Compiling Esterel into Better Circuits

Jia Zeng

Abstract

Producing efficient circuits from a high-level language such as Esterel remains a problem. Sparse state coding requires many more latches used than minimum and waste of reachable state space, while tight state encoding produces slow circuits due to the cost of encoding and decoding.

This paper presents an algorithm to generate small and fast circuitry for Esterel. There are three main parts of the algorithm: state assignment, hardware synthesis, and circuit optimization. The technique is based on Program Dependence Graph. It uses heuristic search in coding space, computes the cost and adjusts until finding a compromise point on latch/logic tradeoff.

The algorithm will be used to compile Esterel into small circuits that meet a timing constraint.

1 Introduction

Esterel is a high-level language designed for real-time systems. It supports high-level control constructs such as concurrent composition, preemption, and exceptions. This aspect makes Esterel a more challenging language to translate into circuitry, but also enable aggressive optimizations because the compiler is able to gain a better understanding of the program's behavior.

State assignment to Esterel is based on the Program Dependence Graph (PDG) of Esterel. Baxter and Bauer present it in their early paper. It is based on the concept of control flow, and preserves all information of an original Esterel program. Esterel supports implicit state machines through explicit and implicit pause statements that delay for a cycle, such as the await statement. States sustain and transfer only between these statements. So we assign states for each of them. Figure 1 gives an example for PDG.

Heuristic search is used in the algorithm to find an efficient state coding. Three main kinds of coding are used in the searching space. First is Berry's [1] one-hot encoding. It produces fast circuits while gives much redundant state space. Second is Edwards's [2,4] group-hot-by-level encoding. It shares latches between those decision nodes whose parents are in the same level but not parallel. Third is some variant of the former two encodings. It shares latches between the decision nodes in some levels, but not for all levels where sharing is possible. In other levels, it still keeps one-hot encoding for the nodes.

We use heuristic search in the state encoding space until we find a resolution with the fewest latches under the requirement, or until the search space is exhausted. We start from one-hot encoding. If it can't meet the requirement given, we fail and return. Otherwise, we'll try variant coding means to delete sharable latches. Every time after re-encoding, the cost of the circuit is re-evaluated. If it is higher than required, the new coding will be thrown away and the former code will be returned. Or we'll repeat the process until we exhaust the searching space. Thus we find the most efficient coding that meets the cost requirement.

The Esterel hardware synthesis is straightforward when the coding has been chosen.

There are two parts of circuit optimization: combinational optimization and sequential optimization. This paper is concentrate on the sequential optimization. And in fact, the sequential optimization has been done at the stage of state encoding before generate real circuits. SIS, the standard public-domain optimizer, will be used to for the combinational optimization after generating circuits.

