Code generation from an Esterel PDG

Cristian Soviani
Department of Computer Science
Columbia University

www.cs.columbia.edu/~soviani
soviani@cs.columbia.edu
Esterel

- Hybrid programing language; both s/w and h/w flavour
- Developed by Gerard Berry starting from 1983 [2]
- Very solid mathematical background
- Synchronous model of time, concurrency, determinism
- Suitable for embedded systems design
- Can both be translated in software and hardware
- s/w: performance (code size / speed) is critical
- Project goal: to efficiently compile Esterel into software
Related work

- Automata Compilers: V3 Compiler [Berry, Gonthier][2] very fast
code size can exponentially grow for large programs

- Netlist Compilers: V5 Compiler [Berry][1]
code size grows linear w/ source input - large programs slow code because of “idle” instructions

- Halt points functions [Bertin, Weil, et al.’s][3][7]
good overall speed / size performance

- EC [Edwards][4]
sees Esterel as an imperative language
code size almost identical to netlist
much quicker, still slower than automata compilers
My work: Esterel PDG to CFG

Starts from PDG (Program Dependency Graph) - concurrent intermediate representation
Generates CFG (Control Flow Graph) - sequential - can be trivially translated to code

- computes data dependencies and remove deps. between mutual exclusive nodes

- uses Edwards [4] technique of thread slicing and interleaving; to minimize context switches, replaces EC’s depth-first with a more efficient algorithm

- uses a modified Simons & Ferrante’s algorithm [5] to order siblings according to data / flow dependencies

- generates the CFG adding guard vars when necessary
What is a PDG

- PDG is a very used intermediate format in compiler design
- Consists from a CDG (Control Dependency Graph) and a DDG (Data Dependency Graph)
- Compared to IC: a more high level abstraction of the program
- PDG can be efficiently optimized
- PDG is a better starting point than other intermediate formats
DDG - red edges

N1

A

B*

C*

B=1

C=1

X=1
Compute the DDG

- my program input is CDG
- DDG will be computed by looking at variable names; use Esterel particularities
  - if a signal is emitted by several instr., the result does no depend on order; they can be read only after emitted by all
  - if a var is written by a thread, another thread can’t read or write it
- remove data deps. between mutual excl. nodes
- more complex analysis is required to detect all dependencies which can be removed
Remove unnecessary data deps.

Dotted edges will be removed
Slice the PDG

Cyclic dependencies between threads

Interleaving is mandatory
Cutting the threads
Fighting for minimum cuts

- following Edwards’ EC: interleave threads
- add “state” variables for thread resuming
- interested in minimum number of cuts
- EC uses a depth first approach
- my approach: detect threads which has to be cut
- a greedy algorithm makes minimum number of cuts
- to do: minimum number of additional variables
Order siblings

- Simons and Ferrante describe a \( O(VE) \) algorithm when a concise CFG exists [5]. Steensgaard extends it. [6]
- The problem is reduced to the ordering of siblings
- External edges are the biggest problem
- For each node a “eec” (external edge condition) set is computed. Based on “eec”, siblings are ordered using a set of rules
- Only particular PDGs have a corresponding concise CFG
- When a concise CFG does not exist, the algorithm stops
- But it points out where guard variables / code duplication are necessary
EEC ordering rules

\( X \in eec(Y) \) iff \( X \) executes if any descendents of \( Y \) executes

\( X \not\in eec(Y) \) \( Y \) has an external edge with respect to \( X \)

- it is possible to simply schedule \( X \) before \( Y \)
- to schedule \( X \) after \( Y \), guard variables are required
- this relationship can be written as \( X < Y \)

Ordering is done by comparing siblings.

- data dependencies have priority

- if no guard variable is needed, the algorithm is guaranteed to insert none
PDG with a concise CFG
PDG without a concise CFG
Generate CFG

- now each region’s children are ordered
- CFG generation is straightforward if no guard variable is needed
- guards variables simulate function calls without introducing overhead
- careful not to introduce unnecessary additional code
- CFG is generated in two steps
- CFG has an obvious translation to code
CFG with additional guard variable
Results

- the program was tested on “problem” handwritten inputs
- both input PDG and generated CFG were exhaustively simulated: results match
- the results are encouraging

To do:
- test it on a real Esterel PDG, generated by ESUIF
- optimize the thread cutting “frontiere”
- optimize sibling ordering when guard variables are required
- optimize the PDG: open problem
Prof. Edwards’ favorite Esterel sample

module Example:
input S, I;
output O;

signal A,R in
every S do
  await I;
  weak abort sustain R
  when immediate A;
  emit O
||
loop
  pause; pause;
present R then emit A end
end
end
end module

PDG (simplified)
PDG

after cutting the threads

0-20
This is the CFG

PDG stats - 29 nodes:
7 regions
7 statements
15 predicates

CFG stats - 24 nodes:
8 predicates
16 statements
worst execution path: 15 instructions
0-21
PDG of a very *** BAD *** sample
Note the abundance of external edges
PDG
after cutting the threads
0-23
This is the CFG

PDG stats - 21 nodes:
8 regions
6 statements
7 predicates

CFG stats - 42 nodes:
16 predicates
26 statements
worst execution path: 27 instructions
simulation space: 2880 input combinations
0-24
References


