Code generation from an Esterel PDG

Cristian Soviani

December 16, 2002

Abstract

If a concise CFG exists for the given PDG, an optimal CFG can be efficiently generated. This is not the case for most Esterel programs. Solving the general problem optimally is NP-complete, so my project will find an non-optimal but efficient CFG.

1 Introduction

From the programmer's point of view, Berry's [2] synchronous programing language Esterel [1] can be the ideal choice for writing a large class of embedded systems software. The real battle is performance. Speed and size are both crucial issues in embedded systems. If the Esterel compiler does not generate very efficient code, the programmer will have to switch back to C.

Berry's Esterel V3/V5 compilers can generate both automata [2] and netlist code [1]. The automata code is build through exhaustive simulation of the program possible states; the code is very fast but code size can grow exponentially with the input source. The netlist code grows linear in terms of input source but speed is much slower as all instructions (most of them idle) are executed each cycle. These methods have a solid theoretical background (they are excellent tools for program analysis) and can be seen as two theoretical extreme cases but run-time performance requires a more pragmatic view.

Bertin, Weil et al.'s Esterel compiler [3] [8] splits the code into small compiled functions between "halt-points". Control and data dependencies are taken into account so these "halt-point" functions can be safely executed top to bottom when scheduled. They are then topologically sorted. Each function can be marked to be executed or not when its turn comes. If executed, it can mark as "executable" other functions (in the next or current cycle). The code speed and size are promising. This approach excludes most idle instructions from execution but does not take advantage from mutual exclusive code sections; each cycle, the scheduler must check the "executable" mark for all functions

In his EC compiler, Edwards [4] notes that despite of many differences, Esterel can be after all seen as an imperative language. Inspired form Lin's work [5] (which compiles a rendesvous like concurrent variant of C), EC translates Esterel's intemediate code (IC) into a concurrent intermediate representation (CCFG - concurrent control flow graph) and than into a sequential CFG (which is actually code). The main challenge is generating the CFG from the multithreaded CCFG when data dependencies require "thread interleaving". EC statically "slices" the threads and introduces additional variables to store the thread "state" at cut points. The variables are later used to "resume" thread execution. Edwards' work is sustained by the experimental results which show very good performance (code speed / size) for a large class of real world Esterel inputs. At this moment, EC can handle only programs where a statically ordering of the instructions can be computed (i.e. they can be topologically sorted); fortunately, this is the case for most real programs.

Although each of above approaches has encouraging results, there is still room for improvement.

My approach is to generate the CFG and thus the code from the PDG (program dependency graph). The PDG is an intermediate representation of the

[1] www.esterel-technologies.com

program, much used in compiler design, consisting from a CDG (control dependency graph) and a DDG (data dependency graph). Instead of being a simple "translation" of the program, like IC, the PDG is a higher program abstraction, ignoring arbitrary links between nodes and keeping only mandatory (control and data) dependencies. Although PDG to CFG translation is not trivial, it is a promising way to get both short and fast code.

Simons and Ferrante describe an efficient algorithm [6] for generating a CFG from a PDG, when a concise CFG exists (i.e. no additional guard variables / code duplication is required). Their ingenious algorithm reduces the problem to the ordering of each parent node's children (siblings) and runs in polynomial time O(VE). The algorithm walks the CFG twice and computes for each node a "eec" set. Briefly, this information is used to detect CDG constraints in scheduling siblings. Sibling ordering is done by inspecting the "eec" sets and data dependencies, using a set of ordering rules. The algorithm stops when finding a concise CFG is not possible but it points out that guard variables / code duplication are needed to solve the problem.

Steensgaard extends Ferrante's work to handle irreductible programs which contain multiple entry loops [7]. This is done by introducing the notions of loop entry and close nodes. His work still considers only PDGs for which a concise CFG exist.

Edwards showed the general problem of finding optimum code is NP-complete. Fortunately, Esterel PDGs have some particularities which allow efficient algorithms. Due to the timing model, the PDG has no loops; due to signal semantics, any signal can be assigned by multiple instructions in arbitrary order; the signal can be read only after it was written by all instructions.