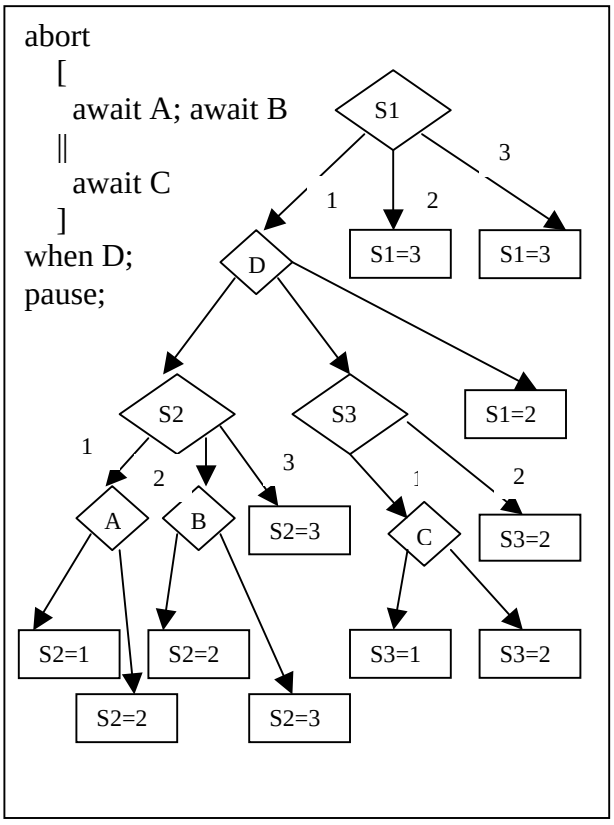


Figure 1 PDG for an example program

2 Related Work

The classic state assignment is based on Simple Finite State Machine. Hachtel and Somenzi [6] describe synthesis of finite state machines. It uses minimization of incomplete specified machines to get reduced reachable states.

Villa and Sangiovanni-Vincentelli [7] present algorithms used in NOVA for optimal state assignment of FSMs. It is based on the state code adjacency concept but more efficient and flexible. NOVA represents constraint satisfaction as a graph-embedding problem. It uses heuristic search to resolve this problem. Its best strategy is "iohybrid_code", which produces results with quality comparable to the results of the maximum adjacent method. Its core algorithm is "ihybrid_code". The set of input constrains is partitioned into satisfied constrain set (SIC) and rejected constrain set (RIC) at the beginning. The algorithm first gives the coding with minimum length under the satisfied constrains. Then it increases the embedding cube to satisfy the RIC within the encoding space that is specified by the user. The iohybrid_code strategy takes similar steps as in ihybrid_code but also takes the output constrains into consider. Generally speaking, output constrains are in lower priority to input ones in this strategy.

Devadas, Ma, Newton, and Sangiovanni-Vincentelli [8] present a method called MUSTANG that is one of the earliest multi-level state assignment methods. It used the state code adjacency concept to reach the aim of maximizing the size and number of common cubes. It builds an attraction graph with weighted edges. An edge's weight is increased if it links to the common fanout and fanin states. MUSTANG was used to help MIS logic synthesis system reducing the number of product terms or literals needed to implement the next-state and output functions.

Berry first outlined the translation of Esterel into circuitry in 1992 [1], refined later to cover reincarnation. It generates a sub-circuit for each statement, and registers only for unit-delay statements. So each leaf state is encoded by one-hot coding. In that case, encoding and decoding circuits are trivial. But it uses many latches and results reachable state space redundancy. Later, Sentovich, Toma and Berry [3,5] described the technique for reducing the number of latches. They rely on computing the reachable state set implicitly using BDDs, then re-synthesizing the circuit using this knowledge to remove sequential redundancies. The whole program is taken as one state machine.

In the compiler we are building, more than one state machine are assigned in different level. It is necessary especially for parallel branches. We can share latches between sequential branches but need to avoid parallel ones.

Edwards [2] proposed three key means to advance Esterel hardware synthesis. First is the PDG. It takes a totally new structural translation to Esterel. Calculating control dependence in the graph, it removes the redundant circuit. That is much more efficient than removing by analyzing the circuit.

Second is a better state encoding. The technique he proposes chooses states encoding at a high level, providing much greater flexibility and larger encoding space to choose from.

Third is to use the don't-care information in logic synthesis. It gives more flexibility to the implementation and helps to generate high-quality circuits.

3 HDCompiler – From PDG to circuitry

HDCompiler is a translator to turn PDG into optimal circuitry for Esterel program. It visited a PDG tree and recorded all the state and their values in a state table. The states were encoded with one of the encoding means. Then it translated the PDG into circuitry based on the coding. After that, comes the circuit optimization.

To optimize the circuit, it deleted all the useless wires firstly– the wire no any other wire took it as a input. Secondly, it visited the PDG tree to find out shareable flip-flops and merge them. Thirdly, it found the slacks between gates by computing and adjusted the circuit with fewer slacks and optimal.

