximal, we must have $\mathrm{lfp}(F) = F^n(x)$.

Implications of Theorem

Our functions are such that if x is two-valued then $F(x)$ is two-valued. This implies the sequence

$$x, F(x), F^2(x), \ldots, F^n(x) \tag{1}$$

is also two-valued. Therefore, the computation can be carried out using purely two-valued variables.

Note that (1) is not necessarily increasing.

Implications of Theorem

Approach:

Program must be proven causal using some other mechanism

Evaluate program through relaxation: start with arbitrary initial guess and evaluate to convergence.

Evaluation carried out with two-valued variables

Iteration strategy can be accelerated using Bourdoncle or my thesis.

Implications of Theorem

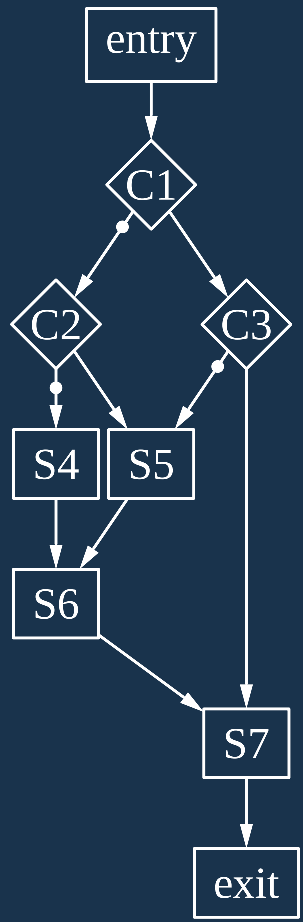
Unroll program according to connectivity.

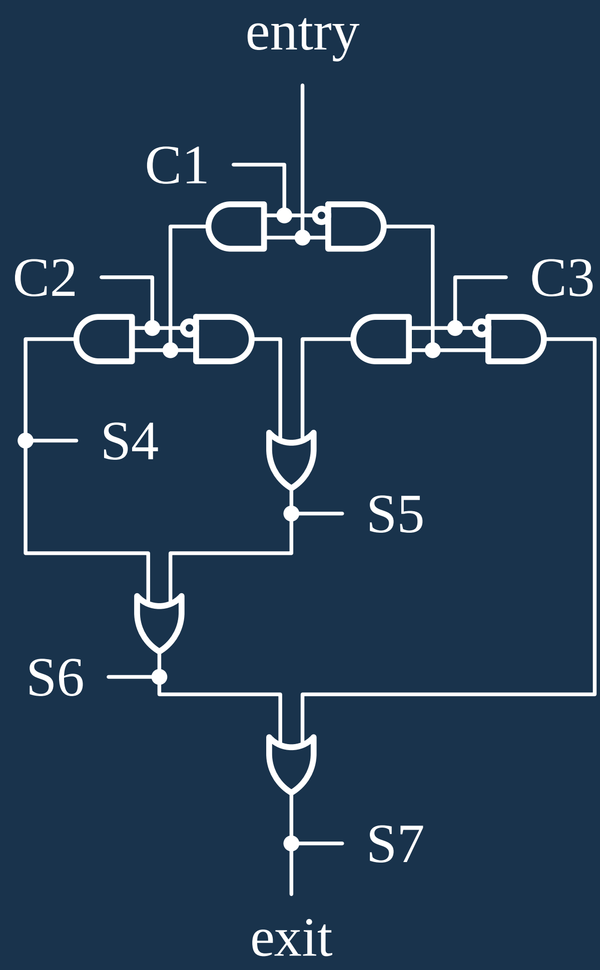
Constant propagate to simplify.

Execute result: two-valued logic only.