2 My work

2.1 Overview

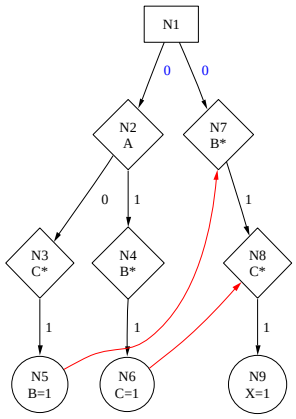
My program takes an Estrel PDG and generates the corresponding CFG in 4 main steps:

- Constructs the DDG, eliminating unnecessary data dependencies
- Slices the PDG using Edwards' technique to remove cyclic dependencies between threads, such that resulting PDG has no interleaving threads and can be simple scheduled; thread cuts are bad for both size and speed, so their number should be reduced as much as possible
- Runs Ferrante's algorithm and orders the siblings in the PDG according to data dependencies and Ferrante's ordering rules; if guard variables are required, their number should be kept low
- Generates the sequential flow (CFG). Code genaration is trivial from this point.

2.2 Compute and Relax data dependencies

This takes the CDG and computes the DDG by looking at data relationship between nodes. It is done in 2 steps. The result is the PDG.

In the first step each predicate is linked to all the statements which refer to the same variable. In the second step each dependency is checked; if the 2 nodes can't be both executed in the same tick, the dependency is removed (this is done by looking at the common ancestors). In the next sample, red edges are kept but the links (N5 to N4) and (N6 to N3) are removed:



My assumption is that any predicate ancestor can have any value, independently. This is not true for

certain cases. But only a more sophisticated tool which simulates all possible states and input values can detect if 2 nodes can be really both executed in the same tick or not (see Esterel constructiveness causality). My program could use the output from such a tool and skip this first module.

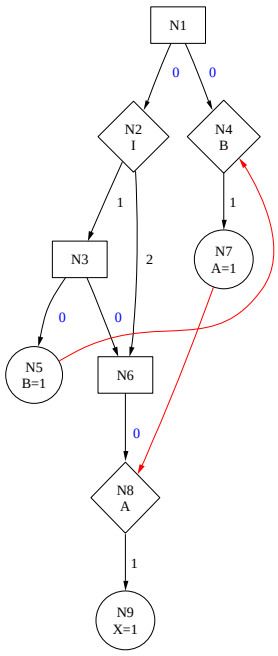
```
compute_relax_ddg()
//step 1
for each predicate p
  for each stat s referring the same var as p
    add data dep from s to p
//step 2
for each data dep s to p
  keep=false
  comm=common region ancestors of s and p
  for each f in comm
    for each child cf of f not in comm
       if cf is ancestor of s XOR p
          keep=true
  if keep==false remove data dep s to p
end
```

2.3 Slicing the CDG

For each region with multiple children I must assure there exists at least an ordering of the siblings (threads). The regions are considered hierarchically, top to bottom, so as less as possible work is moved to upper levels.

For any such region, if there are cyclic data dependencies between siblings, the execution of threads should be "interleaved". This follows the technique used by Edwards in his EC [4] with the mention that EC uses a simple "depth-first" approach but my project reduces the number of context-switches to the minimum possible. This is important as context switches do both increase code size and execution time, and I expect heavy threaded programs to take best advantage of my algorithm.

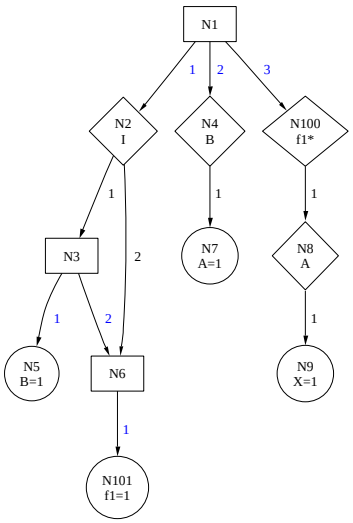
Next there is a simple sample of a PDG which requires interleaving. Note the cyclic data dependencies (red edges) between the two siblings:



The left thread is a "bad" one because N5 and N8 can not be executed in the same "chunk"; N8 should execute after N7, N7 after N4 and N4 after N5. The solution is to cut the "bad" thread in 2 slices; the "upper" part remains in place - it can be scheduled first; the "bottom" part is "relinked" to the region, inserting appropriate guard variables for preserving the flow control - in can be scheduled later. This operation can be done several times and will safely lead to a final result, as there are no cyclic dependencies between signals; in the extreme worst case a one level schedule (netlist - like) could be generated. But the idea is to keep the number of "cuts" low.