The main data structure of the hardware translator includes:

```
class PDGNode {
        int numb;
        Vector parent_numb;
        Vector brunch_val;
        int property;
        String val;
        int wire_entry;
}

class PDGTree {
        PDGNode node;
        PDGTree[] children;
}

class State {
//node# where the state decision made
        int node_numb;
//state number
        int numb;
//possible state values and their codes
        StateCode[] state_codes;
//wire entry for each state value given
        int[] wire_entry;
//wire# for each state value emitted
        int[] wire_emit;
}
```

```
class StateCode {
// to record the value assigned
        int val;
        // code after encoding
        int[] code;
// wire entry for each bit, i.e. each register
        int[] wire_entry;
}

class Wire {
         int numb;
         Vector input_wires;
         int gate_type;
         int[] arrival_t;
         int required_t;
}
```

3.1 State Encoding

As we said previously, there are three main kinds of coding. First is one-hot encoding, second is binary and maximally sharing encoding (compact encoding). And the third is a combination of the first two. That is,

Code-length = log_2(Number of possible state values)

HDCompiler realized the first two encoding. Due to time limit, it didn't realize the third. It will be added to the encoding space and used as an adjustment later. With the one-hot encoding and compact encoding, it's easy to be realized with combine them together in different levels.

3.2 Turning State Machine into Circuitry

To translate a PDG into circuitry, there are two main parts: combinational logic circuit translation, and sequential logic circuit translation, i.e. state machine translation. HDCompiler mainly concentrates on the second part Temporarily.

We assume the wires for the state-value-emit statements have been generated in the first part translation. HDCompiler grouped them and generated wires for each state value emitting, then saved the wire entries in the state table. But for the statement directly under a state decision node, the input wire entry was left unfilled.

For each state in the state table, decoding and encoding circuit was generated separately. To decode a state, we need to generate every bit's value of a state value. A flip-flop was generated for each bit. The input wire for it was set as:

bit[i] = OR(Σ_j Wires for state-value-emit statements for state_value[j] where bit[i] = 1 in the value code.

To encode a state, just follow its bit codes and the wire entry for every bit (i.e. the flip-flop output) stored in the state table. Notice that, to get a state machine running correctly, we also should AND each state machine's output with the wire for the statement to start the state machine. We only considered the case where the state machine has only one start entry, since multi-entry for a state machine is illegal in PDG.

After generating the circuit for the state machine, we went back to fill the input of some wires. These wire were generated for the statements whose execution follows the state decision node's result.

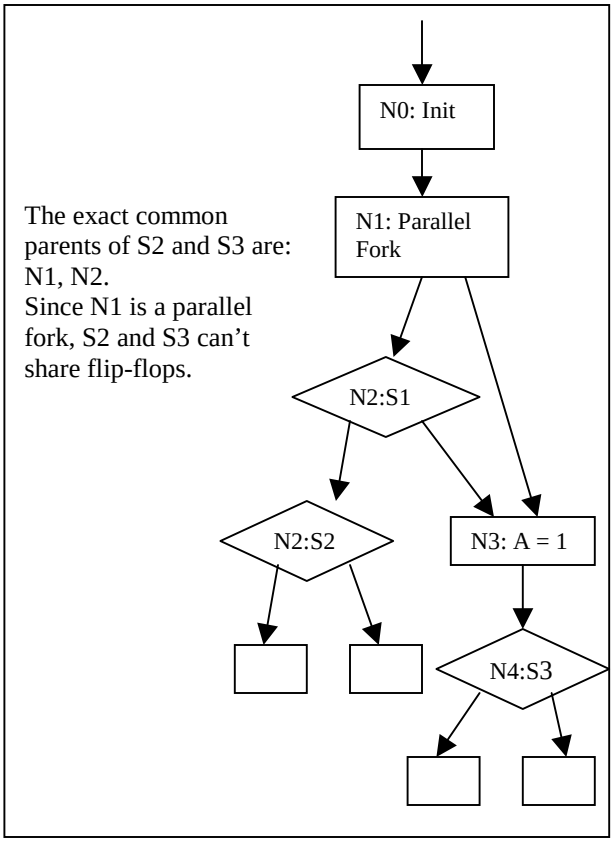
3.3 Flip-Flops Sharing

When there are more than one state machine in a circuit, some of them may share flip-flops. How to decide which of them can share flip-flops? We looked at every two state decision nodes – the entry of a state machine, if none of their *exact common parents* is parallel-fork node or their own, they can share flip-flops. Because it means the two state machines will never run simultaneously.