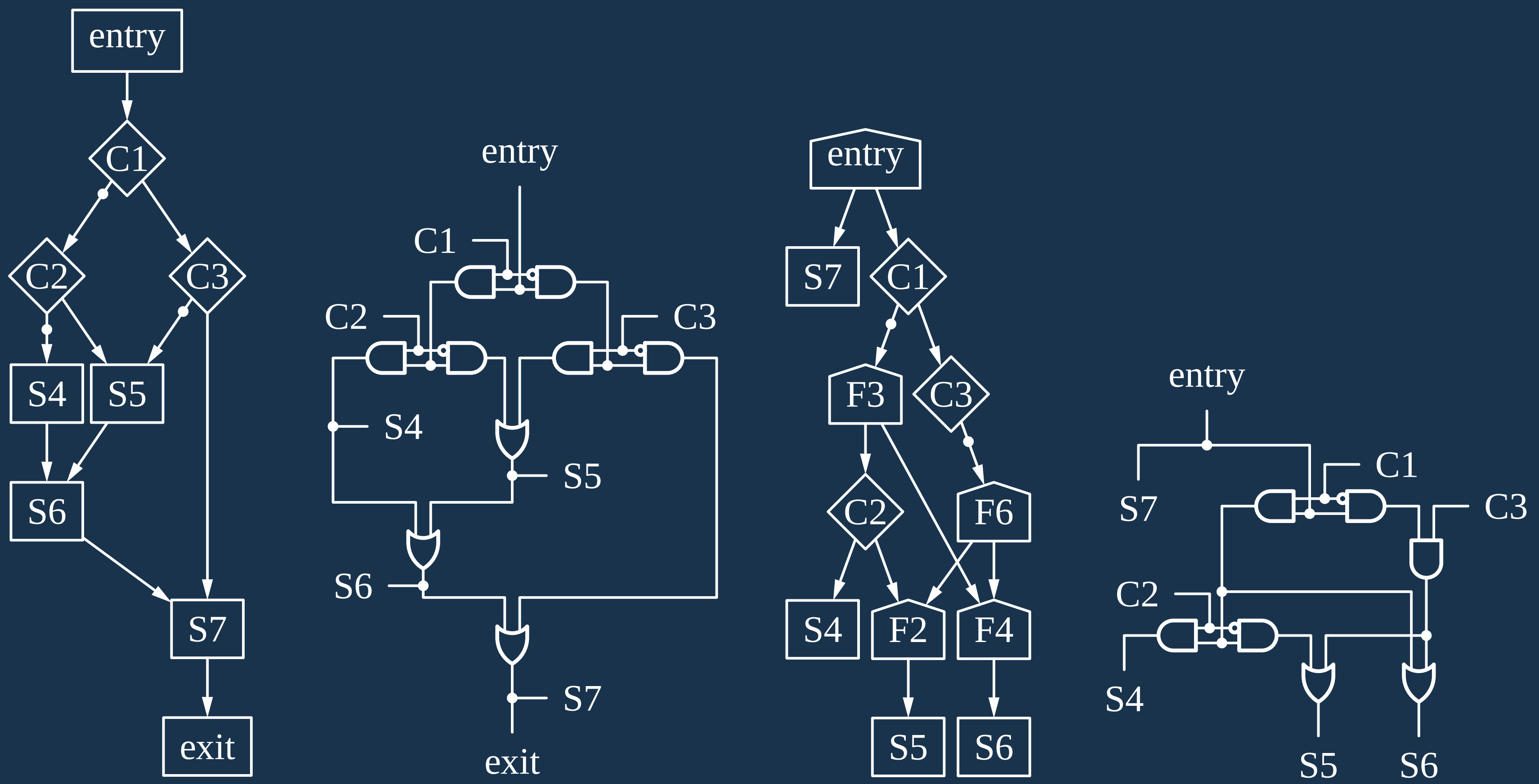
Program Dependence Graph

```
if (C1)
  if (C2)
    S4;
  else
    L: S5;
  S6;
else
  if (C3)
    goto L;
S7;
```





Program Dependence Graph



Program Dependence Graph

Also applicable to software generation

Transform to PDG, then generate code that executes PDG.

Some PDGs can be synthesized directly; others require additional predicates when sequentialized [Ferrante et al., Steensgaard]

Heuristics needed to keep number of predicates minimized.

Discrete-Event Approaches

Pioneered by Weil et al. [CASES 2000]

Efficient, but scheduler is fixed at compile time.

Does not handle statically cyclic programs.

Techniques such as French et al. [DAC 1995] schedule as much as possible beforehand, but retain some dynamic behavior.

Discrete-Event Approaches

Dealing with schizophrenia and causality appear to require code duplication.

Actually not really: just need to execute some code more than once.

Discrete-event scheduler ideal: have it invoke certain subroutines multiple times.

Small loss of efficiency in return for no code size increase.

Conclusions

- New ESUIF compiler
  - Based on SUIF 2 infrastructure
  - Open-source, under development
- Intermediate Representation
  - Numeric exception codes
  - Simple translation into assignments and branches
- Code Generation ideas
  - Static unrolling with two-valued evaluation
  - Program dependence graph approach
  - Discrete-event Approaches