Fortunately, a greedy algorithm works. For each "bad" thread (which must be obviously sliced), the "upper" part will include all nodes which do not depend on nodes on the same thread, using nodes on another threads as "intermediary". The nice result is this simple algorithm make the minimum number of cuts possible.

Next there is the result of the algorithm. Note that there are no more cyclic dependencies between siblings, so they can be scheduled in the order N2, N4, N100:



The following code is an outline of the cutting algorithm. Instead of adding just a predicate and the corresponding statements, a more complex algorithm is actually used to construct a "mirror" tree which preserves the flow control with minimum overhead.

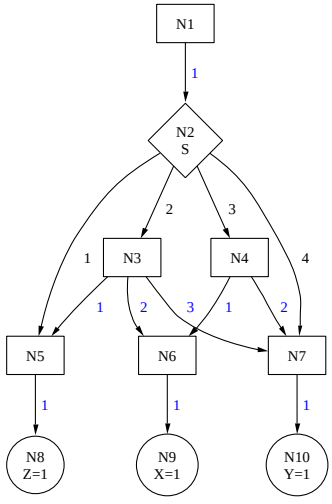
```
slice_pdg()
for all regions f (top->bottom)
  mark all children 'not onepiece'
  while f has a child c not marked 'onepiece'
    slice(f,c)
    mark c as 'onepiece'
end

slice(f,c)
mark all descendents of f as CH
mark all descendents of c as SON
for each n descendent of c (top->bottom)
  if(n has a RMV parent) mark n as RMV
  if(n depends of another node in CH using
     SON nodes as intermediary)
        mark n as RMV

if any node was marked RMV
  create a new pred r
  link r as son of f
  for each n descendent of c (bottom->top)
  if n is RMV and its parents are not RMV
     add a new statement s w/ the same var as r
     link s to n's parent
     relink n to r instead of its parent
```

2.4 Ordering the siblings in the CDG

For each region with multiple children, a scheduling is possible due to the slicing, but additional guard variables/predicates may be necessary. To keep them as few as possible, I use the algorithm developed by Simons & Ferrante [6] which computes each node's "external edge contition" (eec). The technique is also described by Steensgard [7].

By definition, $X \in eec(Y)$ iff X executes if any descendent of Y executes. If $X \notin eec(Y)$, Y has an external edge with respect to X. Briefly, X can be simply scheduled before Y but not after Y, and we can write it as $X < Y$. If we can find such an order for all siblings which also respects data dependencies, no guard variables should be added; otherwise, they are mandatory. A "bad" schedule can be forced by data dependencies and/or by siblings having external edges with respect to each other (i.e. $X \notin ecc(Y)$ and $Y \notin ecc(X)$, meaning $X < Y$ and $Y < X$). Fortunately, this rarely happens so Ferrante's ordering saves us from additional code most of the time.

In the next sample PDG, N4 and N3 schedule their children as shown by the blue numbers. It can be noticed that an additional predicate is required for N5 but not for N6 and N7.




```
order_sib()
```

```
while there unscheduled siblings
  poss=all possible siblings, according to data dep.
  schedule nextsib(poss)
end

nextsib(poss)
  x=first(poss)
  for each y of (poss-x)
    if(y<x) x=y;
  return x;
```

Note that statements and predicated without external edges are first scheduled.