Here we defined exact common parent for two nodes. If Node P is an exact common parent of two nodes X and Y, it must have two properties:

1. P is a common parent of X, Y.
2. Assume P has direct children C1, C2, …, Cn,

$Set_x = \{C_i\}$ = {first node of a route from P to X}
$Set_y = \{C_j\}$ = {first node of a route from P to Y}



There must exist some element different between Set_x and Set_y.

Figure 2 gives an example.

Figure 2 Flip-Flops sharing

Temporarily, we only build the sets of two shareable state machines. In fact, these two-element sets can be merged to make bigger set. That means, three or more state machines can share flip-flops. For example, if machine A and B can share ffs, while A and C can share ffs either, we can conclude A, B and C can share ffs if only B and C can share ffs too. That will be helpful to choose which pairs to be shared in the program with many state machines.

3.4 Slack Computing

Slacks are computed for further optimization. In Esterel, it is assume all inputs are ready at the beginning of a circle. So we take the flip-flop as the start and end of a circuit circle module.

Slack for a circuit module is provides an upper bound of its possible delay increase without violating the timing constrain. It is import for circuitry optimization because it represents the potential capability of obtaining area/power reduction.

A well know procedure to compute slacks for a wire v is:

$$arrival\_t(v) = MAX_{u \in FI(v)} (arrival\_t(u)) + delay(v)$$

$$required\_t(v) = MIN_{w \in FO(v)} (required\_t(u) - delay(w))$$

$$slack(v) = required\_t(v) - arrival\_t(v)$$

We looked at the routes between every two flip-flops. We chain a route forward to compute the arrival time of every wire on it, while backward to compute the required time.

Due to time limit, we only got the slack but haven't used it to optimize the circuitry.

4 Experiment Result

Here we give an example for the test result. Figure 3 shows the input PDG.

In the example, there are 7 state machines. We are going to compare the circuits for different coding, circuitry before and after optimization, and show the slack-computing result.

Comparing the circuits for one-hot encoding before (Figure 4 right) and after optimizing (Figure 4 left), 4 flip-flops are merged without increase the size of combinational logic part.

Table 1 gives the comparing result between the circuits for one-hot encoding and group-by-level encoding (i.e. compact encoding), which are both optimized. We can see the combination logic part is slightly increased in compact coding circuit. But it has much more slacks, also more freedom on circuit optimization.