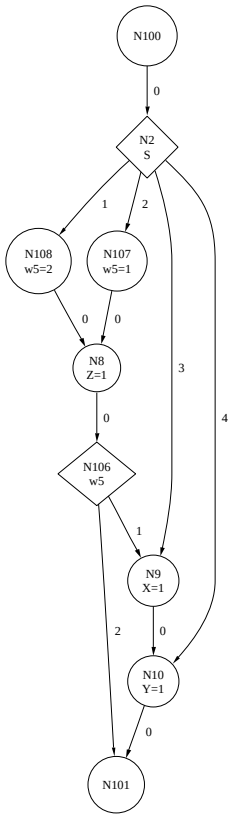
2.5 Generating the CFG

Now we have the scheduling order for all siblings. As a concise CFG do not always exist (in that case its construction is trivial) my program generates the CFG in two steps, to avoid inserting unnecessary additional code.

First, the CDG is walked top to bottom and additional predicates / statements are inserted for any region with multiple parents. “wire” nodes are added to simplify the CFG construction (a wire can have multiple parents but only one child - briefly, a region is replaced by several wires). A parent passes to each child a “wout” node, which briefly is the node where execution resumes after the child completes; this is either an executable node (pred. or statement) or a wire. All regions loose their links (which are moved to wire nodes) and are removed. Finally, all wires are removed (updating the links) so the result of the first step is a valid but unoptimum CFG.

Second, the CFG is walked bottom to top and predicates are “simplified” (when 2 children edges go actually to the same node) or even removed (when there is only one child left); in the same time, the corresponding statements are also simplified / removed. This assures that only necessary additional predicate / statements are kept.

Next: the CFG generated from the previous PDG. Note the additional predicate N106, corresponding guard variable w5 and its assignment in N107 and N108:




```
//step 1
makecfg()
for each node n in CDG (top -> bottom)
  if n is 'stat' or 'pred' wire_ps(n)
  if n is 'reg' wire_f(n)
for each node w of type 'wire'
  relink w's parents to w's child
  remove w
//step 2
for each node n in CFG (bottom ->top)
  if n is additional predicate minpred(n)
end

wire_ps(n)
  get wout from father
  push wout to all children
  is n is 'stat' or 'pred' w/ default case
     link n to wout
wire_f(f)
  if f has multiple parents
```

```
    make new predicate P
    make new node W of type 'wire'
    for each parent pi
       make new statement Si
       link pi to Si, Si to W, P to pi's wout
    wout=P; win=W
  else
    get wout from father
    win=father
  nc=#of f's children
  make nc-1 nodes Wi of type 'wire'
  for each child ci in sched order
    link win to ci
    if(ci is last child)
       push wout to ci
    else
       push W[i+1] to ci
    win=Wi
  delete f
minpred(p)
  while(p has 2 identical children i and j) do
    remove link j
    remove corresponding Sj
    link Sj's parents to Si
  if(p has only one child)
    remove p, remove corresponding S
```

3 Conclusions. To do

I consider the results to be encouraging as my project efficiently handles "problem" handwritten test programs. The input PDG and the generated CFG were exhaustively simulated and the results match. The next step is to test real Esterel input - using Edwards' ESUIF development platform - and to compare the results against other Esterel compilers.

There are several optimizations which can further improve the code performance. The thread "cuts" are done by the greedy algorithm as "low" as possible, but a more careful choice could do better, without affecting the number of cuts. The ordering of "bad" siblings can be improved, to assure minimum additional code when they are required. And finally, the PDG can be optimized exploiting Esterel's particularities.

References

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

[2] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. 19(2):87–152, November 1992.

[3] Valérie Bertin, Michel Poize, and Jacques 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.

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

[5] Bill Lin. Efficient compilation of process-based concurrent programs without run-time scheduling. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, pages 211–217, Paris, France, February 1998.

[6] Barbara Simons and Jeanne Ferrante. An efficient algorithm for constructing a control flow graph for parallel code. Technical Report TR–03.465, IBM, Santa Teresa Laboratory, San Jose, California, February 1993.

[7] Bjarne Steensgaard. Sequentializing program dependence graphs for irreducible programs. Technical Report MSR-TR-93-14, Microsoft, October 1993.

[8] Daniel Weil, Valérie Bertin, Etienne Closse, , Michel Poize, Patrick Venier, and Jacques Pulou. Efficient compilation of Esterel for real-time embedded systems. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 2–8, San Jose, California, November 2000.