Figure 3 PDG for test example

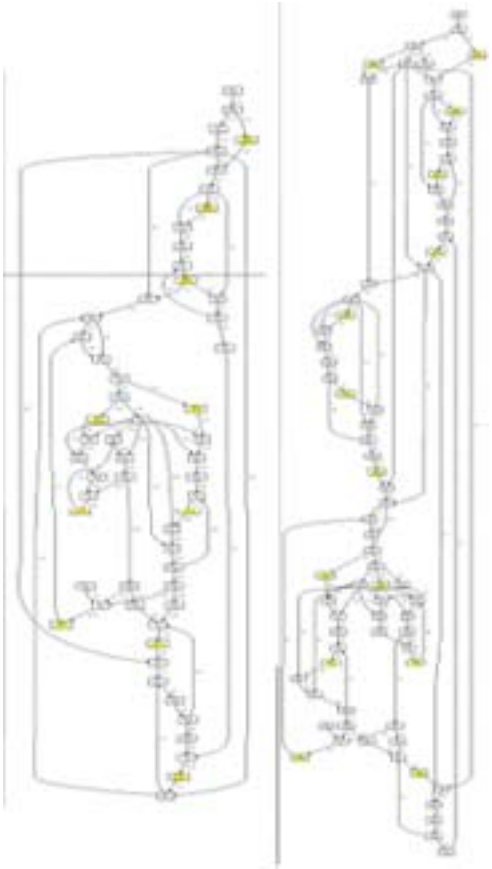


Figure 4 One-hot-encoding circuit after/before optimization

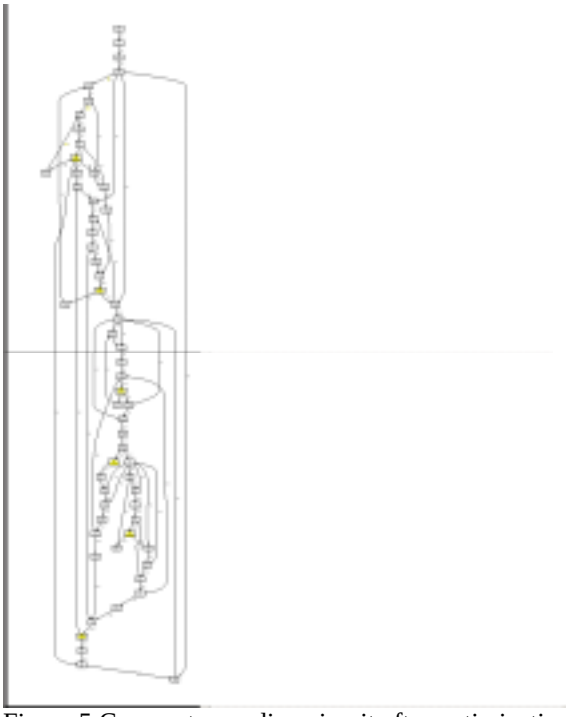


Figure 5 Compact-encoding circuit after optimization

#	Flip-flops	Gates	Wires	Slacks
One-hot	10	56	80	16
Compact	6	85	59	22

Table 1 comparing one-hot and compact encoding circuit

5 Conclusions and Future Work

The HDCompiler was built up a structure for the real Esterel hardware translator. It still needs further improvement. Instead of exact PDG, the input of the HDCompiler is a PDG metamorphose temporarily, which also include the state-machine-by-level information.

Some questions need to be answered in the future work. How to choose state encoding means? How to optimize the circuit based on the slacks? And the program is still buggy. Such as, for the state-emit statements whose exact common parents include parallel fork node, they should be combined with AND gate instead of OR gate to get the state machine running correctly.

7 Bibliography

[1] Gerard 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] Stephen A. Edwards. High-level synthesis from the synchronous language Esterel. In Proceedings of the International Workshop of Logic and Synthesis (IWLS). New Orleans, Louisiana, June 2002.

[3] Gerard Berry. Efficient latch optimization using exclusive sets. In Proceedings of the 34th Design Automation Conference, pages 8-11, Anaheim, California, June 1997.

[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), February 2002

[5] Ellen M. Sentovich, Horia Toma, Gerard Berry. Latch optimization in circuits generated from high-level descriptions. ICCAD'96, November 1996.

[6] Gary D. Hachtel, Fabio Somenzi. Logic Synthesis and Verification Algorithms. Keluwer Academic Publishers. 1996.

[7] Tiziano Villa, Alberto Sangiovanni-Vincentelli. NOVA: State Assignment of Finite State Machines for Optimal Two-Level Logic Implementations. In The Proceedings of the 26th ACM/IEEE Design Automation Conference, pages 327-332, June 1989.

[8] S. Devadas, B. Ma, R. Newton, and A. Sangiovanni-Vincentelli. MUSTANG: State Assignment of Finite State Machines Targeting Multi-level Logic Implementations. IEEE Transactions on -Computer-Aided Design, vol. 7, no. 12, December 1988.

[9] W. Baxter and H. R. Bauer, III. The program dependence graph and vectorization. In Proceedings of the Sixteenth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages, Austin, TX, 1989.

[10] Chunhong Chen, Xiaojian Yang and Majid Sarrafzadeh. Potential Slack: An Effective Metric of Combinational Circuit Performance. ICCAD, pages 198-201. IEEE, November 2000.